“ANOMALY DETECTION IN FINANCIAL PAYMENT SERVICES”

MINOR PROJECT

SUBMITTED in PARTIAL FULFILLMENT of

THE REQUIREMENT FOR THE AWARD OF THE DEGREE of

BACHELOR OF TECHNOLOGY

(COMPUTER SCIENCE AND ENGINEERING)

SUBMITTED BY

AMRIT SINGH – 00696302715

AYUSH SHARMA – 01296302715

VARDAAN ARORA - 40596302715

Under the guidance of

Mr. Adeel Hashmi

Department of Computer Science and Engineering

MAHARAJA SURAJMAL INSTITUTE OF TECHNOLOGY,

JANAKPURI DELHI-58

GURU GOBIND SINGH INDRAPRASTHA UNIVERSITY

DELHI, INDIA

DEC-2018

ACKNOWLEDGEMENT

We truly acknowledge the cooperation and help made by Mr Adeel Hashmi, Assistant Professor, Department of Computer Science and Engineering, Maharaja Surajmal Institute of Technology, C-4, Janakpuri , Delhi, Guru Gobind Singh Indraprastha University, Delhi. He has been a constant source of guidance throughout the course of this project. We would also like to thank him for his help and guidance in understanding anomaly detection in financial payment services. We are also thankful to our friends and family whose silent support led us to complete our project.

(Signature)
Amrit Singh (00696302715)
Ayush Sharma (01296302715)
Vardaan Arora (40596302715)

CERTIFICATE

This is to certify that the project entitled "Anomaly detection in financial payment services" is a bonafide work carried out by Mr. Amrit Singh, Ayush Sharma and Vardaan Arora under my guidance and supervision and submitted in partial fulfillment of B.Tech degree in Computer Science and Engineering of Guru Gobind Singh Indraprastha University, Delhi. The work embodied in this project has not been submitted for any other degree or diploma.

Mr. Adeel Hashmi
Assistant Professor
Dept. of Computer Science and Engineering
MSIT, Delhi

Dr. Naveen Dahiya
Head, Department of Computer Science and Engineering, 2nd shift
MSIT, Delhi.

LIST OF FIGURES

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Fraud, or criminal deception, will always be a costly problem for many profit organisations. Data mining can minimise some of these losses by making use of the massive collections of customer data, particularly in insurance, credit card, and telecommunications industries. However, fraud detection data being highly skewed or imbalanced is the norm. Usually there are many more legitimate than fraudulent examples. This means that by predicting all instances to be legal, a very high success rate is achieved without detecting any fraud. There can be two typical ways to proceed when faced with this problem. The first approach is to apply different algorithms. Each algorithm has its unique strengths, so that it may perform better on particular data instances than the rest. The second approach is to manipulate the class distribution (sampling). The minority class training examples can be increased in proportion to the majority class in order to raise the chances of correct predictions by the algorithm(s), or the majority class training examples can be decreased to decrease high skewness of data.

1.1 Purpose

Imbalanced class distribution is a scenario where the number of observations belonging to one class is significantly lower than those belonging to the other classes. This problem is predominant in scenarios where anomaly detection is crucial like electricity pilferage, fraudulent transactions in banks etc. This happens because machine learning algorithms are usually designed to improve accuracy by reducing the error. Thus, they do not consider the class distribution / proportion or balance of classes. This leads to false classification of many frauds as genuine transactions, which leads to losses to organizations and individuals. To combat this, we explore different resampling techniques to balance the two classes, i.e. majority and minority by using techniques like undersampling and oversampling, and compare recall scores obtained from different machine learning classifiers. We also explore ensemble learning in the form of XGBoost.

1.2 Project Scope

The scope of anomaly detection is not just limited to fraud detection. Outlier detection is a major problem in many notable fields like electricity pilferation, detecting oil spills, detection of diseases. Fraud detection in financial payment services was chosen as to demonstrate the capabilities of the discussed techniques. This can be used by major

financial institutions of the world like banks and credit unions, which can make the network of transactions much secure and sensitive to fraud detection.

1.3 Operating environment

Hardware

- Processor: Intel Celeron family and above
- OS: Windows 7 and above/ Ubuntu 16.04 LTS
- Graphics: Dedicated Graphics
- 4 GB RAM (64-bit)

Software:

- Text editor: PyCharm, Jupyter notebook
- Libraries: Matplotlib, Numpy, Scikit-learn, pandas
- Language: Python

1.4 Methodology

PaySim simulates money transactions based on a sample of real transactions extracted from one month of financial logs from a money service implemented in an African country. The original logs were provided by a multinational company, who is the provider of the financial service which is currently running in more than 14 countries all around the world. This dataset is first analyzed for useful features by using Exploratory Data Analysis. Two things are focussed here: to judge which type of transactions are fraudulent, and to check how the value of attribute i*sFlaggedFraud* is set. Queries are made on the dataset and relations are found. As it turns out, a few of the features are unclear on how the value is set to determine the corresponding *isFraud* value, and hence need to be dropped.

After this, data cleaning is done, which includes imputation of null values. In certain features, imputation lead to loss in original information provided by the dataset and hence the values were replaced with -1 and nan. To get a clear understanding of how *isFraud* is set, two new features are extracted, which determine the error in the balance amounts in sender and receiver accounts.

Once data is ready, resampling techniques are performed. Undersampling is done on the dataset to create a DataFrame containing all fraud instances and equal number of random

non-fraud instances, which are repeated for various c-values to determine a better recall. Various classifiers like Logistic Regression and SVM are run on this data.

Since undersampling leads to loss of information and each datapoint is crucial, another technique was implemented. This time, oversampling with SMOTE was done on the dataset which increases the number of minority data-points by creating synthetic similar data-points.

Classifiers are run on this data again to get a better recall value.

Finally, ensemble learning in XGBoost is performed which allows for weighting the positive class more compared to the negative class, a setting that also allows to account for the skew in the data.

1.5 Design and implementation constraints

- The machine must have all the latest version of libraries installed on the system to be able to run the machine learning model.
- The machine must have at least 4GB of RAM for processing of dataset as the dataset has 6 million data-points.
- As python 3 does not have backwards compatibility with python 2.7 no computers using older version of python can run the script.

CHAPTER 2

LITERATURE SURVEY

2.1 Abstract of the papers referred

2.1.1 A Multiple Resampling Method for Learning from Imbalanced Data Sets [1]

Re-Sampling methods are commonly used for dealing with the class-imbalance problem. Their advantage over other methods is that they are external and thus, easily transportable. Although such approaches can be very simple to implement, tuning them most effectively is not an easy task. In particular, it is unclear whether oversampling is more effective than undersampling and which oversampling or undersampling rate should be used. This paper presents an experimental study of these questions and concludes that combining different expressions of the re-sampling approach is an effective solution to the tuning problem. The proposed combination scheme is evaluated on imbalanced subsets of the Reuters-21578 text collection and is shown to be quite effective for these problems.

2.1.2 SMOTE: Synthetic Minority Over-sampling Technique [2]

An approach to the construction of classifiers from imbalanced datasets is described. A dataset is imbalanced if the classification categories are not approximately equally represented. Often real-world data sets are predominately composed of “normal” examples with only a small percentage of “abnormal” or “interesting” examples. It is also the case that the cost of misclassifying an abnormal (interesting) example as a normal example is often much higher than the cost of the reverse error. Under-sampling of the majority (normal) class has been proposed as a good means of increasing the sensitivity of a classifier to the minority class. This paper shows that a combination of our method of over-sampling the minority (abnormal) class and under-sampling the majority (normal) class can achieve better classifier performance (in ROC space) than only under-sampling the majority class. This paper also shows that a combination of our method of over-sampling the minority class and under-sampling the majority class can achieve better classifier performance (in ROC space) than varying the loss ratios in Ripper or class priors in Naive Bayes. Our method of over-sampling the minority class involves creating synthetic minority class examples. Experiments are performed using C4.5, Ripper and a

Naive Bayes classifier. The method is evaluated using the area under the Receiver Operating Characteristic curve (AUC) and the ROC convex hull strategy.

2.1.3 A comprehensive survey of data mining-based fraud detection research [3]

This survey paper categorises, compares, and summarises from almost all published technical and review articles in automated fraud detection within the last 10 years. It defines the professional fraudster, formalises the main types and subtypes of known fraud, and presents the nature of data evidence collected within affected industries. Within the business context of mining the data to achieve higher cost savings, this research presents methods and techniques together with their problems. Compared to all related reviews on fraud detection, this survey covers much more technical articles and is the only one, to the best of our knowledge, which proposes alternative data and solutions from related domains.

2.1.4 XGBoost: Reliable Large-scale Tree Boosting System [4]

Tree boosting is an important type of machine learning algorithms that is widely used in practice. In this paper, we describe XGBoost, a reliable, distributed machine learning system to scale up tree boosting algorithms. The system is optimized for fast parallel tree construction, and designed to be fault tolerant under the distributed setting. XGBoost can handle tens of millions of samples on a single node, and scales beyond billions of samples with distributed computing.

2.1.5 Fraud in Mobile Financial Services [5]

For many years, financial inclusion was a major challenge globally, due to the high costs involved. To offer financial services, premises had to be constructed, new employees recruited and significant capital investments made. Financial institutions concentrated on the high value consumers that yielded larger revenues and largely excluded the majority of the population. The introduction of mobile telecommunications and later, the adoption of mobile phones to provide financial services changed the dynamics of the industry, bringing financial services closer to the public through existing merchant infrastructure within local communities. The success of M-PESA since its launch in Kenya in 2007 has

increased the appetite for mobile financial service deployments especially in developing countries. Financial institutions such as banks and microfinance institutions are also investing in the provision of mobile financial services. The implementation of mobile financial services, like other financial services, faces risks and challenges. This paper addresses fraud as a challenge in the provision of mobile financial services.

2.1.6 Classification of Imbalanced Data by Using the SMOTE Algorithm [6]

The classification of imbalanced data is a common practice in the context of medical imaging intelligence. The synthetic minority oversampling technique (SMOTE) is a powerful approach to tackling the operational problem. This paper presents a novel approach to improving the conventional SMOTE algorithm by incorporating the locally linear embedding algorithm (LLE). The LLE algorithm is first applied to map the high-dimensional data into a low-dimensional space, where the input data is more separable, and thus can be oversampled by SMOTE. Then the synthetic data points generated by SMOTE are mapped back to the original input space as well through the LLE. Experimental results demonstrate that the underlying approach attains a performance superior to that of the traditional SMOTE.

2.2 Findings from the research papers

2.2.1 A Multiple Resampling Method for Learning from Imbalanced Data Sets [1]

From *"A Multiple Resampling Method for Learning from Imbalanced Data Sets"* by Andrew Estabrooks it is found out that in a concept- learning problem, the data set is said to present a class imbalance if it contains many more examples of one class than the other. Such a situation poses challenges for typical classifiers such as Decision Tree Induction Systems or Multi-Layer Perceptrons that are designed to optimize overall accuracy without taking into account the relative distribution of each class (Japkowicz & Stephen 2002; Estabrooks 2000). As a result, these classifiers tend to ignore small classes while concentrating on classifying the large ones accurately.

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

Figure 2.1: Accuracy of a model = (TP+TN) / (TP+FN+FP+TN)

Figure 2.1 shows that the evaluation of a classification algorithm performance is measured by the Confusion Matrix which contains information about the actual and the predicted.

Random Undersampling aims to balance class distribution by randomly eliminating majority class examples. This is done until the majority and minority class instances are balanced out.

Advantage is that It can help improve run time and storage problems by reducing the number of training data samples when the training data set is huge.

2.2.2 SMOTE: Synthetic Minority Over-sampling Technique [2] [6]

From Paper of "*SMOTE: Synthetic Minority Over-sampling Technique*" by Kevin Bower it is found that SMOTE technique is followed to avoid overfitting which occurs when exact replicas of minority instances are added to the main dataset. A subset of data is taken from the minority class as an example and then new synthetic similar instances are created. These synthetic instances are then added to the original dataset. The new dataset is used as a sample to train the classification models.

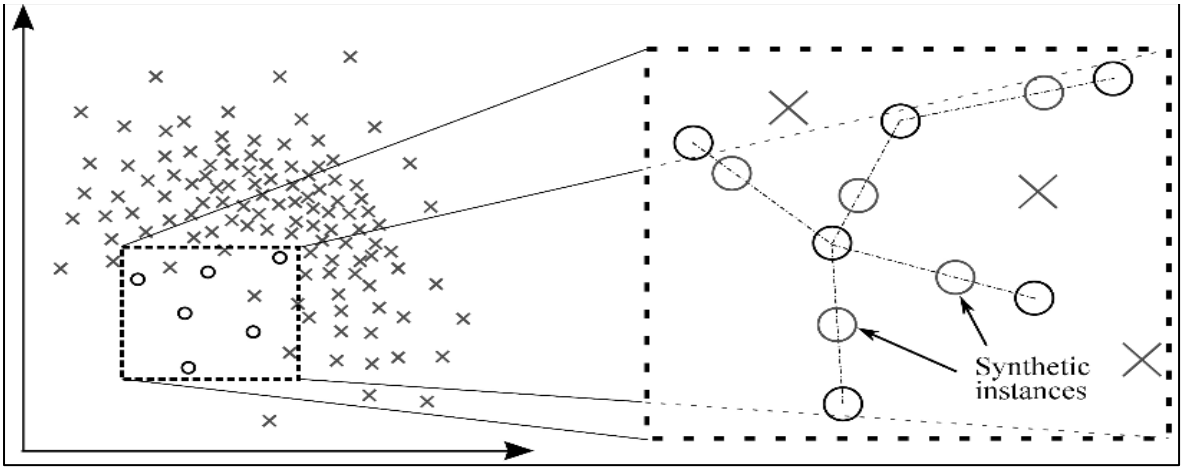

Figure 2.2: Creating synthetic data points in SMOTE

From Figure 2.2, we can see that instead of replicating and adding the observations from the minority class, it overcome imbalances by generates artificial data also called synthetic instances.

2.2.3 A comprehensive survey of data mining-based fraud detection research [3]

From *"A Comprehensive Survey of Data Mining-based Fraud Detection Research"* by Clifton Phua, the term fraud here refers to the abuse of a profit organisation's system without necessarily leading to direct legal consequences. In a competitive environment, fraud can become a business critical problem if it is very prevalent and if the prevention procedures are not fail-safe. Fraud detection, being part of the overall fraud control, automates and helps reduce the manual parts of a screening/checking process.

The fraudster can either commit fraud in the form of aprospective/existing customer (consumer) or a prospective/existing supplier (provider). The external fraudster

has three basic profiles: the average offender, criminal offender, and organised crime offender. Average offenders display random and/or occasional dishonest behaviour when there is opportunity, sudden temptation, or when suffering from financial hardship.In contrast, the more risky external fraudsters are individual criminal offenders and organised/group crime offenders (professional/career fraudsters) because they repeatedly disguise their true identities and/or evolve their modus operandi over time to approximate legal forms and to counter detection systems.

2.2.4 XGBoost: Reliable Large-scale Tree Boosting System [4]

"XGBoost: Reliable Large-scale Tree Boosting System" states that this algorithm uses multiple parameters. To improve the model, parameter tuning is must. It is very difficult to get answers to practical questions like – Which set of parameters you should tune ? What is the ideal value of these parameters to obtain optimal output ?

XGBoost (eXtreme Gradient Boosting) is an advanced implementation of gradient boosting algorithm.

XGBoost parameters:

The overall parameters have been divided into 3 categories by XGBoost authors:
1.General Parameters: Guide the overall functioning
2. Booster Parameters: Guide the individual booster (tree/regression) at each step
3. Learning Task Parameters: Guide the optimization performed.

Boosting :

Boosting is a sequential technique which works on the principle of an ensemble. It combines a set of weak learners and delivers improved prediction accuracy. At any instant t, the model outcomes are weighed based on the outcomes of previous instant t-1. The outcomes predicted correctly are given a lower weight and the ones miss-classified are weighted higher. Note that a weak learner is one which is slightly better than random guessing.

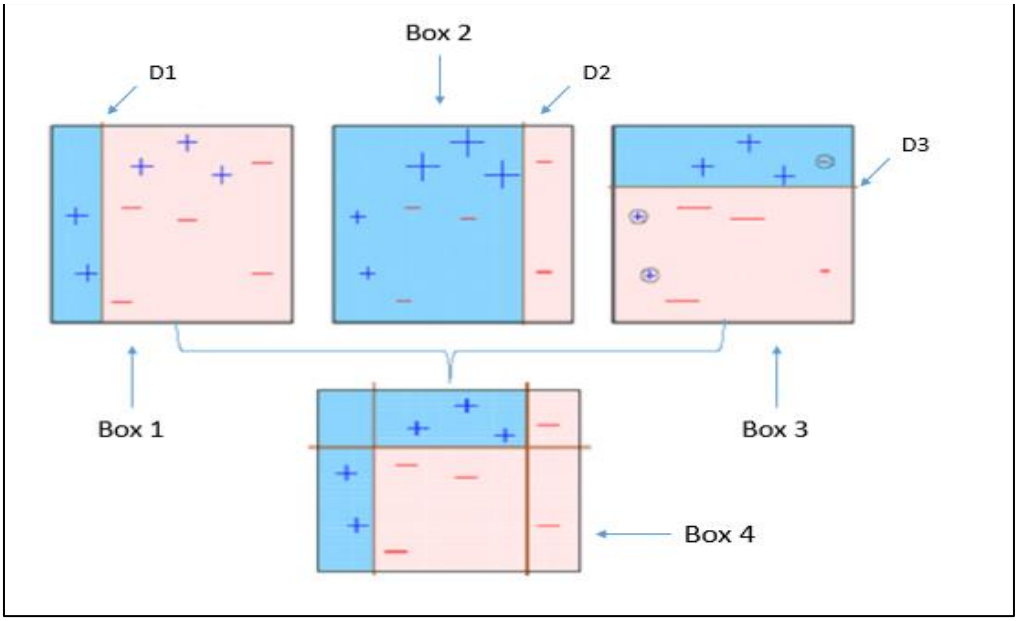

Figure 2.3: Representation of boosting technique

In Figure 2.3, four classifiers (in 4 boxes), are trying to classify + and - classes as homogeneously as possible.

1. Box 1: The first classifier (usually a decision stump) creates a vertical line (split) at D1. It says anything to the left of D1 is + and anything to the right of D1 is -. However, this classifier misclassified three + points.

Note a Decision Stump is a Decision Tree model that only splits off at one level, therefore the final prediction is based on only one feature.

2. Box 2: The second classifier gives more weight to the three + misclassified points (see the bigger size of +) and creates a vertical line at D2. Again it says, anything to the right of D2 is - and left is +. Still, it makes mistakes by incorrectly classifying three - points.

3. Box 3: Again, the third classifier gives more weight to the three - misclassified points and creates a horizontal line at D3. Still, this classifier fails to classify the points (in the circles) correctly.

4. Box 4: This is a weighted combination of the weak classifiers (Box 1,2 and 3). As you can see, it does a good job at classifying all the points correctly.

CHAPTER 3

TECHNOLOGIES IN DOMAIN

3.1 Machine Learning

3.1.1 Introduction

Machine learning (ML) is a field of artificial intelligence that uses statistical techniques to give computer systems the ability to "learn" (e.g., progressively improve performance on a specific task) from data, without being explicitly programmed.

The name machine learning was coined in 1959 by Arthur Samuel. Machine learning explores the study and construction of algorithms that can learn from and make predictions on data – such algorithms overcome following strictly static program instructions by making data-driven predictions or decisions, through building a model from sample inputs. Machine learning is employed in a range of computing tasks where designing and programming explicit algorithms with good performance is difficult or infeasible; example applications include email filtering, detection of network intruders, and computer vision.

Machine learning is closely related to (and often overlaps with) computational statistics, which also focuses on prediction-making through the use of computers. It has strong ties to mathematical optimization, which delivers methods, theory and application domains to the field. Machine learning is sometimes conflated with data mining, where the latter subfield focuses more on exploratory data analysis and is known as unsupervised learning.

Within the field of data analytics, machine learning is a method used to devise complex models and algorithms that lend themselves to prediction; in commercial use, this is known as predictive analytics. These analytical models allow researchers, data scientists, engineers, and analysts to "produce reliable, repeatable decisions and results" and uncover "hidden insights" through learning from historical relationships and trends in the data.

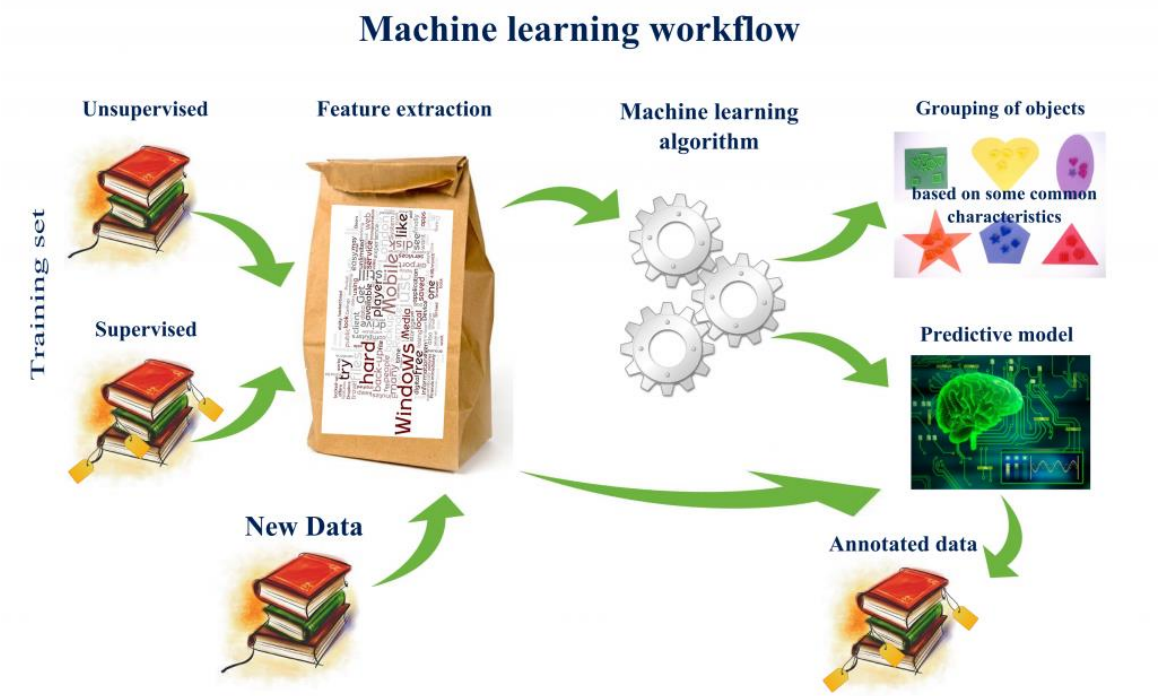

Figure 3.1: Machine Learning Workflow

From figure 3.1, we infer the steps that take place in building a machine learning model. Feature extraction is performed first, and only contributing features make it to the machine learning algorithm, where the model is formed which is used to perform a multitude of tasks, ranging from prediction, classification and clustering.

3.1.2 Need for Machine Learning

Machine Learning is a field which is raised out of Artificial Intelligence(AI). Applying AI, we wanted to build better and intelligent machines. But except for few mere tasks such as finding the shortest path between point A and B, we were unable to program more complex and constantly evolving challenges. There was a realisation that the only way to be able to achieve this task was to let machine learn from itself. This sounds similar to a child learning from its self. So machine learning was developed as a new capability for computers. And now machine learning is present in so many segments of technology, that we don't even realise it while using it.

Finding patterns in data on planet earth is possible only for human brains. The data being very massive, the time taken to compute is increased, and this is where Machine Learning comes into action, to help people with large data in minimum time.

If big data and cloud computing are gaining importance for their contributions, machine learning as technology helps analyse those big chunks of data, easing the task of data scientists in an automated process and gaining equal importance and recognition.

The techniques we use for data mining have been around for many years, but they were not effective as they did not have the competitive power to run the algorithms. If you run deep learning with access to better data, the output we get will lead to dramatic breakthroughs which is machine learning.

3.1.3 Supervised Learning

Supervised learning is a type of machine learning algorithm. A majority of practical machine learning uses supervised learning.

In supervised learning, the system tries to learn from the previous examples that are given. (On the other hand, in unsupervised learning, the system attempts to find the patterns directly from the example given.) Speaking mathematically, supervised learning is where you have both input variables (x) and output variables(Y) and can use an algorithm to derive the mapping function from the input to the output.
The mapping function is expressed as Y = f(X).

In order to solve a given problem of supervised learning, one has to perform the following steps:

1. Determine the type of training examples. Before doing anything else, the user should decide what kind of data is to be used as a training set. In case of handwriting analysis, for example, this might be a single handwritten character, an entire handwritten word, or an entire line of handwriting.
2. Gather a training set. The training set needs to be representative of the real-world use of the function. Thus, a set of input objects is gathered and corresponding outputs are also gathered, either from human experts or from measurements.

3. Determine the input feature representation of the learned function. The accuracy of the learned function depends strongly on how the input object is represented. Typically, the input object is transformed into a feature vector, which contains a number of features that are descriptive of the object. The number of features should not be too large, because of the curse of dimensionality; but should contain enough information to accurately predict the output.
4. Determine the structure of the learned function and corresponding learning algorithm. For example, the engineer may choose to use support vector machines or decision trees.
5. Complete the design. Run the learning algorithm on the gathered training set. Some supervised learning algorithms require the user to determine certain control parameters. These parameters may be adjusted by optimizing performance on a subset (called a validation set) of the training set, or via cross-validation.
6. Evaluate the accuracy of the learned function. After parameter adjustment and learning, the performance of the resulting function should be measured on a test set that is separate from the training set.

3.1.4 Classification

In machine learning and statistics, classification is the problem of identifying to which of a set of categories (sub-populations) a new observation belongs, on the basis of a training set of data containing observations (or instances) whose category membership is known. Examples are assigning a given email to the "spam" or "non-spam" class, and assigning a diagnosis to a given patient based on observed characteristics of the patient (sex, blood pressure, presence or absence of certain symptoms, etc.). Classification is an example of pattern recognition.

In the terminology of machine learning, classification is considered an instance of supervised learning, i.e., learning where a training set of correctly identified observations is available. The corresponding unsupervised procedure is known as clustering, and involves grouping data into categories based on some measure of inherent similarity or distance.

Often, the individual observations are analysed into a set of quantifiable properties, known variously as explanatory variables or features. These properties may variously be

categorical (e.g. "A", "B", "AB" or "O", for blood type), ordinal (e.g. "large", "medium" or "small"), integer-valued (e.g. the number of occurrences of a particular word in an email) or real-valued (e.g. a measurement of blood pressure). Other classifiers work by comparing observations to previous observations by means of a similarity or distance function.

An algorithm that implements classification, especially in a concrete implementation, is known as a classifier. The term "classifier" sometimes also refers to the mathematical function, implemented by a classification algorithm, that maps input data to a category.

Terminology across fields is quite varied. In statistics, where classification is often done with logistic regression or a similar procedure, the properties of observations are termed explanatory variables (or independent variables, regressors, etc.), and the categories to be predicted are known as outcomes, which are considered to be possible values of the dependent variable. In machine learning, the observations are often known as instances, the explanatory variables are termed features (grouped into a feature vector), and the possible categories to be predicted are classes. Other fields may use different terminology: e.g. in community ecology, the term "classification" normally refers to cluster analysis, i.e., a type of unsupervised learning, rather than the supervised learning described in this article.

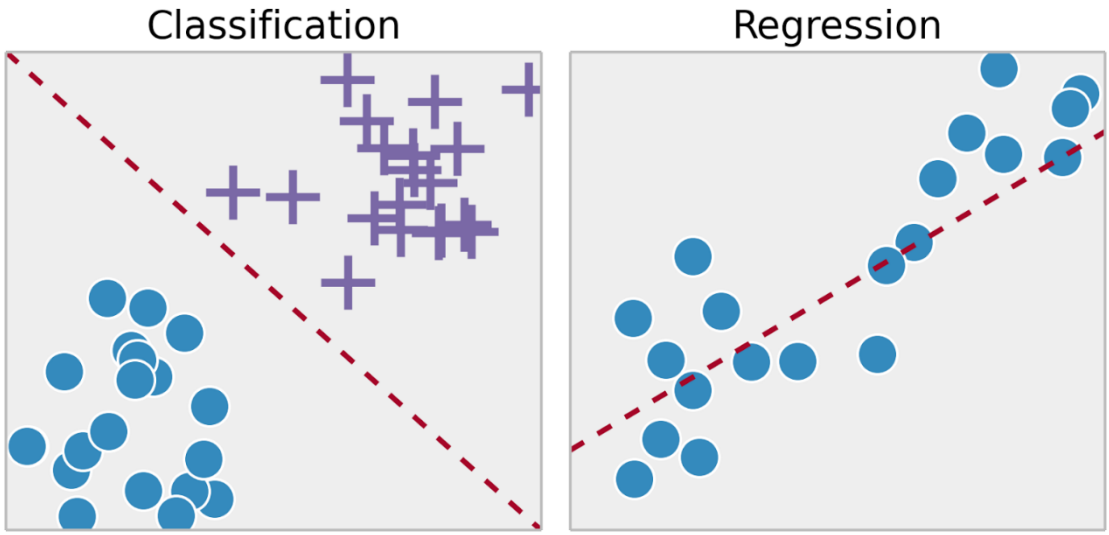

Figure 3.2: Classification vs Regression

From fig 3.2, we see the difference between classification and regression. Regression and classification are both related to prediction, where regression predicts a value from a continuous set, whereas classification predicts the 'belonging' to the class.

For example, the price of a house depending on the 'size' (in some unit) and say 'location' of the house, can be some 'numerical value' (which can be continuous): this relates to regression.

Similarly, the prediction of price can be in words, viz., 'very costly', 'costly', 'affordable', 'cheap', and 'very cheap': this relates to classification. Each class may correspond to some range of values.

CHAPTER 4

RESEARCH METHODOLOGY

4.1 Problem Statement

1. Imbalanced class distribution is a scenario where the number of observations belonging to one class is significantly lower than those belonging to the other classes. [7]
2. This problem is predominant in scenarios where anomaly detection is crucial like electricity pilferage, fraudulent transactions in banks etc. [5] [7]
3. This happens because Machine Learning Algorithms are usually designed to improve accuracy by reducing the error. Thus, they do not consider the class distribution / proportion or balance of classes. [3]
4. Standard classifier algorithms like Decision Tree have a bias towards classes which have number of instances. They tend to only predict the majority class data. The features of the minority class are treated as noise and are often ignored. Thus, there is a high probability of misclassification of the minority class as compared to the majority class. [7]
5. However, while working in an imbalanced domain accuracy is not an appropriate measure to evaluate model performance. For e.g.: A classifier which achieves an accuracy of 98 % with an event rate of 2 % is not accurate, if it classifies all instances as the majority class. And eliminates the 2 % minority class observations as noise.

Examples of imbalanced classes

Thus, to sum it up, while trying to resolve specific business challenges with imbalanced data sets, the classifiers produced by standard machine learning algorithms might not give accurate results. Apart from fraudulent transactions, other examples of a common business problem with imbalanced dataset are:

- Datasets to identify customer churn where a vast majority of customers will continue using the service. Specifically, Telecommunication companies where Churn Rate is lower than 2 %.
- Data sets to identify rare diseases in medical diagnostics etc.
- Natural Disaster like Earthquakes

4.2 Objectives

- Solving imbalanced dataset classification problem in detecting fraudulent transactions by balancing the majority and minority classes through the use of resampling techniques [1]
- Cleaning PaySim generated synthetic dataset for money transactions. [8]
- Using resampling methods (undersampling, oversampling using SMOTE) and ensemble learning (XG-boosting). [1] [2]

4.3 Solution proposed

1. Exploratory Data Analysis

To judge which type of transactions are fraudulent, and to check how the value of attribute i*sFlaggedFraud* is set. Queries are made on the dataset and relations are found. As it turns out, a few of the features are unclear on how the value is set to determine the corresponding *isFraud* value, and hence need to be dropped.

2. Data Cleaning and Feature Engineering

After this, data cleaning is done, which includes imputation of null values. In certain features, imputation lead to loss in original information provided by the dataset and hence the values were replaced with -1 and nan. To get a clear understanding of how *isFraud* is set, two new features are extracted, which determine the error in the balance amounts in sender and receiver accounts.

3. Undersampling

Once data is ready, resampling techniques are performed. Undersampling is done on the dataset to create a DataFrame containing all fraud instances and equal number of random non-fraud instances, which are repeated for various c-values to determine a better recall. Various classifiers like Logistic Regression and SVM are run on this data.

4. Oversampling using SMOTE

SMOTE is an over-sampling approach in which the minority class is oversampled by creating “synthetic” examples rather than by oversampling with replacement. Since undersampling leads to loss of information and each datapoint is crucial, another technique was implemented. This time, oversampling with SMOTE was done on the dataset which increases the number of minority data-points by creating synthetic similar data-points. Classifiers are run on this data again to get a better recall value.

5. Machine Learning to Detect Fraud in Skewed Data.

Finally, ensemble learning in XGBoost is performed which allows for weighting the positive class more compared to the negative class, a setting that also allows to account for the skew in the data.

4.4 Methodology and technologies used

4.4.1 Ensemble learning

XGBoost [4] is an ensemble learning method. Sometimes, it may not be sufficient to rely upon the results of just one machine learning model. Ensemble learning offers a systematic solution to combine the predictive power of multiple learners. The resultant is a single model which gives the aggregated output from several models.

The models that form the ensemble, also known as base learners, could be either from the same learning algorithm or different learning algorithms. Bagging and boosting are two widely used ensemble learners. Though these two techniques can be used with several statistical models, the most predominant usage has been with decision trees.

While decision trees are one of the most easily interpretable models, they exhibit highly variable behavior. Consider a single training dataset that we randomly split into two parts. Now, let’s use each part to train a decision tree in order to obtain two models.

When we fit both these models, they would yield different results. Decision trees are said to be associated with high variance due to this behavior. Bagging or boosting aggregation helps to reduce the variance in any learner. Several decision trees which are generated in parallel, form the base learners of bagging technique. Data sampled with replacement is

fed to these learners for training. The final prediction is the averaged output from all the learners.

4.4.2 Boosting

In boosting, the trees [4] are built sequentially such that each subsequent tree aims to reduce the errors of the previous tree. Each tree learns from its predecessors and updates the residual errors. Hence, the tree that grows next in the sequence will learn from an updated version of the residuals.
The base learners in boosting are weak learners in which the bias is high, and the predictive power is just a tad better than random guessing. Each of these weak learners contributes some vital information for prediction, enabling the boosting technique to produce a strong learner by effectively combining these weak learners. The final strong learner brings down both the bias and the variance.

In contrast to bagging techniques like Random Forest, in which trees are grown to their maximum extent, boosting makes use of trees with fewer splits. Such small trees, which are not very deep, are highly interpretable. Parameters like the number of trees or iterations, the rate at which the gradient boosting learns, and the depth of the tree, could be optimally selected through validation techniques like k-fold cross validation. Having a large number of trees might lead to overfitting. So, it is necessary to carefully choose the stopping criteria for boosting.

The math behind XGBoost: [4]

Boosting consists of three simple steps:

- An initial model F0 is defined to predict the target variable y. This model will be associated with a residual (y – F0)
- A new model h1 is fit to the residuals from the previous step
- Now, F0 and h1 are combined to give F1, the boosted version of F0. The mean squared error from F1 will be lower than that from F0:

$$F_1(x) \leftarrow F_0(x) + h_1(x)$$

To improve the performance of F1, we could model after the residuals of F1 and create a new model F2:

$$F_2(x) \leftarrow F_1(x) + h_2(x)$$

This can be done for *'m'* iterations, until residuals have been minimized as much as possible:

$$F_m(x) \leftarrow F_{m-1}(x) + h_m(x)$$

Here, the additive learners do not disturb the functions created in the previous steps. Instead, they impart information of their own to bring down the errors.

CHAPTER – 5

IMPLEMENTATION AND RESULT

5.1 Architecture

5.1.1 Undersampling

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. [1]

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

5.1.2 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 undersampling, this method also can be divided into two types: Random Oversampling and Informative Oversampling. [1]

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.

5.1.3 SMOTE

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. [7]

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 neighbour.
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.

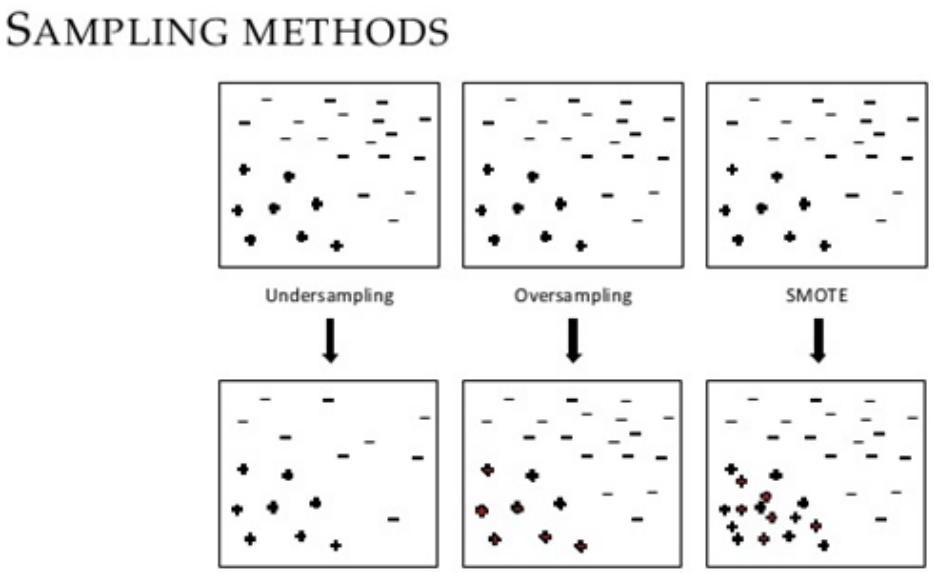

Figure 5.1: Sampling methods

Figure 5.1 shows the comparison between different sampling techniques, where '-' represent the majority class and '.' represent the minority class.

5.1.4 XGBoost

XGBoost (Extreme Gradient Boosting) is an advanced and more efficient implementation of Gradient Boosting Algorithm discussed in the previous section.
Advantages over Other Boosting Techniques

- It is 10 times faster than the normal Gradient Boosting as it implements parallel processing. It is highly flexible as users can define custom optimization objectives and evaluation criteria, has an inbuilt mechanism to handle missing values.
- Unlike gradient boosting which stops splitting a node as soon as it encounters a negative loss, XG Boost splits up to the maximum depth specified and prunes the tree backward and removes splits beyond which there is an only negative loss.

5.2 Implementation of the methodologies

5.2.1 Synthetic dataset generated by the PaySim money transaction simulator

PaySim [8] simulates money transactions based on a sample of real transactions extracted from one month of financial logs from a money service implemented in an African country. The original logs were provided by a multinational company, who is the provider of the financial service which is currently running in more than 14 countries all around the world.

This is a sample of 1 row with headers explanation:

1, PAYMENT ,1060.31, C429214117, 1089.0, 28.69, M1591654462, 0.0, 0.0, 0, 0

step - maps a unit of time in the real world. In this case 1 step is 1 hour of time. Total steps 744 (31 days simulation).

type - CASH-IN, CASH-OUT, DEBIT, PAYMENT and TRANSFER.

amount - amount of the transaction in local currency.

nameOrig - customer who started the transaction

oldbalanceOrg - initial balance before the transaction

newbalanceOrig - new balance after the transaction

nameDest - customer who is the recipient of the transaction

oldbalanceDest - initial balance of recipient before the transaction.

newbalanceDest - new balance of recipient after the transaction.

isFraud - This is the transactions made by the fraudulent agents. In this specific dataset the fraudulent behavior of the agents aims to profit by taking control of customers accounts and try to empty the funds by transferring to another account and then cashing out of the system.

isFlaggedFraud - The model aims to control massive transfers from one account to another and flags illegal attempts. An illegal attempt in this dataset is an attempt to transfer more than 200,000 in a single transaction.

5.2.2 Exploratory Data Analysis

Exploratory data analysis is an approach to analyzing data sets to summarize their main characteristics, often with visual methods. A statistical model can be used or not, but primarily EDA is for seeing what the data can tell us beyond the formal modeling or hypothesis testing task.

```
In [28]: df = pd.read_csv('PS_20174392719_1491204439457_log.csv')
         df = df.rename(columns={'oldbalanceOrg':'oldBalanceOrig', 'newbalanceOrig':'newBalanceOrig', \
                                 'oldbalanceDest':'oldBalanceDest', 'newbalanceDest':'newBalanceDest'})
         print(df.head())
```

```
   step      type    amount     nameOrig  oldBalanceOrig  newBalanceOrig  \
0     1   PAYMENT   9839.64  C1231006815        170136.0       160296.36
1     1   PAYMENT   1864.28  C1666544295         21249.0        19384.72
2     1  TRANSFER    181.00  C1305486145           181.0            0.00
3     1  CASH_OUT    181.00   C840083671           181.0            0.00
4     1   PAYMENT  11668.14  C2048537720         41554.0        29885.86

      nameDest  oldBalanceDest  newBalanceDest  isFraud  isFlaggedFraud
0  M1979787155             0.0             0.0        0               0
1  M2044282225             0.0             0.0        0               0
2   C553264065             0.0             0.0        1               0
3    C38997010         21182.0             0.0        1               0
4  M1230701703             0.0             0.0        0               0
```

a. Types of fraudulent transactions

We find that fraud occurs in only two types of transactions: 'TRANSFER' where money is sent to a customer and 'CASH_OUT' where money is sent to a merchant who pays the customer in cash. Remarkably, the number of fraudulent TRANSFERs almost equals the number of fraudulent CASH_OUTs.

```
In [38]: df.groupby(['type', 'isFraud']).count()['step']
```

```
Out[38]: type      isFraud
         CASH_IN   0          1399284
         CASH_OUT  0          2233384
                   1             4116
         DEBIT     0            41432
         PAYMENT   0          2151495
         TRANSFER  0           528812
                   1             4097
         Name: step, dtype: int64
```

```
In [39]: dfFraudTransfer = df.loc[(df.isFraud == 1) & (df.type == 'TRANSFER')]
         dfFraudCashout = df.loc[(df.isFraud == 1) & (df.type == 'CASH_OUT')]

         print ('\n The number of fraudulent TRANSFERs = {}'.\
                format(len(dfFraudTransfer)))

         print ('\n The number of fraudulent CASH_OUTs = {}'.\
                format(len(dfFraudCashout)))
```

```
 The number of fraudulent TRANSFERs = 4097

 The number of fraudulent CASH_OUTs = 4116
```

b. What determines whether the feature isFlaggedFraud gets set or not?

It turns out that the origin of *isFlaggedFraud* is unclear, contrasting with the description provided. The 16 entries (out of 6 million) where the *isFlaggedFraud* feature is set do not seem to correlate with any explanatory variable. The data is described as *isFlaggedFraud* being set when an attempt is made to 'TRANSFER' an 'amount' greater than 200,000. In fact, as shown below, *isFlaggedFraud* can remain not set despite this condition being met.

```
In [45]: dfFlagged = df.loc[df.isFlaggedFraud == 1]
         dfNotFlagged = df.loc[df.isFlaggedFraud == 0]

         print('\nThe type of transactions where isFlaggedFraud is set: \
         {}'.format(list(dfFlagged.type.drop_duplicates())))

         dfTransfer = df.loc[df.type == 'TRANSFER']

         print('\nMax amount transacted in a TRANSFER where isFlaggedFraud is not set=\
          {}'.format(dfNotFlagged.amount.max()))

         The type of transactions where isFlaggedFraud is set: ['TRANSFER']

         Max amount transacted in a TRANSFER where isFlaggedFraud is not set= 92445516.64
```

Can *oldBalanceDest* and *newBalanceDest* determine *isFlaggedFraud* being set? The old is identical to the new balance in the origin and destination accounts, for every TRANSFER where *isFlaggedFraud* is set. This is presumably because the transaction is halted. Interestingly, *oldBalanceDest* = 0 in every such transaction. However, as shown below, since *isFlaggedFraud* can remain not set in TRANSFERS where *oldBalanceDest* and *newBalanceDest* can both be 0, these conditions do not determine the state of *isFlaggedFraud*.

```
In [50]: print('\nThe number of TRANSFERs where isFlaggedFraud = 0, yet oldBalanceDest = 0 and\
          newBalanceDest = 0 : {}'.\
         format(len(dfTransfer.loc[(dfTransfer.isFlaggedFraud == 0) & \
         (dfTransfer.oldBalanceDest == 0) & (dfTransfer.newBalanceDest == 0)])))

         The number of TRANSFERs where isFlaggedFraud = 0, yet oldBalanceDest = 0 and newBalanceDest = 0 : 4158
```

isFlaggedFraud being set cannot be thresholded on *oldBalanceOrig* since the corresponding range of values overlaps with that for TRANSFERs where *isFlaggedFraud* is not set. We do not need to consider *newBalanceOrig* since it is updated only after the transaction, whereas *isFlaggedFraud* would be set before the transaction takes place.

```
In [51]: print('\nMin, Max of oldBalanceOrig for isFlaggedFraud = 1 TRANSFERs: {}'.\
         format([round(dfFlagged.oldBalanceOrig.min()), round(dfFlagged.oldBalanceOrig.max())]))

         print('\nMin, Max of oldBalanceOrig for isFlaggedFraud = 0 TRANSFERs where \
         oldBalanceOrig = \
         newBalanceOrig: {}'.format(\
         [dfTransfer.loc[(dfTransfer.isFlaggedFraud == 0) & (dfTransfer.oldBalanceOrig \
         == dfTransfer.newBalanceOrig)].oldBalanceOrig.min(), \
         round(dfTransfer.loc[(dfTransfer.isFlaggedFraud == 0) & (dfTransfer.oldBalanceOrig \
                       == dfTransfer.newBalanceOrig)].oldBalanceOrig.max())]))

         Min, Max of oldBalanceOrig for isFlaggedFraud = 1 TRANSFERs: [353874.0, 19585040.0]

         Min, Max of oldBalanceOrig for isFlaggedFraud = 0 TRANSFERs where oldBalanceOrig = newBalanceOrig: [0.0, 575668.0]
```

Can *isFlaggedFraud* be set based on seeing a customer transacting more than once? Duplicate customer names don't exist within transactions where *isFlaggedFraud* is set, but duplicate customer names exist within transactions where *isFlaggedFraud* is not set. It turns out that originators of transactions that have *isFlaggedFraud* set have transacted only once. Very few destination accounts of transactions that have *isFlaggedFraud* set have transacted more than once.

```
In [54]: dfFlagged.nameDest.isin(dfNotFlagged.nameDest)

Out[54]: 2736446     True
         3247297     True
         3760288    False
         5563713    False
         5996407    False
         5996409    False
         6168499    False
         6205439    False
         6266413    False
         6281482    False
         6281484    False
         6296014    False
         6351225    False
         6362460    False
         6362462    False
         6362584    False
         Name: nameDest, dtype: bool
```

It can be easily seen that transactions with *isFlaggedFraud* set occur at all values of *step*, similar to the complementary set of transactions. Thus *isFlaggedFraud* does not correlate with *step* either and is therefore seemingly unrelated to any explanatory variable or feature in the data

Conclusion: Although *isFraud* is always set when *isFlaggedFraud* is set, since *isFlaggedFraud* is set just 16 times in a seemingly meaningless way, we can treat this feature as insignificant and discard it in the dataset without losing information.

c. Are expected merchant accounts accordingly labelled?

It was stated that CASH_IN involves being paid by a merchant (whose name is prefixed by 'M'). However, the present data does not have merchants making CASH_IN transactions to customers.

```
In [55]: print('\nAre there any merchants among originator accounts for CASH_IN \
         transactions? {}'.format(\
         (df.loc[df.type == 'CASH_IN'].nameOrig.str.contains('M')).any()))

         Are there any merchants among originator accounts for CASH_IN transactions? False
```

Similarly, it was stated that CASH_OUT involves paying a merchant. However, for CASH_OUT transactions there are no merchants among the destination accounts.

```
In [56]: print('\nAre there any merchants among destination accounts for CASH_OUT \
         transactions? {}'.format(\
         (df.loc[df.type == 'CASH_OUT'].nameDest.str.contains('M')).any()))

         Are there any merchants among destination accounts for CASH_OUT transactions? False
```

In fact, there are no merchants among any originator accounts. Merchants are only present in destination accounts for all PAYMENTS.

```
In [57]: print('\nAre there merchants among any originator accounts? {}'.format(\
               df.nameOrig.str.contains('M').any()))

         print('\nAre there any transactions having merchants among destination accounts\
          other than the PAYMENT type? {}'.format(\
         (df.loc[df.nameDest.str.contains('M')].type != 'PAYMENT').any()))

         Are there merchants among any originator accounts? False

         Are there any transactions having merchants among destination accounts other than the PAYMENT type? False
```

Conclusion: Among the account labels *nameOrig* and *nameDest*, for all transactions, the merchant prefix of 'M' occurs in an unexpected way.

d. Are there account labels common to fraudulent TRANSFERs and CASH_OUTs?

From the data description, the modus operandi for committing fraud involves first making a TRANSFER to a (fraudulent) account which in turn conducts a CASH_OUT. CASH_OUT involves transacting with a merchant who pays out cash. Thus, within this two-step process, the fraudulent account would be both, the destination in a TRANSFER and the originator in a CASH_OUT. However, the data shows below that there are no such common accounts among fraudulent transactions. Thus, the data is not imprinted with the expected modus-operandi.

```
In [58]: print('\nWithin fraudulent transactions, are there destinations for TRANSFERS \
         that are also originators for CASH_OUTs? {}'.format(\
         (dfFraudTransfer.nameDest.isin(dfFraudCashout.nameOrig)).any()))
         dfNotFraud = df.loc[df.isFraud == 0]

         Within fraudulent transactions, are there destinations for TRANSFERS that are also originators for CASH_OUTs? False
```

Could destination accounts for fraudulent TRANSFERs originate CASHOUTs that are not detected and are labeled as genuine? It turns out there are 3 such accounts. However, 2 out of 3 of these accounts first make a genuine CASH_OUT and only later (as

evidenced by the time step) receive a fraudulent TRANSFER. Thus, fraudulent transactions are not indicated by the *nameOrig* and *nameDest* features.

Fraudulent TRANSFERs whose destination accounts are originators of genuine CASH_OUTs:

	step	type	amount	nameOrig	oldBalanceOrig	\
1030443	65	TRANSFER	1282971.57	C1175896731	1282971.57	
6039814	486	TRANSFER	214793.32	C2140495649	214793.32	
6362556	738	TRANSFER	814689.88	C2029041842	814689.88	

	newBalanceOrig	nameDest	oldBalanceDest	newBalanceDest	isFraud	\
1030443	0.0	C1714931087	0.0	0.0	1	
6039814	0.0	C423543548	0.0	0.0	1	
6362556	0.0	C1023330867	0.0	0.0	1	

	isFlaggedFraud
1030443	0
6039814	0
6362556	0

Conclusion: The *nameOrig* and *nameDest* features neither encode merchant accounts in the expected way, below, we drop these features from the data since they are meaningless.

5.2.3 Data Cleaning

a. Dropping futile features

From the exploratory data analysis (EDA), we know that fraud only occurs in 'TRANSFER's and 'CASH_OUT's. So we assemble only the corresponding data in X for analysis.

```
X = df.loc[(df.type == 'TRANSFER') | (df.type == 'CASH_OUT')]
Y = X['isFraud']
del X['isFraud']

X = X.drop(['nameOrig', 'nameDest', 'isFlaggedFraud'], axis=1)

X.loc[X.type == 'TRANSFER', 'type'] = 0
X.loc[X.type == 'CASH_OUT', 'type'] = 1
X.type = X.type.astype(int)

Xfraud = X.loc[Y == 1]
XnonFraud = X.loc[Y == 0]

X.loc[(X.oldBalanceDest == 0) & (X.newBalanceDest == 0) & (X.amount != 0),
      ['oldBalanceDest', 'newBalanceDest']] = - 1

X.loc[(X.oldBalanceOrig == 0) & (X.newBalanceOrig == 0) & (X.amount != 0),
      ['oldBalanceOrig', 'newBalanceOrig']] = -2
```

Since the destination account balances being zero is a strong indicator of fraud, we do not impute the account balance (before the transaction is made) with a statistic or from a distribution with a subsequent adjustment for the amount transacted. Doing so would mask this indicator of fraud and make fraudulent transactions appear genuine. Instead, below we replace the value of 0 with -1 which will be more useful to a suitable machine-learning (ML) algorithm detecting fraud.

The features *nameOrig*, *nameDest*, and *isFlaggedFraud* were proven futile in our evaluation during Exploratory Data Analysis, hence they have been dropped in this model.

b. Feature Extraction

Motivated by the possibility of zero-balances serving to differentiate between fraudulent and genuine transactions, we take the data-imputation a step further and create 2 new features (columns) recording errors in the originating and destination accounts for each transaction. These new features turn out to be important in obtaining the best performance from the ML algorithm that we will finally use.

```
X['errorBalanceOrig'] = X.newBalanceOrig + X.amount - X.oldBalanceOrig
X['errorBalanceDest'] = X.oldBalanceDest + X.amount - X.newBalanceDest

print('Before undersampling skew = {}\n\n'.format(len(Xfraud) / float(len(X))))
```

```
Before undersampling skew = 0.002964544224336551
```

5.2.4 Machine Learning to detect fraud in Skewed Data

a. Random undersampling

Undersampling by taking random indices of non-fraudulent transactions in the numerical equivalent of the number of fraudulent transactions, then concatenate them as this would make the algorithm sensitive towards detecting the minority class while training. Skew is checked after performing undersampling, which comes out to be 0.5, much better than the original skew.

```
no_frauds = len(Y[Y == 1])
non_fraud_indices = Y[Y == 0].index

list_of_accuracies = []

random_indices = np.random.choice(non_fraud_indices, no_frauds, replace=False)
fraud_indices = Y[Y == 1].index
under_sample_indices = np.concatenate([fraud_indices, random_indices])
under_sample_X = X.loc[under_sample_indices]
under_sample_Y = Y.loc[under_sample_indices]
X_under_train, X_under_test, y_under_train, y_under_test = train_test_split(under_sample_X, under_sample_Y,
                                                                             test_size=0.3, random_state=0)
print('Skew after under-sampling = {}'.format(len(under_sample_Y[under_sample_Y == 1]) / float(len(under_sample_Y))))
```

```
Skew after under-sampling = 0.5
```

b. Plotting bars for 'isFraud' = 0 and 'isFraud' = 1

Bars are plotted for isFraud = 0 and isFraud = 1, to check how many of each instances are now present in the DataFrame.

```
observations = [len(under_sample_Y[under_sample_Y == 1]), len(under_sample_Y[under_sample_Y == 0])]
plt.bar(x=[0, 1], height=observations, tick_label=['isFraud = 1', 'isFraud = 0'])
plt.title("After under-sampling")
plt.show()
```

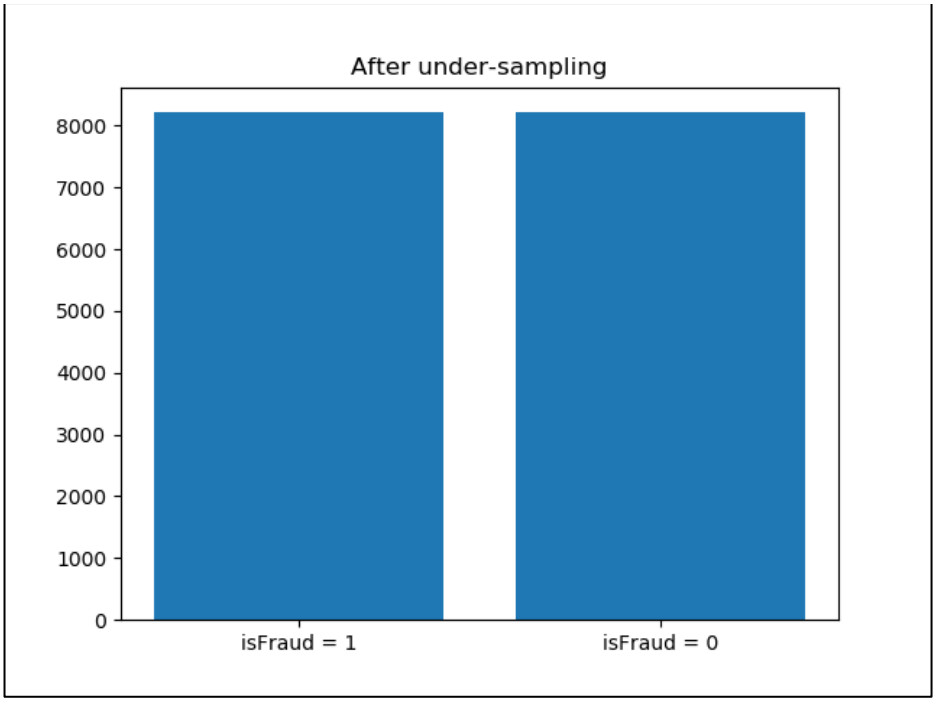

Figure 5.2: Bar graph after undersampling

As seen from Figure 5.2, instances for both *isFraud = 1* and *isFraud = 0* are almost equal.

c. Calculating mean precision and recall

Mean precision and recall are calculated using KFold cross-validation. In *k*-fold cross-validation, the original sample is randomly partitioned into *k* equal sized subsamples. Of the *k* subsamples, a single subsample is retained as the validation data for testing the model, and the remaining $k - 1$ subsamples are used as training data. The cross-validation process is then repeated *k* times, with each of the *k* subsamples used exactly once as the validation data. The *k* results can then be averaged to produce a single estimation. The advantage of this method over repeated random sub-sampling (see below) is that all observations are used for both training and validation, and each observation is used for validation exactly once.

```
62  c_param_range = [0.001, 0.01, 0.1, 1, 10, 100]
63  k_fold = 5
64
65  fold = KFold(n_splits=k_fold, shuffle=False)
66
67  results_table = pd.DataFrame(index=range(len(c_param_range), 3),
68                               columns=['C_parameter', 'Mean recall score', 'Mean precision score'])
69  results_table['C_parameter'] = c_param_range
```

To solve this, as well as minimizing the error as already discussed, you minimize a function that penalizes large values of the parameters. Most often the function is some constant λ. The larger λ is, the less likely it is that the parameters will be increased in magnitude simply to adjust for small perturbations in the data. In our case however, rather than specifying λ, we specify C=1/λ.

```
71      j = 0
72      for c_param in c_param_range:
73          print('-------------------------------------------')
74          print('C parameter: ', c_param)
75          print('-------------------------------------------')
76          print('')
77
78          recall_accs = []
79          precision_accs = []
```

C parameter is inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization. Regularization refers to the method of preventing overfitting, by explicitly controlling the model complexity. It leads to smoothening of the regression line and thus prevents overfitting.

```
80      i = 1
81      for train_index, test_index in fold.split(y_under_train):
82          lr = LogisticRegression(C=c_param, penalty='l1')
83
84          lr.fit(X_under_train.values[train_index], y_under_train.values[train_index])
85
86          y_pred_undersample = lr.predict(X_under_train.values[test_index])
87
88          recall_acc = recall_score(y_under_train.values[test_index], y_pred_undersample)
89          recall_accs.append(recall_acc)
90
91          precision_acc = precision_score(y_under_train.values[test_index], y_pred_undersample)
92          precision_accs.append(precision_acc)
93          print("Iteration {}: recall score = {:.4f}, precision score = {:.4f}".format(i, recall_acc,
94                                                                                        precision_acc))
95          i = i + 1
96
```

For each c value, K-Fold validation is done, then the mean of recall and precision scores of each K-Fold is taken and added to a table. This results_table stores mean recall and precision values for each c parameter.

```
97          results_table.ix[j, 'Mean recall score'] = np.mean(recall_accs)
98          results_table.ix[j, 'Mean precision score'] = np.mean(precision_accs)
99          j += 1
100         print('')
101         print('Mean recall score {:.4f}'.format(np.mean(recall_accs)))
102         print('Mean precision score {:.4f}'.format(np.mean(precision_accs)))
103         print('')
```

Iteration for C parameter = 0.001 and 0.01

```
-------------------------------------------
C parameter:  0.001
-------------------------------------------

Iteration 1: recall score = 0.9474, precision score = 0.9031
Iteration 2: recall score = 0.9401, precision score = 0.8950
Iteration 3: recall score = 0.9501, precision score = 0.8832
Iteration 4: recall score = 0.9394, precision score = 0.8986
Iteration 5: recall score = 0.9761, precision score = 0.9159

Mean recall score 0.9506
Mean precision score 0.8992

-------------------------------------------
C parameter:  0.01
-------------------------------------------

Iteration 1: recall score = 0.9089, precision score = 0.9342
Iteration 2: recall score = 0.9097, precision score = 0.9307
Iteration 3: recall score = 0.9180, precision score = 0.9147
Iteration 4: recall score = 0.8973, precision score = 0.9190
Iteration 5: recall score = 0.9232, precision score = 0.9450

Mean recall score 0.9114
Mean precision score 0.9287
```

Iteration for C parameter = 0.1 and 1

```
-----------------------------------------
C parameter:  0.1
-----------------------------------------

Iteration 1: recall score = 0.8966, precision score = 0.9385
Iteration 2: recall score = 0.9062, precision score = 0.9388
Iteration 3: recall score = 0.8993, precision score = 0.9215
Iteration 4: recall score = 0.8897, precision score = 0.9305
Iteration 5: recall score = 0.9070, precision score = 0.9474

Mean recall score 0.8998
Mean precision score 0.9353

-----------------------------------------
C parameter:  1
-----------------------------------------

Iteration 1: recall score = 0.8948, precision score = 0.9384
Iteration 2: recall score = 0.9071, precision score = 0.9397
Iteration 3: recall score = 0.8984, precision score = 0.9222
Iteration 4: recall score = 0.8897, precision score = 0.9313
Iteration 5: recall score = 0.9070, precision score = 0.9466

Mean recall score 0.8994
Mean precision score 0.9356
```

Iteration for C parameter = 10 and 100

```
-----------------------------------------
C parameter:  10
-----------------------------------------

Iteration 1: recall score = 0.8948, precision score = 0.9384
Iteration 2: recall score = 0.9062, precision score = 0.9397
Iteration 3: recall score = 0.8984, precision score = 0.9222
Iteration 4: recall score = 0.8906, precision score = 0.9313
Iteration 5: recall score = 0.9044, precision score = 0.9464

Mean recall score 0.8989
Mean precision score 0.9356

-----------------------------------------
C parameter:  100
-----------------------------------------

Iteration 1: recall score = 0.8948, precision score = 0.9384
Iteration 2: recall score = 0.9062, precision score = 0.9397
Iteration 3: recall score = 0.8984, precision score = 0.9222
Iteration 4: recall score = 0.8906, precision score = 0.9313
Iteration 5: recall score = 0.9070, precision score = 0.9466

Mean recall score 0.8994
Mean precision score 0.9357
```

e. Selection on the basis of best mean recall score

Best mean recall score is obtained by c parameter = 0.001. The recall score is 0.9506.

```
best_c = results_table
best_c.dtypes.eq(object)
new = best_c.columns[best_c.dtypes.eq(object)]
best_c[new] = best_c[new].apply(pd.to_numeric, errors='coerce', axis=0)
best_c = results_table.loc[results_table['Mean recall score'].idxmax()]['C_parameter']

print('********************************************************************************')
print('Best model to choose from cross validation is with C parameter = ', best_c)
print('********************************************************************************')
```

```
********************************************************************************
Best model to choose from cross validation is with C parameter =  0.001
********************************************************************************

Process finished with exit code 0
```

5.2.5 SMOTE (Synthetic Minority Over-sampling Technique) [2][6]

Rather than replicating the minority observations (e.g., defaulters, fraudsters, churners), Synthetic Minority Oversampling (SMOTE) works by creating synthetic observations based upon the existing minority observations .For each minority class observation, SMOTE calculates the k nearest neighbors.

a. Scaling the dataset, splitting the dataset into train, test and then applying smote on train dataset

As we saw earlier that fraudulent transactions are present in TRANSFER and CASHOUT types only. Hence we make a dataframe containing only these two types and new features only.

First scaling of the features is done for better analysis. Then the dataset is splitted into train and test parts .

SMOTE is then applied on the splitted training data . Splitting is done before SMOTE because if it had been done reversely then the synthetic data information would have been leaked to the test data and thus reducing the efficiency of the model .

SMOTE is applied and thus after SMOTE the labels 0's and 1's become same in number (ie 1104923).

```
types = ["TRANSFER","CASH_OUT"]
df = X[X.type.isin(types)]
```

```
df.type.value_counts()
```

```
CASH_OUT    2237500
TRANSFER     532909
Name: type, dtype: int64
```

```
X_TYPES = np.array(pd.DataFrame(df, columns=['errorBalOrig','errorBalDest']))
Y_TYPES = df["isFraud"]
y_TYPES = np.array(Y_TYPES.reshape(len(Y_TYPES), ))

from sklearn.preprocessing import StandardScaler
sc_TYPES = StandardScaler()
sc_TYPES.fit(X_TYPES)
X_TYPES_sc = sc_TYPES.transform(X_TYPES)
```

```
from collections import Counter
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(X_TYPES_sc, Y_TYPES, test_size=0.6, random_state=0)
```

```
Y_train.value_counts()
```

```
0    1104923
1       3240
Name: isFraud, dtype: int64
```

```
print('Original dataset shape of training data {}'.format(Counter(Y_train)))
sm = SMOTE(random_state=42)
X_TYPES_sm, Y_TYPES_sm = sm.fit_sample(X_train, Y_train)
print('Resampled dataset shape of training data {}'.format(Counter(Y_TYPES_sm)))
```

```
Original dataset shape of training data Counter({0: 1104923, 1: 3240})
Resampled dataset shape of training data Counter({0: 1104923, 1: 1104923})
```

b. Applying Logistic Regression and train it with modified dataset.

After the oversampling using SMOTE , the dataset is ready to be used by model. Logistic regression [9] is then done with the modified dataset with different C parameter values. The recall remains same throughout .

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import KFold
from sklearn.metrics import confusion_matrix, precision_score, precision_recall_curve,auc,roc_auc_score,

c_param_range = [0.001,0.01,0.1,1,10,100]
k_fold =5
fold = KFold(k_fold,shuffle = False)

results_table= pd.DataFrame(index = range(len(c_param_range),3),
                            columns= ['C_parameter','Mean recall score','Mean precision score'])
results_table['C_parameter']= c_param_range

j=0
for c_param in c_param_range:
    print("-"*40)
    print('C parameter: ', c_param)
    print("-"*40)

    recall_accs=[]
    precision_accs=[]

    # Call the logistic regression model with a certain C parameter
    lr = LogisticRegression(C = c_param, penalty = 'l1')

    # Use the training data to fit the model. In this case, we use the portion of the fold to train the
    # with indices[0]. We then predict on the portion assigned as the 'test cross validation' with indic
    lr.fit(X_TYPES_sm,Y_TYPES_sm)

    # Predict values using the test indices in the training data
    y_pred_oversample = lr.predict(X_test)

    # Calculate the recall score and append it to a list for recall scores representing the current c_pa
    recall_acc = recall_score(Y_test,y_pred_oversample)
    recall_accs.append(recall_acc)
```

```
 precision_acc = precision_score(Y_test, y_pred_oversample)
 precision_accs.append(precision_acc)
 print("recall score = {:.4f}, precision score = {:.4f}".format(recall_acc, precision_acc))

 # The mean value of those recall scores is the metric we want to save and get hold of.
 results_table.loc[j,'Mean recall score'] = np.mean(recall_accs)
 results_table.loc[j, 'Mean precision score'] = np.mean(precision_accs)
 j += 1
 print('')
 print('Mean recall score {:.4f}'.format(np.mean(recall_accs)))
 print('Mean precision score {:.4f}'.format(np.mean(precision_accs)))
 print('')
```

```
----------------------------------------
C parameter:  0.001
----------------------------------------
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver wi
ll be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
recall score = 0.9875, precision score = 0.0153

Mean recall score 0.9875
Mean precision score 0.0153

----------------------------------------
C parameter:  0.01
----------------------------------------
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver wi
ll be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
recall score = 0.9875, precision score = 0.0156

Mean recall score 0.9875
Mean precision score 0.0156

----------------------------------------
C parameter:  0.1
----------------------------------------
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver wi
ll be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
recall score = 0.9875, precision score = 0.0156
```

5.2.6 XGBoost (Extreme Gradient Boosting) [4]

It is an example of ensemble learning (many models work together). In Boosting, the models are built in series. In each successive model, parameters are adjusted based on the learning of the previous model. Gradient boosting is a type of boosting technique which works on reducing errors sequentially.

```
In [69]: X.loc[(X.oldBalanceDest == 0) & (X.newBalanceDest == 0) & (X.amount != 0), \
               ['oldBalanceDest', 'newBalanceDest']] = - 1
```

Since the destination account balances being zero is a strong indicator of fraud, we do not impute the account balance (before the transaction is made) with a statistic or from a distribution with a subsequent adjustment for the amount transacted. Doing so would mask this indicator of fraud and make fraudulent transactions appear genuine. Instead, above we replace the value of 0 with -1 which will be more useful to a suitable machine-learning (ML) algorithm detecting fraud.

```
In [70]: X.loc[(X.oldBalanceOrig == 0) & (X.newBalanceOrig == 0) & (X.amount != 0), \
               ['oldBalanceOrig', 'newBalanceOrig']] = np.nan
```

The data also has several transactions with zero balances in the originating account both before and after a non-zero amount is transacted. In this case, the fraction of such transactions is much smaller in fraudulent (0.3%) compared to genuine transactions (47%). Once again, from similar reasoning as above, instead of imputing a numerical value we replace the value of 0 with a null value.

```
In [71]: X['errorBalanceOrig'] = X.newBalanceOrig + X.amount - X.oldBalanceOrig
         X['errorBalanceDest'] = X.oldBalanceDest + X.amount - X.newBalanceDest
```

```
In [82]: from sklearn.model_selection import train_test_split
         from xgboost.sklearn import XGBClassifier
         from sklearn.metrics import average_precision_score
```

```
In [100]: trainX, testX, trainY, testY = train_test_split(X, Y, test_size = 0.4, \
                                                          random_state = randomState)
```

```
In [101]: weights = (Y == 0).sum() / (1.0 * (Y == 1).sum())
          clf = XGBClassifier(max_depth = 3, scale_pos_weight = weights, nthread = 4)
          probabilities = clf.fit(trainX, trainY).predict_proba(testX)
          print('AUPRC = {}'.format(average_precision_score(testY, probabilities[:, 1])))

          AUPRC = 0.9946477428486173
```

The easiest way to improve the performance of the model still further is to increase the *max_depth* parameter of the XGBClassifier at the expense of the longer time spent learning the model.

```
In [105]: weights

Out[105]: 336.3199805186899
```

```
In [106]: clf

Out[106]: XGBClassifier(base_score=0.5, colsample_bylevel=1, colsample_bytree=1,
                        gamma=0, learning_rate=0.1, max_delta_step=0, max_depth=3,
                        min_child_weight=1, missing=None, n_estimators=100, nthread=4,
                        objective='binary:logistic', reg_alpha=0, reg_lambda=1,
                        scale_pos_weight=336.3199805186899, seed=0, silent=True,
                        subsample=1)
```

min_child_weight [default=1]

Defines the minimum sum of weights of all observations required in a child. Used to control over-fitting. Higher values prevent a model from learning relations which might be highly specific to the particular sample selected for a tree.

max_depth [default=6]

Used to control over-fitting as higher depth will allow model to learn relations very specific to a particular sample.

gamma [default=0]

A node is split only when the resulting split gives a positive reduction in the loss function. Gamma specifies the minimum loss reduction required to make a split.

nthread

Number of parallel threads used to run XGBoost

5.3 Result snapshots and evaluation metrics

5.3.1 SMOTE and undersampling

The results including precision and recall values at different C parameter values are stored and plotted for Undersampling with Logistic Regression, Undersampling with Support Vector Machine and SMOTE with logistic regression.

Two graphs are C parameter Values vs Mean precision and C parameter Values vs Mean Recall respectively.

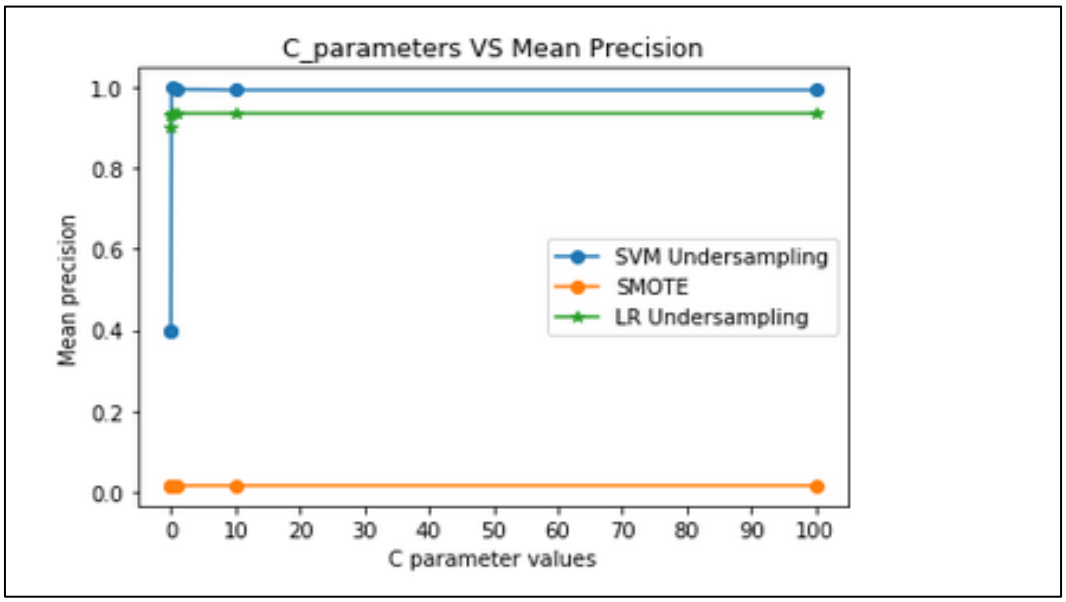

Figure 5.3: C parameter vs mean precision for SMOTE and undersampling

From fig 5.3 it is seen that the precision is better for Undersampling with Logistic Regression for very small values of C. But as C increases Recall values become better for SVM undersampling. SMOTE has worst precision among all three.

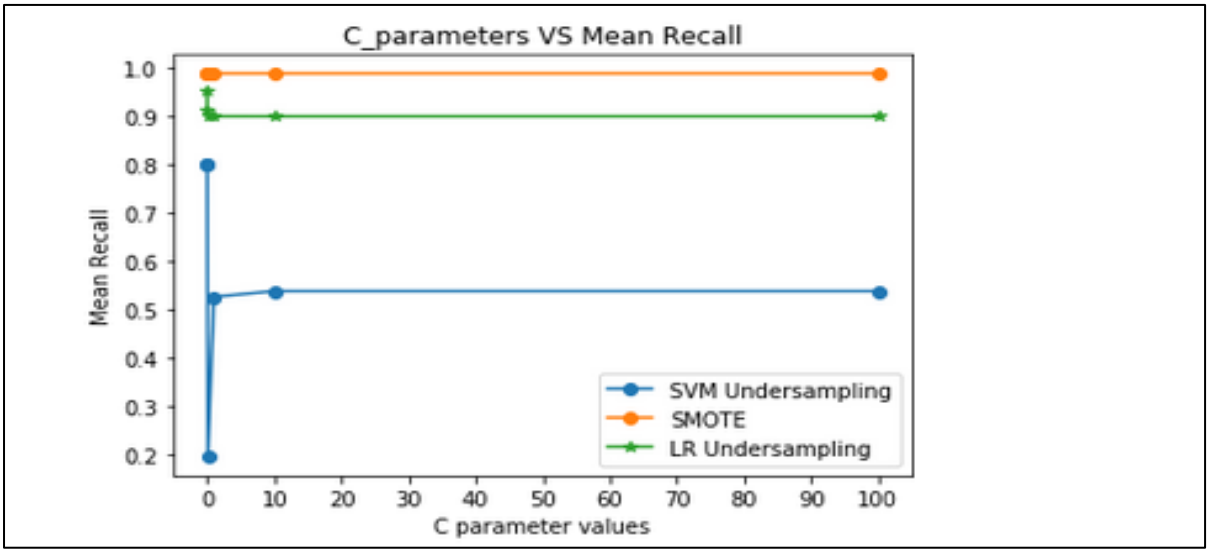

Figure 5.4: C parameter vs mean recall for SMOTE and undersampling

From fig 5.4 it is clear that the recall for SMOTE is better among the three and remains constant for all values of C. SVM undersampling has worst Recall among the three.

5.3.2 XGBoost

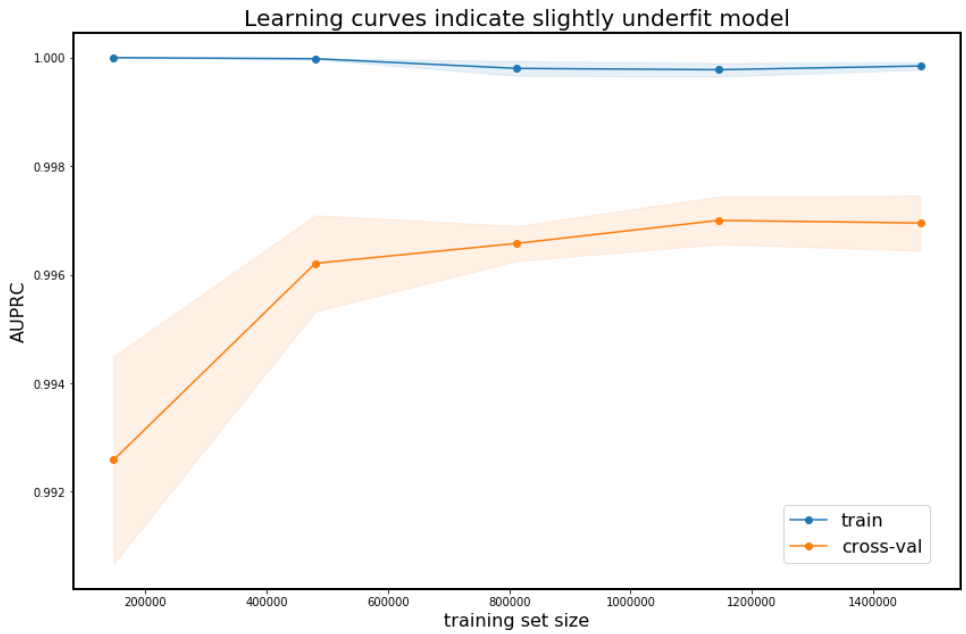

Figure 5.5: AUPRC values for training and test set

Figure 5.5 shows the AUPRC values for the training and test sets, in relation to the size of the training test. As the training set is increased, the test data starts performing better in terms of recall score. The rise is explained as the model is slightly underfit when training set size is low and has a degree of bias.

CHAPTER 6

CONCLUSION

This paper presented an approach for dealing with the class-imbalance problem that consisted of combining different expressions of re-sampling based classifiers in an informed fashion. In particular, our system was built so as to bias the classifiers towards the positive set in order to counteract the negative bias typically developed by classifiers facing a higher proportion of negative than positive examples. The positive bias we included was carefully regulated by an elimination strategy designed to prevent unreliable attributes to participate in the process. The technique was shown to be effective on Paysim synthetic dataset for money transactions as compared to a single classifier or another general-purpose method like Decision Trees.

We thoroughly interrogated the data at the outset to gain insight into which features could be discarded and those which could be valuably engineered. To deal with the large skew in the data, we chose an appropriate metric and used an ML algorithm based on an ensemble of decision trees which works best with strongly imbalanced classes. The method we used should therefore be broadly applicable to a range of such problems.

REFERENCES

[1] Estabrooks, A., Jo, T., & Japkowicz, N. (2004). A multiple resampling method for learning from imbalanced data sets. *Computational intelligence*, *20*(1), 18-36.

[2] Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, *16*, 321-357.

[3] Phua, C., Lee, V., Smith, K., & Gayler, R. (2010). A comprehensive survey of data mining-based fraud detection research. *arXiv preprint arXiv:1009.6119*.

[4] Chen, T., & Guestrin, C. (2016). XGBoost: Reliable Large-scale Tree Boosting System. arXiv 2016; 1–6. *DOI: http://dx. doi. org/10.1145/2939672.2939785*.

[5] Mudiri, J. L. (2013). Fraud in mobile financial services. *Rapport technique, MicroSave*, 30.

[6] Wang, J., Xu, M., Wang, H., & Zhang, J. (2006). Classification of imbalanced data by using the SMOTE algorithm and locally linear embedding. In *Signal Processing, 2006 8th International Conference on* (Vol. 3). IEEE.

[7] https://www.analyticsvidhya.com/blog/2017/03/imbalanced-classification-problem/

[8] https://www.kaggle.com/ntnu-testimon/paysim1

[9] https://www.analyticsvidhya.com/blog/2015/10/basics-logistic-regression/