IMPROVING SOFTWARE DEFECT PREDICTION USING CLUSTER UNDERSAMPLING

BY

MOSES APAMBILA AGEBURE
(10396888)

THIS THESIS IS SUBMITTED TO THE UNIVERSITY OF GHANA, LEGON IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE AWARD OF MPHIL COMPUTER ENGINEERING DEGREE

DEPARTMENT OF COMPUTER ENGINEERING
SCHOOL OF ENGINEERING SCIENCES
UNIVERSITY OF GHANA, LEGON

JULY, 2014

DECLARATION

I, Moses Apambila Agebure, hereby declare that this thesis document except where indicated by referencing, is my own work carried out under supervision in the Department of Computer Engineering, Faculty of Engineering Sciences, University of Ghana, Legon. I further declare that this thesis, either in whole or in part, has not been presented for another degree in this University or elsewhere.

…… ……

Moses Apambila Agebure Date

(Student)

…… ……

Dr Robert A. Sowah Date

(Principal Supervisor)

…… ……

Dr Nathan Amanquah Date

(Co-Supervisor)

DEDICATION

This work is dedicated to GOD ALMIGHTY, and all members of the Agebure family, most especially my dear wife Rose Asampana.

ACKNOWLEDGEMENT

My sincere thanks goes to my supervisors, Dr Robert A. Sowah and Dr Nathan Amanquah for their invaluable guidance, contributions, and encouragements towards the successful completion of this research. I also thank all members of the Department of Computer Engineering, especially Dr. Godfrey Mills, Mr Asiibi Ananga, and my colleagues Kofi Afrifa, Moses Amoasi, Derek Pobi, and Kwadwo for their generous support during my stay in the University.

My warmest appreciation goes to my brother, Andrews Baba Agebure and Prof Elkanah Oyetunji, former Dean of the Faculty of Mathematical Sciences and Director of Academic Quality Assurance Unit, UDS, for the mentoring role they played in my life.

Rose, thank you for understanding and sharing in my dreams.

I also wish to acknowledge the **Carnegie Corporation of New York** through the University of Ghana, under the **UG-Carnegie Next Generation of Academics in Africa** project for financially supporting this research work. I say thank you and may this project stay to support Africans yet unborn.

IMPROVING SOFTWARE DEFECT PREDICTION USING CLUSTER UNDERSAMPLING

Moses Apambila Agebure

ABSTRACT

Adequately learning and classifying datasets that are highly unbalanced has become one of the most challenging task in Data Mining and Machine Learning disciplines. Most datasets are adversely affected by the class imbalance problem due to the limited occurrence of positive examples. This phenomenon adversely affect the ability of classification algorithms to adequately learn from these data to correctly classify positive examples in new datasets. Data sampling techniques presented in Data Mining and Machine Learning literature are often used to manipulate the training data in order to minimize the level of imbalance prior to training classification models. This study presents an undersampling technique that has the capability of further improving the performance of classification algorithms when learning from imbalance datasets. The technique targets the removal of potential problematic instances from the majority class in the course of undersampling. The proposed technique uses Tomek links to detect and remove noisy/inconsistent instances, and data clustering to detect and remove outliers and redundant instances from the majority class. The proposed technique is implemented in Java within the framework of the WEKA machine learning tool. The performance of the proposed sampling technique has been evaluated with WEKA machine learning tool using C4.5 and OneR classification algorithms. Sixteen datasets with varying degrees of imbalance are used. The performance of the models when CUST is used are compared to RUS, ROS, CBU, SMOTE, OSS, and when no sampling is performed prior to training (NONE). The results of CUST are encouraging as compared to the other techniques particularly in datasets that have less than 2% minority instances and larger quantities of repeated instances. The experimental results using AUC and G-Mean showed that CUST resulted in higher performance in most of the datasets than the other methods. The average performance of the classification algorithms across the datasets for each technique also showed that CUST resulted in the highest average performance in all test cases. Statistical comparison of the mean performance also revealed that CUST performed statistically better than ROS, SMOTE, OSS and NONE in all test cases. CUST however, performed statistically the same as RUS and CBU, but with a higher mean performance. The results also confirmed that CUST is a viable alternative to the already existing sampling techniques particularly when the datasets are highly unbalanced with larger quantities of repeated, noisy instances and outliers.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

NASA	National Aeronautics and Space Administration
MDP	Metric Data Program
RUS	Random Undersampling
ROS	Random Oversampling
SMOTE	Synthetic Minority Oversampling Technique
OSS	One-Sided Selection
CBU	Cluster-Based Undersampling
CUST	Cluster Undersampling
BSMOTE	Borderline Synthetic Minority Oversampling Technique
WE	Wilson's editing
EVS	Evolutionary Undersampling
CBO	Cluster-Based Oversampling
AUC	Area Under the receiver-operating characteristic Curve
G-Mean	Geometric Mean
K-S	Kolmogorov-Smirnov
ANOVA	Analysis of Variance
HSD	Honestly Significant Difference
LOC	Line of Code
IDE	Integrated Development Environment
TP	True Positives
TN	True negatives
FP	False Positives
FN	False Negative
TPR	True Positive Rate
TNR	True Negative Rate

FPR	False Positive Rate
FNR	False Negative Rate
Vs	Versus
Prec.	Precision
Acc.	Accuracy

CHAPTER ONE

INTRODUCTION

1.0 Introduction

Classification is a well-studied technique in the machine learning and data mining communities [1], [2], [3]. Classification involves learning from existing data with known classes to predict or classify incoming examples or instances with unknown classes as belonging to a particular class or sub-concept. That is, classification models or classifiers are built or trained using datasets with labelled class attributes and used to classify new or unlabelled examples. Classification has been widely used in real world applications due to its predictive or forecasting nature. For instance, a medical prediction model or system can be built using medical datasets that are based on the symptoms of patients who suffered from a particular ailment and used to predict if a new patient arriving at the hospital is suffering from that ailment or not, given the symptoms exhibited by the new patient.

Classification is data dependent and the algorithms belong predominantly to a discipline of machine learning algorithms referred to as supervised learning algorithms. In supervised learning, classification models are trained using datasets that have labelled class attributes, that is, each example or instance in the training dataset belongs to one of two or more sub-concepts or classes and labelled as such. This implies that the nature or quality of the training data plays a key role on the prediction accuracy or how well a classification model would generalise when presented with new data.

The performance of classification models are influenced negatively by several data quality issues when present in the training data in substantial amount of which noisy instances [4], [5], outliers [6], and class imbalance [7], [8], [9] are examples. Noise and outliers in most cases are erroneously created and can be removed using data pre-processing or cleansing tools. Class imbalance on the other hand is a major data intrinsic problem that plaques so many high risk application domains including software defect prediction or quality estimation [10], [11], [12], medical datasets [13], fraud detection [14], intrusion detection [15], and risk management [16].

Class imbalance unlike noise and outliers is not a problem that arises potentially from mechanical errors or deficiencies in the data generation processes but occurs due to the nature of most application domains and therefore cannot be avoided. For example, consider a standard software system under development, it is generally expected that the number of faults or defects in the software be as minimal as possible. That is, the number of components (e.g. modules, functions, procedures etc.) in the software system that may contain defects in a normal case is expected to be small. If data is generated from this software based on, for instance its modules and the modules (data generated from a module in this case constitutes one data point, instance or example in the dataset) with defects labelled as positive examples and those without defects as negative examples. Then, it is obvious that, the positive examples (modules with defects) would constitute a smaller percentage of the data generated as compared to the negative examples (modules without defects).

In some cases the difference between the number of positive and negative examples could be very high such that the positive examples constitute less than 1% of the

dataset [11]. This phenomenon referred to as class imbalance or unbalanced datasets is reported to undermine the learning ability of most classification algorithms and as a result they tend to predict that new datasets have only the negative class examples and thereby incorrectly classifying the positive class examples, which are of interest.

A classification model trained using a dataset with the above stated distribution would inadequately learn to discriminate the minority class instances and therefore when used to classify a new dataset with similar distribution may return a 99% classification accuracy by only correctly classifying the majority class instances while wrongly classifying the minority instances. In this case, it means that all the defective software modules the process is set to identify would be wrongly classified and thereby defeating the purpose it is set to achieve.

Class imbalance is however, predominant in high risk domains as mentioned above, and in these areas wrongly classifying a positive example comes with a higher cost as compared to that of a negative example [17]. The issue of class imbalance in real world datasets, and its effects on statistical and machine learning algorithms cannot be overlooked. This research therefore looks at some outstanding data level approaches to imbalance learning and how new sampling techniques that take other data quality issues such as noise or inconsistent instances and outliers into consideration when sampling can be adapted to improve the performance and reliability of statistical and machine learning algorithms when learning from imbalance datasets.

1.1 Problem Statement

The case of class imbalance is inherent in most high risk machine learning datasets and therefore possess a greater challenge to the successful application of classification algorithms in such areas. As a result, researchers have explored the use of several class imbalance learning techniques with the goal of improving the performance of classification models. That is, to generate the best possible classification models by maximizing true positives (correctly classifying positive instances) while minimizing false positives (incorrectly classifying negative instances as positive examples). Common among these techniques is the data level or sampling (oversampling and undersampling) approach. Some of the most frequently used sampling techniques include Random undersampling (RUS), Random oversampling (ROS), Synthetic Minority Oversampling Technique (SMOTE) [17], and One-Sided Selection (OSS) [18].

SMOTE was proposed to alleviate the over fitting problem of random oversampling due to the random replication of minority class instances which only increase the size of the dataset but do not add information to it. The mode in which SMOTE creates synthetic instances between a minority instance and its nearest neighbours sometimes worsens the class overlap problem in some datasets depending on the sparseness of the minority instances and is also subject to overgeneralisation [19]. OSS was also introduced to avoid the indiscriminate removal of majority class instances in RUS, which leads to significant loss of information, by only removing redundant, noisy and borderline instances. OSS, however, does not prevent the inclusion of outliers in the sampled data, which has the potential of increasing the misclassification error rates of

classification algorithms [6]. OSS has also been consistently outperformed by RUS in most empirical studies particularly using software defect datasets [4], [20], and [21].

This research therefore seeks to propose a new Cluster Undersampling Technique (CUST) that has the capability of limiting the inclusion of outliers and also repeated and noisy instances in the sampled data thereby eliminating the presence of problematic instances that have the potential of further degrading the performance of classification models.

1.2 Objectives of the Study

The main objective of this study is to design an algorithm for undersampling majority class instances in unbalanced datasets so as to improve the ability of classification algorithms to correctly classify minority instances.

Specific Objectives

To achieve the ultimate objective, the specific objectives of the study include:

- To derive an algorithm that is effective and efficient in undersampling majority class instances in unbalanced datasets
- To implement the algorithm in a Java IDE within the frame work of the WEKA machine learning tool.
- To incorporate the implemented algorithm into the WEKA machine learning tool for testing purpose.
- To assess and evaluate the efficiency of the proposed algorithm by comparing its performance to already existing sampling techniques.

1.3 Justification of the Study

Class imbalance is a naturally occurring problem in most real world machine learning domains such as fraud detection, network intrusion detection, risk management, medical datasets, and software defect prediction among others. It occurs as a result of the limited number of examples of a particular class, mostly the class the process is set to identify, due to the rarity of the phenomenon the class represents.

A key motivation of class imbalance learning is the utmost importance attached to the correct classification of the rather minority instances because they carry a higher cost if they are wrongly classified as compared to the majority instances. As prediction models are usually set to optimize the overall classification accuracy, in most cases they fail to correctly classify the minority instances due to rarity of the instances in the data used to train them.

Data sampling is one of the techniques of minimizing class imbalance problem prior to training classification models, but most of the existing techniques cause other problems in the course of sampling and mostly do not take other data quality issues into consideration. This work is therefore, aimed at designing an efficient sampling algorithm that improves the performance of classification algorithms using a simple logical framework.

This research contributes to both the academia and practice by presenting a comprehensive literature review on class imbalance learning and further proposing and implementing a new undersampling technique (Cluster Undersampling Technique (CUST)) that has been shown to significantly improve the performance of common

classification techniques such as C4.5 decision tree and OneR when learning from imbalance datasets.

1.4 Scope of the Study

The scope of the research covers a study of some selected sampling techniques or data level approaches to imbalance learning and other data quality issues that affect classifier performance in order to design a new and efficient sampling technique for undersampling highly skewed datasets. Other already existing sampling techniques such as random undersampling, random oversampling, synthetic minority over sampling, one-sided selection, and cluster-based undersampling techniques are also implemented to form the foundation for assessing the performance of the proposed sampling technique. These sampling techniques are implemented in Java within the framework of the WEKA machine learning tool.

Two classification algorithms, C4.5 decision tree algorithm and OneR are considered for testing the proposed sampling algorithm because they have been widely used in class imbalance learning and are reported to be efficient. All testing experiments are carried out using the WEKA machine learning tool [48].

In addition, static code attribute datasets from the NASA MDP project [22] and six (6) other datasets from the UCI repository [23] are used for the study. The original scope of the study was to use only software defect datasets from the NASA MDP project, but because the class imbalance problem is not limited to only software defect prediction domain, and following due recommendations from the researcher's supervisors, additional datasets from the UCI repository are used to further test the

proposed sampling technique. The datasets from the UCI repository are not software defect datasets.

1.5 Organisation of Thesis

The remaining chapters of the thesis are organised as follows:

Chapter two entail a review of related background literature on class imbalance problem. A review of some empirical studies that used data level approach to imbalance learning, the classification and sampling techniques as well as a brief review of literature on performance metrics.

Chapter three outlines the research methodology. This includes a description of all algorithms used. It also outlines the experimental procedure/framework, the methods used to measure the performance of the classification models and the statistical methods used to compare the performance of the sampling techniques. A detailed description of the datasets and data pre-processing methods are also given.

Chapter four presents the design of the proposed cluster undersampling technique.

Chapter five entails the implementation and testing of CUST, and a detailed discussion of the experimental result and findings.

Chapter six, which is the concluding chapter contains the conclusion and recommendations based on the findings.

CHAPTER TWO

LITERATURE REVIEW

2.0 Introduction

This chapter presents a review of background literature on the class imbalance problem, empirical studies that used the data level approach to imbalance learning, a discussion of some data sampling techniques, classification algorithms that are used as well as the performance metrics considered for the evaluation of the models.

2.1 The Class-Imbalance Problem

An imbalance dataset in a typical case refers to a dataset with uneven class distribution. This refers not necessarily to a situation where the number of examples in the classes are not equal because it is rare to have a dataset with equal number of examples in each class. But datasets with significant or severe difference in the class distribution. The imbalance problem can be categorised into within-class imbalance and between-class imbalance [24].

Within-class imbalance arises when there is not enough data representation for a certain sub-concept within the minority or majority class due to rarity of data. That is, when the minority or majority class data represents more than one concept and one of the sub-concepts is a rare case and therefore has limited data representation. Borrowing an example from [25], consider an instance where a dataset contains data on healthy and sick people, with the healthy group been the majority and the sick the minority. Assuming that within the sick class, some people are suffering from botulism, a relatively rare illness as compared to the presence of other common illness

such as cold. In this case the imbalance has occurred within the minority class due mainly to the rarity of data for a sub-concept.

Between-class imbalance on the other hand is much concerned with the relative distribution of classes in the dataset. The severity of the imbalance varies with respect to the ratio of majority to minority classes and also the total number of examples in the entire dataset. For example, the class imbalance problem for a dataset with 1,000,000 negative examples and 10,000 positive examples is reasonably different from a dataset with 1000 negative examples and 10 positive examples, though the negative/positive class proportions are identical (100:1). The first problem can be referred to as a problem with relative rarity (relative imbalance) and the latter as a problem with absolute rarity (absolute imbalance) [25]. Relative imbalance is quite a common problem in most real-world applications and has attracted the focus of most knowledge discovery and data engineering research efforts, but in some relative imbalance datasets, the minority concept may be well represented due to the large quantity of data and therefore is accurately learned with little disturbance from the imbalance [24]. However, this is not always the case since most real world data come in limited quantities due to scarcity of specimen from which these data are generated.

Learning from unbalanced datasets or imbalance learning is defined by He [26] as "the learning process for data representation and information extraction with severe data distribution skews to develop effective decision boundaries to support the decision-making process." Most often than not imbalance learning seeks to maximize the correct classification or prediction of minority class examples due to limitations of most traditional classification algorithms to adequately learn from these datasets. For

example, training and testing a classification model with a dataset that has 99% of its instances belonging to the majority class and only 1% minority class instances, the model is most likely to classify all test set as majority class instances and therefore may return a 99% accuracy which would have been highly accepted in machine learning for datasets with even class distribution. But in this case it has only correctly classified the majority class instances while misclassifying the minority class instances which the process is set to achieve. This model though with a very good accuracy is not practically viable.

In critical real-world applications wrongly classifying a minority class instance comes with a higher cost as compared to that of a majority class instance. For instance, in software defect prediction, the aim is often to predict possible areas of software that defects are likely to reside. If a defective module is wrongly classified as a non-defective module by the prediction model then it means that the defects in that module will likely not be fixed during the testing phase since the testing team may not consider it as one of the target modules. The consequence of this is obvious; releasing a software system that will fail in the course of usage and as indicated in [27], the cost of fixing a software defect after it has been released may be 100% more expensive as compared to fixing it during the development process.

Due to the challenges of learning from imbalance datasets and the critical nature of areas where it is prevalent, several imbalance learning approaches have been proposed in literature and targeted at improving the correct classification of minority class instances while minimising false alarm rates. These approaches can be grouped into; data level/sampling approach, kernel-based learning, cost-sensitive learning, active

learning, and one-class learning approach [26]. This study is however focused on data level approach to imbalance learning.

2.2 Empirical Studies using Data Sampling

Class imbalance learning have received tremendous attention from the data mining and machine learning community. Most of the studies are focused on how to improve the classification rates of minority class instances of predictive models using various forms of imbalance learning techniques. Common among these techniques is the data sampling approach, this technique is much preferred because instead of configuring parameters of classification algorithms to maximise the detection of the minority class instances when the built models are presented with new data, the training data is rather manipulated. This serves as a leverage for researchers who do not have the technical expertise to configure the classification algorithms in the learning process [28].

Data sampling concerns itself with modifying the class distribution to yield more balanced datasets. There are two basic approaches to the modification of the class distribution namely oversampling and undersampling. As the names imply, oversampling methods increases the number of minority class instances while undersampling methods discards some majority class instances in order to get a balanced dataset.

Assessing the efficiency of data sampling techniques in software defect prediction, Seiffert et al. [11] examined how data sampling techniques and boosting can improve the performance of C4.5 decision tree algorithm. Three (3) oversampling techniques;

random oversampling, synthetic minority oversampling (SMOTE), borderline-synthetic minority oversampling (BSMOTE) and two undersampling techniques; random undersampling (RUS) and Wilson's Editing (WE) were considered. Using Kolmogorov–Smirnov (K-S) statistic and Area Under the receiver-operating-characteristic Curve (AUC), the authors indicated that all the sampling methods improved the performance of the C4.5 decision tree with RUS outperforming the other methods in thirteen (13) out of the fifteen (15) datasets they used. The boosting method however, performed relatively better than all the sampling methods. The authors however considered only the C4.5 decision tree classification algorithm, and recommended a further exploration of these techniques using other classification algorithms.

Riquelme et al. [29] analysed the impact of SMOTE and resampling on two (2) classification algorithms; Naïve Bayes and C4.5 decision tree algorithm. The authors reported that although sampling did not improve the percentage of correctly classified instances (i.e. the overall accuracy), they did improved the area under the receiver-operating-characteristic curve measure, that is, they classified better the minority class instances, which are of interest. In a similar work, Pelayo & Dick [30] examined stratification, and focused the experiment on SMOTE with the goal of determining if data stratification and SMOTE can improve the classification algorithms ability to recognise defect-prone modules and at what cost. The authors demonstrated that after SMOTE resampling, a more balanced classification accuracy was realised. An improvement of at least 23% in average geometric mean of the classification accuracy on four (4) benchmark datasets.

A novel genetic algorithm-based sampling method (Evolutionary Undersampling method (EVS)) was also proposed by Drown et al. [20] and its efficiency accessed using two classification algorithms; C4.5 decision tree and RIPPER. Comparing its results to that of other sampling methods using AUC, G-Means, and F-measure as performance metrics, they indicated that EVS outperformed OSS, WE, ROS, cluster-based oversampling (CBO), SMOTE, and BSMOTE except RUS which produced competitive results.

Seiffert et al. [4] studied the effects of noise and imbalance on eleven (11) classification algorithms and five (5) sampling techniques. They considered the following sampling techniques; ROS, RUS, OSS, cluster-based oversampling (CBO), WE, SMOTE, and BSMOTE. They indicated that, the presence of noise and imbalance in datasets do affect the performance of classification algorithms and sampling techniques. RUS was however, reported to yield the best performance at all imbalance and noise levels.

García et al. [31], empirically examined the effect of class imbalance ratio and classification algorithm on the effectiveness of data sampling techniques. They used 17 datasets from the UCI repository [23], 8 classifiers and 4 sampling techniques for their experiments. They indicated that, oversampling techniques performed better than undersampling techniques in much skewed datasets, but their performance is relatively the same in datasets that are less skewed. They concluded that, characteristics of classifiers have little influence on the effectiveness of the different data sampling techniques, and the choice of a sampling technique should be done while considering the characteristics of the dataset at hand.

In a summary, the results of the above studies points out that, indeed data sampling is an efficient method for handling the class imbalance problem. It is also worth noting that, most of the undersampling methods, particularly RUS appears to be the favourite among the well-known sampling methods as it produced very competitive results in most of the software defect prediction studies [11], [4], [21], and [20].

2.3 Data Sampling Techniques

2.3.1 Random Resampling Techniques

The simplest and most common among the sampling techniques are random oversampling and random undersampling. Random oversampling randomly replicates the minority class instances thereby increasing the minority class population in the training data, random undersampling on the other hand randomly discards some majority class instances hence, reducing the size of the majority class in the training data [26]. Random undersampling and oversampling have their respective pros and cons. Though these methods are the easiest to use and much faster than other techniques, a major problem associated with random undersampling is the loss of information that occurs due to the random removal of examples from the training data and this problem aggravates when the training datasets are small. This however, is not a problem with oversampling, but rather, the increase in size of the resulting training dataset after oversampling can lead to longer training time and also, the fact that instances are randomly replicated, which adds no new information may result to overfitting [11].

2.3.2 Synthetic Minority Oversampling (SMOTE)

SMOTE [17] oversamples a dataset by creating synthetic minority instances and not by duplicating already existing minority class instances as done in random oversampling. It finds the k nearest neighbours of each instance in the minority class and then generate the synthetic instances in the direction of some or all of the nearest neighbours of an instance depending on the percentage of oversampling required. In the original work, [17] $k = 5$ was used. To create a synthetic instance, the difference between the feature vector of the minority instance under consideration and its nearest neighbour is calculated and multiplied by a random number between 0 and 1. The product is then added to the feature vector of the minority instance under consideration to form the new instance.

Though SMOTE has been used extensively in various fields of application with good results, its mode of creating the synthetic examples is sometimes problematic particularly in very skewed dataset as it blindly generalizes the regions of the minority class without regard to the majority class, this depending on the sparseness of the minority instances in the dataset may led to an increase in class overlapping [19].

2.3.3 Cluster-Based Undersampling

The cluster based undersampling technique proposed by Das et al. [32] is aimed at solving the class imbalance problem by discarding majority instances in overlap regions of the datasets. This is achieved by clustering the training dataset into some k clusters and discarding all majority class instances from clusters which satisfy

$0 < r < 1$ and $r \geq \tau$ where:

$$r = \frac{c^{i}}{|C|} \tag{2.1}$$

c^i is the number of minority instances in cluster i , $|C|$ is the total number of instances in cluster i, and τ is an empirically determined value. They argued that doing this will minimise the class overlap problem whilst reducing the number of majority class instances, hence minimising the imbalance in classes.

The CBU, though is strategically design to eliminate the class overlapping problem as oppose to SMOTE, also has a limitation when sampling from much skewed datasets as it fails to adequately reduce the number of majority class instances. For example, the PC2 dataset considered in this study have approximately 14 minority and 1412 majority instances in the training set after stratification. Clustering this data may result in a lot of clusters having only majority instances, this implies that the majority instances in these clusters would be retained as per the rational of the technique and the few clusters that contain minority instances would be discarded depending on the τ value considered. This implies that only a small percentage of the majority class instances would be discarded.

2.3.4 One-Sided Selection (OSS)

Sampling techniques such as RUS do not take other data quality issues such as the presence of inconsistent/noisy instances, redundant instances, and outliers, which have the potential of affecting accurate estimation of the generalisation of classification models into consideration when sampling. These problems are inherent in most real world datasets as they may be created erroneously or arise due to the

nature of the application domain [5], [6]. It may be argued that these problematic instances can easily be removed during data cleansing processes, but depending on the nature of the data at hand, blindly removing instances from the dataset may further worsen the class imbalance problem.

The main aim of the OSS undersampling techniques as proposed by Kubat and Matwin [18], is to remove noisy/borderline and redundant majority class instances from the training data as a form of undersampling. This, they suggest would reduce the number of majority class instances and thereby minimize the difference in class distribution. OSS undersamples a dataset S by first moving all minority instances and one randomly selected majority instance into a dataset C. C is used to train a 1-NN rule, which is used to classify the instances in S and all misclassified instances are moved from S to C. C is now consistent with S but smaller in size. All majority class instances in C that participate in Tomek links are removed, this process removes noisy and borderline instances. The remaining instances in C are then considered as the final training set. The OSS does not explicitly avoid the inclusion of outliers from the final training set.

The performance of the OSS technique in most empirical studies, particularly in software defect prediction where the presence of noisy and repeated instances is evident, is not encouraging as it has been consistently outperformed by other techniques such as RUS and SMOTE [4], [20], and [21].

2.4 Effects of Outliers

An outlier is defined by Hawkins [33] as "an observation that deviates so much from other observations as to arouse suspicion that it was generated by a different mechanism." According to Acuña and Rodríguez [6] outliers "may arise in a dataset due to mechanical faults, changes in system behaviour, fraudulent behaviour, human error, instrumentation error, or simply through natural deviation from a standard situation… and therefore outlier detection has applications in areas such as: fraud detection, network intrusion, and data cleaning."

Acuña and Rodríguez [6] examined several methods for detecting outliers in multivariate datasets including: robust statistical based outlier detection, detection of outliers using clustering, distance based outlier detection, and density-based local outlier detection methods. They indicated that no outlier detection method can be uniquely recommended and the efficiency of a method depends on the type of outliers that exist in the dataset. For instance, they showed that robust methods are good at detecting scattered outliers but not clustered outliers, which are best detected using clustering methods. Their experimental results showed that, the misclassification error rates of the classifiers decreased by about 25% when outliers are removed from the datasets prior to training. This suggest that, detecting and removing outliers from the majority class as part of an undersampling process to a greater extend would further minimise the misclassification error rates when learning from unbalanced datasets.

2.5 Classification Algorithms

2.5.1 C4.5 Decision Tree Algorithm

C4.5 Decision tree [34] is a tree based learner that improves upon the ID3 [35] learner by adding support for handling missing values, numeric attributes, and tree pruning (a measure adopted to avoid overfitting). It creates classification models using a statistical property called information gain that measures the effectiveness of an attribute to separate the training instances according to their target classification. Information gain is based on entropy, a measure used in information theory to characterise the purity or impurity of an arbitrary collection of examples. For a given collection S, containing negative and positive examples of some target concept, the entropy of S relative to this classification is given as:

$$Entropy(S) = -p_+ \log_2 p_+ - p_- \log_2 p_- \tag{2.2}$$

where p_+ is the number of positive class examples and p_- is the number of negative class examples. The information gain $Gain(S, A)$ of a given attribute A in a collection S is given as:

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{S} Entropy(S_v) \tag{2.3}$$

$Values_{(A)}$ is the set of all possible values for attribute A.

2.5.2 OneR

OneR (1R) [36] is a rule based learner that ranks all attribute with respect to error rates on the training data as opposed to the use of entropy based measures in C4.5. It creates one rule for each attribute value. A rule states that for a given attribute *A* and value *V* the majority class is *C*. The rules that have the highest accuracy on the training data are applied to a hypothesis and those with accuracy below just choosing the majority class instances are pruned from the hypothesis. The final rules are then sorted in order of accuracy in the training data. It treats all attributes with numeric values as continuous and employs a rather straightforward method to divide the range of values into several disjoint intervals. It also treats missing values as legitimate values.

2.6 Static Code Attributes

Static code attributes are basic software metric data that is extracted directly from source code and often carry information of the source code such as its complexity and size. There are basically three categories of static code metrics namely; line of code (LOC) counts, Halstead's metrics [37], and McCabe's metrics [38]. Static code attributes have been extensively and successfully used in literature to build software defect prediction models, notwithstanding the on-going debate on the worth of predictive models built using these attributes.

Fenton & Pfleeger [39] whiles arguing about the worth of static code attributes gave an insightful example in which they demonstrated how the same software functionality is achieved using different programming language constructs, which results to different static measurements for that module. They used this example as a

basis to argue the uselessness of static code attributes. This however, is an indication that static code attributes are always not a complete characterisation of the internals of a function. They further noted that the main McCabe's attribute (cyclomatic complexity, or $v(g)$) is highly correlated with line of code. Shepperd & Ince [40] also presented empirical evidence that the McCabe static attributes offer nothing more than uninformative attributes like lines of code. They remarked that "for a large class of software it (cyclomatic complexity) is no more than a proxy for, and in many cases outperformed by lines of code."

Menzies et al. [41] however, refute the above claims by Fenton & Pfleeger and Shepperd & Ince stating that, "the supposedly better static code attributes, such as Halstead and McCabe, should perform no better than just simple thresholds on lines of code, and the performance of a predictor learned by a data miner should be very poor." They noted that, none of these is true for at least the datasets they used for their study. They showed that, their study found defect predictors with a probability of detection of 71% which is evidently higher as compared to a conclusion arrived at by a panel at IEEE Metrics 2002 [27] that manual software reviews can find ≈ 60% of defects. Also, the panel did not either support Fagan's claim [42] that inspections can find 95% of defects before testing or the claim of Shull that specialized directed inspection methods can catch an extra 35% defects than other methods [43]. It is also argued that methods such as manual code reviews are labour-intensive, and depending on the review methods, 8 to 20 LOC/minute can be inspected and this effort repeats for all members of the review team, which can be as large as four or six [44]. Static code attributes on the other hand, can be automatically and cheaply collected even for very large systems [45]. The significance of static code attributes is not limited only

to the above; as they have been widely used in software quality prediction research and interesting results have been turn out by these researchers [11], [29], [12], and [46]. Menzies et al. [47] while reiterating some of the arguments made in Menzies et al. [29] noted that; defect predictors are easy to use; are widely used and most importantly are useful.

2.7 Methodological Issues

2.7.1 Training and Testing Classification Models

In most machine learning applications, such as software quality prediction, classifiers are often built using already existing data and then deployed to classify new data (i.e. data the classifier has never seen). Building and deploying classifiers without foreknowledge of how well they may generalize (i.e. the error rates) given a new set of data may lead to deploying classifiers that will perform abysmally poor in the field. To avert the occurrence of this problem the classifier has to be tested using an existing dataset that do not form part of the training data or used anyway during classifier training. A common practice is to split the available data into training and testing sets depending on the model validation approach adopted. To achieve valid results the two datasets must not share common instances, since the real world training data and data that the classifier will be confronted with to classify will most likely not have the same instances. The training data (training set) is used to train the classifier and the testing data (test set) is used to test it to determine how well it generalises given a new dataset. Failure to adhere to this principle of training and testing classifiers with independent datasets usually result in an optimistic approximation of the potential real-world performance [48].

The technique of splitting the data into training set and test set is referred to as the holdout method and 60% or 70% or two-thirds of the data is often used as the training set with the remaining 40% or 30% or one-third as the test set. To curtail the occurrence of any statistical bias the holdout process is normally repeated several times with different training and testing data chosen pseudo-randomly and the results of all the holdout experiments averaged to obtain the final performance of the classifier. This technique however, works well when the data size is large with an even class distribution [48]. For instance, consider the NASA MDP software defect dataset, CM1 [22] with a total of 505 modules (instances). In this dataset only 10% of the modules are defective or contain defects and the remaining 90% of the modules are defect-free. Applying the holdout technique on this dataset with 70% training and 30% test set, an extreme situation may arise where the training set may contain only defect-free modules. Training a classifier using this data means that the classifier will only learn to predict the majority class examples (defect-free modules) and will classify any other example (defective modules) as belonging to the majority class because it did not have the privilege to learn from such data during the training process. One way of coping with this problem is using stratified holdout [48].

Stratified holdout ensures that both the majority and minority class examples are evenly represented in each holdout, that is two-thirds (or any split ratio) of the minority and majority class examples are used as the training set and the remaining one-third of both examples is used as the test set thereby eliminating the possibility of one class training set. Just like the simple holdout, the stratified holdout is repeated many times whiles randomly stratifying examples into training and test set at each experiment and the average performance taken as the final performance of the

classifier. Although this technique ensures the presence of examples of all the classes in the training and test set, the issue of biasness is not entirely eliminated. That is, it is possible that some of the instances might never be used as testing data in all the repetitions since they are randomly assign to the training and test set. To ensure that each example is used at least once as part of test set, the *k-fold cross-validation* process is recommended [49].

The k-fold cross-validation splits the dataset into $k>1$ approximately equal folds or sub-samples. The k-folds are mutually exclusive, that is no two folds contain the same examples. Training and testing is repeated k times and k-1 folds is used as the training set and the remaining fold as the test set at each repetition, this is done such that each fold is used only once in the k repetitions as test set. The average of the performance measures of the k repetitions is determined and used as the final performance of the classifier. A variant, the stratified k-fold cross-validation may be used particularly in the case of imbalance datasets to ensure that all the folds have the same number of negative and positive examples. The cross-validation process is repeated many times with different pseudo-randomly chosen sub-samples in order to further reduce biasness. Though the k-fold cross-validation process is computationally expensive, it is quite stable and unbiased for adequately sized datasets, where k=10 folds is normally recommended [49].

The ten-fold cross-validation process is a widely used and recommended technique in class imbalance learning. Seiffert et al. [11] while examining how sampling methods can improve the performance of C4.5 decision tree classification algorithm use ten repetitions of the ten-fold cross validation process to build and evaluate the

classification models. They stated that, repeating the experiments ten times eliminates any biasing that might be introduced by the random components of the partitions and sampling procedures. For similar reasons [30], [29], [4], [50], and [41] also used ten-fold cross validation process in their studies. Considering the advantages of using this process, the stratified 10 folds cross-validation process with ten repetitions is considered for this study.

2.7.2 Performance Metrics

After training a classifier it is tested using the test dataset to determine how well it will generalise given a new dataset. Determining how well a classifier generalises is invariably measuring the error-rate or accuracy (correct classification rate) of the classifier given the test dataset. Considering a binary classification problem, there is always one of four possible groups each example predicted in a classification experiment will belong [41]:

- True positives (TP): positive examples that are correctly predicted.
- True negatives (TN): negative examples that are correctly predicted.
- False positives (FP): negative examples that are incorrectly predicted as positive examples.
- False negatives (FN): positive examples that are incorrectly predicted as negative examples.

These measures are often used to form a confusion metrics [51] as shown in table 2.1 below, that describes the basic class specific predictions of the classifier. Using this confusion metrics almost all existing classification performance metrics can be derived.

Table 2.1: Confusion Metrics of Binary Class Classification

Predicted as positive	Predicted as negative	
TP	FN	Actual positive class
FP	TN	Actual negative class

Overall accuracy and error-rate of a classifier are among the commonly used performance metrics in machine learning, overall accuracy (Acc.) and error rate at a given threshold can be expressed as in equation 2.4 and 2.5 respectively:

$$Acc = \frac{TP + TN}{(TP + TN + FP + FN)} \tag{2.4}$$

$$Error\ rate = \frac{FP + FN}{(TP + TN + FP + FN)} \tag{2.5}$$

That is, accuracy measures the proportion of instances that are correctly classified, and on the other hand, overall error rate is a measure of the proportion of instances that are erroneously classified. Accuracy has an optimal value of 1 or 100% when expressed as a percentage.

These metrics favours the majority class in the case of class-imbalance datasets and are therefore ideal and more appropriate for determining the performance of prediction models that are built using datasets with evenly distributed classes. It is argued that using these methods to determine the performance of predictive models using unbalanced datasets might present misleading results [11], [41]. For instance,

given a dataset with only 10% positive instances and 90% negative instances, a prediction model might turn out 90% classification accuracy whiles in reality have correctly classified only the negative instances and incorrectly classified all the positive instances which the process is set to identify. This model will therefore have 90% accuracy but is practically not viable.

To avoid the problems associated with the use of accuracy performance metrics when the datasets are unbalanced, it is recommended that measures based on class specific metrics be used [52]. As indicated earlier, a typical binary classification experiment returns four possible outcomes; TP, TN, FP, and FN. Based on these measures class specific performance metrics such as sensitivity (true positive rate (TPR) or recall also referred to as the probability of detection in [41]), and false positive rate (FPR) referred to as the probability of false alarm in [41] can be calculated. Also, specificity (or true negative rate (TNR)) and false negative rate (FNR) can be obtained from these four measures as follows:

$$TPR = \frac{TP}{TP + FN} \tag{2.6}$$

$$TNR = \frac{TN}{TN + FP} \tag{2.7}$$

$$FPR = \frac{FP}{TN + FP} \tag{2.8}$$

$$FNR = \frac{FN}{TP + FN} \tag{2.9}$$

Optimum classification is achieved when sensitivity and specificity have values of 1 each, that is, the classifier has been able to classify all positive and negative class examples correctly in which case FPR and FNR will be 0 (i.e. at their optimum values). This however, is not always the case in real world applications with unbalanced dataset, since there is always a higher risk of misclassifying minority class instances. There is therefore a trade-off between sensitivity and specificity and the price of high sensitivity is higher false positive rate (i.e. lower specificity) [41]. The trade-off between sensitivity and specificity and the performance of the classifier across the entire range of possible decision threshold can be visualized with a receiver operating characteristic curve with sensitivity on the *y-axis* and the complement of specificity, the false positive rate on the *x-axis* [53].

Area Under receiver-operating characteristic Curve (AUC) provides a single measure from a Receiver-Operating Characteristic (ROC) curve. The AUC measure is independent of a selected decision threshold and thus gives a classifier's performance without considering prior probabilities or misclassification cost. These attributes of AUC makes it a good metric for classifier performance under unbalanced datasets [61]. The AUC has a value between 0 and 1, with 0 indicating the worst performance and 1 the highest performance of a classifier and is given as:

$$AUC = \frac{1 + TPR - FPR}{2} \tag{2.10}$$

Kubat and Matwin [18] also suggested the Geometric mean (G-Mean) as a good metric for classification problems involving class imbalance. The G-Mean like the

AUC provides a single numeric measure of a classifier's performance using its true positive rate and true negative rate. It gives a value between 0 and 1, with 0 indicating worse performance and 1 highest performance. It is calculated as:

$$G - Mean = \sqrt{TPR \times TNR} \tag{2.11}$$

Table 2.2: Summary of Selected Empirical Studies Reviewed

Article	**Purpose**	**classification algorithms explored**	**Sampling methods**	**Conclusion**	**Performance measure**
Seiffert et al. [4]	Studied the effects of noise and class-imbalance on sampling and classification algorithms	They considered 11 classification algorithms	Compared RUS, ROS, OSS, WE, SMOTE, BSMOTE, and CBOS	RUS, WE, and BSM are reported to be more robust than the others and also noise do have an effect on sampling and classification algorithms	Used AUC to measure learners performance and a four factor ANOVA to compare performance of learners
Riquelme et al. [29]	Finding defective modules from highly unbalanced datasets	Naïve Bayes and C4.5 decision trees algorithms	Explored SMOTE and resampling	They reported that the methods improved the AUC measure of both learners	AUC
Seiffert et al. [11]	Examined five sampling methods and boosting, how they can improve the performance of C4.5	C4.5 decision trees	Compared RUS, WE, ROS, SMOTE, BSMOTE and AdaBoost using K-S Statistic and AUC.	They concluded all methods significantly improved the performance of the C4.5. RUS performed better than all sampling methods but boosting outperformed it.	AUC and K-S statistic to measure learners performance and ANOVA to compare results
Pelayo and Dick [30]	Studied the application of Novel Resampling Strategies to Software Defect Prediction	C4.5 decision trees	They focused on SMOTE with the goal to determine if SMOTE can improve recognition of defect-prone modules, and at what cost	An improvement of at least 23% in the average G-mean classification accuracy was recorded	G-mean
Drown et al. [20]	Proposed a novel genetic algorithm-based sampling method (Evolutionary Undersampling method (EVS))	C4.5 decision trees and RIPPER	Compared the impact of EVS to (RUS, OSS, WE) and (ROS, CBOS, SMOTE, BSMOTE) using AUC, G-Means and the F-measure	EVS outperformed all over and undersampling sampling methods except RUS which produced competitive results. But EVS performed better with smaller data sets	Used AUC and G-mean to measure the performance of learners, and ANOVA and Tukey's Honestly Significant Difference (HSD) ranking to compare the results of learners
V. García et al. [31]	Assessed effect of imbalance ratio and classifiers on sampling techniques	8 classifiers	SMOTE, gg-SMOTE, RUS, and WE+MSS	Oversampling techniques performed better than undersampling in most skewed datasets, and classifier characteristics have no effects on sampling techniques.	True positive rate, true negative rate, G-mean, and IBA

CHAPTER THREE

METHODOLOGY

3.0 Introduction

This chapter describes the general experimental setup including: the datasets used, data pre-processing methods, data sampling techniques, classification algorithms, performance metrics as well as the experimental framework. It also gives a justification for every course of action taken.

3.1 Experimental setup

3.1.1 Datasets

The sixteen datasets used in this research are sourced from two publicly available data repositories, National Aeronautics and Space Administration (NASA) Metric Data Program (MDP) repository [22], University of California, Irvine repository [23] and one dataset from the WEKA sample datasets that are distributed with it.

The NASA datasets were obtained from various software projects carried out at NASA through their Metric Data Program (MDP) [22], the datasets are also made available through the PROMISE repository [54]. This research adapts the cleansed version of the datasets labelled D' by Shepperd et al. [55] in order to allow easy verification of the results and reproducibility of the experiments since the original version of the datasets contain some problems such as implausible and missing values. The cleansed versions of the data are also available at [54]. The cleansing of the data was carried out by Shepperd et al. [55] to get rid of anomalous data. This was necessitated by the fact that different versions of the NASA datasets were available from the NASA MDP repository and PROMISE repository with inconsistent

and erroneous data and the inconsistencies in these datasets raises concerns of data integrity [55]. Shepperd et al. [55] further stated that, these problems hamper the empirical validation and replication of experiments by researchers. The aim was then to provide unified datasets to ensure data integrity and ease the difficulty of empirical validation of experimental results and the replication of experiments. Ten of the cleansed versions of the NASA MDP datasets are considered for this research. These datasets are from various software projects such as science instrumentations, ground data processing, and flight control systems. Table 3.2 below shows the programming language and project or software from which the datasets were created, Table 3.1 shows a summary of the datasets showing number of attributes, number of examples, and number and percentage of positive and negative examples.

The five datasets sourced from the UCI repository are originally multiple class datasets but are modified into binary class problems for the purpose of imbalance learning. The class combinations used in this study are used by other researchers for various research work involving imbalance learning [28], [29]. The last dataset named unbalanced, is an unbalanced dataset included in the WEKA sample datasets. Table 3.3 provides brief descriptions of these datasets and a summary of their class distributions shown in the last few rows of Table 3.1.

These NASA MDP and UCI datasets are considered for this study because they are publicly available, so the experiments carried out in this study can be easily verified or replicated by other researchers. Secondly, because the datasets have been widely used in imbalance learning and are also void of problematic attribute values such as implausible and mission values.

Table 3.1: Summary of Datasets

Dataset	Num. of Attributes	Num. of Instances	Positive Instances		Negative Instances	
			Number	Percentage (%)	Number	Percentage (%)
CM1	38	344	42	12.21	302	87.79
KC1	22	2096	325	15.51	1771	84.49
KC3	40	200	36	18.00	164	82.00
MC1	39	9277	68	0.73	9209	99.27
MC2	40	127	44	34.65	83	65.35
MW1	38	264	27	10.23	237	89.77
PC1	38	759	61	8.04	698	91.96
PC2	37	1585	16	1.01	1569	98.99
PC3	38	1125	140	12.44	985	87.56
PC4	38	1399	178	12.72	1221	87.28
Abalone9v18	9	731	42	5.75	689	94.25
Abalone19	9	4177	32	0.766	4145	99.24
Ecoli4	8	336	20	5.95	316	94.05
Glass2	10	214	17	7.94	197	92.06
Yeast2v8	9	264	20	7.58	244	92.42
unbalanced	33	856	12	1.40	844	98.60

Table 3.2: Programming Languages and Projects NASA Dataset were created from

Dataset	Language	Project
CM1	C	Is a NASA spacecraft instrument for data collection and processing
KC1	C++	The KC1 project is a single Computer Software Configuration Item (CSCI) within a large ground system
KC3	Java	is an application that collects, processes, and delivers satellite metadata
MC1	(C) C++	Is a system for a combustion experiment of a space shuttle
MC2	C++	Is a video guidance system
MW1	C	Is an application from a zero gravity combustion experiment
PC1	C	is a flight software for an earth orbiting satellite
PC2	C	Is from a dynamic simulator for attitude control systems
PC3	C	Is from a flight software for earth orbiting satellite
PC4	C	Is from a flight software for earth orbiting satellite

Table 3.3: Description of UCI Datasets

Dataset	Description
Abalone9v18	Used to predict the age of abalone, obtained by cutting the shell and counting the number of rings. Abalone with 9 rings and 18 rings are used.
Abalone19	Used to predict the age of abalone, obtained by cutting the shell and counting the number of rings. Abalone with 19 rings is used against all other number of rings.
Ecoli4	Collected from different sources including GenProtEC and SWISSPROT. Used to predict localization site of protein
Glass2	From the composition and characteristics of glass, meant for predicting types of glass.
Yeast2v8	Contains data of protein localization sites in *yeast* bacteria, based on several bio-statistical tests
unbalanced	

3.1.2 Data Sampling Techniques

Besides the proposed undersampling technique, five (5) other data sampling techniques are considered in this study. These techniques are used as the basis to assess the efficiency of the proposed technique. They include, cluster-based undersampling, random undersampling, random oversampling, synthetic minority oversampling and one-sided selection. For the purpose of this study the cluster-based undersampling, random undersampling, random oversampling, and one-sided selection techniques are implemented within the framework of the WEKA machine learning tool using NetBeans java IDE. To ensure the implemented algorithms conform to standards of the WEKA tool, code templates and guidelines provided by [48], [60] are followed. The java archive (jar) file of synthetic minority oversampling technique implementation created by Chawla et al. [17] and made available online for academic use is used. It is worth noting that, SMOTE is one of the sampling algorithms included in the stable versions of WEKA but because the developer version of WEKA is used for this study, SMOTE is downloaded as a separate file and incorporated into the learning framework.

Random Sampling

The main random sampling techniques considered in this study are the random undersampling and oversampling techniques. Since the right proportions of majority instances to undersample or the proportion of minority class instance to oversample is not known a prior, several sampling parameters are used and the best chosen. Six undersampling parameters are considered for random undersampling in this study, 5, 10, 25, 50, 75, and 90. This implies that, if undersampling is done with parameter 25, then only 25% of the majority class instances is retained that is 75% of the majority class instances are discarded. Also, random oversampling is done with seven parameters, 50, 100, 200, 300, 500, 750, and 1000. When oversampling is done with a parameter 100, it means that the number of minority class instances is increased by 100% or in other words the number of minority class instances is double. The sampling parameter that yield the best results per each dataset and classification algorithm is considered and the performance of the given classification algorithm recorded for further analysis. These parameters were used by some previous studies including [11], and [20].

Cluster-Based undersampling technique (CBU)

The cluster-based undersampling technique proposed by Das et al. [32] is aim at solving the class imbalance problem by discarding majority instances in overlap regions of the datasets. The main idea is that clustering the dataset into some clusters and discarding majority class instances from clusters where the ratio of minority class instances in the cluster to the total number of instances in the cluster is greater than or equal to an empirically determined threshold, will minimise the class overlap problem while also reducing the number of majority class instances, hence minimising the

imbalance in classes. Twelve clustering parameters ($k = \{5, 10, 15, 30, 50, 55, 70, 75, 100, 110, 120, and\ 130\}$) are considered in this study. The wide range of clustering parameters is considered because of the variation in the size of the datasets. The experiments for each k are also repeated with ten different threshold values ($\tau = \{0.1, 0.09, 0.08, 0.07, 0.06, 0.05, 0.04, 0.03, 0.02, 0.01\}$). The highest classification performance for the range of τ values for each k is recorded and the highest results for the range of k values is considered the performance of the respective classification algorithm for the given dataset.

Synthetic Minority Oversampling Technique

SMOTE requires the user to specify the number of nearest neighbours, percentage of minority class instances to oversample and a random seed. The default number of nearest neighbours, $k = 5$ and a random seed of 10 is used for all experiments. The sampling parameters outlined above for random oversampling are also used for SMOTE and the highest classification performance recorded by the respective classification algorithm across the range of parameters is considered the performance of the algorithm for the dataset under consideration.

One-Sided Selection

The one-sided selection sampling technique only requires a random seed for the selection of the majority class instance that is added to the minority instances for training the 1-NN rule. A random seed of 10 is used in this study.

3.1.3 Classification Algorithms

Two classification algorithms, C4.5 decision tree algorithm [34] implemented in WEKA as J48 decision tree and OneR [36] a rule based learner are used in this study. These algorithms are chosen because they are widely used and also represent different approaches to learning.

C4.5 Decision tree

J48 is the WEKA version of the C4.5 Decision tree, WEKA requires the user to specify among others whether to prune the tree or not, the confidence factor for pruning, and the number of instances per leave. The default values of the J48 learner specified in the WEKA machine learning tool are used for all experiments unless otherwise stated. That is, the experiments are carried out with a confidence factor of 0.25 and a minimum of 2 instances per leaf node.

OneR (1R)

The OneR implementation in WEKA has only two options that the user may specify; the debug option which tell the classifier to output additional information to the console, this option is not important in this study and therefore set to false (default value); the other option is the minimum bucket size which specifies the minimum number of instances per rule. The default bucket size, that is, 6 instances per rule is used in this study.

3.1.4 Tools used

The tools used for this study include the WEKA machine learning tool [48] and NetBeans java IDE. WEKA is a java based machine learning tool developed by the

University of Waikato and consist of a collection of implementation of machine learning algorithms. The algorithms include those for data pre-processing, clustering, regression, classification and association rule mining. It also provides functions for data visualization and is freely available for download for academic purposes. The tool can either by used from the graphical user interface or by referencing its java archive (jar) file from a java IDE. Using the tool from a java IDE gives the experimenter addition advantage as he/she has complete control over the learning process and can therefore implement and use other algorithms that are not part of the WEKA tool. The second method is adapted for the experiments in this research since some of the data pre-processing methods needed are not part of the tool.

3.1.5 Performance Metrics

The two main performance metrics used to measure the classification performance of the algorithms on each dataset are the Area Under the receiver-operating characteristic Curve (AUC) and the Geometric Mean (G-Mean). The WEKA machine learning tool provides a function for the calculation of AUC and this function is used for determining the AUC of models built in this study. The average AUC value of the 10x10 way cross validation process is determined and used as the final performance measure of each classification algorithm. The calculation of G-Mean is not provided in WEKA and is therefore calculated using a function added to the learning process by the user.

3.1.6 Analysis of Results

Given the nature of this study, there is the need to empirically examine the results obtain using the various sampling techniques and classification algorithms to establish

if the performance of the proposed technique is the same or significantly better or worse than the other sampling techniques and the models built without sampling the training data. This study considers a single factor Analysis of Variance (ANOVA) model [56]. The factor is sampling technique with seven levels (NONE, RUS, ROS, SMOTE, CBU, OSS, and CUST). The ANOVA model is used to test the hypothesis that the effects of all levels of the experimental factor are the same against the alternative hypothesis that at least one level of the factor is significantly different (i.e., the AUC or G-Mean results of all sampling techniques are equal or at least the results of one technique is significantly different from the rest). The single-factor analysis of variance model is given as:

$$y_{ij} = \mu + \tau_i + \in_{ij} \begin{cases} i=1,2,...,a \\ j=1,2,...,n \end{cases} \tag{3.1}$$

where y_{ij} is the ij^{th} observation, μ is the overall mean, τ_i is i^{th} treatment, and $\in_{ij}$ is a random error component that incorporate all sources of variability in the experiments.

If the null hypothesis is rejected or in other words the alternative hypothesis is accepted, then further analysis is done using the Tukey's Honestly Significant Difference (HSD) test [56] to determine which level of the factor (sampling technique) is significantly different. The HSD procedure makes use of the distribution of the Studentized range statistic:

$$q = \frac{\bar{y}_{\max} - \bar{y}_{\min}}{\sqrt{MS_E / n}} \tag{3.2}$$

where $\bar{y}_{\max}$ is the largest and $\bar{y}_{\min}$ the smallest mean out of a group of p sample means, MS_E the Mean Squared error, and n the sample size. For equal sample sizes, the HSD test declares two means significantly different if the absolute value of their sample difference exceeds:

$$T_\alpha = q_\alpha(a, f)\sqrt{\frac{MS_E}{n}} \tag{3.3}$$

where a is the number of treatments, and f the degree of freedom associated with the MS_E.

3.1.7 Experimental Method

Five sampling techniques are used to sample the training data prior to training the classification models. In addition the classification models are train with the raw training data (i.e., training data that is not sampled) and this is referred to as NONE.

The models are built and evaluated using the tenfold cross validation process. That is, each dataset is divided into ten equal folds, the stratified tenfold cross validation process is used to ensure that each fold maintains the original class ratio of the datasets (i.e. all the folds contain the same number of majority and minority class examples). Nine of the ten folds are used as training set and the remaining fold is kept as hold out (testing set) and is used to test the performance of the model built on the nine folds. Prior to sampling the training set all instances that are common (seen) to both training and test sets are discarded form the training set.

The process is repeated ten times so that each fold is used once as a test set. In each case the sampling techniques are not applied to the hold out data (i.e., the hold out data is not modified in anyway). The tenfold cross validation process is run ten times and at each run the dataset is randomized. This repetition eliminates any biasing that might be introduced by the sampling or the stratification process.

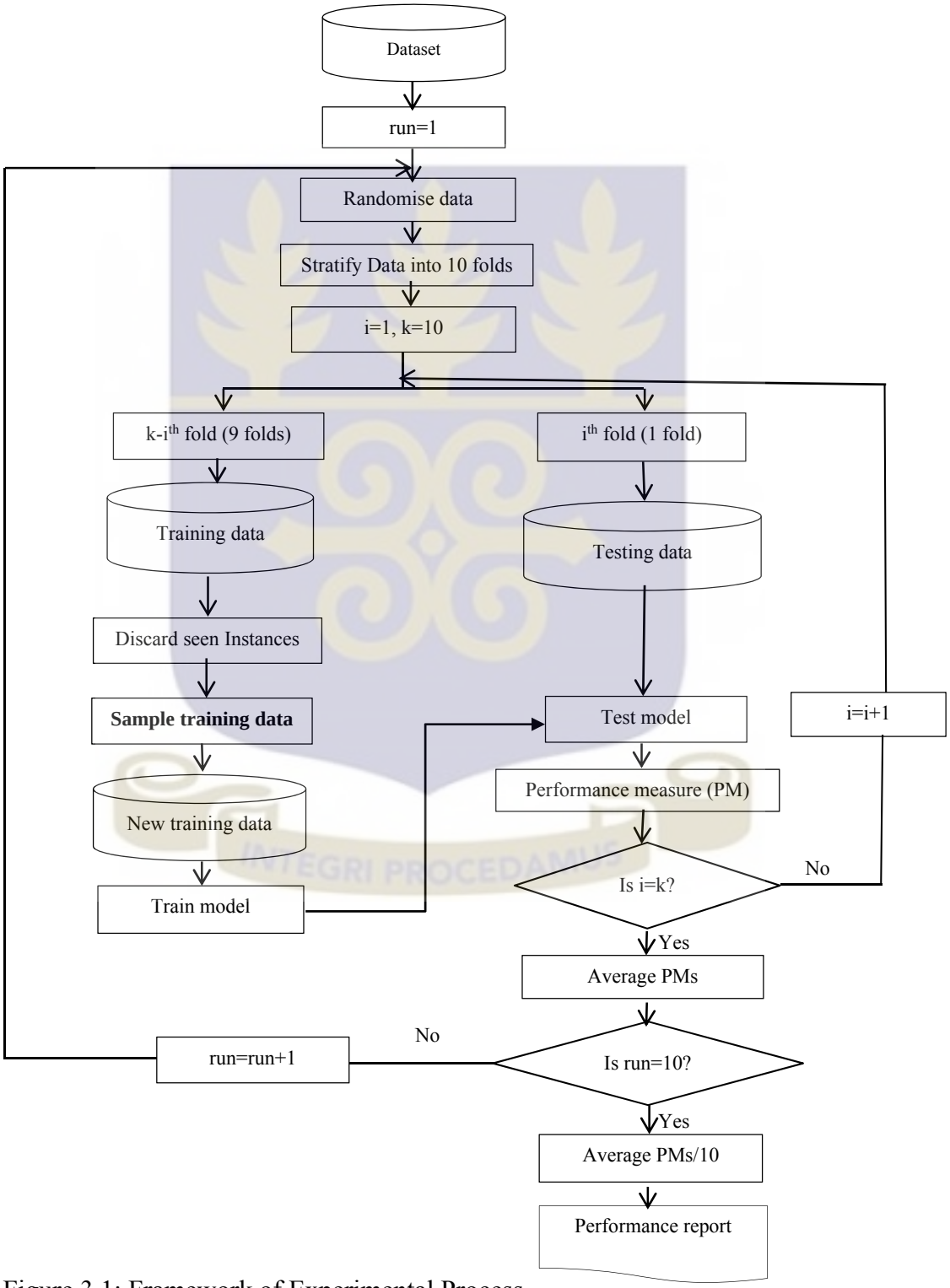

Figure 3.1: Framework of Experimental Process

The classification performance of the models on each holdout data of the tenfold cross validation process is summed and the average calculated and used as the performance of the model. Also, since the cross validation process is run ten times, the average results of the ten runs of cross validation process are further average and this serves as the final results of the classification model under consideration. The experimental process illustrated using a flow chart is shown in Figure 3.1 above.

CHAPTER FOUR

DESIGN OF CLUSTER UNDERSAMPLING TECHNIQUE

4.0 Introduction

The chapter gives an account of the design of the proposed technique, Cluster Undersampling Technique (CUST).

4.1 Design of CUST

CUST seeks to improve the performance of classification algorithms when learning from imbalance datasets by undersampling the majority class instances in the training set taking into account the presence of inconsistent instances, repeated instances, and outliers in the majority class. The hypothesis therefore is that eliminating inconsistent, repeated instances and outliers in the course of undersampling will yield a subsample of the majority class that is void of problematic instances that have the potential of negatively affecting a classifiers performance.

The CUST undersampling process is in two stages, the first stage is the removal of inconsistent instances from the majority class, and the second is the main undersampling process where outliers and repeated instances are removed along with other instances.

The first part of the process uses a technique derived from Tomek links [57] used in OSS to remove noise and borderline instances. Tomek links is based on the closeness of two instances belonging to different classes. Given two instances I_j and I_k belonging to different classes, and the distance between them $dist(I_j, I_k)$, if there is no other instances I_l such that $dist(I_j, I_l) < dist(I_j, I_k)$ then the pair (I_j, I_k) is called a

Tomek link. If two instances form a Tomek link, one of them is either noisy or both are on or near the class boundary.

This implies that for the instances to form a Tomek link, $dist(I_j, I_k)$ must not necessarily be equal to zero, however, the criterion used in the proposed technique, contrary to that used in OSS, only allows a majority class instance to be discarded if and only if $dist(I_j, I_k) = 0$. That is, the instances must have exactly the same attributes but different class labels. Thus, only inconsistent or noisy majority class instances are removed. The minority instances are not discarded because doing so will further worsen the class imbalance problem.

The call of the function that performs the removal of inconsistent instances is optional, in that, if the experimenter is certain that the dataset he/she is presenting do not contain inconsistent instances the option can be set to false and the process begins from the second stage, this is to avoid unnecessary wastage of computational resources and time.

After the removal of inconsistent instances from the majority class or the call of the function been set to false, the second phase of undersampling process starts with the remaining data passed through from the first stage. The second phase is based on the idea of using clustering to identify outliers in datasets [6], [58], and [59]. The main assumption of this technique is that, clustering the majority class instances in the training set into k clusters, instances that portray similar characteristics would fall naturally into the same clusters with the outliers either forming smaller clusters or single instance clusters away from the main clusters.

Clustering of instances is done using the simple k-means clustering algorithm implemented in WEKA. The default number of clusters in CUST is set to five (5) but the experimenter has the option to change it to any value, k that is desired. The number of instances to sample from each cluster is determined using equation 4.1 below:

$$Maj^i = r \times \frac{MI}{MA} \times MC^i \;; 1 \leq i \leq k,\ MA \neq 0 \tag{4.1}$$

where Maj^i is the number of instances to sample from cluster i calculated to the nearest whole number, MI is the total number of minority class instances in the training set, MA is the total number of majority class instance after inconsistent instances are removed in the first stage, MC^i is the number of instances in cluster i, and r is a parameter that specifies the ratio of majority to minority instances in the final training set. If $r = 0.75$ the sampled majority instances would be $\approx 25\%$ less than the minority instances, if $r = 1$, the ratio is $\approx 1:1$ meaning they are approximately equal in number, and if $r = 2$, the ratio would be $\approx 2:1$, meaning the majority instances sampled are twice as much as the minority instances. The default value of r is set to 1 in the algorithm but, since the best ratio of majority/minority instances that yields best classifier performance is not always 1:1 [11], the experimenter has the option to vary the value of r until the performance of the classifier is maximized. The optimum value of r depends on the number of clusters, dataset, and classification algorithm at hand.

Equation 4.1 by its nature discriminates against undersampling from clusters that have smaller number of instances, thereby limiting the number of outliers, which potentially reside in these smaller clusters from been included in the sampled data.

After determining the number of instances to sample from a cluster using equation 4.1, the instances in the cluster are randomized using a random seed that is provided by the experimenter and the instance that comes first is selected. All duplicates of this instance in the cluster are discard and the remaining instances randomized. The instance that comes first is selected again and all its duplicates discarded. This process is repeated until the required number of instances to be sampled from the cluster is realized or there are no instances left in the cluster. Discarding all the occurrence of an instance in a cluster after it has been selected is aimed at avoiding duplicates of the instance from been selected, thereby eliminating the occurrence of repeated majority class instances in the final training set. This process is carried out in all the clusters and the selected instances put together to form the set of majority class instances in the final training set. The process used to sample instances from each cluster is much like that of random undersampling but it performs an additional task of discarding all duplicates of an instance after it has been selected. The algorithm of CUST is shown in table 4.1 below.

Table 4.1: Algorithm of CUST

1. Separate the training data into minority and majority groups;
   Min containing minority instances, and
   Maj containing majority instances
   let *MI*=number of instances in Min, and
   MA= number of instances in Maj

2. removeInconsistent (True/false) (* function to remove inconsistent instances*)
   If $removeInconsistent = True$
   for $i \leftarrow 1\ to\ MI$
   for $j \leftarrow 1\ to\ MA$

if $EuclideanDistance(Min_i,\ Maj_j) == 0$ then remove Maj_j from Maj
endif
endfor
endfor
$Maj = remaining\ Maj;\ MA = number\ of\ instances\ in\ Maj$
endif
return (*end of removeInconsistent*)

3. Cluster Majority instances in *Maj* into k clusters using k-means algorithm

4 for $i \leftarrow 1\ to\ k$ (* extract data in cluster i *)
$MC^i = number\ of\ instances\ in\ cluster\ i$
Compute: $Maj^i = r \times \frac{MI}{MA} \times MC^i$ to the nearest whole number (*number of instances to sample from cluster i*)
While $Maj^i > 0$
If $MC^i = 0$
Break
endif

rand: randomize instances in cluster i

push $Instance_1$ (* Add first instance to sampled data*)

for $j \leftarrow 1\ to\ MC^i - 1$ (*Remove all repetition of selected instance from cluster*)
if $EuclideanDistance(Instance_1,\ Instance_{j+1}) == 0$
remove $Instance_{j+1}$ from cluster
endif
endfor
$MC^i = number\ of\ instances\ remaining\ in\ cluster\ i$
$Maj^i - -$
endwhile
endfor
return(*end of doCluster*)
End of algorithm

CHAPTER FIVE

IMPLEMENTATION AND TESTING OF CUST

5.0 Introduction

This chapter presents the implementation of CUST as well as a discussion of the results from the various testing experiments carried out.

5.1 Implementation of CUST

The detail source code for the implementation of the cluster undersampling technique is provided in the appendices. This code listing shows the complete source code of the implementation that demonstrates the design of CUST. This includes the importation of WEKA libraries through to the implementation of the CUST function.

As stated in chapter three, the cluster undersampling technique is implemented using NetBeans Java IDE within the framework of the WEKA machine learning tool. The completed CUST program is packaged into a jar file so that its performance could be verified and tested using the WEKA tool.

The CUST carries out two main tasks in the course of undersampling. The first task is to remove inconsistent majority class instances from the training set, and the second is to cluster the remaining majority class instances and undersample a defined number of instances from each cluster depending on its size. To carry out the first task, for-loops are used to step through the majority class instances comparing each minority class instance to each majority class instance and if the Euclidean distance between them is zero (0), that is, the minority class instance and majority class instance have exactly

the same attribute values, the majority class instance is discarded. The extract of code that does this is shown below:

```
EuclideanDistance dist = new EuclideanDistance(getInputFormat());
if(m_RemoveInconsistent){
   for(int i=0; i<pTrain.numInstances(); i++){
      for(int l=0; l<nTrain.numInstances(); l++){
      Double dst = dist.distance(pTrain.instance(i), nTrain.instance(l));
         if(dst.equals(0.00)){
           nTrain.remove(l);
         }
      }
   }
}
```

After discarding the inconsistent majority class instances, the remaining data is then pushed to the second stage for clustering and sampling. The library of k-means clustering algorithm implemented in WEKA is used for clustering the data. In the clustering process an additional attribute is added to the data, this additional attribute shows the cluster to which an instance has been assigned so as to facilitate easy sorting of the instances into the respective clusters for undersampling. After clustering the data, a for-loop is used to extract the instances for each cluster at a time, after which the additional attribute that define the cluster is removed, and the number of instances to sample from the cluster calculated. A while-loop is then used to control the undersampling process. At each recursion of the while-loop the data is randomized and the first instance selected and all repetitions of the selected instance removed from the data. An extract of the code that does this is shown below:

```
while(ms>0){
   int num= newnTrain.numInstances();
```

```
  if (num==0){
      break;
  }
 Instances samData= newnTrain.stringFreeStructure();

 Random rand = new Random(getRandomSeed());
 newnTrain.randomize(rand);

 push((Instance) newnTrain.instance(0).copy());  //Add instance to sampled data

//removes all repetitions of selected instance
  for(int ni=0; ni< newnTrain.numInstances()-1; ni++){
    Double dst = dist.distance(newnTrain.instance(0), newnTrain.instance(ni+1));
      if(dst!=0.00){
         samData.add(newnTrain.instance(ni+1));
      }
  }
newnTrain =new Instances(samData);
ms--;
} //end while
```

After all instances have been processed by the filter, the sampled majority class instances and minority class instances are then made available for collection and the input data cleared from memory.

5.2 Testing of CUST

After designing and implementing an algorithm of this nature, its true efficiency can only be assessed through thorough testing and experimentation. As indicated in section 3.1 of chapter three, the main test-bed in this study is the WEKA machine learning tool. Ten NASA MDP defect datasets and six other datasets from the UCI repository are used to examine the extent to which CUST can improve the performance of commonly used machine learning algorithms when learning from imbalance datasets. The number of clusters specified in section 3.1.2 of chapter three is also used to examine CUST with $r = \{1.0, 1.25, 1.50, 1.75, 2.0\ 2.5, 3, 10, 25, 50\}$.

The WEKA tool provides two options through which none-core algorithms such as CUST can be incorporated into it. The first option is adding the jar file of the algorithm to the CLASSPATH of the user's computer from which WEKA can automatically detect, load and make it available for usage from its GUI [60]. The second option is adding the jar file of WEKA and the algorithm to the reference library of any java IDE and performing the experiments from there. The second option is adapted for this study.

For the purpose of illustration, Figure 5.1 and 5.2 below show screen shots of CUST in the WEKA GUI, Figure 5.1 shows CUST been added to the list of supervised instance filters and figure 5.2 shows CUST interface that contains the fields where the various options can be set by the user.

To use CUST in a java IDE, which is the method adopted for testing it, the procedure for using any other supervised instance filter specified in [60] is followed. That is, importing the needed libraries, instantiating the filter, setting the options and applying the filter to the training data. Remember that the filter is applied only to the training data in this study. The code extract below shows how the filter is setup in a java IDE.

```
CUST cust=new CUST ();
cust.setClassValue ("1");
cust.setRemoveInconsistent ("1");
cust.setRatio (1);
cust.setnumClusters (5);
cust.setRandomSeed (1);
cust.setInputFormat (Data);
Instances filteredData=CUST.useFilter (Data, cust);
```

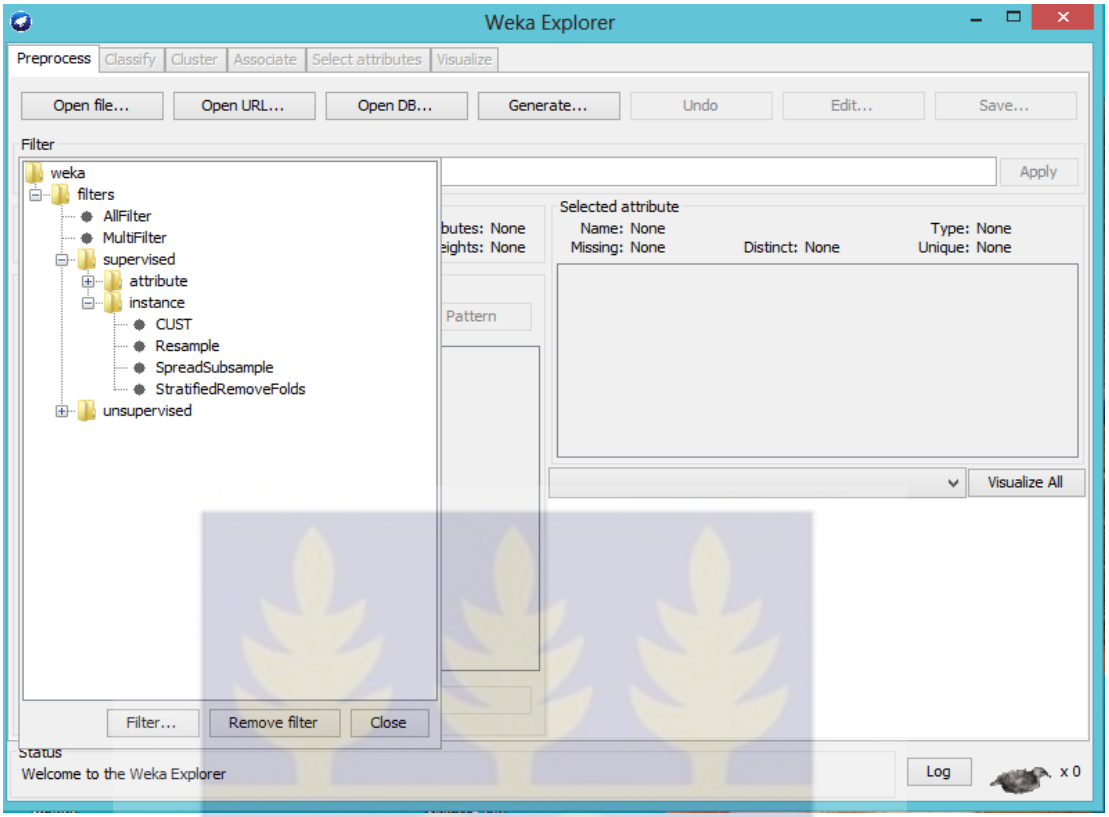

Figure 5.1: Window showing CUST been added to Supervise Instance Filters in WEKA GUI

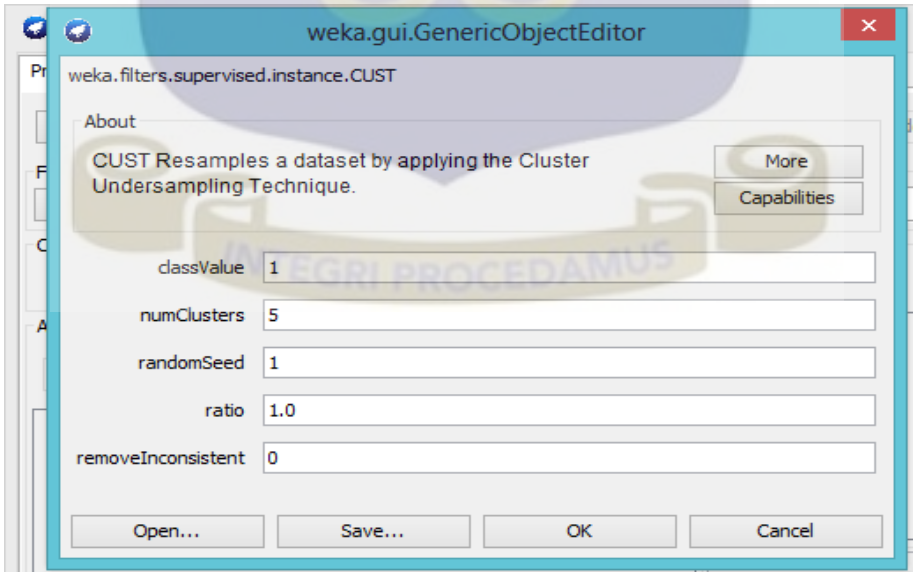

Figure 5.2: CUST Interface showing Options in WEKA GUI

5.3 Results and Discussion

This section presents the results of the experiments described in chapter three. First the performance of the classification algorithms when CUST is used to undersample the training data is compared to the performance when the training data is not sampled prior to training the algorithms (NONE) and subsequently the results of CUST is compared to the results of the other sampling techniques. The last section presents an ANOVA analysis of the results to establish if there is significant difference in the results of the sampling techniques. The performance of the algorithms using the AUC and G-Mean performance metrics are presented and discussed.

These results are shown in table 5.1 through table 5.4 below. Table 5.1 and table 5.2 show the results of C4.5 algorithm and table 5.3 and table 5.4 show the results of the OneR algorithm in terms of AUC and G-Mean metrics, respectively. The first column in each of these tables indicates the datasets, the first row the sampling techniques and the entries under are the performance of the respective classification algorithms given the sampling techniques and corresponding datasets. The result of the sampling technique that produce the highest performance for each dataset is bold-faced in the tables. The results using other metrics such as true positive rate (recall), false positive rate, precision, F-measure, and overall accuracy, which are intermittently referenced in the discussions are shown in the appendices.

Table 5.1: Results of C4.5 using AUC Performance Measure

Dataset	NONE	CUST	RUS	CBU	SMOTE	ROS	OSS
Abalone9v18	0.623	**0.758**	0.665	0.694	0.755	0.677	0.666
Abalone19	0.500	**0.751**	0.561	0.624	0.690	0.606	0.500
Ecoli4	0.810	**0.966**	0.876	0.839	0.920	0.845	0.871
Glass2	0.737	**0.885**	0.763	0.763	0.800	0.718	0.773
Yeast2v8	0.720	**0.931**	0.751	0.766	0.826	0.739	0.738
unbalanced	0.500	**0.827**	0.635	0.628	0.678	0.575	0.487
CM1	0.589	**0.703**	0.654	0.685	0.619	0.612	0.581
KC1	0.670	**0.723**	0.700	0.716	0.693	0.622	0.705
KC3	0.597	**0.733**	0.622	0.677	0.616	0.624	0.615
MC1	0.646	**0.876**	0.756	0.651	0.700	0.679	0.645
MC2	0.602	**0.694**	0.628	0.649	0.669	0.626	0.624
MW1	0.527	**0.708**	0.653	0.697	0.611	0.615	0.652
PC1	0.663	**0.793**	0.751	0.754	0.713	0.684	0.685
PC2	0.501	**0.927**	0.751	0.718	0.615	0.557	0.515
PC3	0.630	**0.741**	0.688	0.735	0.665	0.630	0.646
PC4	0.761	**0.865**	0.827	0.820	0.778	0.758	0.789
Average	**0.630**	***0.805***	**0.705**	**0.714**	**0.709**	**0.660**	**0.656**

Table 5.2: Results of C4.5 using G-Mean Performance Measure

	NONE	CUST	RUS	CBU	SMOTE	ROS	OSS
Abalone9v18	0.320	**0.692**	0.661	0.609	0.598	0.329	0.362
Abalone19	0.000	**0.731**	0.126	0.442	0.058	0.000	0.000
Ecoli4	0.752	**0.940**	0.831	0.794	0.905	0.739	0.805
Glass2	0.317	**0.842**	0.430	0.515	0.407	0.223	0.442
Yeast2v8	0.561	**0.835**	0.675	0.665	0.669	0.533	0.599
unbalanced	0.000	**0.822**	0.386	0.080	0.199	0.171	0.000
CM1	0.318	**0.677**	0.620	0.650	0.536	0.386	0.423
KC1	0.518	0.675	0.680	**0.687**	0.593	0.517	0.657
KC3	0.430	**0.678**	0.584	0.652	0.558	0.441	0.512
MC1	0.291	**0.849**	0.689	0.369	0.333	0.320	0.378
MC2	0.529	**0.656**	0.566	0.592	0.639	0.574	0.562
MW1	0.311	**0.685**	0.561	0.635	0.445	0.396	0.508
PC1	0.377	**0.733**	0.716	0.723	0.605	0.421	0.554
PC2	0.000	**0.894**	0.560	0.530	0.166	0.048	0.010
PC3	0.452	0.677	0.683	**0.725**	0.582	0.486	0.565
PC4	0.683	**0.871**	0.832	0.817	0.763	0.683	0.781
Average	**0.366**	***0.766***	**0.600**	**0.593**	**0.503**	**0.392**	**0.447**

Table 5.3: Results of OneR using AUC Performance Measure

Dataset	NONE	CUST	RUS	CBU	SMOTE	ROS	OSS
Abalone9v18	0.560	**0.738**	0.690	0.707	0.569	0.532	0.574
Abalone19	0.500	**0.710**	0.536	0.540	0.500	0.504	0.500
Ecoli4	0.801	**0.929**	0.885	0.813	0.859	0.771	0.841
Glass2	0.497	0.646	0.589	**0.680**	0.549	0.501	0.490
Yeast2v8	0.769	0.773	0.769	0.770	0.769	0.650	**0.821**
unbalanced	0.500	**0.784**	0.593	0.770	0.500	0.541	0.500
CM1	0.517	0.667	0.636	**0.673**	0.541	0.573	0.509
KC1	0.572	0.689	0.682	**0.713**	0.658	0.661	0.655
KC3	0.523	**0.650**	0.574	0.611	0.615	0.564	0.583
MC1	0.554	**0.833**	0.683	0.604	0.561	0.594	0.556
MC2	0.557	**0.629**	0.523	0.569	0.625	0.561	0.588
MW1	0.510	**0.712**	0.646	0.697	0.654	0.550	0.662
PC1	0.556	**0.744**	0.693	0.698	0.554	0.607	0.573
PC2	0.499	**0.894**	0.692	0.649	0.518	0.543	0.499
PC3	0.539	**0.736**	0.675	0.723	0.621	0.570	0.559
PC4	0.623	**0.831**	0.827	0.749	0.750	0.592	0.639
Average	**0.567**	***0.748***	**0.668**	**0.686**	**0.615**	**0.582**	**0.597**

Table 5.4: Results of OneR using G-Mean Performance Measure

Datasets	NONE	CUST	RUS	CBU	SMOTE	ROS	OSS
Abalone9v18	0.234	**0.730**	0.675	0.682	0.240	0.135	0.279
Abalone19	0.000	**0.698**	0.178	0.267	0.000	0.016	0.000
Ecoli4	0.714	**0.924**	0.859	0.733	0.839	0.652	0.785
Glass2	0.000	**0.613**	0.488	0.593	0.349	0.021	0.000
Yeast2v8	0.670	0.655	0.670	**0.672**	0.649	0.397	0.807
unbalanced	0.000	**0.724**	0.236	0.709	0.000	0.093	0.000
CM1	0.129	**0.654**	0.600	0.637	0.299	0.457	0.145
KC1	0.424	0.685	0.679	**0.701**	0.644	0.659	0.621
KC3	0.192	**0.595**	0.421	0.541	0.438	0.508	0.430
MC1	0.234	**0.827**	0.598	0.385	0.260	0.392	0.242
MC2	0.420	**0.622**	0.447	0.515	0.587	0.501	0.532
MW1	0.093	**0.687**	0.514	0.638	0.514	0.272	0.514
PC1	0.277	**0.738**	0.664	0.687	0.340	0.499	0.337
PC2	0.000	**0.887**	0.510	0.423	0.044	0.139	0.000
PC3	0.252	**0.729**	0.660	0.718	0.541	0.519	0.380
PC4	0.502	**0.817**	0.813	0.732	0.730	0.444	0.538
Average	**0.259**	***0.724***	**0.563**	**0.602**	**0.405**	**0.356**	**0.351**

5.3.1 CUST versus NONE

This section compares the performance of the classification algorithms when CUST is used and when the data is not sampled prior to learning, NONE. This is to establish that CUST can improve the performance of common classification algorithms such as C4.5 decision tree and OneR when learning from imbalance datasets.

Figure 5.3 and figure 5.4 illustrate the performance of C4.5 when CUST is used to undersample the training data versus NONE. Figure 5.3 shows the results in terms of AUC performance metric and figure 5.4 shows that of G-Mean performance metrics across all the datasets. From figures 5.3 it is shown that CUST improved the performance of the C4.5 learner by a minimum increase in AUC value of 0.052 that is, 7.83% performance increase in the KC1 dataset and a maximum of 0.425 constituting an increase of 84.923% in the PC2 dataset.

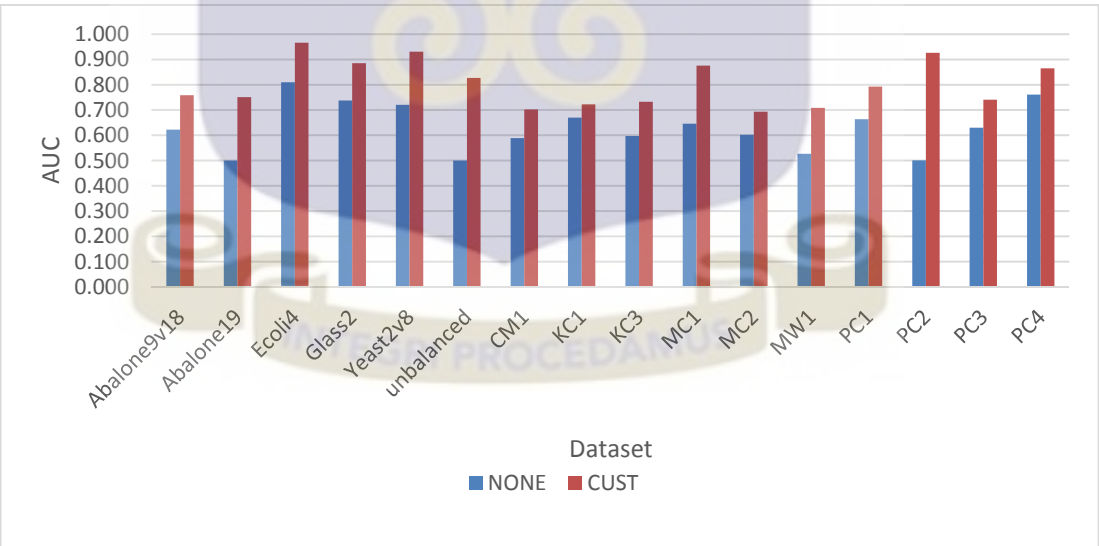

Figure 5.3: Plot of AUC Values of CUST and NONE for C4.5 Learner

Figure 5.4, which illustrates the C4.5 performances using G-Mean showed a minimum increase of 0.128 corresponding to 24.00% increase in performance in MC2

dataset, a higher increase of 0.559 corresponding to 192.09% increase in performance in the MC1 dataset. The performance of the classification algorithm also increased from 0.00 G-Mean in the PC2, abalone19, and unbalanced datasets to 0.894, 0.731, and 0.822, respectively. However, the 0.00 G-mean values in these datasets is as a result of the C4.5 learner's failure on the average to correctly classify a single minority class instance when the training data is not sampled prior to training the learner, hence it recorded an average recall or true positive rate of 0.00. This is however, also true for the learner when the AUC performance metric is used, that is, the 0.501, 0.500, and 0.500 for PC2, abalone19, and unbalanced datasets, respectively, in figure 5.3 above for NONE are obtained with 0.00 recall.

In a summary, the results showed that CUST improved the classification performance of the C4.5 decision tree learner across all the sixteen datasets considered and the improvement is higher in the datasets that are most skewed (PC2, MC1, abalone19, and unbalanced).

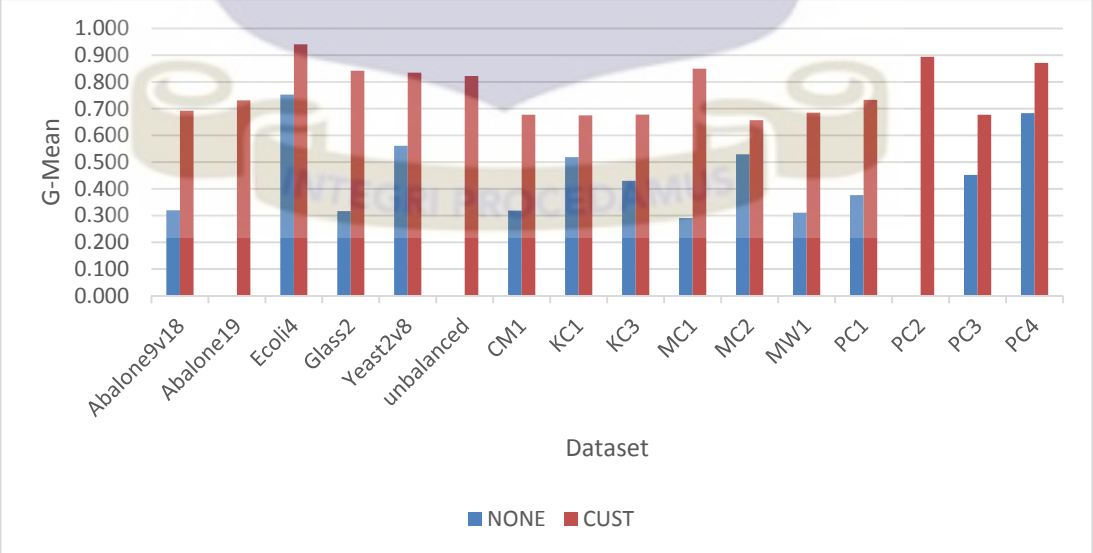

Figure 5.4: Plot of G-Mean Values of CUST and NONE for C4.5 Learner

The performance of the OneR learner for CUST and NONE illustrated in figure 5.5 and figure 5.6 follow similar patterns as the C4.5 learner. However, for the OneR learner as shown in figure 5.5, the minimum difference in performance in AUC between CUST and NONE occurred in the yeast2v8 dataset with a value of 0.004 corresponding to 0.54% increase in performance from 0.769 for NONE to 0.773 for CUST. The maximum difference occurred in the PC2 dataset with a value of 0.399 indicating 78.984% increase in performance from 0.499 for NONE to 0.894 for CUST. In the remaining fourteen datasets CUST recorded an improvement in the learner's performance between 12.83% and 56.70%.

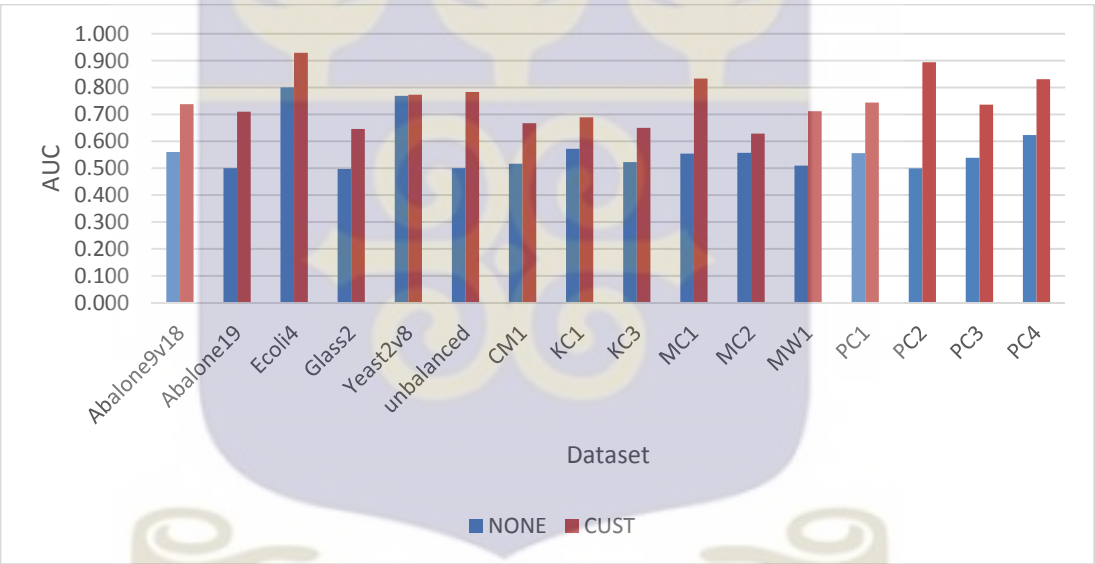

Figure 5.5: Plot of AUC Values of CUST and NONE for OneR Learner

Figure 5.6 illustrates the performance of the OneR learner in terms of G-Mean performance metrics for CUST and NONE. It is clear from figure 5.6 that CUST improved the performance of the learner in most of the datasets. However, the classification algorithm performed better in the yeast2v8 dataset when it is not sampled. That is, the performance of the classification algorithm depreciated from 0.670 for NONE to 0.655 for CUST representation a decline of 2.22% in

performance. The difference in performance using G-Mean in the remaining datasets is however higher as compare to that of AUC. The minimum improvement in performance occurred in the MC2 dataset with a value of 0.202 indicating 47.99% improvement and the maximum again occurring in the PC2, abalone19, unbalanced, and glass2 datasets with difference of 0.887, 0.698, 0.724, and 0.613 respectively. The performance of the learner increased from 0.00 G-Mean measure to the respective values. The learner also recorded significant increase in performance, above 100% in the following datasets, abalone9v18, CM1, KC3, MC2, MW1, and PC3 with 211.75%, 406.98%, 210.92%, 253.40%, 635.57%, and 189.34% increase respectively. These values largely confirm the claim that CUST can significantly improve the performance of the OneR learner when learning from imbalance datasets.

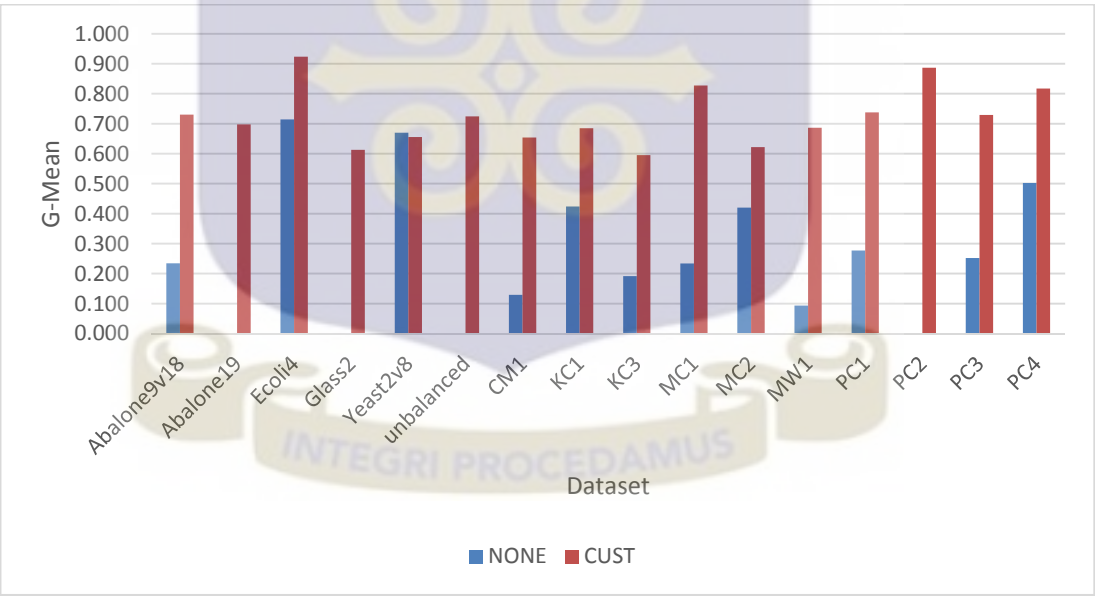

Figure 5.6: Plot of G-Mean Values of CUST and NONE for OneR Learner

It has been established that CUST thus improve the performance of common learners such as C4.5 decision tree and OneR. Nonetheless, a significant conclusion cannot be drawn without comparing the performance of CUST to other sampling techniques.

The subsequent section presents a comparison of the performance of CUST to five well known sampling techniques.

5.3.2 CUST vs other sampling techniques

The performance of CUST is compared to the performance of other three undersampling techniques; RUS, CBU, and OSS and two oversampling techniques; ROS and SMOTE in this section. The third column to the last column of table 5.1 through table 5.4 show the performance in terms of AUC and G-Mean of C4.5 and OneR learners while applying these sampling techniques on each of the datasets prior to training.

Figure 5.7 below illustrates the AUC measures of C4.5 for the various datasets and sampling techniques as in table 5.1. It is clear from this figure that CUST outperformed all the other sampling techniques across all the datasets. However, the difference between CUST and the best performing other sampling technique(s) for some of the datasets is not large. For instance, the difference in performance between CUST and CBU in the KC1 dataset is only 0.007, also 0.006 between CUST and CBU in the PC3 dataset, and 0.003 between CUST and SMOTE in the abalone9v18 dataset.

Comparing how the other five sampling techniques improve the performance of the C4.5 classification algorithm using AUC measure, ROS and OSS performed relatively worse as compared to the other three techniques. SMOTE outperformed all the other sampling techniques in the six datasets from the UCI repository, but only performed marginally better than CBU in the MC1 dataset and RUS in the MC2 dataset. ROS

outperformed CBU in the MC1, and ecoli4 datasets, RUS in the KC3, abalone9v18, and abalone19 datasets, and SMOTE in MW1 and KC3 datasets as shown in figure 5.7. OSS, also performed relatively better than RUS in abalone9v18, glass2, and KC1 datasets, CBU in ecoli4 and glass2 datasets, and SMOTE in KC1, MW1, and PC4 datasets. CBU and RUS performed competitively, with RUS outperforming CBU in only ecoli4, unbalanced, MC1, PC2, and PC4 datasets.

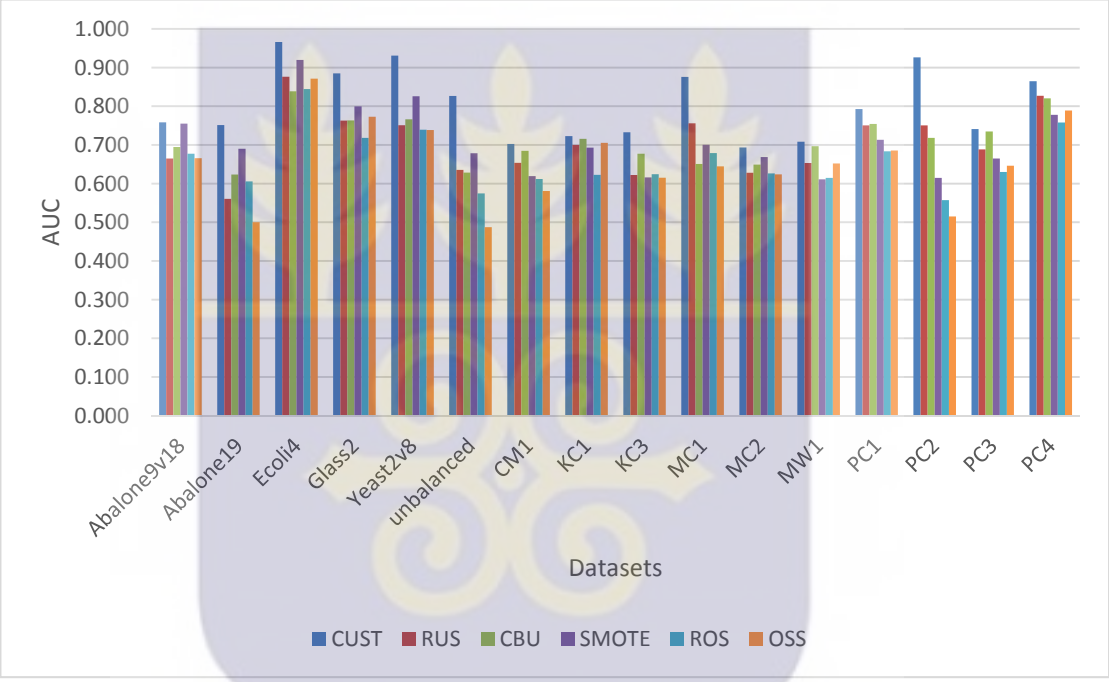

Figure 5.7: Plot of CUST vs other Sampling Techniques for C4.5, AUC

The performance of the C4.5 learner using G-Mean measure is illustrated in figure 4.8 below. From figure 4.8, CBU and RUS outperformed CUST in the KC1 and PC3 datasets, though the difference between RUS and CUST is not large (i.e. 0.005 in KC1 and 0.006 in PC3). Also, the performance of the C4.5 algorithm in the MC1, PC2, abalone19, and unbalanced datasets, indicates that CUST is much robust when handling much skewed datasets as compared to the other sampling techniques as their performance dwindled in these datasets.

Figure 5.8 also shows that, SMOTE performed relatively better than RUS in the MC2 and ecoli4 datasets, CBU in ecoli4, yeast2v8, unbalanced, and MC2 datasets. SMOTE outperformed ROS in all the datasets and is outperformed by OSS in glass2, KC1, MC1, and PC4 datasets. OSS outperformed CBU in only the MC1 and ecoli4 datasets, RUS in glass2 dataset, and is outperformed by ROS in abalone9v18, abalone19, unbalanced, MC2, and PC2 datasets. ROS however, outperformed RUS in the MC2 dataset and CBU in the unbalanced dataset. RUS only outperformed CBU in four (ecoli4, MC1, PC2, and PC4) out of the sixteen datasets considered using G-Mean measure.

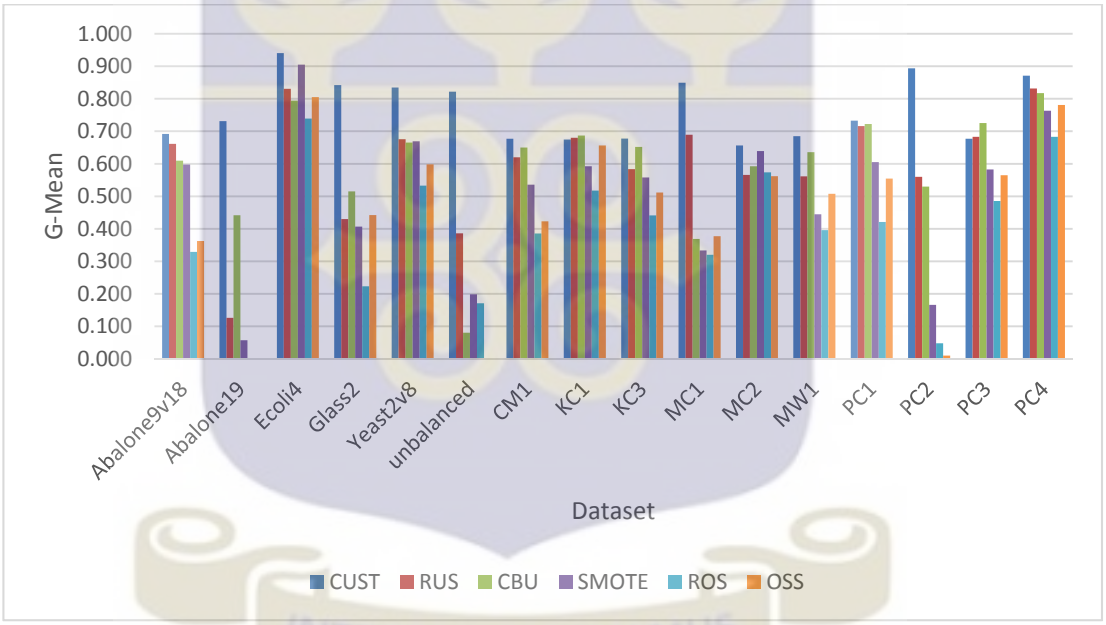

Figure 5.8: Plot of CUST vs other Sampling Techniques for C4.5, G-Mean

The performance of the OneR learner in terms of AUC measure for the various sampling techniques across the sixteen datasets are shown in table 5.3 and is illustrated graphically in figure 5.9 below. From figure 5.9, CUST outperformed all the other sampling techniques in all datasets with the exception of CBU that performed relatively better in glass2, CM1 and KC1 datasets. The difference in

performance between CUST and RUS, CUST and SMOTE in the PC4 and MC2 datasets is only 0.004, and CUST and CBU in the yeast2 dataset is also only 0.003. Also, CBU outperformed CUST in the CM1 dataset marginally by a difference of 0.006.

Amongst the five sampling techniques, ROS, OSS and SMOTE performed relatively worse with the OneR learner using AUC measure. ROS performed marginally better than SMOTE in the abalone19, unbalanced, CM1, KC1, PC1, MW1 and PC2 datasets, and RUS in the MC2 dataset as shown in figure 5.9. OSS on the other hand performed better than ROS in seven datasets, RUS in KC3, MC2, and MW1 datasets, SMOTE in abalone9v18, MC2, MW1, and PC1 datasets, and CBU in MC2 and ecoli4 datasets. It however, outperformed all the techniques in the yeast2v8 dataset. SMOTE outperformed CBU and RUS in MC2 and KC3, and also RUS in MW1 datasets using the AUC measure.

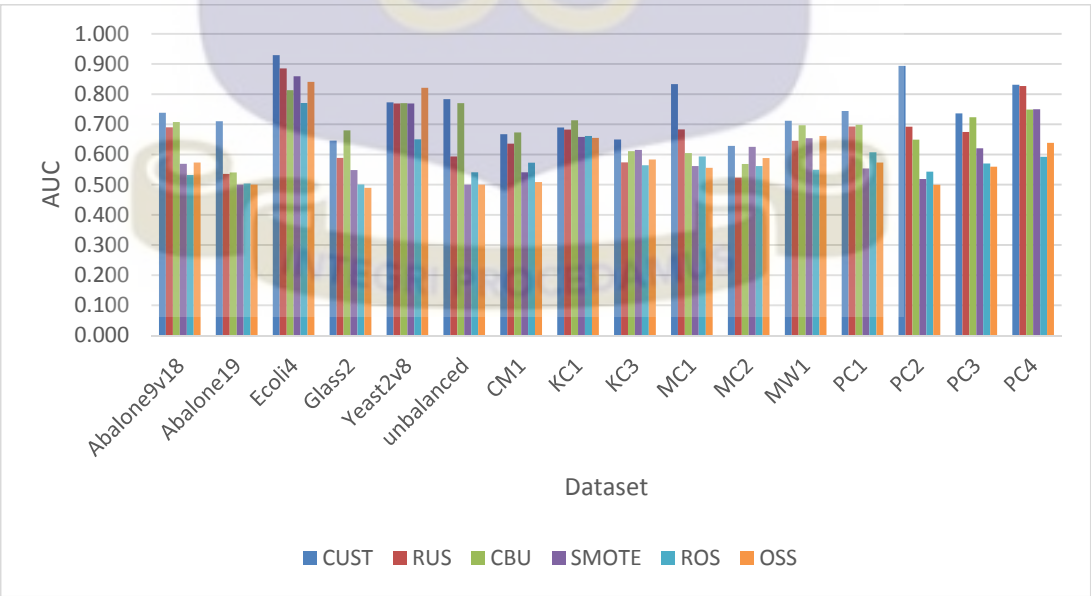

Figure 5.9: Plot of CUST vs other Sampling Techniques for OneR, AUC

The performance of the OneR learner using G-Mean measure shown in figure 5.10 indicates that CUST performed better than the other sampling techniques in fourteen out of the sixteen datasets. CBU outperformed CUST in the remaining KC1 dataset by a difference of 0.024 and OSS outperformed it in the yeast2v8 dataset. Comparatively, the performance of the learner is relatively poor in the abalone19, glass2, unbalanced, MC1 and PC2 datasets when the other sampling techniques are used, particularly ROS, SMOTE, and OSS while that of CUST rather increased. Noticeable are the results of OSS, ROS, and SMOTE, where they recorded approximately 0.00 G-Mean values in these datasets. Also, CBU witness its lowest performance in the abalone19 dataset followed by the MC1 dataset. This to a greater extent confirms the accession made above that, CUST is more robust in improving the performance of learners when learning from much skewed datasets than the other techniques.

Figure 5.10 shows that OSS, ROS, and SMOTE recorded the worse performance in most of the datasets. OSS recorded 0.00 G-Mean values in the abalone19, glass2, unbalanced, and PC2 datasets and only produced G-Mean values above 0.0500 in five out of the sixteen datasets considered. OSS however, recorded the highest G-Mean value among all the sampling techniques in the yeast2v8 dataset, and outperformed RUS and CBU in the MC2 dataset. OSS also outperformed SMOTE in the abalone19v8 dataset. SMOTE on the other hand recorded 0.00 G-Mean values in two (abalone19 and unbalanced) datasets and also a value of 0.044 in the PC2 dataset. SMOTE also performed better than RUS in KC3 and MC2 datasets and CBU in ecoli4 and MC2 datasets. ROS recorded its lowest performance 0.016, 0.021, and 0.093 in abalone19, glass2, and unbalanced datasets respectively. RUS also performed

better than CBU in only four (ecoli4, MC1, PC2, and PC4) datasets using G-Mean measure.

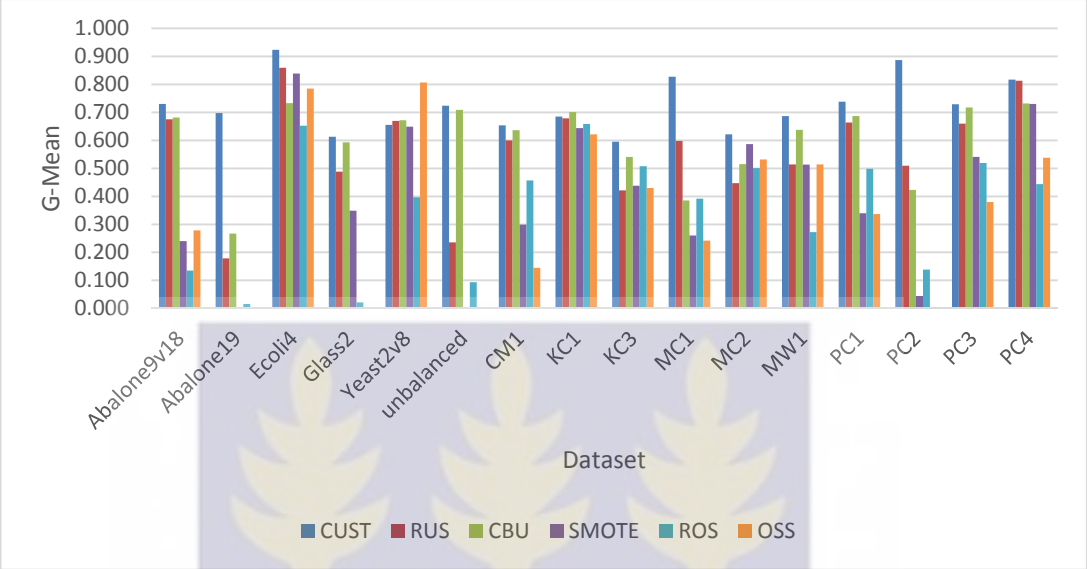

Figure 5.10: Plot of CUST vs other Sampling Techniques for OneR, G-Mean

The above presentation points out that no single sampling technique clearly outperformed the rest in all the test cases considered, though CUST, CBU, and RUS largely dominated the others using both AUC and G-Mean measure for both learners. The boxplots in figure 5.11 through figure 5.14 below further illustrates how different the classification algorithms performed across the datasets given each sampling technique. The plots show the minimum, maximum, and mean performance of the learners using each sampling technique across the datasets.

From figure 5.11 and 5.12, which illustrates the performance of the C4.5 classification algorithms using AUC and G-Mean measures, respectively, it is evident that CUST has the highest average performance followed by CBU, SMOTE, RUS, ROS, OSS, and NONE when AUC measure is used, and RUS, CBU, SMOTE, OSS, ROS, and NONE when G-Mean measure is used. Figure 5.13 and figure 5.14 also illustrate that,

for the OneR learner, CUST on the average again performed relatively better than the other techniques followed by CBU, RUS, SMOTE, OSS, ROS, and NONE using AUC measure, and CBU, RUS, SMOTE, ROS, OSS, and NONE using G-Mean measure.

This however, does not indicate that there is statistical significant difference in the performance of the techniques, further statistical analysis is required to establish whether the difference in classifier performance with respect to the sampling techniques is significant and this is presented in section 5.3.3 below.

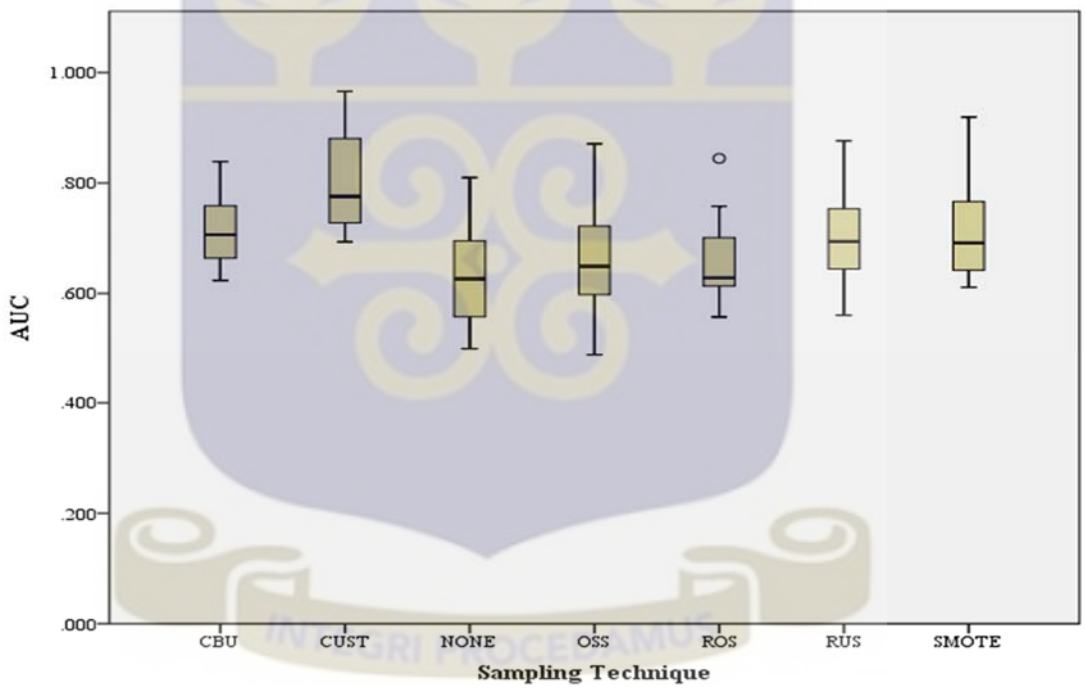

Figure 5.11: Boxplot of C4.5 Performance by Sampling Technique, AUC

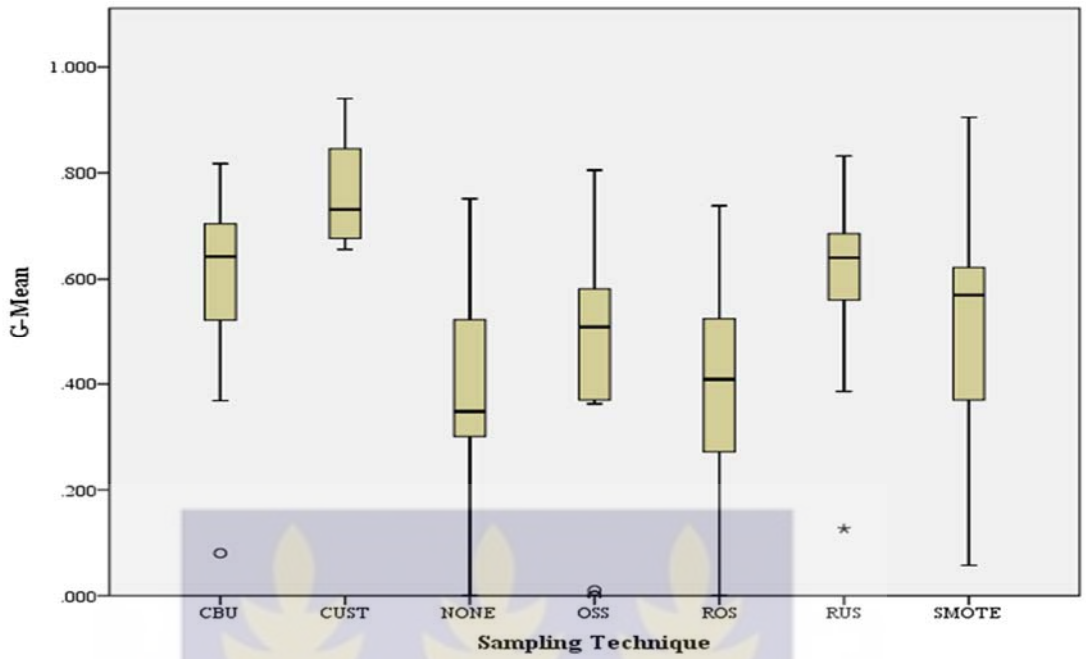

Figure 5.12: Boxplot of C4.5 Performance by Sampling Technique, G-Mean

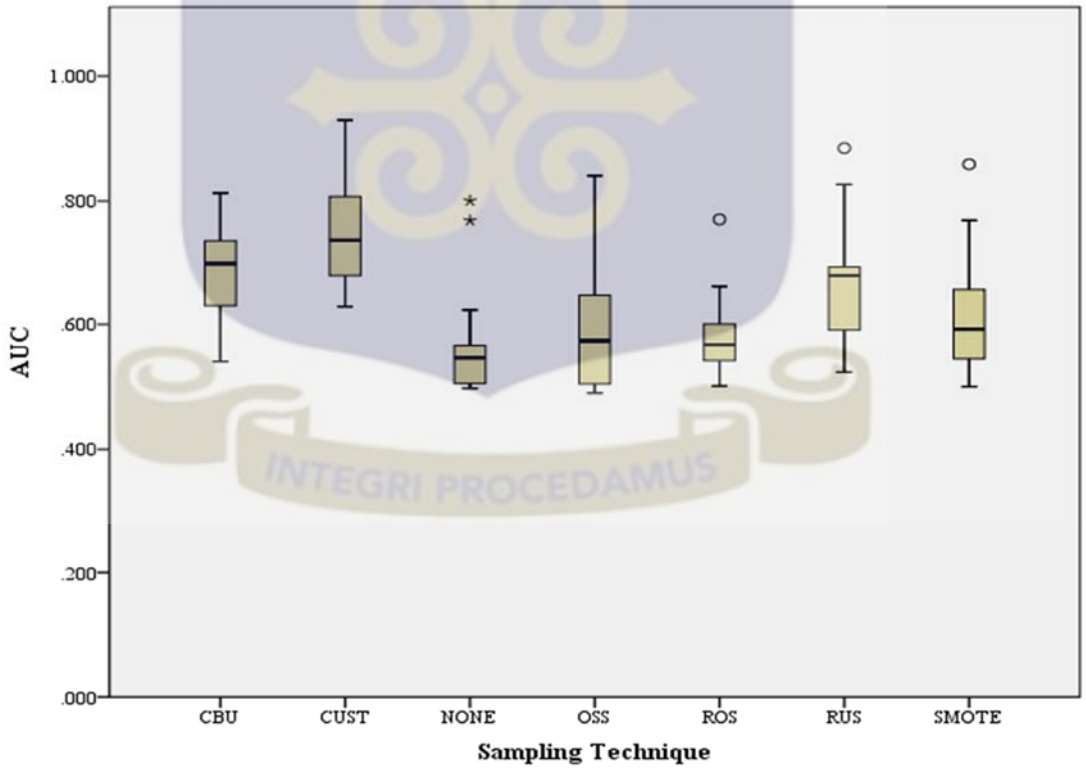

Figure 5.13: Boxplot of OneR Performance by Sampling Technique, AUC

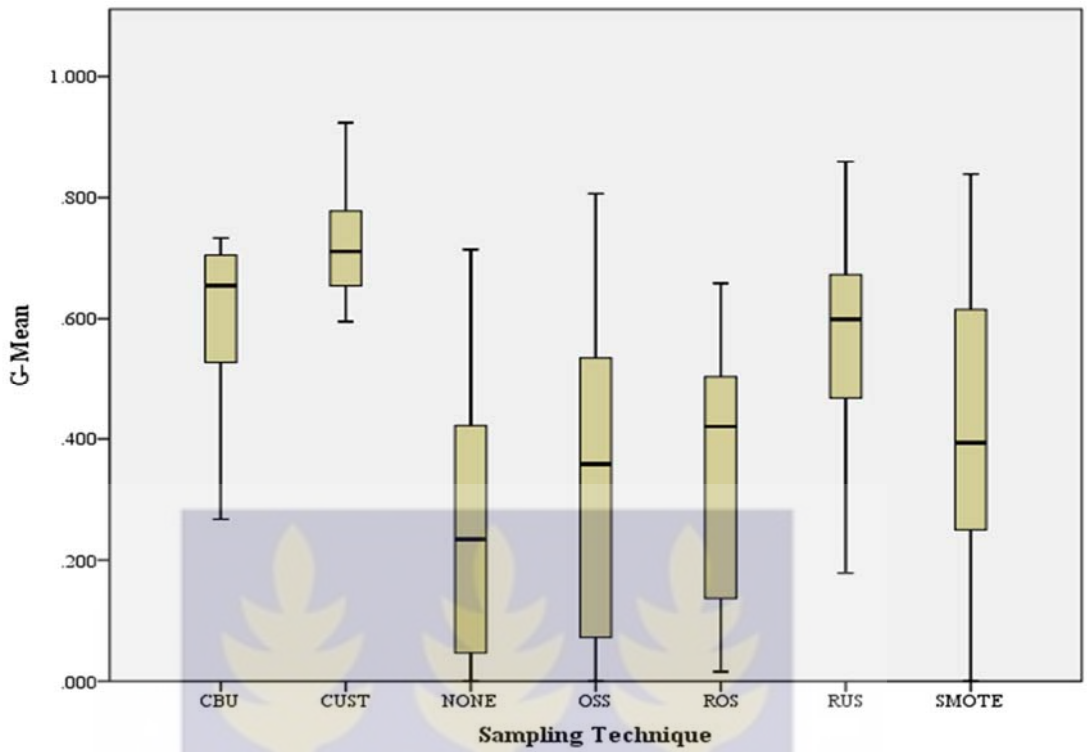

Figure 5.14: Boxplot of OneR Performance by Sampling Technique, G-Mean

5.3.3 ANOVA Analysis

This section presents an examination of the statistical significance of the results presented above in table 5.1, table 5.2, table 5.3, and table 5.4 using the ANOVA model specified in section 3.1.6 of chapter three. The ANOVA models of table 5.1, table 5.2, table 5.3, and table 5.4 are shown in row two, three, four, and five of table 5.5 respectively. Table 5.5 shows the Degrees of freedom (DF), sum of squares (SS), mean squares (MS), f-values, and p values of the models.

Table 5.5: ANOVA Models of C4.5 and OneR Algorithms (AUC and G-Mean)

Experiment	DF	SS	MS	F Value	P
C4.5 AUC	6	0.315612105	0.052602018	6.97	0.0000
C4.5 G-Mean	6	1.88493462	0.31415577	7.83	0.0000
OneR AUC	6	0.408899859	0.068149977	8.19	0.0000
OneR G-Mean	6	2.66498242	0.444163737	10.22	0.0000

The results of the ANOVA models shown in table 5.5 all have p-values equal to 0.000 (p=0.000). This implies that in all four test cases, at 5% confidence level (α=0.05) the choice of a sampling technique has significant impact on the performance (AUC and G-Mean measures) of the C4.5 decision tree and OneR classification algorithms. This means that the performance of at least one of the sampling techniques is significantly different from the rest in each of the test cases. Since the ANOVA models suggests that there is significant difference in the performance of the sampling techniques, the Tukey's Honestly Significance Difference (HSD) test specified in section 3.1.6 of Chapter three is used to carry out a post-hoc pairwise comparison to determine which of the technique(s) resulted in significantly different performance.

The HSD test results of the C4.5 decision tree learner is shown in table 5.6a and 5.6b, and the results of the OneR learner shown in table 5.7a and 5.7b. The first column in each table contains the sampling techniques, the second and third columns show the mean performance and rank of the respective sampling techniques. Table 5.6a and 5.7a show the results based on AUC measure, and table 5.6b and 5.7b show the mean performance and rank of the respective techniques based on the G-Mean measure. If two techniques have the same letter in a column it implies that their performance are not statistically different.

The results in table 5.6a shows that basing the classifier's (C4.5) performance on AUC measure, CUST performed statistically better than RUS, SMOTE, OSS, ROS and NONE but not CBU, that is, there is no statistical difference between the performance of the proposed technique CUST and CBU at 5% confidence level. CBU however, performed statistically the same as the rest of the techniques RUS, SMOTE,

OSS, ROS, and NONE at 5% confidence level. The HSD test based on the G-Mean measure, as in table 5.6b, at 5% confidence level indicates that the performance of the C4.5 classifier is significantly different when CUST is used as compared to SMOTE, ROS, OSS, and NONE. CBU and RUS also improved the classifiers performance significantly with respect to NONE. The performance of CBU, RUS, SMOTE, ROS, and OSS are however not significantly different.

The results in table 5.7a also indicates that at 5% confidence level there is statistical significant difference in the performance of the OneR learner when CUST is used to sample the training data and when SMOTE, ROS, OSS and NONE are used. There is however, no significant difference in the learner's performance when CUST, CBU and RUS are used. The results of CBU is also statistically different from ROS and NONE but not SMOTE and OSS. Although the results of RUS is not significantly different from CUST, it is also not different from the results of SMOTE, OSS, and ROS. From table 5.7b, CUST performed statistically better than SMOTE, ROS, OSS, and NONE. CBU also performed statistically better than ROS, OSS, and NONE. There is no statistical difference in the performance of CUST, RUS, and CBU. The results of RUS is however, not statistically different from that of SMOTE, ROS, and OSS, though it performed the same as CUST and CBU.

In a summary, CUST statistically outperformed the two oversampling techniques SMOTE and ROS, and also OSS and NONE in all cases considered. It also outperformed RUS with the C4.5 classification algorithm when AUC measure is used, but performed statistically the same though with higher average performances in the remaining three cases as this well know undersampling technique, RUS, which is

reported in literature [4], [20], to be a very competitive sampling technique. RUS, though performed statistically the same as CUST, failed in all cases to outperform SMOTE, ROS and OSS, which CUST statistically outperformed. CBU, a lesser known undersampling technique, which has never been studied using these datasets particularly the software defect prediction datasets, also performed statistically the same as CUST and RUS, and also failed to outperform SMOTE in all cases and ROS and OSS in most of the cases. The two oversampling techniques, SMOTE and ROS, and OSS have in all cases performed statistically the same as NONE. This implies that amongst the sampling techniques considered CUST performed relatively better since it statistically outperformed SMOTE, ROS and OSS in most cases which CBU and RUS failed to.

Table 5.6a: HSD results for C4.5, AUC

Tech	AUC	
	Mean	HSD
CUST	0.8051	A
CBU	0.7135	BA
SMOTE	0.7093	B
RUS	0.7051	B
ROS	0.6604	B
OSS	0.6557	B
NONE	0.6298	B

Table 5.6b: HSD results for C4.5, G-Mean

Tech	G-Mean	
	Mean	HSD
CUST	0.7661	A
RUS	0.6000	BA
CBU	0.5928	BA
SMOTE	0.5035	BC
OSS	0.4474	BC
ROS	0.3917	BC
NONE	0.3662	C

Table 5.7a: HSD results for OneR, AUC

Tech	AUC	
	Mean	HSD
CUST	0.7478	A
CBU	0.6854	BA
RUS	0.6683	BAC
SMOTE	0.6152	BDC
OSS	0.5968	BDC
ROS	0.5821	DC
NONE	0.5673	D

Table 5.7b: HSD results for OneR, G-Mean

Tech	G-Mean	
	Mean	HSD
CUST	0.7241	A
CBU	0.6021	BA
RUS	0.5633	BAC
SMOTE	0.4046	BDC
ROS	0.3565	DC
OSS	0.3506	DC
NONE	0.2588	D

5.3.4 Precision and Recall

In class imbalance learning, particularly when data sampling approach is adapted, precision is often traded for higher recall and the minimum allowed precision to the best of the researcher's knowledge depends on the field of application. However, some of the experiments in this study recorded zero precision which is not as a result of higher recall but rather because the classifiers recorded zero true positives after the respective sampling techniques are used on the datasets. This occurred in some of the much skewed datasets when OSS, ROS, and SMOTE are used to sample the training data prior to training. This is not the case with CUST, RUS, and CBU. In the datasets where CUST recorded lower precision certainly resulted to higher recall. For instance, the OneR learner recorded a recall of 1 and a lower precision of 0.048 when CUST is used to sample the training data of PC2 dataset. Though the precision is lower, the results are much better when compare to results reported in previous studies such as Menzies et al. [41], which reported a recall of 0.72 at a precision of 0.0202 using PC2 dataset. This implies that the results reported in this study particularly for CUST is

within acceptable range. Detail results on precision and recall are included in the appendices.

5.3.5 Computational Time of Sampling Techniques

Figure 5.15 below shows the average time in nanoseconds taken by each sampling techniques to sample the training data of the respective datasets. The sampling time illustrated is for only the sampling parameters that resulted in the highest performance of the algorithms. From figure 5.15 it is shown that the sampling time of CUST and CBU is higher than all the other techniques across the datasets, but marginal in the smaller datasets and extremely large in the larger datasets. Thus, the sampling time increases as the size of the datasets increases. This phenomenon can be attributed to the use of the k-means clustering algorithm in CUST and CBU, whose computational complexity depends on the number of clusters, size of the dataset, the dimension of the dataset, and the number of iterations performed during clustering.

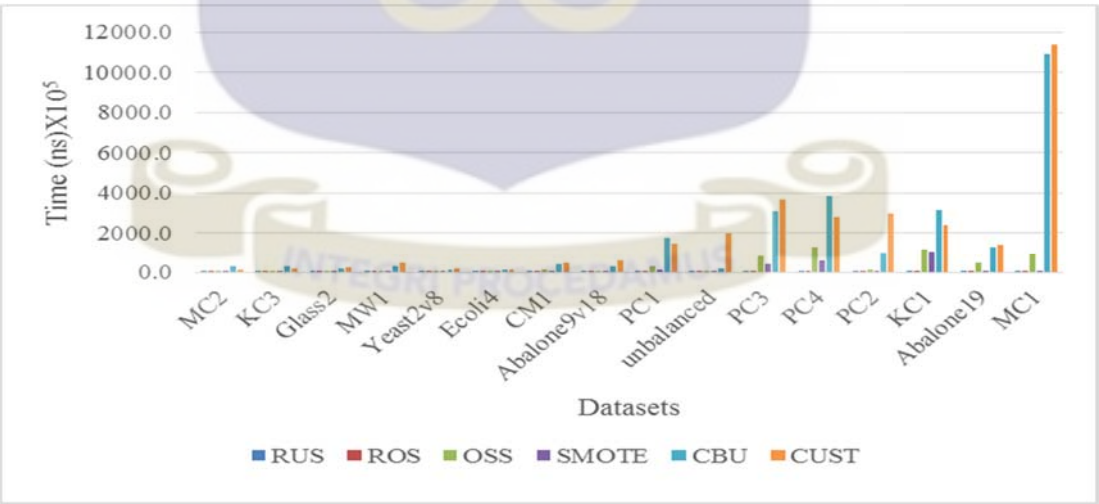

Figure 5.15: Computational Time of Sampling Techniques

To assess the dynamics of the computational time of CUST and CBU with respect to the number of clusters, they are further tested using the smallest (MC2) and largest

(MC1) datasets and in each case the sampling parameters are held constant and the number of clusters varied. The results of these tests are illustrated in figure 5.16 for MC1 dataset and figure 5.17 for MC2 datasets. The results suggests that the number of clusters considered in these algorithms significantly influence their sampling time.

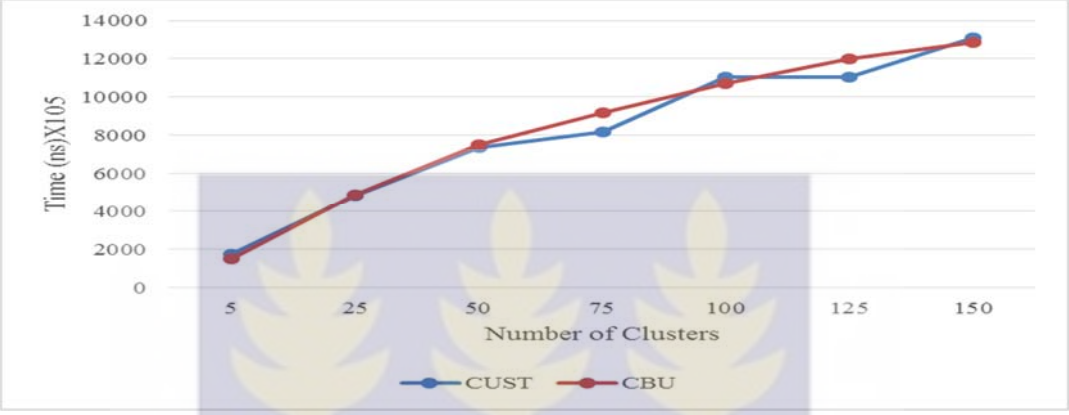

Figure 5.16: Sampling time of CUST and CBU for MC1 dataset

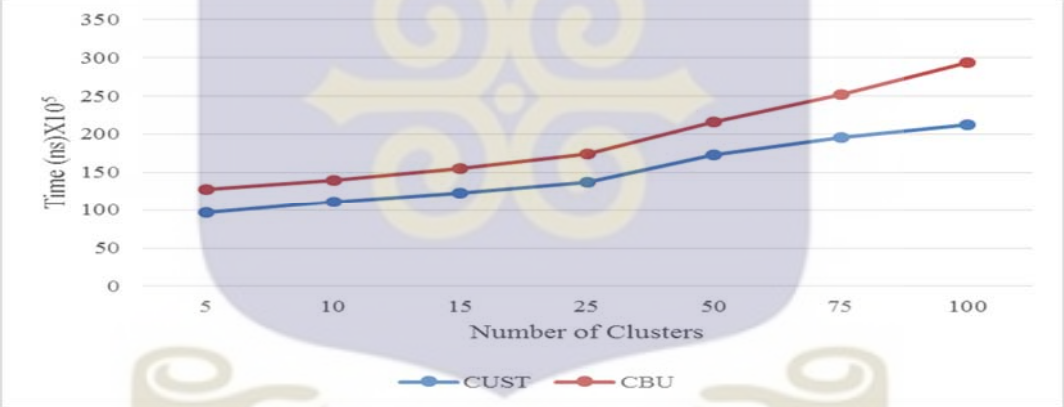

Figure 5.17: Sampling time of CUST and CBU for MC2 dataset

Considering the computation complexity of the techniques and their performance detailed in the above sections, the main variables that determines the choice of a sampling technique are the size of the dataset and how skewed the dataset is. For small to medium size and much skewed datasets, CUST appears to be the ultimate choice. However, for very large datasets with relatively higher proportions of minority instances and where computational resources is a limitation, then RUS can be used instead.

CHAPTER SIX

CONCLUSION AND RECOMMENDATION

6.0 Introduction

This is the final chapter of the work and presents the conclusion and recommendations outlining possible future research directions.

6.1 Conclusion

Data classification or class prediction using machine learning algorithms is widely studied and the aim is often to build prediction models that can identify possible positive instances in arriving data streams as much as possible in order to allow optimum allocation of limited resources or timely interventions to avert possible problems or effects these instances might cause. Class imbalance, which is inherent to most real world datasets militate against the classifiers ability to adequately learn to correctly identify these positive instances since they always constitute the minority in the datasets. Several class imbalance learning approaches have been devised to tackle the class imbalance problem to which the data level or sampling approach is an example. The sampling approach solves the imbalance problem by altering the distribution of instances in the training data either by discarding some of the majority class instances (undersampling) or increasing the population of minority class instances (oversampling). Some of the most well studied sampling techniques in literature include; RUS and OSS which are undersampling techniques, and SMOTE and ROS which are oversampling techniques.

There are however, other well-known data quality issues such as inconsistent/noisy instances, repeated/redundant instances, and outliers, which have the potential to

negatively affect the performance of classification algorithms when present in training data in substantial quantities. The above quality issues are as well found in datasets that are imbalance. This study therefore designed, implemented and examined a Cluster Undersampling Technique that solves the class imbalance problem by eliminating majority class instances from the training data that fall into any of the above categories, thereby producing a final training set that is virtually void of irrelevant instances.

The proposed technique was empirically examined using two machine learning algorithms; C4.5 decision tree and OneR algorithm, ten real-world software defect datasets from the NASA repository and six other datasets from the UCI repository. The performance of the classification algorithms when CUST is used to sample the training data prior to training was compared to the performance without sampling the training data. The results using AUC and G-Mean performance measures showed that using CUST significantly improved the performance of the algorithms in all the datasets considered.

The performance of CUST was also compared to five existing sampling techniques which include RUS, CBU, SMOTE, OSS, and ROS. The AUC and G-Mean measures showed that CUST yielded better results in at least thirteen out of the sixteen datasets using both C4.5 and OneR learners. RUS and CBU produced competitive results particularly in datasets with fewer repeated instances and higher percentages of minority class instances. A Tukey's HSD test on the mean performance of the classification algorithms at α=0.05 showed that CUST statistically performed better that SMOTE, ROS, OSS and NONE but RUS and CBU yielded competitive results

though they did not perform statistically different from SMOTE, ROS and OSS in most cases.

However, analysis of the computation time of the sampling techniques showed that CUST and CBU becomes computational expensive when used on very large datasets with a larger number of clusters. This is as a result of the use of k-means algorithm whose computation complexity depend on the value of k, number of instances, the dimension of the dataset and the number of iterations performed during clustering.

6.2 Recommendations

The performance of the proposed technique across the various datasets suggests that it increased the performance of the learners much better in datasets with lesser percentages of minority class instances. It would therefore be of interest to carry out further studies using different datasets with varying size and degrees of imbalance and classification algorithms to confirm and further establish the practicality and other possible limitations of CUST.

As this study focused much on addressing the class imbalance problem in software defect datasets and other few areas, assessing the applicability and efficiency of CUST in other application areas where the class imbalance problem is also predominant such as fraud detection, intrusion detection, and cancer detection is highly recommended.

Considering the computational complexity of CUST, it is also of interest to research into how the time requirements of CUST can be minimized either by tackling it at the

algorithmic level or how other outlier detection techniques with lesser computational time can be used in the algorithm instead of the k-means algorithm.

REFERENCES

[1] X. Li, W. Ying, J. Tuo, B. Li, and W. Liu, "Applications of classification trees to consumer credit scoring methods in commercial banks." *IEEE International Conference on Systems, Man and Cybernetics*, 5, 4112–4117, 2004.

[2] V. U. B. Challagulla, F. B. Bastani, I-L. Yen, and R. A. Paul, "Empirical Assessment of Machine Learning based Software Defect Prediction Techniques," *Proc. of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'05),* 2005

[3] R. Malhotra and A. Jain, "Fault Prediction Using Statistical and Machine Learning Methods for Improving Software Quality," *Journal of Information Processing Systems*, Vol. 8, No. 2, June 2012, available on line at http://dx.doi.org/10.3745/JIPS.2012.8.2.241.

[4] C. Seiffert, J. V. Hulse, T. M. Khoshgoftaar, and A. Felleco, "An empirical study of the classification performance of learners on imbalance and noisy software quality data," Information Science, doi:10.1016/j.ins.2010.12.016, 2011

[5] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson, "Reflections on the NASA MDP Data Sets," *Selected Papers Special Issue on Evaluation and Assessment in Software Engineering,* IET, 2011.

[6] E. Acuña and C. Rodríguez, "An empirical study of the effect of outliers on the misclassification error rate," *Transactions on Knowledge and Data Engineering,* 2005.

[7] J. Zhang, and I. Mani, "kNN approach to unbalanced data distributions: A case study involving information extraction." In *Proceedings of the ICML'2003 workshop on learning from imbalanced datasets,* 2003.

[8] M. Maloof, "Learning when data sets are imbalanced and when costs are unequal and unknown." In *Proceedings of the ICML'03 workshop on learning from imbalanced data sets*, 2003.

[9] N. V. Chawla, "C4.5 and imbalanced datasets: Investigating the effect of sampling method, probabilistic estimate, and decision tree structure." In *Proceedings of the ICML'03 workshop on class imbalances*, 2003.

[10] T. M Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl, "Accuracy of Software Quality Models over multiple releases," *Annals of Software Engineering* 9(1–4): 103–116.

[11] C. Seiffert, T. M. Khoshgoftaar, and J. V. Hulse, "Improving Software-Quality Predictions with Data Sampling and boosting," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans,* Vol. 39, No. 6, 2009.

[12] T. M. Khoshgoftaar and N. Seliya, "Comparative Assessment of Software Quality Classification Techniques: An Empirical Case Study," *Empirical Software Engineering*, vol. 9, pp. 229-257, 2004.

[13] H. Parvin, B. Minaei-Bidgoli, H. Alizadeh, "Iranian Cancer Patient Detection Using a New Method for Learning at Imbalanced Datasets" in *Proc. IDEAL 2011, Lecture Notes in Computer Science*, Vol. 6936, 2011, pp. 299-306

[14] M. Di Martino, F. Decia, J. M., and A. Fernåndez, "Improving Electric Fraud Detection Using Class Imbalance Strategies," *ICPRAM 2012,* 2012

[15] N. Qazi and K. Raza, "Effect of Feature Selection, SMOTE and under Sampling on Class Imbalance Classification," *uksim, 2012 UKSim 14th International Conference on Computer Modelling and Simulation*, pp.145-150, 2012.

[16] V. García, A. I. Marqués, J. S. Sánchez, "Improving Risk Predictions by Pre-processing Imbalanced Credit Data," *19th International Conference, ICONIP 2012, Lecture Notes in Computer Science,* Vol. 7664, Doha, Qatar, November 12-15, 2012, Proceedings Part II, pp. 68-75.

[17] N. V. Chawla, L. O. Hall, K W. Bowyer, and W. P. Kegelmeyer, "SMOTE: Synthetic minority oversampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321-357, 2002.

[18] M. Kubat and S. Matwin, "Addressing the curse of imbalanced training sets: One sided selection," in *Proc. 14th Int. Conf. Mach. Learn.,* 1997, pp. 179–186.

[19] T. Maciejewski and J Stefanowski, "Local Neighbourhood Extension of SMOTE for Mining Imbalanced Data," *IEEE*, 2011.

[20] D. J. Drown, T. M. Khoshgoftaar, and N. Seliya, "Evolutionary Sampling and Software Quality Modeling of High-Assurance Systems," *IEEE Transactions on Systems, Man, and Cybernetics—Part A: Systems and Humans*, Vol. 39, No. 5, SEPTEMBER 2009

[21] T. M. Khoshgoftaar, K. Gao, and N. Seliya, "Attribute Selection and Imbalanced Data: Problems in Software Defect Prediction," In *proc. 22nd International Conference on Tools with Artificial Intelligence,* 2010, DOI 10.1109/ICTAI.2010.27.

[22] http://mdp.ivv.nasa.gov/

[23] A. Asuncion and D. J. Newman, UCI Machine Learning Repository [http://www.ics.uci.edu/~mlearn/MLRepository.html]. Irvine, CA: University of California, Department of Information and Computer Science, 2007.

[24] H. He and E. A. Garcia, "Learning from Imbalanced Data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, No. 9, September 2009.

[25] G. M. Weiss, "Foundations of Imbalanced Learning," in *Imbalanced Learning: Foundations, Algorithms, and Applications,* 1st ed., H. He and Y. Ma, Ed. Hoboken, New Jersey: John Wiley & Sons, Inc., 2013, pp. 14-38.

[26] H. He, "Introduction," in *Imbalanced Learning: Foundations, Algorithms, and Applications,* 1st ed., H. He and Y. Ma, Ed. Hoboken, New Jersey: John Wiley & Sons, Inc., 2013, pp. 1-8.

[27] F. Shull, V. Basili, B. Boehm, A. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What we have learned about fighting defects", *Proc. Eighth IEEE Symposium on Software Metrics,* pp. 249-258, 2002.

[28] R. Barandela, J.S. Sánchez, V. García, and E. Rangel, "Strategies for learning in class imbalance problems," Pattern Recognition 36 (3) (2003) 849–851.

[29] J. C. Riquelme, R. Ruiz, D. Rodriguez, and J. Moreno, "Finding defective modules from highly unbalanced datasets," *Actas de los Talleres de las Jornadas de Ingenier´ıa del Software y Bases de Datos*, vol. 2, no. 1, pp. 67–74, 2008.

[30] L. Pelayo, and S. Dick, "Applying Novel Resampling Strategies to Software Defect Prediction," In *proceeding of: Fuzzy Information Processing Society,* 2007. ISBN: 1-4244-1213-7.

[31] V. García, J.S. Sánchez, and R.A. Mollineda, "On the effectiveness of pre-processing methods when dealing with different levels of class imbalance," Knowledge-Based Systems 25 (2012) 13–21.

[32] B. Das, N. C. Krishnan, D. J. Cook, "Handling Imbalanced and Overlapping Classes in Smart Environments Prompting Dataset", *Springer Book on Data Mining for Services in Studies in Computational Intelligence,* 2012.

[33] D. Hawkins, *Identification of Outliers*. Chapman and Hall. London, 1980.

[34] J. R. Quinlan. *C4.5: programs for machine learning.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[35] J. Aczel and J. Daroczy, *On Measures of Information and Their Characterizations*. New York: Academic, 1975.

[36] R. C. Holte. *Very simple classification rules perform well on most commonly used datasets*. In Machine Learning, pages 63–91, 1993.

[37] M. H. Halstead, *Elements of Software Science,* New York: Elsevier North - Holland, 1977.

[38] T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, pp. 308-320, December 1976.

[39] N.E. Fenton and S. Pfleeger, *Software Metrics: A Rigorous and Practical Approach,* International Thompson Press, 1997.

[40] M. Shepped and D. Ince, "A Critique of Three Metrics," *J. Systems and Software,* vol. 26, no. 3, pp. 197-210, Sept. 1994.

[41] T. Menzies, J. Greenwald, and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," *IEEE Trans. Software Eng.*, vol. 33, no. 1, pp. 2-13, Jan. 2007.

[42] M. Fagan, "Advances in Software Inspections," *IEEE Trans. Software Eng.*, pp. 744-751, July 1986.

[43] F. Shull, I. Rus, and V. Basili, "How Perspective-Based Reading Can Improve Requirements Inspections," *IEEE Computer,* vol. 33, no. 7, pp. 73-79, July 2000, http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/82.77.pdf.

[44] T. Menzies, D. Raffo, S. Setamanit, Y. Hu, and S. Tootoonian, "Model-Based Tests of Truisms," In *Proc. IEEE Automated Software Eng.* Conf., 2002.

[45] N. Nagappan and T. Ball, "Static Analysis Tools as Early Indicators of Pre-Release Defect Density," In *Proc. International Conf. Software Eng.,* pp. 580-586, 2005.

[46] T. M. Khoshgoftaar and N. Seliya, "Fault Prediction Modelling for Software Quality Estimation: Comparing Commonly Used Techniques," *Empirical Software Engineering*, vol. 8, pp. 255-283, 2003.

[47] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and Ayşe Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Automated Software Eng.*, DOI 10.1007/s10515-010-0069-5, May 2010

[48] I. H. Witten, F. Eibe, and M. A. Hall, *Data mining: Practical machine learning tools and techniques*, Morgan Kaufmann Series in Data Management Systems, 3rd ed., Burlington, USA, Morgan Kaufmann Publishers Inc., 2011.

[49] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," In *Proc. of the 14th international joint conference on Artificial intelligence,* Vol. 2, pp. 1137–1143, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 1995.

[50] P. Chaudhary, N. Mohan, and P. S. Sandhu, "An Empirical Assessment for Software Defect Forecast on Qualitative and Quantitative Factors using Simple Decision Table Majority Classifier," *International Journal of Research in Engineering and Technology (IJRET)*, Vol. 1, No. 3, 2012 ISSN 2277 – 4378.

[51] G. M. Weiss, "Mining with rarity: A unifying framework," SIGKDD Explorer, vol. 6, no. 1, pp. 7–19, 2004.

[52] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson, "Further Thoughts on Precision" *International Conference on Evaluation and Assessment in Software Engineering*, 2011.

[53] M. Gönen, "Receiver Operating Characteristic (ROC) Curves," *Statistics and Data Analysis, SUGI 31 proceedings*, available online at www2.sas.com.

[54] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan, The PROMISE Repository of empirical software engineering data http://promisedata.googlecode.com, West Virginia University, Department of Computer Science, 2012

[55] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data Quality: Some Comments on the NASA Software Defect Data Sets," (Draft), November, 2011, available online at: http://nasa-softwaredefectdatasets.wikispaces.com , (accessed on 2nd October, 2013)

[56] D. C. Montgomery, "Experiments with a Single Factor: The Analysis of Variance," in *Design and Analysis of Experiments,* 5th ed., New York, USA: John Wiley & Sons, Inc., 2001, pp. 60-118.

[57] I. Tomek, "Two modifications of CNN," *IEEE Transactions on System, Man, and Cybernetics*, vol. 6, pp. 769-772, 1976.

[58] D. Rocke and D. Woodruff, "Identification of outliers in multivariate data," *Journal of the American Statistical Association*, vol. 91, no. 435, pp. 1047-1061, 1996.

[59] L. Kaufman, and P.J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley, New York, 1990.

[60] R. R. Bouckaert, F. Eibe, M. Hall, R. Kirkby, P. Reutemann, A. Seewald, and D. Scuse, "WEKA Manual for Version 3-7-9," University of Waikato, February, 2013.

APPENDICES

APPENIX A

As indicated in chapter three, section 3.1.2, in order to use the implemented filters in WEKA some standard source codes in the WEKA user manual [60] and book [48] are referenced.

I. FULL JAVA CODE OF CUST IMPLEMENTATION

```
package weka.filters.supervised.instance;
/**
*
* @author magebure
*/
import weka.core.Capabilities;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.Option;
import weka.core.OptionHandler;
import weka.core.RevisionUtils;
import weka.core.TechnicalInformation;
import weka.core.TechnicalInformationHandler;
import weka.core.Utils;
import weka.core.Capabilities.Capability;
import weka.core.TechnicalInformation.Field;
import weka.core.TechnicalInformation.Type;
import weka.filters.Filter;
import weka.filters.SupervisedFilter;
import weka.clusterers.SimpleKMeans;
import weka.core.EuclideanDistance;
import weka.filters.unsupervised.attribute.AddCluster;
import weka.filters.unsupervised.attribute.Remove;

import java.util.Enumeration;
import java.util.Random;
import java.util.Vector;

public class CUST extends Filter
implements SupervisedFilter, OptionHandler, TechnicalInformationHandler {

/** the random seed to use. */
protected int m_RandomSeed = 1;

/** the ratio of majority/minority instances to create. */
protected double m_Ratio = 1.0;

/** the index of the class value to undersample. */
protected String m_ClassValueIndex = "1";

/** the number of clusters. */
protected int m_numCluster = 5;
```

```
/** whether to remove inconsistent instances. */
protected String mb_RemoveInconsistent = "0";

/** whether to remove inconsistent instances. */
protected boolean m_RemoveInconsistent = false;
/**
* Returns a string describing this classifier.
*
* @return        a description of the classifier suitable for
*            displaying in the explorer/experimenter gui
*/
public String globalInfo() {
 return "CUST Resamples a dataset by applying the Cluster Undersampling"+
 " Technique. The original dataset must fit entirely in memory." +
 " The sample ratio and number of clusters may be specified." +
 " For more information, see \n\n"
 + getTechnicalInformation().toString();
}

/**
  * Returns an instance of a TechnicalInformation object, containing
  * detailed information about the technical background of this class,
  *
  * @return        the technical information about this class
  */
public TechnicalInformation getTechnicalInformation() {
   TechnicalInformation result = new TechnicalInformation(Type.ARTICLE);

   result.setValue(Field.AUTHOR, "M. A. Agebure");
   result.setValue(Field.TITLE, "Cluster Undersampling Technique");
   result.setValue(Field.PROJECT, "MPhil Thesis, UG");
   result.setValue(Field.YEAR, "2014");

   return result;
}

 /**
 * Returns the Capabilities of this filter.
 *
 * @return            the capabilities of this object
 * @see               Capabilities
 */
 public Capabilities getCapabilities() {
  Capabilities result = super.getCapabilities();
  result.disableAll();

  // attributes
  result.enableAllAttributes();
  result.enable(Capability.MISSING_VALUES);

  // class
  result.enable(Capability.NOMINAL_CLASS);
  result.enable(Capability.MISSING_CLASS_VALUES);

  return result;
 }
```

```
    /**
     * Returns an enumeration describing the available options.
     *
     * @return an enumeration of all the available options.
     */
  public Enumeration listOptions() {
    Vector newVector = new Vector();

    newVector.addElement(new Option(
    "\tSpecifies the random number seed\n"
    + "\t(default 1)",
    "S", 1, "-S <num>"));

    newVector.addElement(new Option(
    "\tSpecifies Ratio of of maj:min instances to create.\n"
    + "\t(default 1.0)\n",
    "P", 1, "-P <num>"));

    newVector.addElement(new Option(
    "\tSpecifies number of clusters to create.\n"
    + "\t(default 5)\n",
    "Z", 1, "-K <num>"));

    newVector.addElement(new Option(
    "\tSpecifies the index of the nominal class value to undersample\n"
    +"\t(default 1))\n",
    "C", 1, "-C <value-index>"));

    newVector.addElement(new Option(
     "\tSpecifies whether to discard inconsistent instances or not\n"
    +"\t(default is 0: dont discard))\n",
     "I", 1, "-I <value-index>"));

    return newVector.elements();
  }

  /**
   * Parses a given list of options.
   *
   <!-- options-start -->
   * Valid options are: <p/>
   *
   * <pre> -S &lt;num&gt;
   *  Specifies the random number seed
   *  (default 1)</pre>
   *
   * <pre> -P &lt;num&gt;
   *  Specifies ratio of maj:min instances to create.
   *  (default 1.0)
   * </pre>
   *
   * <pre> -C &lt;value-index&gt;
   *  Specifies the index of the nominal class value to undersample
   *  (default 1:)
   * </pre>
   *
```

```
   * <pre> -C &lt;num&gt;
   *  Specifies whether to discard inconsistent instances or not
   *  (default 0: dont discard))
   * </pre>
   <!-- options-end -->
   *
   * @param options the list of options as an array of strings
   * @throws Exception if an option is not supported
   */

  public void setOptions(String[] options) throws Exception {
    String seedStr = Utils.getOption('S', options);
    if (seedStr.length() != 0) {
      setRandomSeed(Integer.parseInt(seedStr));
    } else {
      setRandomSeed(1);
    }

    String ratioStr = Utils.getOption('P', options);
    if (ratioStr.length() != 0) {
      setRatio(new Double(ratioStr).doubleValue());
    } else {
      setRatio(1.0);
    }

    String numClusStr = Utils.getOption('K', options);
    if (numClusStr.length() != 0) {
      setnumClusters(Integer.parseInt(numClusStr));
    } else {
      setnumClusters(5);
    }

    String classValueIndexStr = Utils.getOption( 'C', options);
    if (classValueIndexStr.length() != 0) {
      setClassValue(classValueIndexStr);
    } else {
      setClassValue("1");
    }

     String removeInconStr = Utils.getOption( 'I', options);
     if (removeInconStr.length()!=0) {
       setRemoveInconsistent(removeInconStr);
     } else {
       mb_RemoveInconsistent="0";
     }
   }

  /**
   * Gets the current settings of the filter.
   *
   * @return an array    of strings suitable for passing to setOptions
   */
   public String[] getOptions() {
    Vector<String>    result;

    result = new Vector<String>();
```

```
  result.add("-C");
  result.add(getClassValue());

  result.add("-P");
  result.add("" + getRatio());

  result.add("-K");
  result.add("" + getnumClusters());

  result.add("-S");
  result.add("" + getRandomSeed());

  result.add("-I");
  result.add("" + getRemoveInconsistent());

  return result.toArray(new String[result.size()]);
}

/**
 * Returns the tip text for this property.
 *
 * @return          tip text for this property suitable for
 *                  displaying in the explorer/experimenter gui
 */
public String randomSeedTipText() {
  return "The random number seed.";
}

/**
 * Gets the random number seed.
 *
 * @return          the random number seed.
 */
public int getRandomSeed() {
  return m_RandomSeed;
}
/**
 * Sets the random number seed.
 *
 * @param value     the new random number seed.
 */
public void setRandomSeed(int value) {
  m_RandomSeed = value;
}

/**
 * Returns the tip text for this property.
 *
 * @return          tip text for this property suitable for
 *                  displaying in the explorer/experimenter gui
 */
public String ratioTipText() {
  return "The ratio of Maj:Min instances in sampled data.";
}
```

```
/**
 * Sets the ratio of Majority:Minority instances.
 *
 * @param value    the ratio to use
 */
public void setRatio(double value) {
  if (value > 0)
    m_Ratio = value;
  else
    System.err.println("ratio must be > 0!");
}

/**
 * Gets the ratio value.
 *
 * @return         the ratio value
 */
public double getRatio() {
  return m_Ratio;
}

/**
 * Returns the tip text for this property.
 *
 * @return         tip text for this property suitable for
 *             displaying in the explorer/experimenter gui
 */
public String numClustersTipText() {
  return "The number of clusters to create.";
}

/**
 * Sets the number of clusters to create.
 *
 * @param value    the number of clusters to use
 */
public void setnumClusters(int value) {
  if (value > 0)
    m_numCluster = value;
  else
    System.err.println("numcluster must be > 0!");
}

/**
 * Gets the number of clusters to create.
 *
 * @return         the number of clusters to create
 */
public int getnumClusters() {
  return m_numCluster;
}

/**
 * Returns the tip text for this property.
 *
 * @return         tip text for this property suitable for
```

```
     *            displaying in the explorer/experimenter gui
     */
    public String classValueTipText() {
      return "The index of the class value to which CUST should be applied." +
       " Use a value of 1 to auto-detect the non-empty minority class.";
    }

     /**
     * Sets the index of the class value to which CUST should be applied.
     *
     * @param value     the class value index
     */
     public void setClassValue(String value) {
      m_ClassValueIndex = value;
     }

     /**
     * Gets the index of the class value to which CUST should be applied.
     *
     * @retur the index of the class value to which CUST should be applied
     */
     public String getClassValue() {
      return m_ClassValueIndex;
     }

     /**
     * Returns the tip text for this property.
     *
     * @return          tip text for this property suitable for
     *             displaying in the explorer/experimenter gui
     */
    public String RemoveInconsistentTipText() {
      return "Whether majority inconsistent instances should be discarded.";
    }
     /**
     * Sets whether majority inconsistent instances should be discarded.
     *
     * @param value     the True/False
     */
     public void setRemoveInconsistent(String value) {
      mb_RemoveInconsistent = value;
      if (mb_RemoveInconsistent.equals("0")) {
         m_RemoveInconsistent = false;
      }else {
       m_RemoveInconsistent = true;
      }
     }
     /**
     * Gets the value as to whether to remove inconsistent instances.
     *
     * @return True or False
     */
     public String getRemoveInconsistent() {
      return mb_RemoveInconsistent;
     }
```

```
 /**
  * Sets the format of the input instances.
  *
  * @param instanceInfo     an Instances object containing the input
  *                         instance structure (any instances contained in
  *                         the object are ignored - only the structure is required).
  * @return      true if the outputFormat may be collected immediately
  * @throws Exception    if the input format can't be set successfully
  */
public boolean setInputFormat(Instances instanceInfo) throws Exception {
    super.setInputFormat(instanceInfo);
    super.setOutputFormat(instanceInfo);
    return true;
}
/**
 * Input an instance for filtering. Filter requires all
 * training instances be read before producing output.
 *
 * @param instance         the input instance
* @return              true if the filtered instance may now be
*                      collected with output().
* @throws IllegalStateException if no input structure has been defined
*/

public boolean input(Instance instance) {
  if (getInputFormat() == null) {
   throw new IllegalStateException("No input instance format defined");
  }
  if (m_NewBatch) {
    resetQueue();
    m_NewBatch = false;
  }
  if (m_FirstBatchDone) {
    push(instance);
    return true;
  } else {
    bufferInput(instance);
    return false;
  }
}

/**
 * Signify that this batch of input to the filter is finished.
 * If the filter requires all instances prior to filtering,
 * output() may now be called to retrieve the filtered instances.
 *
 * @return         true if there are instances pending output
 * @throws IllegalStateException if no input structure has been defined
 * @throws Exception     if provided options cannot be executed
 *                       on input instances
 */
public boolean batchFinished() throws Exception {
  if (getInputFormat() == null) {
   throw new IllegalStateException("No input instance format defined");
  }
  if (!m_FirstBatchDone) {
```

```
    // Do CUST, and clear the input instances.
    doCUST();
  }
  flushInput();
  m_NewBatch = true;
  m_FirstBatchDone = true;
  return (numPendingOutput() != 0);
}

protected void doCUST() throws Exception {
Instances pTrain=getInputFormat().stringFreeStructure();
Instances nTrain=getInputFormat().stringFreeStructure();

int classVIndex=0;
if(getClassValue().equals("0"))
classVIndex=1;

//*** Segment training data into +ve and -ve sets
Enumeration instanceEnum = getInputFormat().enumerateInstances();
   while(instanceEnum.hasMoreElements()) {
      Instance instance = (Instance) instanceEnum.nextElement();
      if ((int) instance.classValue() == classVIndex) {
         push((Instance) instance.copy());
         pTrain.add(instance);
      }else{
         nTrain.add(instance);
      }
    }

EuclideanDistance dist = new EuclideanDistance(getInputFormat());

//removes moisy majority class instances
if(m_RemoveInconsistent){
    for(int i=0; i<pTrain.numInstances(); i++){
       for(int l=0; l<nTrain.numInstances(); l++){
       Double dst = dist.distance(pTrain.instance(i), nTrain.instance(l));
          if(dst.equals(0.00)){
            nTrain.remove(l);
          }
       }
    }
}
int ntr=nTrain.numInstances();
int ptn=pTrain.numInstances();
double pt=ptn/ntr;

//Cluster majority instances
SimpleKMeans kmeans = new SimpleKMeans();
  AddCluster addcls=new AddCluster();
     kmeans.setNumClusters(getnumClusters());
     addcls.setClusterer(kmeans);
     addcls.setInputFormat(nTrain);
 nTrain = AddCluster.useFilter(nTrain, addcls);
 nTrain.setClassIndex(nTrain.numAttributes() -1);

 for (int ncls=0; ncls<getnumClusters(); ncls++){//Extract data for ith cluster
```

```
    Instances clsTrain=nTrain.stringFreeStructure();
          for (int j=0; j<nTrain.numInstances(); j++){
             if (nTrain.instance(j).classValue()==ncls){
                clsTrain.add(nTrain.instance(j));
             }
          }
//Remove cluster attribute
String[] opts = new String[]{ "-R", "last"};
  Remove remove = new Remove();
  remove.setOptions(opts);
  remove.setInputFormat(clsTrain);
  Instances newnTrain = Filter.useFilter(clsTrain, remove);
  newnTrain.setClassIndex(newnTrain.numAttributes()-1);

int ntn=newnTrain.numInstances();

double dd=(getRatio()*pt*ntn);
int ms=(int)Math.round(dd);  // # of instances to undersample from cluster

while(ms>0){
    int num=newnTrain.numInstances();
    if (num==0){
        break;
    }
    Instances samData=newnTrain.stringFreeStructure();

  Random rand = new Random(getRandomSeed());
  newnTrain.randomize(rand);
  push((Instance) newnTrain.instance(0).copy());

  for(int ni=0; ni<newnTrain.numInstances()-1; ni++){
  Double dst = dist.distance(newnTrain.instance(0), newnTrain.instance(ni+1));
       if(dst!=0.00){
          samData.add(newnTrain.instance(ni+1));//n
       }
  }
   newnTrain=new Instances(samData);
ms--;
}///end while
}//end cls
}

public static void main(String[] args) {
runFilter(new CUST(), args);
}
}
```

II. CODE EXTRACT OF FUNCTION FOR CBU

```
protected void doCBU() throws Exception {
Instances train=getInputFormat().stringFreeStructure();

SimpleKMeans kmeans = new SimpleKMeans();
    AddCluster addcls=new AddCluster();
       kmeans.setNumClusters(getnumClusters());
```

```
        addcls.setClusterer(kmeans);
        addcls.setInputFormat(getInputFormat());
    train = AddCluster.useFilter(getInputFormat(), addcls);
    train.setClassIndex(train.numAttributes() -1);

 for (int ncls=0; ncls<getnumClusters(); ncls++){
    Instances clsTrain=new Instances(train, 0);
    for (int j=0; j<train.numInstances(); j++){
     if (train.instance(j).classValue()==ncls){
         clsTrain.add(train.instance(j));
     }
}
//Remove cluster attribute
String[] opts = new String[]{ "-R", "last"};
    Remove remove = new Remove();
    remove.setOptions(opts);
    remove.setInputFormat(clsTrain);
Instances newTrain = Filter.useFilter(clsTrain, remove);
newTrain.setClassIndex(newTrain.numAttributes()-1);

//*** Create place holders for training data
Instances pTrain=getInputFormat().stringFreeStructure();
Instances nTrain=getInputFormat().stringFreeStructure();

//*** Segment training data into +ve and -ve sets
int classVIndex=0;
if (getClassValue().equals("0"))
    classVIndex=1;
Enumeration instanceEnum = newTrain.enumerateInstances();
  while(instanceEnum.hasMoreElements()) {
      Instance instance = (Instance) instanceEnum.nextElement();
      if ((int) instance.classValue() == classVIndex) {
          push((Instance) instance.copy());
          pTrain.add(instance);
      }else{
          nTrain.add(instance);
      }
  }
double tn=newTrain.numInstances();
double ptn=pTrain.numInstances();
double ratio=ptn/tn;

Enumeration instanceEnumn = nTrain.enumerateInstances();
if(ratio==0.0){
  while(instanceEnumn.hasMoreElements()) {
      Instance instance = (Instance) instanceEnumn.nextElement();
          push((Instance) instance.copy());
  }
}
else if(ratio>0.0 && ratio<1.0){
 if(ratio<getRatio()){
  while(instanceEnumn.hasMoreElements()) {
      Instance instance = (Instance) instanceEnumn.nextElement();
          push((Instance) instance.copy());
  }
  }
```

```
}
}
}

public static void main(String[] args) {
runFilter(new CBU(), args);
}
}
```

III. CODE EXTRACT OF FUNCTION FOR RUS

```
protected void doRUS() throws Exception {
Random rand=new Random(getRandomSeed());
Instances pTrain=getInputFormat().stringFreeStructure();
Instances nTrain=getInputFormat().stringFreeStructure();

   double uPercent=getPercentage();
   if(getPercentage()>=100){
      uPercent=100;
      System.err.println("Percentage to undersample "
            +"must be less than 100%. Undersampling will"
            +"proceed with 100%, all selected class instances"
            +"will be discarded.");
   }

int classVIndex=0;
if (getClassValue().equals("0"))
   classVIndex=1;

 Enumeration instanceEnum = getInputFormat().enumerateInstances();
  while(instanceEnum.hasMoreElements()) {
     Instance instance = (Instance) instanceEnum.nextElement();
     if ((int) instance.classValue() == classVIndex) {
        push((Instance) instance.copy());
        pTrain.add(instance);
     }else{
        nTrain.add(instance);
     }
  }
  int nCount=nTrain.numInstances();
  double percent=(uPercent*nCount)/100;
  int sCount=(int)Math.round(percent);

  while(sCount>0){
     int sam=rand.nextInt(nCount);
     push((Instance) nTrain.instance(sam).copy());
     nTrain.remove(sam);
     nCount--;
     sCount--;
  }
}

public static void main(String[] args) {
runFilter(new RUS(), args);
```

```
}
}
```

IV. CODE EXRACT OF FUNCTION FOR ROS

```
protected void doROS() throws Exception {
  Random rand=new Random(getRandomSeed());
  Instances pTrain=getInputFormat().stringFreeStructure();
  Instances nTrain=getInputFormat().stringFreeStructure();

 int classVIndex=1;
 if (getClassValue().equals("0"))
  classVIndex=0;
 Enumeration instanceEnum = getInputFormat().enumerateInstances();
  while(instanceEnum.hasMoreElements()) {
    Instance instance = (Instance) instanceEnum.nextElement();
    push((Instance) instance.copy());
    if ((int) instance.classValue() == classVIndex) {
       pTrain.add(instance);
    }else{
       nTrain.add(instance);
    }
  }
  int pCount=pTrain.numInstances();
  double percent=(getPercentage()*pCount)/100;
  int sCount=(int)Math.round(percent);

  while(sCount>0){
     int sam=rand.nextInt(pCount);
     push((Instance) pTrain.instance(sam).copy());
     sCount--;
  }
 }

public static void main(String[] args) {
runFilter(new ROS(), args);
}
}
```

V. CODE EXTRACT OF FUNCTION FOR OSS

```
protected void doOSS() throws Exception {

Random rand=new Random(getRandomSeed());
Instances pTrain=getInputFormat().stringFreeStructure();
Instances nTrain=getInputFormat().stringFreeStructure();


int classVIndex=0;
if (getClassValue().equals("0"))
   classVIndex=1;

 Enumeration instanceEnum = getInputFormat().enumerateInstances();
  while(instanceEnum.hasMoreElements()) {
```

```
    Instance instance = (Instance) instanceEnum.nextElement();
    if ((int) instance.classValue() == classVIndex) {
       push((Instance) instance.copy());
       pTrain.add(instance);
    }else{
       nTrain.add(instance);
    }
  }
  int nCount=nTrain.numInstances();
  int sam=rand.nextInt(nCount);
  pTrain.add(nTrain.instance(sam));
  nTrain.remove(sam);

  double [] pred=new double[nTrain.numInstances()];
  double [] orig=new double[nTrain.numInstances()];
  IBk nBk=new IBk();
  nBk.buildClassifier(pTrain);
  for(int i=0; i<nTrain.numInstances(); i++){
     orig [i]=nTrain.instance(i).classValue();
     pred [i]=nBk.classifyInstance(nTrain.instance(i));
  }

  for(int j=0; j<orig.length; j++){
   if (orig [j]!=pred[j]){
     pTrain.add(nTrain.instance(j));
   }
  }

 Instances npTrain=pTrain.stringFreeStructure();
 Instances nnTrain=pTrain.stringFreeStructure();
 for (int j=0; j<pTrain.numInstances(); j++){
   if (pTrain.instance(j).classValue()==0){
     npTrain.add(pTrain.instance(j));
   }
   else { nnTrain.add(pTrain.instance(j));
   }
 }
pTrain.clear();
nTrain.clear();

EuclideanDistance dist = new EuclideanDistance(getInputFormat());
for(int pt=0; pt<npTrain.numInstances(); pt++){
  int [] arPt=new int[nnTrain.numInstances()];
  double vDist=0;
  for(int nt=0; nt<nnTrain.numInstances(); nt++){
    Double Dist = dist.distance(npTrain.instance(pt), nnTrain.instance(nt));
     if(nt==0 || Dist<vDist){
        arPt=new int [nnTrain.numInstances()];
        vDist=Dist;
        arPt [nt]=1;
     }
     else{
        if(Dist==vDist)
         arPt [nt]=1;
     }
  }
```

```
    Instances temp=nnTrain.stringFreeStructure();
    int d=0;
    while(d<arPt.length){
        if(arPt [d]!=1)
            temp.add(nnTrain.instance(d));
        d++;
    }
    nnTrain=new Instances(temp);
    temp.clear();
}
 Enumeration instanceEnumn = nnTrain.enumerateInstances();
   while(instanceEnumn.hasMoreElements()) {
      Instance instance = (Instance) instanceEnumn.nextElement();
       push((Instance) instance.copy());
   }
}

public static void main(String[] args) {
runFilter(new OSS(), args);
}
}
```

APPENDIX B

I. Other performance measures of learners without sampling (NONE)

	C4.5					OneR				
Dataset	FPR	Recall	Prec.	F	Acc.	FPR	Recall	Prec.	F	Acc.
Abalone9v18	0.015	0.195	0.384	0.242	93.982	0.006	0.127	0.364	0.179	94.393
Abalone19	0.000	0.000	0.000	0.000	99.234	0.000	0.000	0.000	0.000	99.234
Ecoli4	0.011	0.670	0.745	0.679	96.958	0.009	0.610	0.762	0.653	96.840
Glass2	0.034	0.265	0.303	0.268	90.957	0.006	0.000	0.000	0.000	91.519
Yeast2v8	0.003	0.450	0.695	0.532	95.520	0.012	0.550	0.720	0.603	95.447
unbalanced	0.000	0.000	0.000	0.000	98.599	0.000	0.000	0.000	0.000	98.599
CM1	0.083	0.181	0.235	0.193	82.739	0.036	0.070	0.158	0.094	85.503
KC1	0.067	0.296	0.454	0.351	83.383	0.052	0.196	0.420	0.261	83.164
KC3	0.089	0.301	0.396	0.320	80.050	0.073	0.118	0.176	0.129	78.100
MC1	0.003	0.157	0.299	0.191	99.132	0.000	0.109	0.457	0.171	99.304
MC2	0.243	0.442	0.487	0.444	64.929	0.174	0.288	0.457	0.332	64.013
MW1	0.039	0.217	0.339	0.244	88.499	0.039	0.058	0.106	0.068	86.949
PC1	0.030	0.208	0.395	0.248	90.922	0.016	0.127	0.385	0.182	91.555
PC2	0.002	0.000	0.000	0.000	98.777	0.001	0.000	0.000	0.000	98.871
PC3	0.067	0.245	0.342	0.275	84.718	0.016	0.093	0.464	0.147	87.351
PC4	0.062	0.511	0.570	0.522	88.413	0.021	0.267	0.664	0.372	88.856
Average	**0.047**	**0.259**	**0.353**	**0.282**	**89.801**	**0.029**	**0.163**	**0.321**	**0.199**	**89.981**

II. Other performance measures of learners with RUS

	C4.5					OneR				
Dataset	FPR	Recall	Prec.	F	Acc.	FPR	Recall	Prec.	F	Acc.
Abalone9v18	0.344	0.705	0.118	0.199	65.847	0.334	0.714	0.127	0.211	66.845
Abalone19	0.036	0.081	0.010	0.016	95.708	0.039	0.110	0.021	0.034	95.470
Ecoli4	0.111	0.830	0.400	0.505	88.586	0.060	0.830	0.536	0.612	93.360
Glass2	0.075	0.385	0.313	0.308	88.329	0.473	0.650	0.102	0.172	53.494
Yeast2v8	0.198	0.675	0.325	0.375	79.224	0.012	0.550	0.720	0.603	95.447
unbalanced	0.152	0.405	0.035	0.063	84.219	0.053	0.240	0.055	0.082	93.623
CM1	0.458	0.750	0.188	0.297	56.686	0.512	0.785	0.179	0.289	52.364
KC1	0.270	0.644	0.315	0.417	71.683	0.282	0.647	0.299	0.407	70.669
KC3	0.306	0.553	0.297	0.373	66.950	0.149	0.298	0.339	0.284	75.100
MC1	0.065	0.530	0.059	0.106	93.168	0.050	0.417	0.062	0.105	94.580
MC2	0.309	0.538	0.476	0.476	63.897	0.303	0.452	0.452	0.429	61.013
MW1	0.188	0.480	0.268	0.318	77.863	0.116	0.407	0.289	0.319	83.474
PC1	0.300	0.751	0.184	0.294	70.436	0.474	0.859	0.140	0.240	55.262
PC2	0.098	0.530	0.055	0.098	89.856	0.101	0.485	0.051	0.089	89.521
PC3	0.242	0.628	0.271	0.376	74.197	0.439	0.789	0.208	0.327	58.926
PC4	0.225	0.896	0.374	0.525	79.029	0.319	0.973	0.310	0.469	71.794
Average	**0.211**	**0.586**	**0.230**	**0.297**	**77.855**	**0.232**	**0.575**	**0.243**	**0.292**	**75.684**

III. Other performance measures of learners with ROS

	C4.5					OneR				
Dataset	FPR	Recall	Prec.	F	Acc.	FPR	Recall	Prec.	F	Acc.
Abalone9v18	0.015	0.194	0.401	0.244	93.955	0.009	0.073	0.190	0.101	93.819
Abalone19	0.000	0.000	0.000	0.000	99.198	0.001	0.008	0.025	0.012	99.181
Ecoli4	0.015	0.635	0.733	0.648	96.453	0.014	0.555	0.683	0.586	96.037
Glass2	0.037	0.190	0.204	0.182	90.074	0.014	0.015	0.025	0.018	90.965
Yeast2v8	0.016	0.430	0.571	0.468	94.201	0.014	0.315	0.423	0.348	93.481
unbalanced	0.014	0.160	0.100	0.112	97.454	0.008	0.090	0.058	0.064	97.888
CM1	0.121	0.345	0.305	0.308	81.367	0.185	0.331	0.198	0.241	75.542
KC1	0.167	0.430	0.323	0.367	77.057	0.325	0.648	0.270	0.380	67.050
KC3	0.158	0.405	0.365	0.367	76.400	0.322	0.450	0.237	0.302	63.750
MC1	0.009	0.352	0.230	0.267	98.602	0.023	0.210	0.092	0.115	97.113
MC2	0.289	0.533	0.511	0.504	64.872	0.305	0.427	0.436	0.412	60.218
MW1	0.105	0.330	0.255	0.268	83.722	0.079	0.178	0.213	0.181	84.573
PC1	0.067	0.381	0.335	0.346	88.814	0.121	0.335	0.195	0.242	83.541
PC2	0.008	0.050	0.053	0.046	98.285	0.019	0.105	0.061	0.069	97.211
PC3	0.111	0.373	0.331	0.344	82.489	0.216	0.356	0.192	0.247	73.039
PC4	0.085	0.602	0.513	0.549	87.505	0.044	0.228	0.453	0.295	86.348
Average	**0.076**	**0.338**	**0.327**	**0.314**	**88.153**	**0.106**	**0.270**	**0.234**	**0.226**	**84.985**

IV. Other performance measures of learners with CUST

	C4.5					OneR				
Dataset	FPR	Recall	Prec.	F	Acc.	FPR	Recall	Prec.	F	Acc.
Abalone9v18	0.157	0.590	0.205	0.294	82.908	0.021	0.710	0.161	0.261	76.335
Abalone19	0.333	0.817	0.019	0.038	66.769	0.032	0.792	0.016	0.032	63.039
Ecoli4	0.113	1.000	0.443	0.592	89.340	0.000	0.950	0.495	0.623	91.087
Glass2	0.198	0.900	0.321	0.451	80.844	0.042	0.800	0.116	0.200	51.429
Yeast2v8	0.126	0.823	0.620	0.689	86.483	0.000	0.600	0.556	0.520	91.994
unbalanced	0.285	0.950	0.049	0.092	71.833	0.333	0.900	0.040	0.076	67.081
CM1	0.398	0.790	0.220	0.342	62.160	0.400	0.735	0.210	0.322	61.672
KC1	0.212	0.576	0.347	0.430	76.096	0.249	0.628	0.318	0.421	73.192
KC3	0.293	0.708	0.375	0.462	71.000	0.125	0.425	0.545	0.425	79.000
MC1	0.130	0.850	0.046	0.088	87.011	0.141	0.807	0.043	0.081	85.868
MC2	0.314	0.680	0.595	0.602	68.654	0.403	0.660	0.483	0.556	62.115
MW1	0.232	0.633	0.282	0.366	75.256	0.176	0.600	0.305	0.386	79.915
PC1	0.149	0.667	0.283	0.387	83.926	0.249	0.738	0.218	0.329	74.967
PC2	0.150	0.950	0.064	0.119	85.107	0.212	1.000	0.048	0.091	78.990
PC3	0.212	0.614	0.288	0.386	76.718	0.363	0.829	0.249	0.383	66.670
PC4	0.181	0.927	0.430	0.586	83.272	0.315	0.977	0.315	0.475	72.188
Average	**0.218**	**0.780**	**0.287**	**0.370**	**77.961**	**0.191**	**0.759**	**0.257**	**0.324**	**73.471**

V. Other performance measures of learners with OSS

	C4.5					OneR				
Dataset	FPR	Recall	Prec.	F	Acc.	FPR	Recall	Prec.	F	Acc.
Abalone9v18	0.021	0.228	0.363	0.262	93.571	0.008	0.156	0.403	0.213	94.365
Abalone19	0.000	0.000	0.000	0.000	99.234	0.000	0.000	0.000	0.000	99.234
Ecoli4	0.037	0.730	0.626	0.634	94.941	0.033	0.715	0.658	0.639	95.178
Glass2	0.066	0.395	0.332	0.326	89.041	0.021	0.000	0.000	0.000	90.158
Yeast2v8	0.010	0.490	0.676	0.547	95.215	0.099	0.741	0.639	0.669	87.282
unbalanced	0.002	0.000	0.000	0.000	98.413	0.000	0.000	0.000	0.000	98.576
CM1	0.134	0.292	0.226	0.246	79.587	0.063	0.080	0.115	0.089	83.239
KC1	0.157	0.516	0.383	0.436	79.236	0.150	0.460	0.365	0.403	78.960
KC3	0.164	0.407	0.355	0.361	75.950	0.141	0.308	0.344	0.300	76.050
MC1	0.006	0.209	0.238	0.209	98.801	0.001	0.112	0.459	0.175	99.279
MC2	0.562	0.749	0.430	0.537	56.321	0.541	0.717	0.418	0.517	54.705
MW1	0.064	0.387	0.414	0.372	87.966	0.064	0.387	0.424	0.376	87.993
PC1	0.072	0.373	0.335	0.338	88.339	0.019	0.165	0.456	0.229	91.554
PC2	0.002	0.010	0.005	0.007	98.796	0.001	0.000	0.000	0.000	98.859
PC3	0.565	0.371	0.336	0.343	82.596	0.062	0.181	0.298	0.216	84.373
PC4	0.094	0.678	0.520	0.584	87.734	0.030	0.308	0.619	0.401	88.563
Average	**0.122**	**0.365**	**0.327**	**0.325**	**87.859**	**0.077**	**0.271**	**0.325**	**0.264**	**88.023**

VI. Other performance measures of learners with SMOTE

	C4.5					OneR				
Dataset	FPR	Recall	Prec.	F	Acc.	FPR	Recall	Prec.	F	Acc.
Abalone9v18	0.036	0.425	0.407	0.406	93.299	0.012	0.150	0.283	0.188	93.982
Abalone19	0.007	0.033	0.025	0.029	98.588	0.000	0.000	0.000	0.000	99.234
Ecoli4	0.016	0.850	0.800	0.807	97.620	0.032	0.750	0.683	0.663	95.544
Glass2	0.036	0.036	0.350	0.350	92.035	0.253	0.350	0.111	0.146	70.996
Yeast2v8	0.099	0.650	0.311	0.413	88.219	0.012	0.550	0.650	0.583	95.456
unbalanced	0.014	0.200	0.133	0.150	97.430	0.000	0.000	0.000	0.000	98.599
CM1	0.188	0.417	0.235	0.293	76.379	0.107	0.190	0.174	0.168	80.666
KC1	0.108	0.401	0.408	0.399	81.570	0.228	0.544	0.310	0.391	73.647
KC3	0.155	0.432	0.422	0.400	77.100	0.075	0.306	0.415	0.337	81.350
MC1	0.005	0.187	0.213	0.188	98.866	0.006	0.129	0.130	0.118	98.740
MC2	0.297	0.669	0.618	0.559	67.301	0.345	0.596	0.495	0.513	63.167
MW1	0.133	0.360	0.229	0.266	81.509	0.114	0.422	0.313	0.334	83.714
PC1	0.096	0.433	0.297	0.342	86.615	0.086	0.193	0.177	0.164	85.601
PC2	0.013	0.130	0.081	0.092	97.799	0.003	0.040	0.028	0.029	98.714
PC3	0.128	0.402	0.310	0.346	81.343	0.127	0.369	0.289	0.315	80.983
PC4	0.082	0.642	0.542	0.581	88.249	0.188	0.688	0.345	0.452	79.622
Average	**0.088**	**0.392**	**0.336**	**0.351**	**87.745**	**0.099**	**0.330**	**0.275**	**0.275**	**86.251**

VII. Other performance measures of learners with CBU

	C4.5					OneR				
Dataset	FPR	Recall	Prec.	F	Acc.	FPR	Recall	Prec.	F	Acc.
Abalone9v18	0.123	0.472	0.218	0.283	85.382	0.350	0.765	0.131	0.220	65.659
Abalone19	0.342	0.464	0.009	0.018	65.691	0.112	0.193	0.015	0.026	88.221
Ecoli4	0.057	0.735	0.572	0.585	93.029	0.009	0.635	0.773	0.672	96.990
Glass2	0.082	0.470	0.349	0.363	88.206	0.640	1.000	0.121	0.214	41.080
Yeast2v8	0.105	0.610	0.381	0.424	87.301	0.020	0.560	0.699	0.592	94.809
unbalanced	0.020	0.075	0.034	0.046	96.695	0.383	0.490	0.029	0.045	61.511
CM1	0.409	0.754	0.210	0.325	61.049	0.371	0.718	0.224	0.328	63.991
KC1	0.399	0.792	0.269	0.401	63.081	0.405	0.831	0.275	0.413	63.168
KC3	0.281	0.643	0.341	0.433	70.550	0.234	0.457	0.310	0.351	71.000
MC1	0.007	0.202	0.217	0.189	98.747	0.019	0.228	0.144	0.144	97.546
MC2	0.283	0.555	0.509	0.514	66.128	0.402	0.540	0.415	0.450	57.795
MW1	0.191	0.580	0.275	0.358	78.670	0.171	0.565	0.309	0.376	80.175
PC1	0.199	0.674	0.236	0.345	79.105	0.308	0.705	0.171	0.272	69.274
PC2	0.161	0.555	0.043	0.075	83.655	0.147	0.445	0.044	0.070	84.908
PC3	0.406	0.726	0.283	0.405	73.122	0.265	0.712	0.280	0.398	73.184
PC4	0.211	0.851	0.379	0.521	79.715	0.400	0.898	0.251	0.391	63.766
Average	**0.205**	**0.572**	**0.270**	**0.330**	**79.383**	**0.265**	**0.609**	**0.262**	**0.310**	**73.317**