DEVELOPMENT OF ADVANCED DATA SAMPLING SCHEMES TO ALLEVIATE CLASS IMBALANCE PROBLEM IN DATA MINING CLASSIFICATION ALGORITHMS

BY

SAKINAT OLUWABUKONLA FOLORUNSO
MATRICULATION NUMBER: 112644

B. Tech. Computer Science (Akure), M. Sc. Computer Science (Ibadan)

A Thesis in the Department of Computer Science,
Submitted to the Faculty of Science,
In partial fulfilment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

of the

UNIVERSITY OF IBADAN, IBADAN

NIGERIA

SEPTEMBER, 2015

Certification

This is to certify that this research was carried out by Sakinat Oluwabukonla **FOLORUNSO** with matriculation number **112644** in the Department of Computer Science, University of Ibadan, Ibadan, Nigeria.

.....

Supervisor

Dr. A. B. Adeyemo

B. Sc. (Ife), PGD, M. Tech., Ph. D (Akure)

Department of Computer Science

University of Ibadan, Ibadan, Nigeria.

.....

Head of Department

Dr. A. B. Adeyemo

Department of Computer Science

University of Ibadan, Ibadan, Nigeria.

Dedication

This work is dedicated to the glory of Almighty Allah.

Also to my Mother, Alhaja R. A. Tijani and to the loving memory of my late father, Alhaji (Chief) R. O. Tijani.

Acknowledgements

My immense gratitude goes to Almighty Allah, the most Beneficent, the Merciful, the Firm. It is by His mercy that I have gone this far.

I wish to express my profound gratitude to my supervisor, Dr. A. B. Adeyemo for his guidance, supervisory act and being meticulous when going through this thesis. Your concern and tutelage is an addition to my life. I thank you for impacting knowledge in me. May God bless you (Amen).

My appreciation also goes to Dr S. O. Akinola, for the words of encouragement. To my mentor, Prof. A. O. Osofisan and my Sisters/Mentors, Dr Y. O. Folajimi, Dr B. O. Oladejo and Dr I. T. Ayorinde, I am very grateful. Dr O. B. Akinkunmi, Dr O.W.F. Onifade, Dr A. B. C. Roberts and Dr Osunade and all the entire academic staff and non- academic staff of the department, thank you for your support.

I also wish to express my profound gratitude to Professor Gustavo E.A.P.A. Batista of Departamento de Ciências de Computação, Universidade de São Paulo, Campus de São Carlos, Brazil for his priceless and selfless contribution to the success and completion of this thesis. Also, Diego Furtado Silva is highly appreciated for his selfless and immense support.

The effort of all the academic and non-academic staff of the Department of Mathematical sciences, Olabisi Onabanjo University, Ago Iwoye, Ogun State cannot be measured with ordinary thanks. I appreciate the undying support, assistance and mentoring of Dr. D. A. Agunbiade, Dr. T. O. Olatayo, Dr. O. S. Odetunde, Dr B. O. Oguntunde, Mrs K-K. A. Abdullah and Mr Taiwo Abbas. I also wish to express my appreciation to Dr. O. M. Ajibade and Mrs S. O. Ogunsanya of Geology Department, Olabisi Onabanjo University, Ago Iwoye, Ogun State for their endless support.

I also appreciate the support and contribution of Prof. O. Longe of Adeleke University, Ede and Dr Gabriel Iwasokun of Department of Computer Science, The Federal University of Akure, Ondo State (FUTA). My gratitude also goes to Mr Hameed Olaide of IT department of IITA, Nigeria.

The effort and assistance from my mother, Alhaja R. A. Tijani and My Uncle, Dr. M. A. Tijani are priceless and cannot be measured in cash or kind. Special thanks and

unquantifiable indebtedness to all my siblings for their moral and financial support: Mr Oludare Samsudeen Tijani, Mr Ayodele Kabir Tijani, Alhaja Fatimah Oluwaseyi Salami and Mr Oyeniyi Quadri Tijani. I want to thank you all for being there for me at all times.

My appreciation also goes to you all my in-laws: Folorunsos, Nurenis, Aderintos especially Mrs Biodun Rafiat Olagunju and Mrs Lola Salewa Akindele, who always play the dual role of Sister and Mother-in-law.

I must not forget to thank the entire members of Badrul-din As-salat women circle of Nigeria especially Murshid Taofeek Salaudeen and Alhaja R. A. Adabanija for their support and all the members of Association for Computer Machinery (ACM), Ibadan-Nigeria Chapter. My undying gratitude goes to my close friends especially Dr. Oluwafemi Oriola, Alhaji Fagbenro, Alhaji Dr. and Alhaja Muyibi, Alhaji Sulaiman Salami, Mrs Aderemi, Khalifa Musa and to those who because of space cannot be mentioned here. Thank you all.

My appreciation goes to all my students who have graduated and those still in school especially Adediji Michael (Angel) and Awoyemi Emmanuel of Olabisi Onabanjo University, Ago Iwoye, Ogun State.

My profound gratitude also goes to my children: Temitope Sariy Abeke, Mubarak Temidayo Adisa, Temilade Hamdalat Ajoke and Abdul Mateen Temidara Alao, who were always with me day and night praying persistently for the success of this work. Thank You for your understanding, perseverance and love. I am indeed grateful to you all.

Finally, many thanks to my husband, the man of my youth, Alhaji Dr. Rasheed Olayode Folorunso for being there for me always, for the sleepless night, for your prayers and for being my eyes to read this work. Almighty Allah will strengthen you and make all that you touch to prosper in your sight.

ABSTRACT

Classification is the process of finding a set of models that distinguish data classes to predict unknown class label in data mining. The class imbalance problem occurs when standard classifiers are majority-biased while the minority class is ignored. Existing classifiers tend to maximise overall prediction accuracy and minimise error at the expense of the minority class. However, research had shown that misclassification cost of the minority class is higher and should not be ignored since it is the class of interest. This work was therefore designed to develop advanced data sampling schemes that improve the classification performance of imbalance datasets with the view of increasing the recall of the minority class.

Synthetic Minority Oversampling Technique (SMOTE) was extended to SMOTE+300% and combined with existing under-sampling schemes: Random Under-Sampling (RUS), Neighbourhood Cleaning Rule (NCL), Wilson's Edited Nearest Neighbour (ENN) and Condense Nearest Neighbour (CNN). Five advanced data sampling scheme algorithms: SMOTE300ENN, SMOTE300RUS, SMOTE300NCL, SMOTENCL and SMOTERUS were coded using JAVA and implemented in WEKA, a data mining tool as an Application Programming Interface. The existing and developed schemes were applied to 886 Diabetes Mellitus (DM), 1,163 Senior Secondary School Certificate Result (SSSCR) and 786 Contraceptive Methods (CM) datasets. The datasets were collected in Ilesha and Ibadan, Nigeria. Their performances were determined with different classification algorithms using Receiver Operating Characteristics (ROC), recall of the minority class and performance gain metrics. Friedman's Test at $p = 0.05$ was used to analyse these schemes against the classification algorithms.

The ROC metric revealed that the mean rank values for DM, SSSCR and CM datasets treated with the advanced schemes ranged from 6.9-13.8, 3.8-12.8 and 6.6-13.5, respectively when compared with the existing schemes which ranged from 3.4-7.8, 2.6-12.6 and 2.8-7.9, respectively. These results signifies improved classification performance. The Recall metric analysis for the DM, SSSCR and CM datasets in the advanced schemes ranged from 9.4-13.0, 6.3-14.0 and 7.3-13.6, respectively when compared with the existing schemes 2.0-7.5, 2.5-8.9 and 2.1-7.4, respectively. These results show increased detection of the minority class. Performance gains by the advanced

schemes over the original dataset (DM, SSCE and CM) were: SMOTE300ENN (27.1%), SMOTE300RUS (11.6%), SMOTE300NCL (15.5%), SMOTENCL (8.3%) and SMOTERUS (7.3%). Significant difference was observed amongst all the schemes. The higher the mean rank value and performance gain, the better the scheme. The SMOTE300ENN scheme gave the highest ROC and recall values in the three datasets which were 13.8, 12.8, 12.3 and 13.0, 14.0, 13.6, respectively.

The developed Synthetic Minority Oversampling Technique 300 Wilson's Edited Nearest Neighbour scheme significantly improved classification performance and increased the recall of the minority class over the existing schemes using the same dataset. It is therefore recommended for classification of imbalanced datasets.

Keywords: Imbalanced dataset, Receiver operating characteristics, Data reduction techniques, Cost sensitive learning

Word count: 445

Table of contents

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

Introduction

1.1 Background to the study

Data mining is defined as the process of discovering patterns in data (Witten *et al*., 2011). It can also be referred to as the extraction or "mining" of knowledge from large amounts of data (Han and Kamber, 2001).

Data mining has attracted lots of attention from the information industry due to availability of large amont of data and the burning need to transform these data into information and knowledge. Data mining techniques such as class description, association analysis, classification and prediction, cluster analysis and outlier analysis are used to specify the kind of patterns to be found in data mining tasks. Classification is the process of finding new set of models that describes and distinguish data classes and concepts to be able to predict unknown class of objects. These newly created models are based on the analysis of the training data whose class label is known. Since the class label of each training sample is provided, this step is known as supervised learning. The new model may be presented in various forms such as classification (IF-THEN) rules or mathematical formulae. These rules can be used to categorize future data samples, as well as provide a better understanding of the dataset contents.

Classification can be used to predict the class label of data objects. In many application domains, users may be interested in mining descriptions that distinguishes a target class from its contrasting classes in the same dataset. Classification models/algorithm/classifiers/learners in data mining include Decision Trees, Artificial Neural Networks, *k*-Nearest Neigbhour classifiers and Random Forest. Hence, classification is an important task but a general problem in data mining and machine learning.

Some of the issues regarding dataset for classification are data cleaning, relevance analysis, data transformation, class imbalance problem and comparison of classifiers. These sub-problems impede learning. When constructing a classification model, the learning algorithm reveals the underlying relationship between the attribute set and class label and identifies a model that best fits the training data. This model should accurately predict the class label of previously unknown problem.

However, standard classifiers usually perform poorly on imbalanced data sets because they are designed to generalize from training data and output the simplest hypothesis that best fits the data. Therefore, this simplest hypothesis pays less attention to rare cases. However, in many cases, identification of these rare objects/minority class is of crucial importance; classification performances on the small/rare/minority classes are the main concerns in determining the property of a classification model (Sun *et al*., 2006, Hoens, 2012).

Most traditional classifiers operate on data drawn from the same distribution as the training data and assume that maximizing accuracy is the principal goal. Also, in a problem with imbalance level of 99%, a learning algorithm that minimizes error rate could decide to classify all examples as the majority class, in order to achieve a low error rate of 1%. A practical example is a domain trying to predict terrorist and non-terrorist or a cancer patient to a non-cancer patient. The size of the samples representing non-terrorist and non-cancer patients are more than the terrorist and cancer patient. Most classifiers will assume that the cost of misclassification for these two classes (terrorist and non-terrorist or cancer and non-cancer patients) is the same. But the cost of predicting a non-terrorist is much lower than actual terrorist who carries a bomb at a cinema. Nevertheless, all minority examples will be wrongly classified in this case (Xu-Ying *et. al,* 2009). This class imbalance

problem had been observed to cause a significant deterioration in the performance of standard classifiers (Barandela *et. al.*, 2003a; Johnson *et. al.*, 2012).

1.2 Motivation of study

Traditional classifiers such as Decision Trees, Artificial Neural Networks (ANN), and Support Vector Machines (SVM) are ineffective at identifying samples from minority class which is the class of interest during classification (Garcia *et al.*, 2012). New techniques are required to ensure that classifier can effectively identify these most important yet rarely occurring examples.

Secondly, there is also the advantage of low storage requirement (a reduced dataset) and a high computational advantage when dataset are reduced. Therefore, enhanced sampling schemes, which are external, independent of any classifier and also versatile, are desirable. This study therefore is motivated by the need to identify specific domains for which an imbalance was shown to hurt the performance of standard classifiers. Also to show whether these class imbalances are always damaging to classification and to what extent do different types of imbalances affect classification performances.

1.3 Justification for the study

The justification for this research is that most standard classifiers are working towards achieving a generalised *accuracy* and low error rate which are biased towards the majority class while completely ignoring the minority class. But this minority class is the class of interest. Following closely, is *class distribution* of a domain where the classifiers assume that the classification algorithm will work on the dataset drawn from the same class distribution with training and testing dataset but this is not always true. Furthermore, the *Error cost* which is characterised by the situation whereby the classifier assumes that errors coming from the different classes in the dataset are the same but this is not correct as misclassification cost of the minority class is higher than the majority class.

1.4 Research aim and objectives

The aim of this study is to develop enhanced data sampling schemes for improving the performance of imbalance datasets trained on classification models that can increase the RECALL of the minority class which is the class of interest.

The specific objectives of the study are to:

i. develop enhanced data sampling schemes to alleviate the effects of class imbalanced problem;

ii. evaluate the performance of these new data sampling schemes on different classifiers as well as on homogeneous and heterogeneous ensembles and compare their performances

1.5 Research methodology

The following methodologies were used in this study:

i. Extensive review of literature in related work

ii. Development of a theoretical taxonomy of the relationship between under-sampling schemes in class imbalance learning, their underlying data reduction techniques and their time complexity.

iii. Development of the enhanced data sampling schemes; SMOTE300ENN, SMOTENCL, SMOTERUS, SMOTE300NCL and SMOTE300RUS using Java programming language.

iv. Extension of WEKA, a data mining tool to accommodate these enhanced data sampling schemes.

v. Testing of the new data sampling schemes on selected datasets obtained locally: Contraceptive Methods (CM), Senior Secondary School Result (SSS Result), Diabetes Mellitus disease (DM) and Tuberculosis (TB) dataset in Nigeria.

vi. Testing of the enhanced and existing data sampling schemes (CNN, ENN and NCL) on various base classifiers (Decision Tree, RIPPER, Artificial Neural Network (ANN), Random Tree, Fast Decision Tree Learner (REPTree), Support Vector Machine (SVM) and *k*-Nearest Neighbours Classifier (1B3)), homogeneous ensembles (Boosting (ADABoostM1), BAGGING, Random Subspace (Decision Forest), Random Forest, Random Committee and MultiClass Classifier) and compared the result with heterogeneous ensemble (STACKING using Ripper, Decision Tree, 1B3, SVM and MLP as base classifiers in this order and Decision tree as the meta classifier).

vii. Evaluating the results obtained from the study of these datasets using the following metrics; Receiver Operating Characteristics Area Under Curve (ROC_AUC), Kappa

Statistics, Root Mean Square Error (RMSE), RECALL of the minority class and Performance Loss/gain.

viii. Analyses of the results obtained with performance metrics using both parametric and non-parametric statistical methods; ANOVA, Box and whisker plots and Friedman test at statistical significance level of 0.05%, confidence level of 95% in SPSS package.

1.6 Scope of the study

This study spans datasets with imbalance class distribution where the imbalance distribution among the classes in the dataset hindered the performance of classifiers. The study focuses on how to increase the RECALL of the minority class which is the class of interest. The study also identified specific domains for which an imbalanced dataset was shown to hinder the performance of standard classifiers, determine whether these imbalances were always damaging and to what extent different types of imbalances affect classification performances. CM, SSS Result and DM datasets were examples of such cases where the traditional classifier trained on them is overwhelmed by the number of the majority class thereby misclassifying the minority class, which is the class of interest.

1.7 Organisation of thesis

The rest of this thesis is organised as follows:

Chapter two gives an extensive review on class imbalance learning and several solutions reported in the literature. These include the sampling schemes, ensemble techniques, evaluation metrics and related work. Chapter three gives a comprehensive explanation on the methods used, the model development and the experimental setup. Chapter four presents the results obtained and the detailed discussion on the various results obtained. Finally, Chapter five gives the summary of the study, conclusion drawn from the study and the recommendations for future work are presented.

1.8 Glossary of terms

Association analysis: This is the discovery of association rules showing attributes-value conditions that occur frequently together in a given dataset.

Bias:	This is defined by Mitchell (1980) as a rule or method that causes an algorithm to choose one generalised output over another as explained by Wilson and Martinez, (1997b).
CBOS:	Cluster-based Oversampling
Cluster analysis:	This technique analyses data objects without consulting a known class label.
ENN:	Wilson's Edited Nearest Neighbour
MLP:	Multi Layer Perceptron
NCL:	Neighbour Cleaning Rule
Noise:	This is a random error or variance in a measured variable.
OSS:	One Sided Selection
Outlier analysis:	This is the study of data objects that do not comply with the general behaviour or models of the data.
Patterns:	These are rules generated from mining a dataset. A pattern represents knowledge if is easily understood by humans, valid on test data with some degree of certainty and novel.
REPTree:	Reduced Error Pruning Tree
RIPPER:	Repeated Incremental Pruning to Produce Error Reduction
RNN:	Reduced Nearest Neighbour
ROC:	Receiver Operating Characteristics
ROS:	Random Oversampling
RUS:	Random Under Sampling
Samples:	This could be used synonymously with examples, instances or objects. This is referred to as data tuples (rows or records) in a dataset.
SMO:	Sequential Minimization Optimization
SMOTE:	Synthetic Minority Oversampling TEchnique
SNN:	Selective Nearest Neighbour
Supervised learning:	This is a step in which the class label of each training samples is not known, and the number or set of classes to be learned may not be known in advance.
TLink:	Tomek Link

Training dataset: These are data tuples analysed to build the model collectively from.

CHAPTER TWO

Literature Review

This chapter presents the review of literature on class imbalance problem and related work.

2.1 Class Imbalance Problem

The class imbalance problem corresponds to the domain for which one class is represented by a large number of examples while the other is represented by few (Japkowicz, 2003). Class imbalance learning is the learning problem in which instances in some classes heavily outnumber the instances in other classes. Imbalance Data Set (IDS) corresponds to domain that suffers this problem (Wang *et al.*, 2009).

In such cases, standard classifiers tend to be overwhelmed by the large classes and ignore the small ones. This imbalance causes sub-optimal classification performance or even worse (Chawla *et al.*, 2004, Fernanez *et al.*, 2011). It is a fundamental problem of data mining research (Yang and Wu, 2005) and pattern recognition (Ghanem *et al.*, 2010). When the prediction model is trained on such an imbalance dataset, it tends to show a strong bias towards the majority class, since typical learning algorithms intend to maximize the overall prediction accuracy. In fact, if 95% of the entire dataset belongs to

the majority class, the model could ignore the remaining 5% of minority class and predict that all of the test data are in the majority class. Though the accuracy will be 95%, the instances belonging to the minority class will be absolutely mis-classified (Hido and Kashima, 2008). The mis-classification cost for the minority class, however is usually much higher than that of majority class and should not be ignored (Hido and Kashima, 2008, Thai-Nghe *et al.*, 2009).

Domain suffering naturally from class imbalances include detection of oil spill in satellite radar images (Kubat and Matwin, 1997), diagnosis of diseases in medicine such as rare diseases (cancer) and rare gene mutation (albino), medical diagnosis (Yang and Ma, 2010), network monitoring, intrusion detection (Vegard, 2010), earth quakes and nuclear explosion and helicopter (Guo *et al.*, 2008), risk management (Chawla *et al.*, 2004), text classification (Estabrooks *et al.*, 2004), education (the ratio of the number of "pass student" to "fail student") and detection of fraudulent or default banking (Thai- Nghe *et al.*, 2009), species distribution prediction in ecology and conservational biology (Johnson *et al.*, 2012), information retrieval and filtering (Lewis and Gale, 1994), response optimization in Customer Relationship Management (CRM) (Lessmann, 2004), document classification (Manevitz and Yousef, 2001), image retrieval (Chen *et al*., 2001), Deoxyribo Nucleic Acid (DNA) Microarray time series (Pearson *et al*., 2003), spam-detection and filtering (Kolcz *et al*., 2003) and sentence boundary detection in speech (Liu *et al*., 2006). In practical applications, the ratio of the small to large classes can be drastic such as 1:100, 1:1000, or 1 to 10,000 and sometimes even more (Chawla *et al.*, 2004). In a classification problem, algorithm is used to construct a model by learning from training set which contains examples with class labels (Boontarika and Maythapolnum, 2011).

2.2 Problems associated with class imbalance

Class imbalance occurs when there are significantly fewer training instances of one class compared to other classes (Thai- Nghe *et al.*, 2009, Chawla *et al.*, 2004). In some applications, some data are naturally imbalanced. Examples are in credit card fraud and rare disease case (cancer). However, imbalance data set can also occur in areas where data are too expensive to be obtained for the minority class e.g. shuttle failure (Chawla *et al.*, 2004, Guo *et al.*, 2008) or limitation in collecting data such as cost, privacy, and the large effort required to obtain a representative data set (Thai-Nghe *et al.*, 2009) thus, creating

'artificial' imbalances (Chawla *et al.*, 2004). Class imbalance gives rise to various difficulties when learning.

2.2.1 Difficulties encountered in imbalanced classification

Some of the difficulties associated with imbalance dataset classification when allied according to Lopez *et al*., (2013), Batista *et al.*, (2004), Nguyen, (2011), Johnson *et al.*, (2012), Fernandez *et al*., (2011) includes:

a. **Small Sample Size**

This corresponds to the situation where the size of the minority class is extremely small due to the fact that there is either limitation in collecting data, data are too expensive or the datasets are naturally imbalanced. So, learning algorithm could not make generalisations about the class distribution because of lack of information or enough data. In this situation, the minority class becomes poorly represented. The combination of imbalanced data and the small sample size problem presents a new contest as the minority class can be poorly represented and the classifier to learn this data space become too specific, leading to over fitting.

b. **Class Overlapping**

The problem of overlapping between classes appears when a region of the data space contains a similar quantity of training data from each class as shown in Figure 2.1. This problem may lead to developing an inference with almost the same apriori probabilities in this overlapping area, which makes it very hard or even impossible to distinguish between the two classes. Classification of imbalance dataset becomes sub-optimal when allied with class overlapping problem. However, any linearly separable problem can be solved by any base classifier irrespective of the class imbalance problem.

c. **Small Disjuncts**

The imbalance class problem is identical with small disjuncts problem. This small disjuncts problem is a condition that arises when sample from minority classes are represented within sub-clusters which happen as a direct result of underrepresented concept as established by Weiss and Provost, 2003; Galar *et al*., 2012 and Rahman

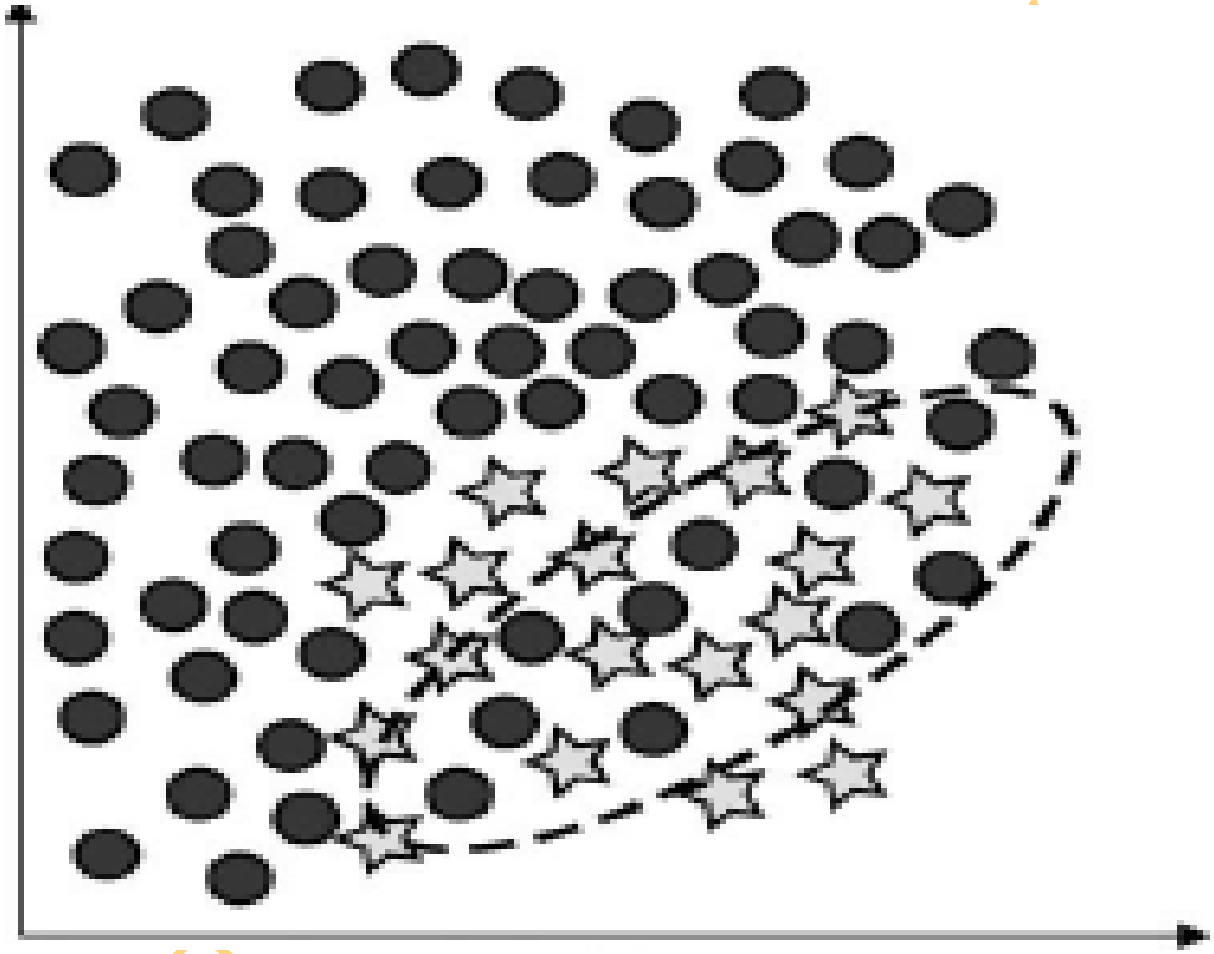


Figure 2.1: Class overlapping problem (Galar *et al*., 2012)

and Raju, 2014 and shown in Figure 2.2. Although these small disjuncts are hidden in most problems, their existence highly increases the complexity of the problem in the case of imbalance because it becomes hard to know whether these samples represent an actual sub-concept or are merely attributed to noise as concurred by Jo and Japkowicz (2004).

d. **Dataset Shift**

This phenomenon occurs when there is difference in the distribution of training and test samples of the same dataset as confirmed by Quinonero *et al.*, (2009). This issue is significant in the presence of class imbalance dataset as a single mis-classification on the minority class can cause a sub-optimal performance in classifiers. This issue is especially relevant when dealing with imbalanced classification, because in highly imbalanced domains, the minority class is particularly sensitive to singular classification errors, due to the typically low number of examples it presents.

e. Concept Complexity

This is an important factor in a classifier ability to alleviate class imbalance problem. Concept complexity in data corresponds to the level of separabilty of classes within the dataset (Japkowicz and Stephen, 2002). The class imbalance factor starts affecting the classifier generalisation ability as the degree of data complexity increases.

High complexity refers to inseparable datasets with highly overlapped classes, complex boundaries and high noise level. When samples of different classes overlap in the feature space, finding the ideal class boundary becomes tough (Nguyen *et al.*, 2009).

f. Noise

The class imbalance problem is more significant when the data sets have a high level of noise. Noise in datasets can emerge from various sources like data samples are poorly acquired or incorrectly labelled, or extracted features are not sufficient for classification as explained by Nguyen *et al.*, (2009).

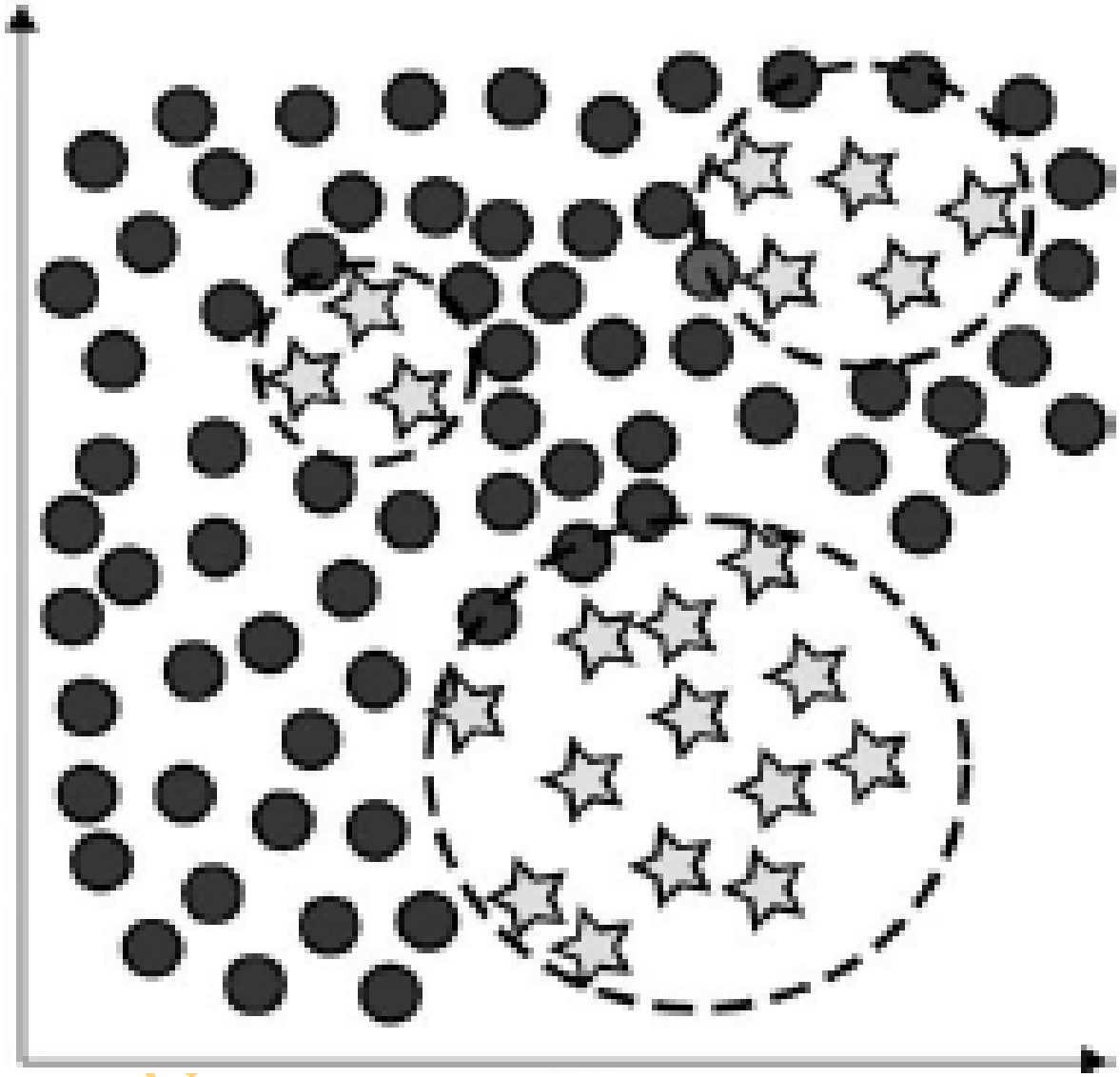


Figure 2.2: Small class disjuct (Galar *et al*., 2012)

2.2.2 Multiple Class Problems

Typically, there are two types of classes for imbalance datasets namely: bi-class and multiple classes (more than two classes or multi-class). In a bi-class application, the imbalanced problem is observed as one class, represented by a large amount of samples while the other is represented only by a few. The class with the few training sample are usually associated with high identification importance, is referred to as the positive class; the other one is the negative class (Thai-Nghe *et al.*, 2010, Sun *et al.*, 2006). In practice, most applications have more than two classes where unbalanced class distribution hinders the performance of the classifier. They suffer from more classification difficulties.

Most of the solutions reported to alleviate class imbalance problem so far are mainly two-class imbalance problems. Most real-world applications however have more than two classes with imbalanced distributions. They pose new challenges that are not observed in two-class problems. The multi-class classification problem is an extension of the traditional binary class problem where a dataset consists of k different classes instead of two. Though class imbalance exists in binary class datasets where one class severely outnumbers the other class, it also extends to multiple classes where the effects of imbalance are even more problematic. That is, given k different classes; there are multiple ways for class imbalance to manifest itself in the dataset. One typical way is that there is one "super majority" class which contains most of the instances in the dataset. Another typical example of class imbalance in multi-class datasets is the result of a single minority class. In such instances, each $k-1$ instances consists of roughly $1/(k-1)$ of the dataset, and the "minority" class makes up the rest (Hoens *et al.*, 2012). Multi-class imbalance problems suffer from more classification difficulties.

2.3 Methods of multiple classes' problem decomposition

There are several methods by which multi-class classification can be resolved. These are discussed in sections 2.3.1 to 2.3.3:

2.3.1 Direct multiclass classification

This scheme works by performing classification on the learning algorithm directly. This involves using the learning algorithm directly without any changes in parameters to alleviate a multiple class problem. Examples of such algorithms are K-Nearest Neighbour, Decision Tree, Bayes Classifier (Naïve Bayes) and Support Vector Machine.

2.3.2 **Multiclass Extension: Decomposition**

This is a technique of processing multiple class by transforming the problem into multiple or several binary (two-class) classification sub-problems. Decomposing a big problem has some advantages which according to Wang (2011) includes:

a. Individual classifiers are likely to be simpler than a classifier learnt from the whole data set.
b. They can be trained simultaneously for less modelling time
c. They can be trained independently which allows different feature spaces, feature dimensions and architectures. The change in one classifier will not affect the others.

However, the potential drawbacks of decomposition method according to Wang (2011) are:

a. each individual classifier is trained without full data knowledge and
b. It can cause classification ambiguity or uncovered data regions with respect to each type of decomposition.

2.3.3 **Methods of Decomposing Multiple Class Problems**

There are several approaches/methods in decomposing a multi classification problem. These methods as outlined by Boontarika and Maythapolnun, (2011) are discussed in 2.3.3.1 to 2.3.3.4:

2.3.3.1 One-**versus-One (OVO) Method**

This approach creates a classifier for each pair of classes. The training set for each pair classifier (i, j) includes only those instances that belong to either class $i \; or \; j$. A new instance, x, belongs to the class upon which most pair classifiers agree. The prediction decision is quoted from the majority vote technique. There are $n\frac{(n-1)}{2}$ classifiers to be computed, where n is the number of classes in the dataset. It is evident, that the disadvantage of this scheme is that there is need to generate a large number of classifiers, especially if there are a large number of classes in the training set. For example, if there is a training set of 1,000 classes, then 499,500 classifiers are needed. On the other hand, the

size of the training set for each classifier is small because all instances that do not belong to that pair of classes are excluded as discussed by Awad *et al*. (2009).

2.3.3.2 One-versus-All (OVA) Method

It creates a classifier for each class in the dataset. The training set is pre-processed such that, for a classifier j, instances that belong to class j are marked as class $(+1)$ and instances that do not belong to class j are marked as class (-1). In the OVA scheme, one computes n classifiers, where n is the number of pages that users have visited (at the end of each session). A new instance, x, is predicted by assigning it to the class that its classifier outputs has the largest positive value (that is maximal marginal). The main advantage of this method is that it introduces redundancy which creates generalisation in the classification but causes an over fitting problem when applied to a small sample size because each classifier uses data from two classes of interest and ignores the rest as conferred by Zhou and Tuck (2007).

The advantage of OVA scheme when compared to the OVO scheme is that it has fewer classifiers. On the other hand, the size of the training set is larger for OVA scheme than for an OVO scheme because the whole original training set was used to compute each classifier.

2.3.3.3 P-Against-Q (PAQ) Method

This is a generalised concept of a coding scheme. A code word length is equivalent to the sum of P and Q, where $P \geq 1$ and $Q \geq 1$. P is the number of "on" (binary 1) bits, and Q is the number of "off" (binary 0) bits. OVA is a PAQ scheme with $P = 1 \; and \; m = k$ as debated by Ou *et al*, (2004).

2.3.2.4 Error-Correcting Code Design Method

Error-correcting output code is defined to be a matrix of binary values. The length of a code is the number of columns in the code. The number of rows in the code is equal to the number of classes in the multiclass learning problem. A "code word" is a row in the code. A good error-correcting output code for a k-class problem should satisfy two properties: Row separation where each code word should be well-separated in Hamming distance from each of the other code words and column separation where each bit-position function f_i should be uncorrelated with the functions to be learned for the other bit positions $f_j, j \neq$

i. The power of a code to correct errors is directly related to the row separation as concluded by Dietterich and Bakiri, (1995).

2.4 Evaluation metrics

Accuracy is the most common evaluation metrics used by most traditional application. But accuracy is not suitable to evaluate imbalance data sets as it places more weight on the majority class than the minority class as affirmed by (Weiss and Provost 2003; Guo *et al.*, 2008). However, it has been observed that for extremely skewed class distribution, the RECALL/True Positive Rate (TPR) of the minority class is often 0, which means that there are no classification rules generated for the minority class as conferred by Guo *et al.* (2008). The under listed metrics in sections 2.4.1 to 2.4.10 are the most frequently used.

2.4.1 Confusion Matrix

In a bi-class problem, the confusion matrix records the result of correctly and incorrectly recognised examples of each class (Galar *et al.*, 2012; Thai-Nghe *et al.*, 2009). Table 2.1 presents the confusion matrix of a bi–class problem. The positive class represents the minority class while the negative class represent the majority class. True Positive (TP) shows the number of positive class correctly classified as positive, while True Negative (TN) shows the number of negative class correctly classified as negative class. False Positive (FP) shows the number of negative classes that were incorrectly classified as the positive class while false negative (FN) shows the number of positive classes that were incorrectly classified as negative class. The Recall/Sensitivity/True Positive Rate (TPR) is the likelihood that a positive class is correctly classified as positive as depicted by (equation 2.3). Positive Predictive Value (PPV)/Precision is the likelihood that positive prediction is correct as depicted by (equation 2.2) while NPV is the likelihood that a negative predictions correct as depicted by (equation 2.7). False Negative Rate (FNR) is the likelihood that a positive example is classified as negative example as depicted by (equation 2.5). This confusion matrix could be extended and expanded to multiple class problems.

$$\text{Accuracy} = \frac{(\text{TP}+\text{TP})}{(\text{TP}+\text{FN}+\text{FP}+\text{TN})} \qquad (2.1)$$

$$\text{Precision/PPV} = \frac{TP}{(TP+FP)} \quad (2.2)$$

$$\text{Recall (TPR or Sensitivity)} = \frac{TP}{(TP+FN)} \quad (2.3)$$

$$FPR = \frac{FP}{(FP+TN)} \quad (2.4)$$

$$FNR = \frac{FN}{(TP+FN)} \quad (2.5)$$

$$\text{Specificity/TNR} = \frac{TN}{(TN+FP)} \quad (2.6)$$

$$NPV = \frac{TN}{(TN+FN)} \quad (2.7)$$

2.4.2 **F_ measure**

This metric harmonises the mean between Recall and Precision and it is depicted by (Equation 2.8). It can be calculated by picking its values from Table 2.1. It generally focus the learning accuracy on positive class from completeness and efficiency aspect respectively (Ding, 2011). It is high when both Recall and Precision are high and can be adjusted through changing the value of β (Guo *et al.*, 2008, Thai-Nghe. *et al.*, 2009). The relative importance of precision versus recall is denoted by β and it is usually set to 1 (Chawla, 2005).

$$\text{F\_ Measure} = \frac{(1+\beta^2)(\text{Recall}\times\text{Precision})}{\beta^2\times(\text{Recall}+\text{Precision})} \quad (2.8)$$

Table 2.1 Confusion Matrix

	Positive Prediction	**Negative Prediction**
Positive Class	True Positive (TP)	False Negative (FN)
Negative Class	False Positive (FP)	True Negative (TN)

2.4.3 **Kappa statistic or Cohen's kappa coefficient**

It is used to measure the agreement between predicted and observed categorisation of a dataset, while correcting for an agreement that occurs by chance. Its maximum value is 100% (perfect agreement) and the expected value for random predictor with the column total is 0 (no agreement) (Witten *et al.*, 2011). Cohen's kappa coefficient is a statistical measure of inter-rater agreement or inter-annotator agreement or qualitative (categorical) items (Carletta, 1996). It is generally thought to be a more robust measure than simple percent agreement calculation since κ takes into account the agreement occurring by chance.

The equation for κ is depicted in (equation 2.9):

$$k = \frac{\Pr(a) - \Pr(e)}{1 - \Pr(e)} \qquad (2.9)$$

Where Pr (a) is the relative observed agreement among raters, and Pr (e) is the hypothetical probability of chance agreement, using the observed data to calculate the probabilities of each observer randomly in each category. If the raters are in complete agreement, then $k = 1$. If there is no agreement among the raters other than what would be expected by chance (as defined by Pr (e)), $k = 0$. Landis and Koch (1977) characterized values Kappa Statistic, $k < 0$ as indicating no agreement and, $0 < k \leq 0.20$ as slight, $0.21 < k \leq 0.40$ as fair, $0.41 < k \leq 0.60$ as moderate, $0.61 < k \leq 0.80$ as substantial, and $0.81 < k \leq 1$ as almost perfect agreement.

2.4.4 **G- Means Criterion**

Also known as geometric means and it combines the performance of both positive class and negative class i.e. geometric mean of the accuracies measured separately on each class (Positive and Negative) and depicted by (Equation 2.10) calculated from Table 2.1. It is calculated as the product of the prediction accuracies for both classes. High prediction accuracy on both positive and negative class will give rise to a high G-means value (Ding, 2011). Also, it measures the avoidance of the over fitting to the negative class and the degree to which the positive class is ignored.

$$\text{G-Mean} = \sqrt{(Specificity \times Sensitivity)} \quad (2.10)$$

2.4.5 Matthews Correlation Coefficient (MCC)

This is a strong metric that considers both accuracies and error rates on both classes, since all the four values in the confusion matrix are involved in this formula. A high MCC value means the learner should have high accuracies on positive and negative classes, and also have less mis-classification on the two classes. Therefore, MCC can be considered as the best singular assessment metric so far (Ding, 2011). This is represented in (Equation 2.11).

$$MCC = \frac{(TP \times TN) - (FP \times FN)}{\sqrt{(P_c \times N_C \times P_r \times N_r)}} \quad (2.11)$$

Where $P_c = TP+FN$

$N_c = TN+FP$

$P_r = TP+FP$

$N_r = FN+TN$

2.4.6 ROC (Receiver Operating characteristic) and AUC (Area Under the Curve of ROC)

ROC graph is a technique for visualising, organising and selecting classifier based on their performances (Fawcett, 2003). It has properties that make them especially useful for domains with skewed class distribution and unequal classification error cost (Fawcett, 2006). It is a two-dimensional graph in which TP rate is plotted on the Y- axis and FP rate on the X- axis. ROC graph depicts relative trade-offs between benefits (TPR) and costs (FPR) as depicted in Figure 2.3. So far, all the metrics discussed are based on fixed values of TP, TN, FP, FN, where such values can be easily collected when the class labels and predicted values are both discrete. However, in some other cases, such as the Bayesian network, or some neural network, or some ensemble classifiers, the prediction on testing data are continuous values, and a threshold have to be chosen to discretize them. Shifting the threshold within certain range can produce different groups of TP, TN, FP, FN values. By linking these TP and FP values jointly and plotting them on a 2-D axis, a Receiver Operating Characteristics (ROC) graph is constructed, as depicted in Figure 2.3. The ideal model should produce a point in Position A—the top left corner of Figure 2.3, where TPR

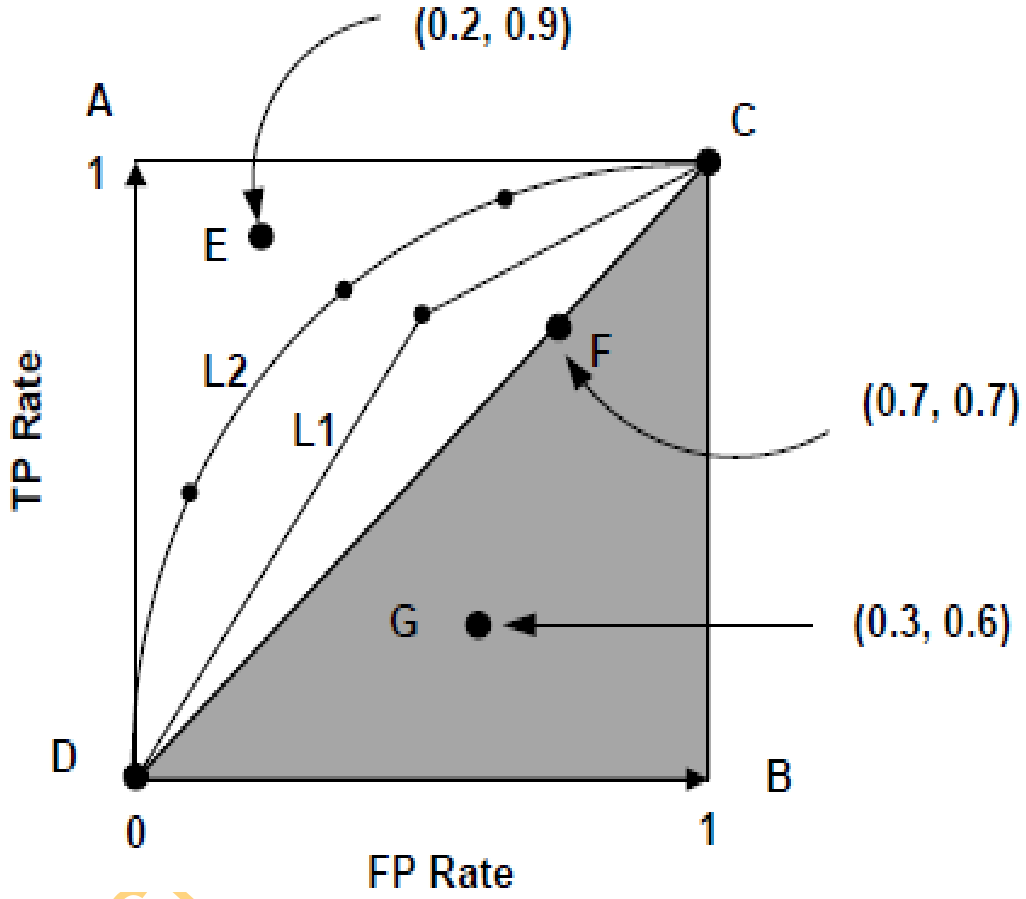


Figure 2.3: ROC_AUC graph (Ding, 2011)

is 1 and FPR is 0; and the worst model should be the point B at the bottom right corner. Hence, a good classification model should be as close to the top left corner as possible. Meanwhile, a model making random guess will be located on the diagonal, where the TPR and FPR are equal to each other. Note that the point D on the bottom left corner means the classifier predicts every examples as negative, and point C on the top right means all the predictions are positive. The ROC curve is created by connecting all groups of TPR and FPR values and point D and C together. The closer the ROC curve approaches to the top-left corner, the better the classification performance is (Weiss and Provost, 2003). However, directly comparing two or more ROC curves are challenging and impractical, e.g., two curves may be interleaved together and it is hard to claim the better one. Thus, a single numerical value to represent the effectiveness of the ROC curve is necessary, which brings the Area under the ROC curves (ROC_AUC).

Clearly, the ROC_AUC value is ranged from 0 to 1, and the higher it is, the better the classifier. Although the ROC curve provides a straight visualization for performance evaluation, it also has a particular limitation when it is applied to the highly imbalanced data set (Davis and Goadrich, 2006).

ROC attractive property is that they are insensitive to changes in class distribution. If the proportion of positive to negative instances changes in a test set, the ROC curves will not change. For ROC, graphs are based upon TPR and FPR, in which dimension is a strict columnar ratio, so do not depend on class distribution (Fawcett, 2006). (Equation 2.12) represents the formula for calculation.

$$\text{ROC\_AUC} = \frac{1 + (\text{TPR} - \text{FPR})}{2} \qquad (2.12)$$

2.4.7 **Precision-Recall Curves (PRC)**

The PRC depicts the relationship between precision and recall as the classification threshold varies (Thai-Nghe *et al.*, 2009; Davis and Goadrich, 2006). The recall (measures how often a positive class instance in the dataset is predicted as a positive class instance) is plotted on the X-axis and precision (which measures how often an instance which was predicted as positive is actually positive) is on the Y-axis.

2.4.8 H-Measure

H–Measure was proposed by Thai-Nghe *et al.* (2011). It uses asymmetric Beta distribution B42 to evaluate classifier when learning from imbalance datasets. H-Measure is used to replace the implicit cost weight distribution in the AUC. AUC has serious deficiency, since it implicitly uses different mis-classification cost distributions for different classifiers (Nguyen, 2011).

2.4.9 Cross Validation (CV)

This is a methodology often used when independent testing data are not available. Given a Training data set, equally split the set into ***K*** folds and then iteratively choose one fold for testing, and the others for training, until all folds have been used exactly once for testing. ***K*** can be pre-specified by user, and is normally chosen to be 5, 7 and 10. When ***K*** equals to the total number of examples in the data set, it is also called leave-one-out cross validation (LOOCV) (Ding, 2011).

2.4.10 Root Mean Square Error (RMSE)

The Root Mean Square Error (RMSE) is a frequently used measure of the difference between values predicted by a model and the values actually observed from the environment that is being modelled. These individual differences are also called residuals, and the RMSE serves to aggregate them into single measure of predictive power. (www.ctec.ufal.br/professor/crfj).

The RMSE of a model prediction with respect to the estimated variable X_{model} is defined as the square root of the mean squared error:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(X_{obsi} - X_{modeli})^2}{n}} \qquad (2.12)$$

Where X_{obs} is observed values and X_{model} is modelled values at time/place *i*.

2.5 Challenges faced by class imbalance problem

Some of the challenges faced by class imbalance problem according to Wasiowski and Chen (2010) were stated as follows:

i. It is not always easy to distinguish between noise examples and minority class examples as they are completely ignored by the classifier.

ii. The use of standard accuracy rate that benefits the covering of the majority examples.

iii. Classification rules that predict the positive class are often highly specialised and their covering is low, hence they are discarded in favour of more general rules.

iv. The combination of imbalance and the small sample size poses a problem to class imbalance learning

2.6 **Solutions to class imbalance problem**

Numerous existing solutions to Class Imbalance Problem were developed both at data and algorithmic levels. Almost all the solutions developed were designed for a two-class problem, where the imbalance problem observed is that one class is highly under represented but associated with a higher identification importance. All the existing solutions to class imbalance learning manipulates the training size, class prior, cost matrix and placement of decision boundary (Liu et al, 2008). At the data level, the objective is to re-balance the class distribution by re-sampling the data space while at the algorithm level, solutions try to adapt existing classifier learning algorithm to strengthen learning with regards to the minority class. The main advantage at the data level techniques is that they are independent of the underlying classifier (Fernandez *et al*. 2011, Ding, 2011)

At the data level; re-sampling technique balances the class distribution in the training data, by either adding examples to the minority class (Oversampling) or removing examples from the majority class (under- sampling) (Yang and Wu, 2006; Ding, 2011) or combination of oversampling and under-sampling (Sun *et al.*, 2006; Guo *et al.*, 2008, Ding, 2011). The resulting sampled dataset is then made more amenable to traditional algorithm which can then be used to classify the data. Figure 2.4 (a) shows an imbalance dataset where there were many more majority classes than the minority classes while Figure 2.4 (b) shows a balanced dataset with well-defined clusters.

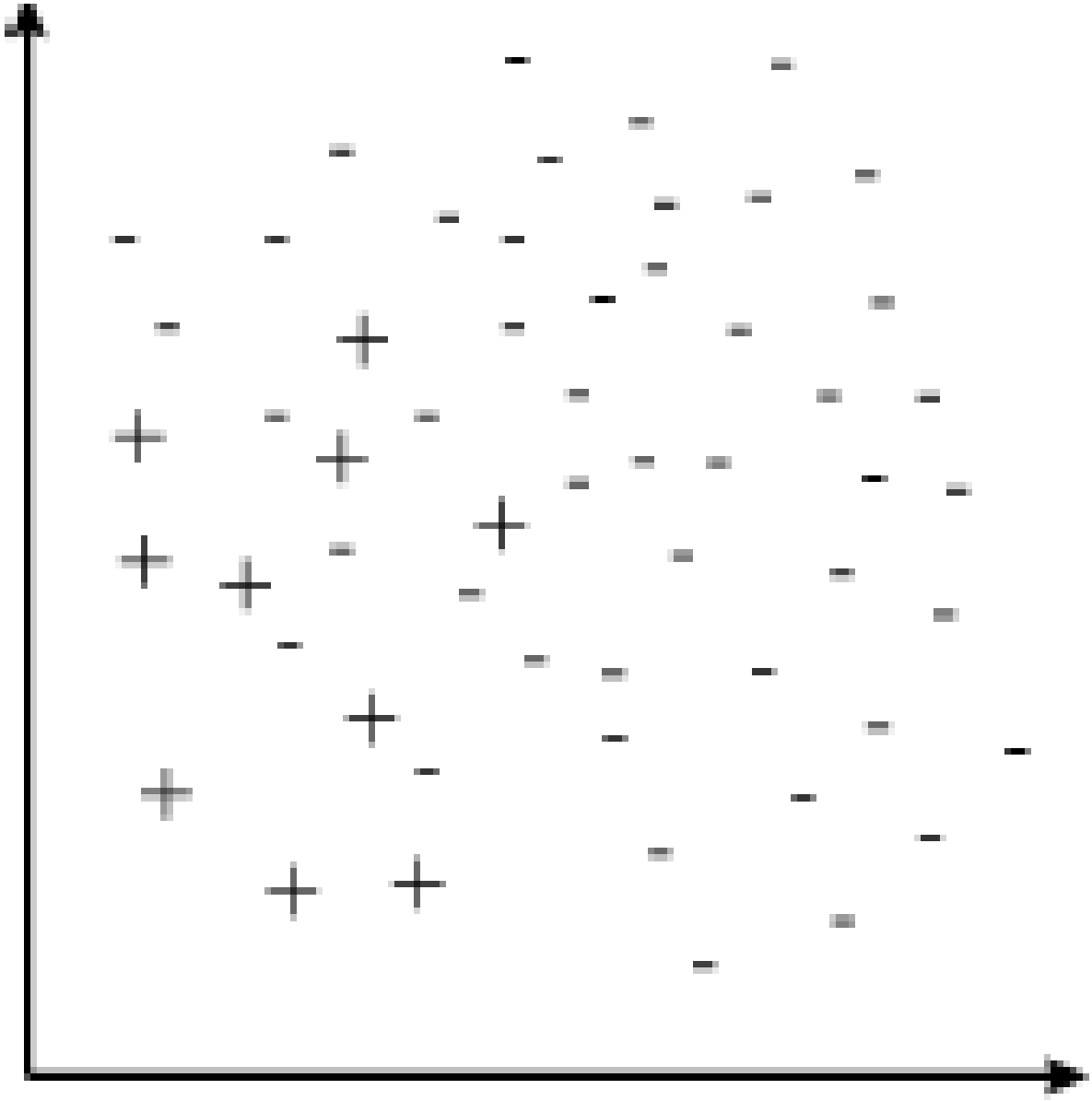


Figure 2.4(a): An Imbalanced dataset

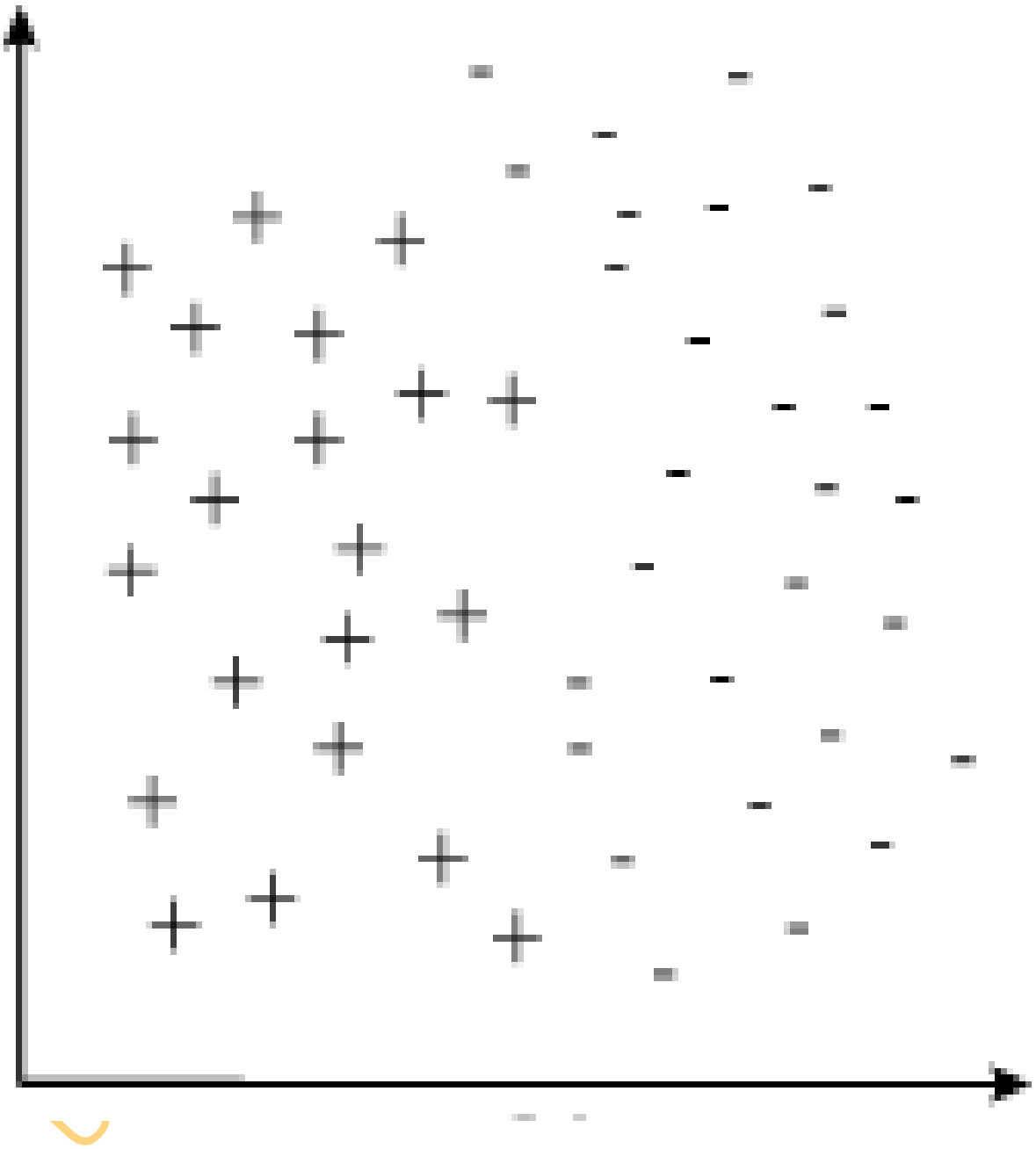


Figure 2.4(b): A balanced dataset

2.6.1 Sampling schemes

The under listed sampling schemes were commonly used for dealing with the class imbalance problem. Their advantage over other methods is that they are external and thus easily transportable. Though, these approaches can be simple to implement, tuning them can be difficult.

2.6.1.1. Under-sampling schemes

Sampling can be used as a data reduction technique because it allows a large dataset to be represented by a smaller subset of the dataset. This technique removes examples from the dataset to a desired distribution. There are two types of under-sampling: random and informed.

a. **Random Under-Sampling (RUS)**

This is random elimination of majority class examples. RUS makes no attempt to "intelligently" remove examples from the training data. Instead, RUS simply removes examples from the majority class at random until a desired class distribution is achieved. It can discard potentially useful data that could be important for the induction process, and this can make the decision boundary between minority and majority harder to learn (Ding, 2011; Seiffert *et al*., 2010). It creates a subset of the original dataset by the eliminated instances.

b. **Informed Under sampling.**

This technique removes instances from the dataset intelligently. Examples of these techniques are:

i. **Reduced Nearest Neighbour (RNN)**

This algorithm starts with S = T where S and T are datasets and removes each instance from S if such removal does not cause any other instances in T to be mis-classified. It is able to remove noisy and internal instances while retaining border points (Gates, 1972). This rule is an extension of the Condensed Nearest Neighbour (CNN) rule and it corrects the case of inconsistency in CNN (Miloud-Aouidate and Baba-Ali, 2011).

ii. **Selective Nearest Neighbour (SNN)**

This method extends CNN such that every member of dataset, T must be closer to a member of S of the same class rather than to a member of T (instead of S) of a different class. Though, this algorithm is still sensitive to noise, there is great reduction in training set as well as a higher accuracy than CNN. It is more complex than most other reduction technique and learning rate is significantly greater (Ritters *et al.,* 1975).

iii. Wilson's Edited Nearest Neighbour Rule (ENN)

Wilson, (1972) proposed an edited k- NN rule to improve the 1- NN rule. In his rule, editing the reference set is first performed: Each sample in the reference set is classified using the 3-NN rule and the set formed by eliminating it from the reference set. All the samples mis-classified are then deleted from the reference set. Afterward, any input sample is classified using the 1-NN rule and the edited reference set. Ties would be randomly broken whenever they occur (Wilson, 1972).

iv. Neighbourhood Cleaning Rule (NCL)

In this technique, ENN rule is used to identify and remove majority class. The algorithm first finds the three nearest Neighbours for each of $E_i$examples in the training set. If E_i belong to the majority class and it is mis-classified by its three Nearest Neighbours (3-NN), then Ei is removed. If E_i belongs to the minority class and it is mis-classified by its 3-NN to be the majority class, then removes the three nearest Neighbour. In order to avoid excessive reduction of small classes, only examples from classes mis-classified by 2-NN of its 3-NN are removed (Laurikala, 2001).

v. Tomek Links (TLink)

TLink under sampling technique was proposed by Ivan Tomek (Tomek, 1976) as a method of enhancing the Nearest-Neighbor Rule. Tlink algorithm removes both noise and borderline example. Let E_i,E_j, belong to different classes, $d(E_i, E_j)$ is the distance between them. A (E_i, E_j) pair is called a Tomek link if there is no example E_1, such that $d(E_i, E_1) <$

$d(E_i, E_j)$ *or* $d(E_j, E_1) < d(E_i, E_j)$. Examples participating in Tomek link are either borderline or noisy (Tomek, 1976). Figure 2.5 (a) shows an imbalanced dataset while Figure 2.5 (b) shows a more balanced dataset where points forming Tomek Link have been found and removed. Only majority class examples have also been removed.

vi. Condensed Nearest Neighbour Rule (CNN)

This technique is used to find a consistent minimum subset of examples and also to identify redundant examples that do not affect classification. A subset E_i of E is consistent with E if using a 1-Nearest Neighbour, E_i correctly classifies the example E. Let E be the original training set. Let E_i contains all positive examples from S and one randomly selected negative example. Then, classify E with the 1- NN rule using the examples in E_i. Move the entire mis-classified example from E to E_i (Hart, 1968).

This algorithm is sensitive to noise, thus causing even more misclassifications than before misclassification. Figure 2.6(a) presents an imbalance dataset while Figure 2.6(b) shows a balanced dataset after applying the CNN algorithm.

vii. **One - Sided Selection (OSS)**

This is a combination of Tomek link followed by the application of CNN. Tomek Link is used to remove noisy and border line majority class examples. Then, CNN will remove example from the majority class that are distant from decision border and redundant to create a constituent subset. Then a consistent subset of the majority class is formed. The learner always keeps all positive examples as they are too rare to be wasted, only noisy and negative examples are pruned out (Kubat and Matwin, 1997). It is an efficient algorithm especially in the case of high imbalanced data, but it requires significant execution time and processing resources (Jo and Japkowicz, 2004; Batista *et al.,* 2004 and Bekkar and Alitouche, 2013).

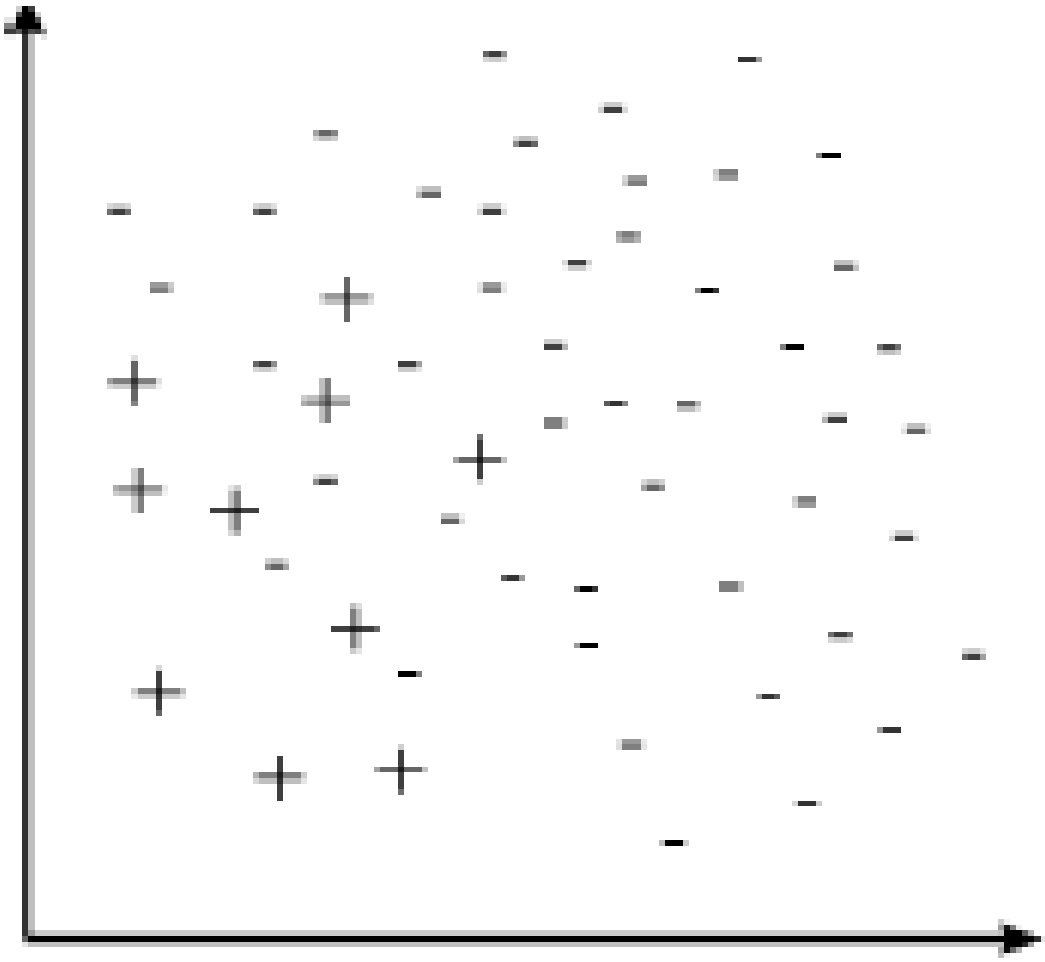


Figure 2.5(a): Imbalanced Dataset

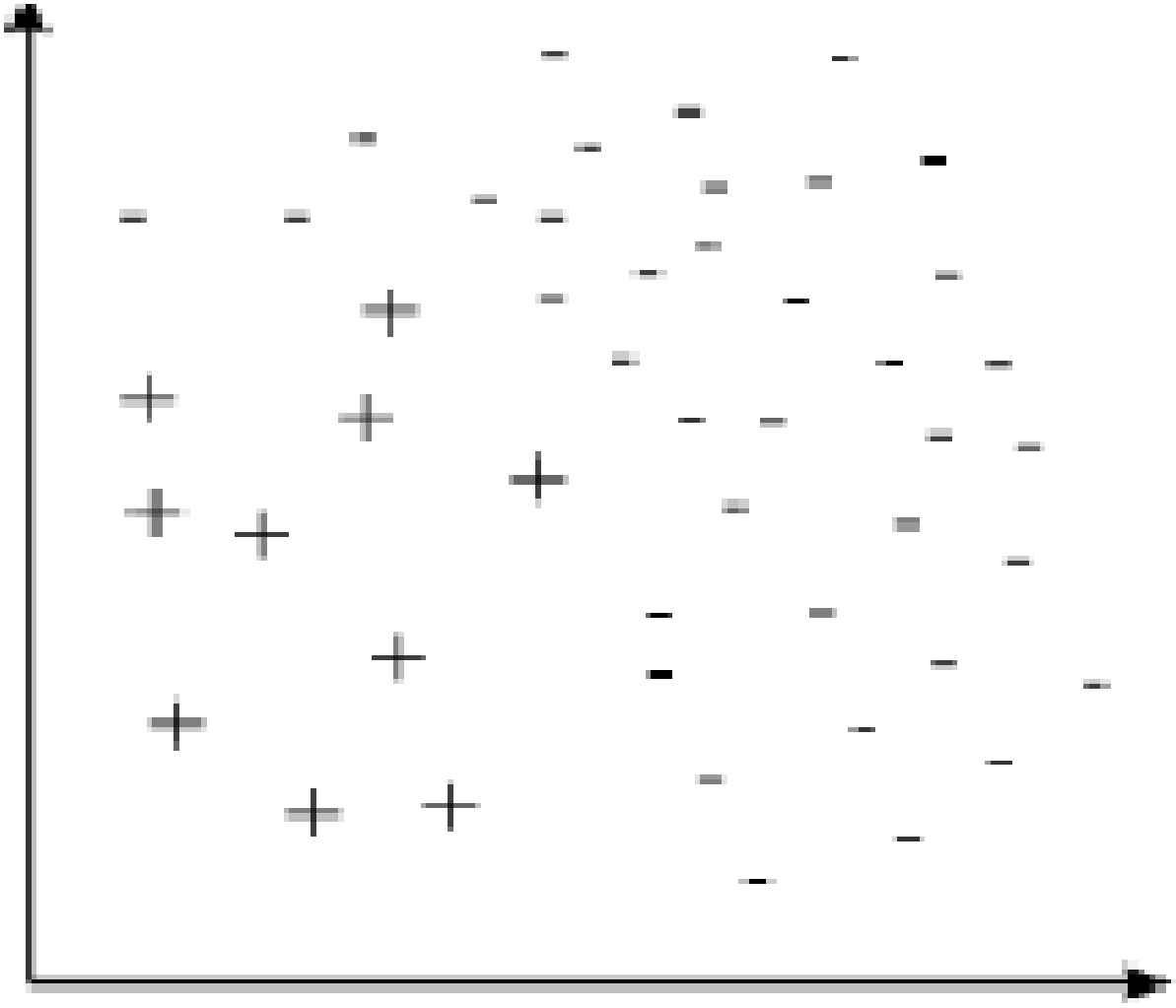


Figure 2.5(b): Balanced Dataset using TLink

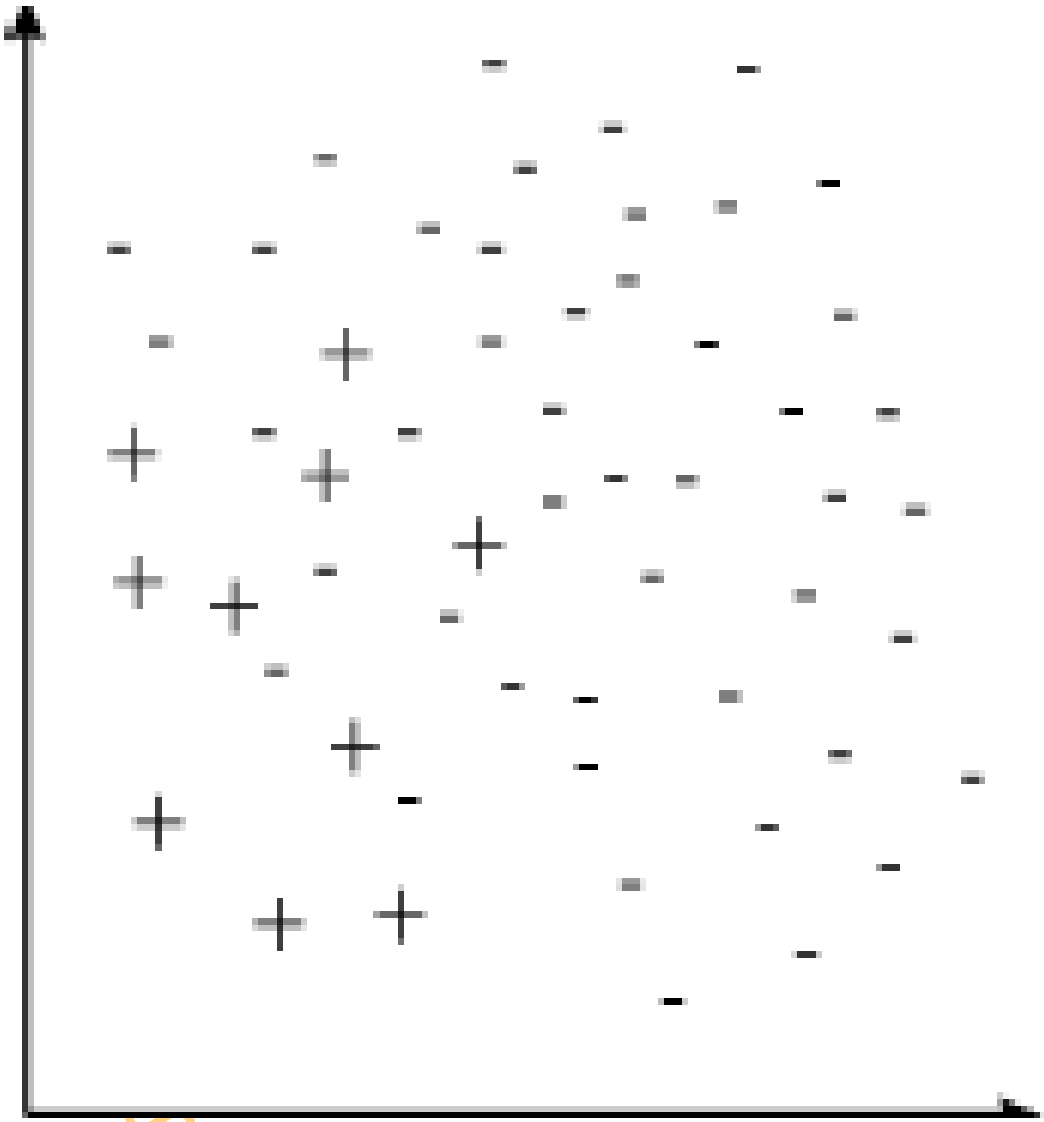


Figure 2.6(a): Imbalanced dataset

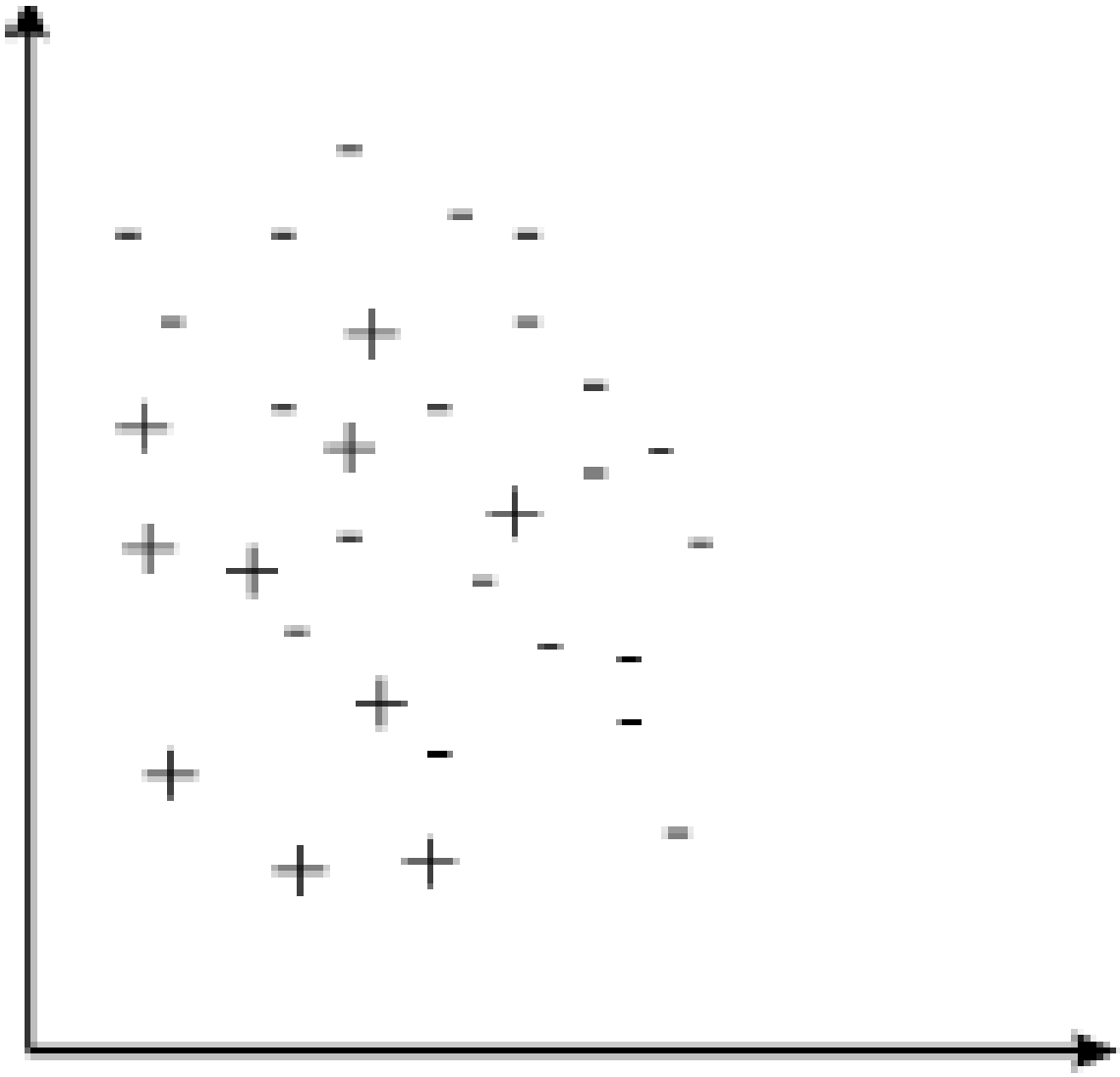


Figure 2.6(b): a balanced dataset after CNN

i. CNN + Tomek Link

Here, the learner will first select a consistent subset of the negative examples using CNN and then use Tomek link to remove them. The training set becomes more balanced. The objective is to evaluate with OSS as finding Tomek link is computationally demanding, it would be computationally cheaper if it was performed on a reduced data set (Batista *et al.*, 2004).

2.6.1.2 Over-sampling schemes

This is a replication of minority class examples (Thai-Nghe *et al.*, 2010), but can increase the likelihood of occurring over fitting and time consuming for the learning process (Guo *et al.*, 2008). It creates a superset of the original dataset replicating some instances or creating new instances from existing ones (Fernandez *et al.*, 2011). There are two types to this technique; Random and Informed over sampling.

a. Random OverSampling (ROS)

This is the continuous replication of the minority class at random until a more balanced or desired distribution is reached.

b. Informed oversampling

This technique intelligently picks data point from the minority class to be oversampled rather than picking them at random. This technique includes:

i. SMOTE (Synthetic Minority Oversampling TEchnique)

This technique generates synthetic examples by operating in feature space rather than data space. The minority class is oversampled by taking each minority class sample and introducing synthetic examples along the line segments joining any/all of the *k* minority class nearest neighbours. This technique overcomes the over fitting problem and broadens the decision region of the minority class examples. Synthetic samples were generated in the following ways (Chawla *et al.*, 2002):

a. The difference between the feature vector (sample) under consideration and its Nearest Neighbour was taken.

b. This difference was multiplied by a random number between 0 and 1, and
c. Add the result of (b) above to the feature vector under consideration.

Synthetic samples for nominal variable were generated in the following way: (Chawla *et al.*, 2003)

a. Take the majority vote between the feature vector in consideration and its k-Nearest Neighbours for the nominal feature value.
b. Choose at random if there is a tie
c. Assign the value to the new synthetic minority class sample

ii. **Cluster-based oversampling (CBOS)**

This technique attempts to even out the between-class imbalance as well as the within-class imbalance. There may be subsets of the examples of one class that are isolated in the feature-space from other examples of the same class, creating a within-class imbalance. Small subsets of isolated examples are called small disjuncts. Small disjuncts often cause degraded classifier performance, and CBOS aims to eliminate them without removing data (Jo and Japkowicz, 2004).

iii. **ADASYN (ADAptive SYNtethic)**

The essential idea of ADASYN is to use a weighted distribution for different minority class examples according to their level of difficulty in learning, where more synthetic data is generated for minority class examples that are harder to learn compared to those minority examples that are easier to learn. As a result, the ADASYN approach improves learning with respect to the data distributions in two ways:

a. reducing the bias introduced by the class imbalance, and
b. adaptively shifting the classification decision boundary toward the difficult examples; therefore improving learning performance.

These two objectives are accomplished by a dynamic adjustment of weights and an adaptive learning procedure according to data distributions. The new

method was tested on six different dataset form University of California, Irvine (UCI) data repository and evaluate using ROC, Precision, Recall, F-measure and G-mean using Decision tree as base classifier. The result obtained from ADASYN was compared with SMOTE and the original dataset and shows that for the overall winning, ADASYN outperformed the other methods. The conclusion was that ADASYN can autonomously shift the classifier decision boundary to be more focused on those difficult to learn examples, therefore improving learning performance. (He *et al*., 2008).

iv. **Border_Line SMOTE**

This oversampling technique presents two new minority oversampling methods named Borderline- SMOTE1 and Borderline-SMOTE2, in which only the borderline examples of the minority class are oversampled. This technique selects minority examples which are considered to be on the border of the minority decision region in the feature-space and only performs SMOTE to oversample those instances, rather than oversampling them all or a random subset. It first finds out the borderline minority examples; then synthetic examples are generated from them and added to the original training set. Borderline-SMOTE2 not only generates synthetic examples from each minority samples that is on the decision border and it's positive nearest neighbours but also does that from its nearest negative neighbour in the majority class region. The TPR and F- Measure of the minority class was used as metric with four datasets from UCI data repository and Decision Tree classifier was applied. Borderline-SMOTE1 and Borderline-SMOTE2 were compared with SMOTE and ROS. The result of the experiment revealed that both new schemes behaved excellent but Borderline-SMOTE2 was super on TPR because it generated synthetic examples from both the minority borderline examples and their nearest neighbours of the majority class, however, the procedure caused overlap between the two classes, thus decreases its F-value to some extent. The conclusion was that experiments indicated that the two new methods behaved better, which validated the efficiency of the methods. Some of the future recommendations were to

combine the new methods with under-sampling methods and integrate the new methods to some data mining algorithms (Han *et al*., 2005).

2.6.1.3 Advanced Sampling

This is also called Hybrid Sampling methods. It combines both oversampling and under-sampling methods to achieve better classification. It also adds the advantage that the dataset can be balanced without losing too much information or loses too much information (Ding, 2011). This type of technique includes SMOTE + ENN, SMOTE + Tomek link and CNN+ TL (Batista *et al.*, 2004).

2.6.2 Solution at the algorithm level

At the algorithm level, solutions try to adapt existing classifier learning algorithm to strengthen learning with regards to the minor class solutions. This is achieved either by:

2.6.2.1 Adjusting algorithm itself

This is also the same as adjusting the decision threshold. This technique forces decision making of the classifier to be biased towards the expensive class that is minority class (Boontarika and Maythapolnun, 2011). The classifier is manipulated internally to solve the class imbalance problem but it is not re-usable for another application domain (Guo *et al.*, 2008). Examples: For Support Vector Machine (SVM), few attempts have dealt with the imbalanced training-data problem (Karakoulas and Taylor, 1999; Lin *et al*., 2002; Veropoulos *et al*., 1999) while Lin *et a*l., (2002); Veropoulos *et al*., (1999) and Wu and Chang, (2003) used different penalty constants for different classes of data.

2.6.2.2 One-Class Learning

Also called recognition-based learning which learn examples mainly or only from one class rather than two-class (discrimination-based). It guarantees that some rules will be learned for minority class (Zhang and Mani, 2003). Japkowicz *et al*. (1995) developed a recognition based Multi Layer Perceptron (MLP) for unbalanced dataset. In this case, modelling is performed using examples from the positive (minority) class only and the one–class model often performs reasonably (Raskutti and Kowalyczyk, 2004). The problem of one-class classification is harder than the standard two-class classification problem. In two class classification, when examples of majority and minority are both

available, a decision boundary is supported from both sides by examples of each of the classes. In the case of one-class classification only the minority class is available, just one side of the boundary is supported. Based on the examples of one class only, it is hard to decide how tight the boundary should fit around the minority class. The absence of majority examples makes it also very hard to estimate the classification error (Juszczack and Duin, 2003). Examples are one- class SVM (Wang *et al*, 2009, Chawla *et al*., 2004, Guo *et al.*, 2008, Manevitz and Yousef, 2001).

2.6.2.3 **Cost Sensitive Learning (CSL)**

This technique incorporates cost in the decision making of classification. This is achieved by adjusting the cost of various classes to counter the class imbalance. Most classifiers assume that the mis-classification costs are the same (Thai-Nghe. *et al.*, 2009). This assumption is not correct. For example, the cost of mis-classifying a terrorist as a non-terrorist is higher than mis-classifying non- terrorist as terrorist. Cost could be money, a waste of time, or even severity of an illness (Nguyen, 2011; Elkan, 2001). The purpose of CSL is to build a model with minimum mis-classification cost. The costing is achieved using a cost matrix which encodes the penalty of classifying samples from one class to another (Sun, 2007; Galar *et al.*, 2012).

This method is most direct for dealing with highly skewed class distribution with unequal mis-classification cost (McCarthy *et al.*, 2005). A cost sensitive learner can accept cost information from a user and assign different costs to different type of mis-classification errors. But not all learners are cost sensitive. Altering the class probability thresholds used to assign the classification value and rebalancing the proportion of the positive and negative training examples in the training set are the commonest method of implementing CSL in learners (Elkan, 2001, McCarthy *et al.*, 2005, Domigos, 1999, Liu and Zhou, 2006). Cost matrix of a binary class presented in Table 2.2 corresponds to the confusion matrix presented in Table 2.1 and it will provide the costs associated with the four outcomes in the confusion matrix denoted by cost of true positive (C_{TP}), cost of false positive (C_{FP}), cost of false negative (C_{FN}) and cost of true negative (C_{TN}). With CSL, no cost is assigned to correct classification such that $C_{TP} = C_{TN} = 0$. The cost assigned to the positive (minority) class is often higher than the negative (majority) class since the positive is the class of interest ($C_{FN} > C_{FP}$). It could also be extended to multiple class problems.

The disadvantages of CSL include:

a. Many learning algorithms are not cost sensitive.
b. Mis-classifications cost are often not known
c. AdaCost (Sun, 2007), CSB1 MetaCost (Domigos, 1999) and AdaC2 (Sun et al., 2006) are some few examples of CSL.

2.6.2.4 **Ensemble Learning**

Combining classifiers could also be referred to as committee of learners, mixture of experts, classifier ensembles, and multiple classifier systems consensus theory (Kuncheva and Whitaker, 2003). They combine the power of multiple (usually weak) classifiers on similar datasets to provide accurate predictions for future instances (Hoens and Chawla, 2010).

The basic idea is to construct several classifiers from the original data and then aggregate their predictions when instances are presented and this improves the generalisation ability of each classifier: each classifier is known to make errors, but since they have been trained on different datasets or they have different behaviours over different part of the input space, mis-classified examples are not necessarily the same. To generate a model, several classifiers, called base classifiers, are trained, and they can be constructed from different classification algorithms, which make up a heterogeneous, or from the same algorithm, which result in a homogeneous ensemble (Boontarika and Maythapolnum, 2011).

Table 2.2: Cost Matrix

	Actual negative	Actual negative
Predicted negative	$C(0,0) = C_{TP}$	$C(0,1) = C_{FP}$
Predicted positive	$C(1,0) = C_{FN}$	$C(1,1) = C_{TN}$

A heterogeneous ensemble includes classifiers of various learning algorithm which gives diversity to the model. In a homogeneous ensemble, diversity is introduced by training different classifiers with different sets of data (Singhi and Liu, 2005). The training data is often varied in such a way as to give each classifier a (slightly) different dataset so as to avoid over fitting. There are several training parameters and factors which can be manipulated to create ensemble members and these according to Sun (2007) include:

a. The initial condition
b. The training data
c. The architecture of the classifier and
d. The training algorithm.

2.6.2.4.1 Why Ensemble Might Be Better Than Single Classifier?

There are three fundamental reasons that make ensemble to be better than single classifier. These, according to Dietterich (2000) include among other things, the following:

a. The Statistical Problem

This arises when the hypothesis space is too large for the amount of available data. Hence, there are many hypotheses with the same accuracy on the data and the learning algorithm chooses only one of them. There is a risk that the accuracy of the chosen hypothesis is low on an unseen data.

b. **The Computation Problem**

An ensemble constructed by running the local search from many different starting points may provide a better approximation to the true unknown function than any of the individual classifier.

c. The Representational Problem

It arises when the true function *f* cannot be represented by any of the hypotheses space.

2.6.2.4.2 Ensemble Methods

This refers to collection of classifiers that are minor variants of the same classifier, whereas "multiple classifier systems" is a broader category that also includes those combinations that consider the hybridization of different models (Galar *et al.*, 2012). When forming ensembles, creating diverse classifiers (but maintaining their consistency with the training set) is a key factor to make them accurate. A necessary and sufficient condition for an ensemble of classifiers to be more accurate than any of its individual members is if classifiers are accurate and diverse. One key to successful ensemble methods is to construct ensembles with error rates below 0.5 whose errors are least somewhat uncorrelated (Dietterich, 2000). Ensembles can either be homogeneous, in which every base classifier is constructed with the same algorithm, or heterogeneous, in which different algorithms are used to learn the ensemble members (Chawla and Sylvester; 2007, Gilpin and Dunlavy, 2009).

2.6.2.4.3 Methods of combination of ensembles

The following are some of the methods of combination of ensembles.

a. **Bayesian Voting**

This primarily addresses component of ensembles. The Bayesian Committee is not optimal as it does not address the computational and representational problems in any significant way.

b. **Manipulating the Training Examples**

The single learning algorithm is run several times each time with a different subset of the training examples. This technique works best especially for unstable learning algorithms whose output classifier undergoes major changes in response to small changes in the training data. Examples are Bagging (Bootstrap AGGregatING) (Breinman, 1996), Boosting (Freud and Schapire (1995, 1996), Cross Validated Committee (Parmanto *et al.*, 1996).

c. **Manipulating the Input Features**

This technique creates different subsets of the input features of the training set available to the learning algorithm. The resulting ensemble will be accurate and

diverse but only works when the input features are highly redundant. Examples are Decision Forest (Ho, 1998), Rotation Forest (Rodriguez *et al.*, 2006), Random Forest (Breinman, 2001).

d. **Manipulating the Output Targets**

This method manipulates the *y* values given to the learning algorithm. Examples are Error Correcting Output Coding methods (Dietterich and Bakiri, 1995) and AdaBoost.OC (Schapire, 1997).

e. **Injecting Randomness**

This is another method of ensemble creation. When randomness is injected into the classifier, the resulting classifier will be different and will be applied to the same dataset.

2.6.2.4.4 Diversity in Ensembles

This is the degree to which classifiers make different decisions on one problem. It allows voted accuracy to be greater than that of single classifier. Generally, larger diversity causes better recall for minority, but worse for minority classes. The best F-measure and G- mean value do not appear at the status with high accuracy/low diversity or the status with low accuracy/high diversity. Proper diversity degree results in better performance (Wang *et al.* 2009).

2.6.2.4.5 Measure of Diversity

There are different measures of diversity of ensembles. Ten statistics were studied (Kuncheva and Whitaker, 2003) which can measure diversity among binary classifier outputs (correct or incorrect vote for the class label). Four averaged pairwise measures are Q-Statistics, The Correlation Co-efficient,ρ, The Disagreement Measure and the Double-Fault measure. The six non- pairwise measures are the Entropy Measure E, the Measure of "Difficulty"; θ, the Kohavi-Wolpert variance, the Measure of Inter rater agreement κ, the Generalised Diversity, and the coincident failure diversity

2.7 LEARNING ALGORITHMS

This section presented the various learning algorithms or learner or classifier used in this study

2.7.1 Random Forest

This algorithm is a refinement of bagged trees to construct a collection of decision trees with controlled variations. This method combines Breinman's bagging and random subspace methods. This algorithm improves on bagging by de-correlating the trees. It grows the trees in parallel independently of one another (Chandrahasan *et al*., 2011). A random forest consists of a collection of tree-structured classifier $\{h(x, \theta_k), k = 1 \ldots n\}$ where the $\{\theta_k\}$ are independent identically distributed random vectors and each tree casts a unit vote for the most popular class at input x. This is a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest. The generalisation error for forests converges to a limit, as the number of trees in the forest becomes large. The framework in terms of strength of the individual predictors and their correlations give insight into the ability of the random forest to predict (Breinman, 2001).

2.7.2 Random Subspace Method (Decision Forest)

This is an ensemble of decision tree based classifier that maintains the highest accuracy on training data and improves on generalisation accuracy as it grows in computational complexity. The classifier consists of multiple trees contrasted systematically by pseudo randomly selecting subsets of components of the feature vector, that is, trees constructed in randomly chosen subspaces. In each pass, such a selection is made and a subspace is fixed where all points have a constant value (say, zero) in the unselected dimensions. All samples are projected to this subspace, and a decision tree is constructed using the projected training samples. In classification, a sample of an unknown class is projected to the same subspace and classified using the corresponding tree. For a given feature space of n dimensions, there are 2^n such selections that can be made, and with each selection a decision tree was constructed. Decision trees were generated using only the selected feature components. Each tree generalises classification to unseen points in different ways by invariances in the unselected feature dimensions. Decisions of the trees were combined by getting the average of the estimates of posterior probabilities at the leaves (Ho, 1998).

2.7.3 **Random Committee**

This classifier builds an ensemble of randomizable base classifiers (Random Tree). Each base classifier is built using a different random number seed (but based on the same data). The final prediction is a straight average of the predictions generated by the individual base classifiers (Bouckaert et al, 2010).

2.7.4 **MultiClass Classifier**

This is a meta classifier for handling multiple class with 2-class classifiers. This classifier is capable of applying OVO, OVA and error correcting output codes on datasets for increased accuracy.

2.7.5 **Boosting**

This algorithm uses the whole dataset to train each classifier serially, but after each round, it gives more focus to difficult instances, with the goal of correctly classifying examples in the next iteration that were incorrectly classified during the current iteration. Hence, it gives more focus to examples that are harder to classify, the quantity of focus is measured by a weight, which initially is equal for all instances. The boosting algorithm takes as input a training set of m examples $S = (X_1, Y_1, \ldots X_m Y_m)$ $where$ X_i is an instance drawn from some space X and represented in some manner (typically, a vector of attribute values), and $Y_i \in Y$ is the class label associated with X_i. The boosting algorithm has access to another unspecified learning algorithm, called the weak learning algorithm, which is denoted generically as WeakLearn. The boosting algorithm calls WeakLearn repeatedly in a series of rounds. On round t, the booster provides WeakLearn with a distribution D_t over the training set S. In response, WeakLearn computes a classifier or hypothesis $h_t: X \rightarrow Y$ which should correctly classify a fraction of the training set that has large probability with respect to D_t. That is, the weak learner's goal is to find a hypothesis h_t which minimizes the (training) error $\varepsilon_t = pr_i{\sim}D_t[h_t(x_i) \neq y_i]$. It should be noted that this error is measured with respect to the distribution D_t that was provided to the weak learner. This process continues for T rounds, and, at last, the booster combines the weak hypotheses $h_1 \ldots h_T$ into a single final hypothesis h_{fn} (Freud and Schapire, 1996 and 1999).

2.7.6 Stacking

This is an ensemble method by which different classifiers could be combined using another classifier at a meta-level. The outputs from base classifier and the corresponding true class labels would be used as the new dataset of learning in the meta-level. This dataset will be used to construct a meta classifier in order to learn any existing pattern of mis-classification by base classifiers (Singhi and Liu, 2005). This is done by partitioning the data set into a held-in data set and a held-out data set; training the models on the held-in data; and then choosing whichever of those trained models performs best on the held-out data. This is the cross-validation technique. Stacking exploits this prior belief further. It does this, by using performance on the held-out data to combine the models rather than choose among them, thereby typically getting performance better than any single one of the trained models.

2.7.7 Repeated Incremental Pruning to Produce Error Reduction (RIPPER)

This class implements a propositional rule learner, Repeated Incremental Pruning to Produce Error Reduction (RIPPER), which was proposed by William W. Cohen as an optimized version of Incremental Reduced Error Prunning (IREP) (Cohen, 1995). RIPPER is a program for inducing sets of classification rules. Each rule is a conjuction of conditions on attribute values. Rules are returned as an ordered list and the first rule that evaluates to true is used to assign the classification.

2.7.8 Boostrap AGGregatING (BAGGING)

The concept of bagging is to construct an ensemble which consists of X training different classifiers with boostrap replicas of the original training dataset. A new dataset is formed to train each classifier by randomly drawing (with replacement) instances from the original dataset (usually maintaining the original dataset size). Hence, diversity is obtained with the re-sampling procedure by the usage of different data subset. Finally, when an unknown instance is presented to each individual classifier, a majority or weighted vote is used to infer the class (Breinman, 1996).

2.7.9 Support Vector Machine (SVM)

Support Vector Machine (SVM) is one of the binary classiffiers based on maximum margin strategy introduced by Vapnik and Lerner, 1963. Originally, SVM was for linear

bi-class classiffication with margin, where margin means the minimal distance from the separating hyper plane to the closest data points. SVM seek an optimal separating hyper plane, where the margin is maximal. The solution is based only on those data points at the margin. These points are called as support vectors. The linear SVMs have been extended to nonlinear examples when the nonlinear separated problem is transformed into a high dimensional feature space using a set of nonlinear basis functions. However, the SVMs are not necessary to implement this transformation to determine the separating hyper plane in the possibly high dimensional feature space. Instead, a kernel representation can be used, where the solution is written as a weighted sum of the values of a certain kernel function evaluated at the support vectors (Sun, 2007). The kernel function is thus the key component in this approach. Gaussian radial basis functions and polynomial kernel functions are often used in practice. When perfect separation is not possible, slack variables are introduced for sample vectors to balance the tradeoff between maximizing the width of the margin and minimizing the associated error.

2.7.10 Artificial Neural Network (ANN)

Neural networks have the topology of a directed graph and loosely simulate the structure of biological neural networks in human brains. They are composed of processing nodes that transfer activities to each other via connections. These one-way inter-unit connections hold the processing ability of the network through weights obtained by learning from a set of training data. Each node evaluates the input values, calculates a total for the combined input values, compares the total with a threshold value, and determines what its own output will be. A neural network's learning is defined as changes in the memory weight matrix. There is a variety of strategies to train the network, including applications of numerical and statistical methods such as back propagation errors, differential equations, least-squares fitting and others (Sun, 2007). Back propagation network is a feed-forward network with one input layer with many inputs, one output layer with many outputs, and one or more hidden layers. The activation function of a hidden node is often a sigmoid-function. Reported experimental results by Japkowicz and Stephen 2002 indicated that the back propagation performed deficiently with imbalanced data sets. The main reason is the small class is inadequately weighted in the network.

2.7.11 *k*- Nearest Neighbour

k-Nearest Neighbour is an instance-based classifier, which uses specific training instances to make predictions without having to maintain a model derived from data (Aha *et.al*., 1991). The conceptual idea of the *k*-Nearest Neighbour algorithm is simple and intuitive. Given a test sample, the algorithm computes the distance (or similarity) between the test sample and all of the training samples to determine its *k*-nearest neighbours. The class of the test sample is decided by the most abundant class within the *k*-nearest neighbour samples. In the presence of the imbalanced training data, samples of the small classes occur sparsely in the data space. Given a test sample, the calculated k-nearest neighbours bear higher probabilities of samples from the prevalent classes. Hence, test cases from the small classes are prone to being incorrectly classified (Sun, 2007).

2.7.12 Reduced Error Pruning (REP tree)

This is a fast decision tree learner that builds a decision/regression tree using information gain/variance and prunes it using reduced-error pruning (with back fitting). Only sorts values for numeric attributes once. Missing values are dealt with by splitting the corresponding instances into pieces (i.e. as in decision tree).

2.9 Critical appraisal and comparison of the under sampling techniques

This section takes a critical appraisal of various under sampling techniques reported in the literature against issues in their underlying data reduction techniques.

One of the techniques of alleviating class imbalance learning is based on prototype selection/data pre-processing or data reduction (Garcia *et al.*, 2012). Most under sampling techniques are modifications of classic prototype selection methods to balance the dataset (Batista *et al*., 2004). These under sampling techniques aimed at obtaining a representative training set with a lower size compared to the original one and with similar or even higher classification accuracy for new incoming data. A formal specification of the under samples problem is as follows: let S $\subseteq$ Training Set be the subset of the selected samples resulting from the execution of an under sampling algorithm, then one classifies a new pattern X_j from reduced dataset by the K-nearest neighbour rule acting over S than of the Training Set (Garcia *et al.*, 2012).

2.9.1 **Nearest Neigbhours (NN)**

Distance is evaluated from all training points to sample point and the point with the lowest distance is called Nearest Neighbour (NN) (Bathia and Vandana, 2010). Given a set of previously labeled training set (TS), NN rule assigns a sample to the same class as the closest Neighbour in the set, according to a similarity/distance in the feature space (Vazquez *et al.*, 2005; Dasarathy, *et al*., 2000). The basic NN algorithm retains all of the training instances in dataset, hence, requires relatively large storage, high computational complexity and low noise tolerant (Garcia *et al.*, 2012). It learns very quickly (O(n) time) for it only needs to read in the training set without further processing (Wilson and Martinez, 1997c) and generalizes accurately for most applications (Wilson and Martinez, 2000). Also, noisy instance are stored as NN stores all instances in the training set (TS) which degrades generalization accuracy. Two broad groups techniques of under-sampling reported in the literature were when the algorithm tries to remove erroneously labeled dataset and also "clean" the possible overlapping between regions of different classes referred to as Editing (Vazquez *et al.*, 2005). The second group, which is aimed at selecting the minimal subset of training set (TS) but also lead to the same performance as the NN rule using the whole training set (TS) referred to as Condensing (Dasarathy *et al.*, 2000). These techniques include: Reduced Nearest Neighbour (RNN), Selective Nearest Neighbour (SNN), Condensed Nearest Neighbour (*CNN*), One-Sided Selection (OSS), Neighbour Cleaning Rule (NCL) and Tomek Link.

2.9.2 **Properties of Under-samplingTechniques**.

According to Wilson and Martinez (2000, 2000b), the properties of under sampling techniques includes the following:

i. **Representation**

This factor seeks to retain a subset of the original instances (collection of training examples). Representation methods could be hyper-rectangles, rules or prototypes (Dasarathy *et al.*, 2000). Choices to be made when designing training set for under sampling algorithm are whether to under sample the original TS into a subset or to modify using a new representation. The problem with using the original dataset is that there is difference in the class distribution.

ii. Direction of search

When searching to form a subset or modify training set (TS) to keep from the TS, various direction of search were involved. These include the under listed factors:

a. Incremental search

This begins with an empty subset S, and adds each instance of T to S, if it meets some criteria. The advantage here is that instances can be added to S continuously with the same criteria after training is completed. Another advantage is that they are faster and uses less storage during learning. However, they are prone to errors: initial criteria were based on limited information, thus sensitive to order of presentation. e. g. CNN.

b. Decremental search

Here, all examples are available for examination, so a decision can be made on which instance is best to be removed during learning. This search begins with S = T; then searches for instances to remove from S. Examples of algorithm belonging to this search methods are RNN, SNN, ENN. Though, it can result in greater storage reduction but is often more computationally expensive.

c. Batch

All examples in the TS are available for examination. The examples that meet the removal criteria are removed from the TS at once. The algorithm is relieved from having to constantly update the list of nearest neighbours when instances are removed individually, but it suffers from increased time complexity.

iii. Border points Versus Central points

There are four types of negative examples to be removed when under sampling/reducing training set (Kubat and Matwin, 1997): Those that suffer from the class-label, borderline examples that are close to the boundary between the positive and negative regions. They are unreliable: even a small amount of attribute noise can send the example to the wrong side of decision surface, those that are redundant so that their part can be taken over by other examples and safe examples

that are worth being kept for future classification tasks. The decision to retain border point, central or some other point distinguishes one under sampling technique from another. Border points forms decision boundaries between classes so, removing them leaves smooth decision boundary behind. Noisy points are point that does not agree with their neighbours. 'Internal' or centre points do not affect decision boundaries so much and thus their removal will have relatively little effect on classification.

iv. Similarity (Distance) function

This is the distance between neighbours. It is used to decide which neighbours are closest to an input vector and can have a drastic effect on learning algorithm. The training points are assigned weights according to their distance from sample data point (Bathia and Vandana, 2010). Some examples of this function are:

a. Linear Distance function

It measures the distance between two input vectors with their number of attribute/variables (Wilson and Martinez, 2000). It cannot handle instances with both linear and nominal attributes. Examples include Euclidean distance function and Mahalanobis.

b. **Value Difference Metric for Nominal Attributes** (Stanfill and Waltz, 1986)

It is suitable for nominal attributes but inappropriate for direct use on continuous attributes. Example is Value Difference Metric (VDM).

c. **Interpolated Value Difference Metric** (Wilson and Martinez, 2000)

This function is appropriate for linear (discrete, but ordered), continuous (real valued) and nominal (discrete, unordered) attributes. Known example is Interpolate Value Difference Metric (IVDM).

d. **Heterogeneous Distance function** (Wilson and Martinez, 1997)

This handles application with both continuous and nominal attributes. Common examples are the Heterogeneous Euclidean-Overlap Metric (HOEM) and Heterogeneous Value Difference Metric (HVDM).

v. Voting

k is the number of neighbours used to decide the output of a class of an input vector. The under sampling algorithm has to decide on value of *k* which is typically a small, odd integer (1, 3, or 5). Cover and Hart (1967) proposed a *k*-NN rule in which NN is calculated on the basis of value of *k* that specifies how many NN are to be considered to define class of a sample data point.

vi. Evaluation strategies

Relative strength and weaknesses of each under-sampling algorithm should be compared based on a number of characteristics namely; storage reduction, speed increase, generalization accuracy, noise tolerance, learning speed (Wilson and Martinez, 2000, Wilson and Martinez, (2000b), and algorithm complexity (Jankowski and Grochowski, 2004).

vii. Types of selection

This factor only classifies the technique into Condensation, Edition or the mixture of the two algorithms as represented below:

a. Condensing

This technique aims at selecting a sufficiently minimal subset of training instances without a significant degradation of accuracy (Sanchez, 2004). It includes the techniques which aim to retain the points which are closer to the decision boundaries, also called border points. The intuition behind retaining border points is that internal points do not affect the decision boundaries as much as border points, and thus can be removed with relatively little effect on classification. Nevertheless, the reduction capability of condensation methods is normally high due to the fact that there are fewer border points than internal points in most of the data but can often result in marginally poorer classification/recognition performance (Dasarathy *et al*., 2000).

b. Edition

They remove points that are noisy or do not agree with their Neighbours. This removes close border points, leaving smoother decision boundaries behind.

However, such algorithms do not remove internal points that do not necessarily contribute to the decision boundaries. The effect obtained is related to the improvement of generalization accuracy in test data, although the reduction rate obtained is lower. However, the focus of this algorithm is not on reducing the training set but on defining a high quality training set by removing outliers (sanchez, 2004).It 'cleans' possible overlapping among region from different classes.

c. Hybrid

Hybrid methods try to find the smallest subset *S* which maintains or even increases the generalization accuracy in test data. To achieve this, it allows the removal of internal and border points based on criteria followed by the two previous strategies. The k-NN classifier is highly adaptable to these methods, obtaining great improvements even with a very small subset of instances selected.

2.9.3 Comparison of under-sampling Technique

A comparison of all under-sampling techniques (RNN, SNN, CNN, OSS, ENN, and NCL) with their characteristics were compared and presented in Table 2.3. They all shared similar advantages that is, reduction in the size of the training data, improvement in query time, low memory requirement and reduction in the recognition rate. Their drawbacks are a high computational complexity, high cost and time consuming.

The Reduced Nearest Neighbour (RNN), Selective Nearest Neighbour (SNN), Condense Nearest Neighbour (CNN) and One-Sided Selection (OSS) all attempted to create a subset which is a smaller version of the original dataset/training samples.

Wilson's Edited Nearest Neighbour (ENN) and Neighbourhood Cleaning Rule (NCL) presented their reduced dataset as a modified version of the original dataset while All K-NN presented its reduced version as a mixture of both subset and modification.

The direction of search for samples to under sample could be in the form of incremental, decremental and batch mode. While RNN, SNN, ENN and NCL performed a decremental search for samples to be removed, CNN searches the training set in the incremental mode

while All K- NN batches all the training samples before searching. ENN, NCL and All K-NN served more as a "cleaning agent" rather than data reduction. Hence, their popular use in Class Imbalance Problem (CIP). They attempted to remove noisy, erroneous and border point that could hinder classification performance. Both NCL and All k-NN were modifications of ENN. CNN, RNN and SNN served to effectively reduce the dataset by removing samples from the dataset but a consistent subset is not guaranteed. OSS is the combination of All K-NN and CNN.

ENN and NCL used Heterogeneous Value Distance Metric (HVDM) to detect its nearest neighbours while RNN, SNN, *CNN*, OSS and All K-NN detected its nearest neighbour, using Euclidean distance function.

All the under sampling technique used 1–NN, except ENN and NCL which performed removal of samples from the dataset with 3-NN. RNN, SNN and CNN selected samples to be removed from the training set by condensing the training set while ENN and NCL selected prototypes by editing the training set. Since OSS is the combination of CNN and All K–NN, it inherited the properties of the techniques by condensing the training set before editing the remaining samples.

Table 2.3: Comparison of under-sampling technique

Properties	**RNN** (Gates,1972)	**SNN** (Ritters *et al*., 1975)	**ENN** (Wilson,1972)	**CNN** (Hart,1968)	**ALL K-NN** (Tomek,1976)	**OSS** (Kubat& Matwin,1997)	**NCL** Laurikala, (2001)
Presentation	Subset	Subset	Modified	Subset	Mixed	Subset	Modified
Direction of search	Decremental	Decremental	Decremental	Increamental	Batch	Batch	Decremental
Border versus Point	Noisy and Internal point	Internal point	Noisy and Border point	Internal point	Noisy and border point	Noisy and border point	Internal and border point
Distance function	Euclidean	Euclidean	HVDM	Euclidean	Euclidean	Euclidean	HVDM
Voting	1-NN	1-NN	3-NN	1-NN	1-NN	1-NN	3-NN
Algorithm Complexity	O (n^3)	O (n^3)	O (n^2)	O (n^3)	O (n^2)	O (n^3)	O(n^2)
Type of Selection	Condensing	Condensing	Editing	Condensing	Editing	Editing and Condensing	Editing

2.10 Related work

This section presents the review of existing work done by researchers to alleviate the class imbalance problem in specific domains.

Kubat and Matwin (1997) discussed the criteria to evaluate the utility of classifiers induced from such training sets, give explanation of the poor behaviour of some learners under these circumstances and suggest as a solution a simple technique called OSS of examples. They combined two under-sampling schemes to reduce original dataset to a consistent subset. They first applied Tomek Link (TL) to remove noisy and border line examples from the dataset. Then, Condense Nearest neighbour (CNN) was applied to remove redundant examples to create a consistent subset of the original dataset. Accuracies on both minority and majority classes, average accuracy on both classes and geometric mean was used to measure the performance on 1-NN and Decision Trees classifiers. The domain on the schemes were applied significantly profited from the scheme. The sensitivity of imbalanced distribution of examples can be mitigated by OSS scheme and should be applied only if values of the accuracies of either the majority or minority class are low. The solution developed addresses only two class problem. The combination of under-sampling reduces the dataset greatly thereby removing too much information.

Barandela *et al*. (2003b) conducted research to explore issues related to the class imbalance problem. They focused resampling the dataset and also on internally biasing the discrimination-based process, as well as on a combination of them. The solution developed was evaluated over four real datasets using Nearest neighbour (NN) classifier and ROC and geometric mean as metric to measure performance. Editing schemes: Wilson's Edited Nearest Neighbour (WE) and *k*-Nearest Centroid Neighbour (NCN), were used to delete noisy examples from majority class alone. Condensing scheme: Modified Selective (MS) was combined with WE and *k*-NCN respectively to also downsize only the majority classes of the dataset and also to both classes. Also, a weighted distance function is assigned to respective classes and not individual examples to internally bias the discrimination procedure during classification. So, WE, WE+MS, *k*-NCN, *k*-NCN+MS and MS schemes were used to under sample only the majority class, MS, WE and WE+MS schemes were used to under sample both classes and were learned on both discrimination and non-discrimination NN classifier. The weighted distance procedure (discrimination-based)

produced an improvement in performance measure. Also that repeated application of editing shows similar or better results than those of the single editing.

Batista *et al.* (2004) performed experimental evaluation involving ten sampling techniques (ROS, RUS, TLink, CNN, OSS, NCL and SMOTE) and methods: CNN+TL, SMOTE + TL and SMOTE + ENN schemes were proposed by the author as a data cleaning process after oversampling. This experiment was performed on 13 dataset from UCI data repository. The Decision Tree learner was used with Pruning and non-Pruning properties. ROC_AUC was used as the performance metric and Hsu's Multiple Comparison with the Best (MCB) was used for statistical analysis. Also, the number of rules/branches and mean number of conditions per rule were reported for the original and oversampled datasets. Two of the proposed methods are generally ranked among the best for datasets with fewer minority classes. They concluded that datasets with larger number of minority class samples should be treated with ROS as it will also produce meaningful results. They recommended investigation into why allocating half of the training examples to the minority class does not always provide optimal classification results. Further, Tomek Links and NCL that do not allow user to specify the resulting class distribution should be improved upon.

Lessmann (2004) aimed at improving the detection of respondents of mailing campaign (response optimisation) in Customer Response Management (CRM) which is a class imbalance problem using SVM. He first evaluates SVM's capabilities of handling this problem internally by adjusting its parameterization (kernel). Then, combining all parameter settings for the linear and Gaussian SVM to obtain a total of 130 experiments. Secondly, he resampled the dataset using ROS and RUS to 1:2 and 1:1 respectively and learned on SVM. F-Measure and Geometric mean were used as evaluation metrics. The result revealed that SVM can account for class imbalance through internal parametization within the model selection stage. SVM is also robust

Hulse *et al.* (2007) applied seven sampling techniques (RUS, ROS, OSS, CBOS, WE, SMOTE, and Border_line SMOTE) and also their variation to make 31 sampling techniques with 35 different benchmark datasets and 11 commonly used learning algorithm (C4.5N, C4.5D, IB2, IB5, NB, MLP, RBF, RIPPER, RF, LogisticRegression

and SVM). Statistical analysis was performed using a 1-way ANOVA to understand the statistical significance of the results obtained and Tukey's Honestly Significant Difference (HSD) was used to create homogenous subsets. The objective of the study was to present a comprehensive and systemic experimental analysis of learning from and providing practical guidance to machine learning practitioners when building classifiers from imbalanced data. The data clearly demonstrated that sampling is often critical to improving classifier performance, especially optimising threshold-dependent measures such as the geometric mean, TPR, F-measure and Accuracy and ROC_AUC and Kolmogorov-Smirnov (K/S). It was concluded that RUS and ROS performed better than the rest especially CBOS which performed worst. Furthermore, that individual learner responds differently to the application of sampling.

Gu (2007) developed an effective scoring model to predict potential cross-sell take-ups but the dataset is imbalanced and the classes were overlapped. He proposed combination of random forest based techniques and sampling methods to identify the potential buyer. Firstly, the dataset was cleaned by the elimination of dangerous negative instances. The dataset were divided into the minor instances set *P* and the major instances set *N* where *N* was further divided into *n* subsets equal size. For each N_i, he trained a Random Forest from the rest instances in *N* and the entire *P*. The trick was that for every classification tree in the forest, the class distribution in the corresponding training data was not balanced, that is, more negative instances than positive instances. Then, all instances in N_i that were incorrectly classified by *RF* were removed. Secondly, he trained a variant of random forest for which each tree was based towards the positive class to classify the dataset where a majority voting was made for prediction. The proposed method was then compared with SMOTE, RUS, ROS, TLink and *CNN* on 8 UCI datasets to obtain ROC_AUC metric. The method was used to evaluate the customer dataset as well as 8 datasets from UCI data repository where the proposed scheme performed best on 4 of the UCI dataset when compared with standard sampling schemes. The proposed method achieved the best performance with the highest AUC score.

Nguyen *et al.* (2009) introduced a new learning approach that aimed at tackling the class imbalance problem. They first proposed a new under-sampling method based on clustering. The clustering technique was employed to partition the training instances of

each class independently into a smaller set of training patterns such that each cluster contains samples from the same class and each class can have several clusters. Then, a weight was assigned to each training prototype to address the class imbalance problem. The weighting strategy was introduced in the cost function such that the class distributions becomes roughly even. In the extreme imbalance cases, where the number of minority instances is small, unsupervised learning were used to resample only the majority instances, cluster centres were selected as prototype samples but all the minority class samples were kept. The proposed approach which combines unsupervised and supervised learning to deal with the class imbalance problem can be applied to any classifier. MLP classifier was applied to five imbalanced dataset from UCI data repository (Blake and Merz, 1998) and G-mean, F-measure, specificity and sensitivity were used as evaluation metric. Experimental results showed that the proposed approach can effectively improve the classification accuracy of the minority class, while maintaining the overall classification performance.

Awokola (2010) applied REP Tree, RIPPER, Ridor rules and Decision Tree classifiers to predict the presence of diabetes mellitus disease in patients in a teaching hospital. Accuracy metric was used to evaluate the classifiers. But the dataset is highly skewed and this was not put into consideration. The result showed that most of the diabetes patients had TYPE2 diabetes and were 40 years of age. The class imbalance in this dataset gave a sub-optimal performance during classification. The minority class, GDM, was not well detected by any of the classifiers.

Agboola (2010) applied Random Tree, Decision Tree, Decision Stump, Best First Decision Tree, Simple Classification And Regression Tree (CART), LogitBoost Alternating Tree (LADTree), NAIVEBAYES and Functional Tree to Senior Secondary School (SSS) Result examination result dataset to discover the reasons for the abysmal results recorded in the two examinations that were collected in five model secondary schools in Ibadan, Nigeria. Accuracy and RMSE metric were used for evaluation and pair t-Test and correlation were used to analyse the results obtained. Random tree performed best for having obtained the highest accuracy value and lowest RMSE. The conclusion is that the results of the two exams were significantly different and their result does not depend on

one another. But the dataset were highly skewed and the minority class was not predicted by any of the classification algorithm considered.

Georgescu *et al.* (2010) applied several techniques for data reduction (Principal Component Analysis (PCA), Partial Least Square (PLS), Structurally Random Matrices (SRM) and Orthogonal Matching Pursuit (OMR)) to publicly available datasets (IONOSPHERE, Wisconsin Breast Cancer and NASA) with the goal of comparing how an increasing level of compression affects the performance of SVM-type classifiers. However, these data reduction techniques cannot remove the class imbalance problem in an imbalanced dataset.

Hoens and Chawla (2010) aimed at overcoming the class imbalance problem by proposing an ensemble framework that combines random sub-space method with sampling schemes (SMOTE and RUS). This is a special case of RSM + sampling where SMOTE or RUS were used within each randomly selected subspace. In the RSM during the training phase each classifier is trained on a subset of the data in which some features were removed. After removing a subset of the features, SMOTE/RUS was then applied to the dataset, which was subsequently used to train the classifier. ROC metric was used for evaluation and comparison with Random Forest, Random Subspace, BAGGING Boosting (AdaBoost.M1). Friedman and Bonferroni-Dunn test were used as statistical tool. RSM+SMOTE performs significantly better than other classifiers followed by RSM+RUS. Since SMOTE is dependent upon the features, and in ensemble methods having classifiers with different biases is optimal, RSM+SMOTE provided better performance over other techniques. However, it is not recommended for low dimensional dataset (with fewer features e.g. 2).

Asha *et al.* (2011 and 2012) proposed the use of Classification based on Predictive Association Rules (CPAR), Predictive Rule Mining (PRM) and First Order Inductive Learner (FOIL) with Statistical test along with Laplace accuracy as rule evaluation measures with different testing modes. The performance of these methods on tuberculosis dataset were analysed with two classes; Pulmonary Tuberculosis (PTB) and Retroviral PTB (RPTB) that is those having TB with HIV. Though results obtained showed that

CPAR and PRM learned the dataset classes correctly, no consideration was given to the class distribution of the used dataset.

Johnson *et al.,* (2012) intended to model species' distribution with focus on the problem of class imbalance. The study focused on nine species of small to medium sized birds belonging to the Vireo genus prevalent in the North-eastern United States. Eight models were used for learning: both pruned and un-pruned Decision tree, Classification and Regression Trees (CART), Logistic Regression (LR), MAXimum ENTropy (MAXTENT: a method based specifically on the concept of the ecological niche), Naïve Bayes (NB), Hellinger Distance Decision Trees (HDDT), Random Forest (RF) and RF with SMOTE and evaluated using ROC_AUC, AUPR and mean Correlation. Distribution maps were used to display results geographically to show distributions predicted with several modelling methods. Results showed that though not dominant on any given species dataset, HDDT is handily the dominant performer with respect to ROC_AUC, AUPR and mean Correlation. They concluded that AUPR is a useful tool for evaluating species distribution models. Secondly, HDDT generally modelled species with performance competitive with MAXTENT, an established specie distribution model. They recommended the use of ROC_AUC and AUPR together as evaluation metric.

Nagabhushanam *et al*. (2013) explored the need to develop a data mining solution to make diagnosis of tuberculosis as accurate as possible and helps decide if it is reasonable to start tuberculosis treatment on suspected patients without waiting for the exact medical test results or not. They proposed the use of Sugeno-type "adaptive network-based fuzzy inference system" (ANFIS) to predict the existence of *Mycobacterium tuberculosis* (the causative agent of tuberculosis). They also implemented a MLP and PART learning model using the same data set and used RMSE to evaluate their performances but did not consider the class imbalance nature of the dataset.

Rahman and Davis (2013) examined the performance of over-sampling (SMOTE) and under-sampling (Cluster based) techniques to balance cardiovascular data. They modify Yen and Lee, (2009)'s cluster based under-sampling method by first separating the data into two subsets: majority and minority class samples. Then, the majority class is further separated into *K* clusters. The aim is to reduce the gap between the majority and minority

samples. All the majority sample clusters/subsets are separately combined with minority samples to make *k* different training sets and are then classified using decision tree and Fuzzy Unordered Rule Induction algorithm. The dataset with the highest accuracy were kept for further data mining process. Using ROC metric and accuracy, SMOTE shows good classification performance and in some cases, very close to the performance of the proposed method. The proposed method is found useful for datasets where class labels are not certain. However, only two classification algorithms were used on a single dataset.

Habibi *et al.*, (2015) examined a predictive model using features related to the diabetes TYPE2 risk factors. The diabetes dataset used was obtained from a database in a diabetes control system in Tabriz, Iran. They used Decision tree classifier to build the model, ROC_AUC and Kappa Statistics as the main evaluation measure and implemented in WEKA. The classification result was sub-optimal as the dataset was highly skewed.

Fattahi *et al.*, (2015) introduced a new ensemble based method consisting of SMOTE and Rotation Forest. They constructed classifiers with obtaining rotating subspaces of the original dataset using PCA. The study was evaluated using 20 binary imbalanced dataset from the KEEL dataset repository using RMSE, ROC_AUC, FNR and Kappa statistics as performance metrics. The Kappa-Error diagram was plotted for the analysis of the result obtained

2.11 Remarks

This chapter reviewed the Class Imbalance Problem, existing solutions and domains where it was shown that the class imbalance problem give sub optimal classification performance.

Two school of thoughts proposed solutions to alleviated Class Imbalance Problem. The first postulation applied solution at the data level which is external to classification algorithms. The major drawback at this level is that there is loss of information on the datasets and overfitting during classification.

The second school of thought believed that solution should be applied at the algorithm level. The major drawback at this level is that the classification algorithms are specific for that dataset and cannot be reused.

Thus, the research gap identified is that there is need for solution which will be external and portable to classification algorithm, prevent information loss, avoid over fitting, increases the RECALL of the minority class and gain in performance after applying the solution through the developed enhanced data sampling schemes.

CHAPTER THREE

Research Methodology

This chapter gives a detailed explanation of the methodology used in the study

3.1 The flow diagram of the data mining process used for the study is presented in Figure 3.1. The research data on Diabetes Mellitus (DM) disease were obtained from Wesley Guilds Hospital, Ilesha (Awokola, 2010), Senior Secondary School Result (SSS Result) examination results were obtained from West African Examination Council (WAEC) office in Ibadan (Agboola, 2010), Tuberculosis dataset was obtained from Ijaye State Hospital, Abeokuta and Contraceptive Methods (CM) dataset was obtained from health center, Ibadan North East Local Government, Ibadan. The obtained datasets were pre-processed with the both the existing and enhanced data sampling schemes before classification.

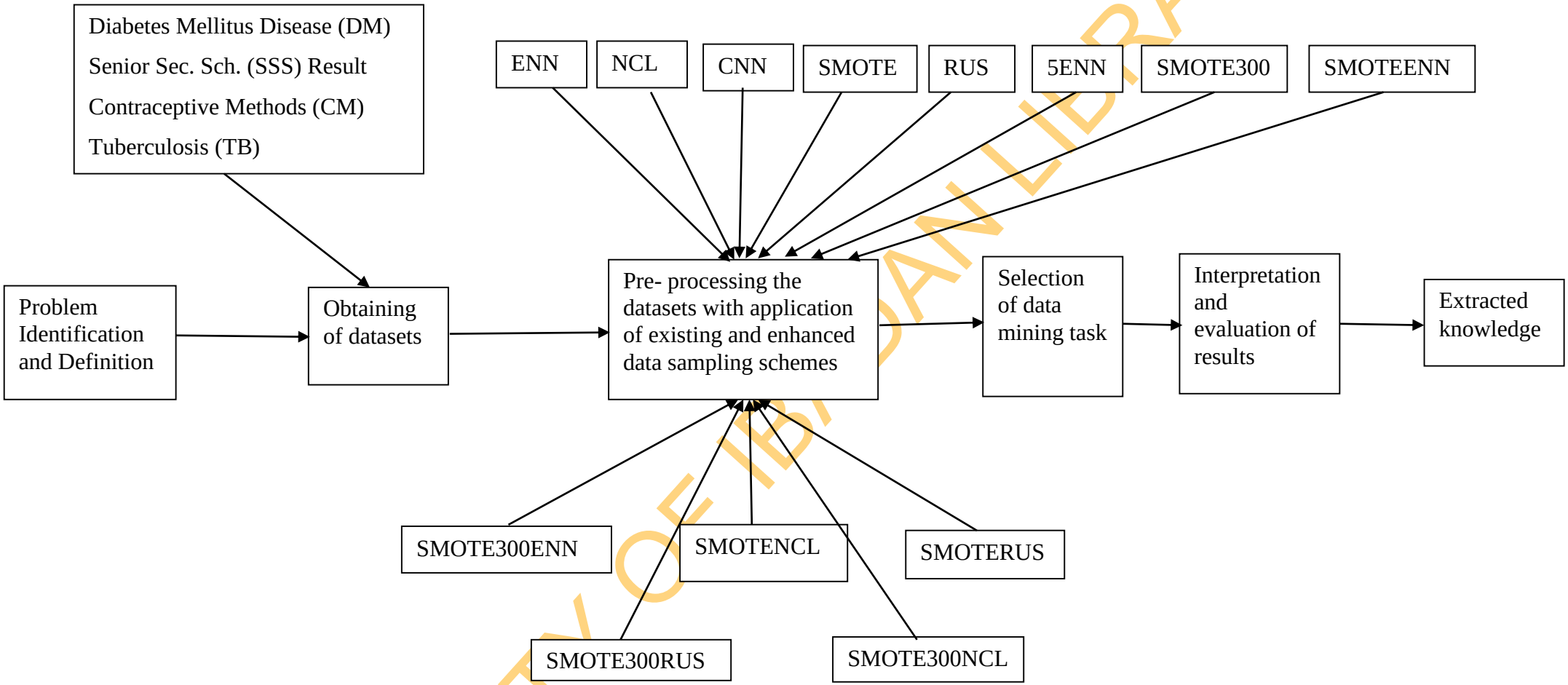


Figure 3.1 The research methodology

3.2 Model Development

This section presents the algorithm for enhanced sampling schemes developed in the study. The enhanced data sampling schemes namely SMOTE300ENN, SMOTENCL, SMOTERUS, SMOTE3OORUS and SMOTE300NCL were coded using Java programming language and implemented in Waikato Environment for Knowledge Analysis (WEKA) version 3.6.8 as additional filters available for use in the filter library.

3.2.1 The Enhanced Data Sampling Schemes Algorithms

Given a dataset, T, which consists of pair (x_i, y_i), where $i = 1,2, \ldots m$ *where* $x_i$denote the input attributes and y_i denote the class labels. T contains n instances with m attributes each and either belong to a positive (minority) or negative (majority) class. The minority class C which is also the class of interest is a subset of $y_i: C \subset y_i$. The minority class was oversampled by taking each minority class samples and introducing synthetic examples along the line segments joining any/all of the k nearest Neighbours. In this study, k = 5-Nearest Neighbour was used and the rate of oversampling used was 300%. The reason for this choice is that different rate of oversampling had been tried on various datasets with 300% being the best. The existing SMOTE uses 100% rate of oversampling. This technique generated synthetic examples in a less application specific manner, by operating in "feature space" rather than "data space".

Synthetic samples for continuous variable were generated in the following ways:

a. The difference between the feature vector (sample) under consideration and its Nearest Neighbour was taken.
b. This difference was multiplied by a random number between 0 and 1, and
c. Add result of (b) above to the feature vector under consideration.

Synthetic samples for nominal variable were generated in the following ways:

a. Take the majority vote between the feature vector in consideration and its k-Nearest Neighbours for the nominal feature value.
b. Choose at random if there is a tie
c. Assign the value to the new synthetic minority class sample

The illustration for this is as follows:

Consider a sample (6, 4) and let (4, 3) be its Nearest Neighbour.
(6, 4) is the sample for which *k*- Nearest Neighbors are being identified
(4, 3) is one of *k*-Nearest Neigbhour

Let:
f1_1 = 6 f2_1 = 4, f2_1 – f1_1 = -2
f1_2 = 4, f2_2 = 3, f1_2 – f2_2 = -1

The new samples will be generated as
(f1', f2') = (6, 4) + rand (0-1) * (-2, -1)
rand(0-1) generates a random number between 0 and 1

This caused the selection of a random point along the line segment between three specific features as against the original one specific feature. This approach effectively forced the decision region of the minority class to become more general. Then, Wilson's Edited Nearest Neighbour (ENN), Neighbourhood Cleaning Rule (NCL) and Random Under-sampling (RUS) under-sampling schemes were applied respectively to both original SMOTE and SMOTE+300% to remove noisy, erroneous, internal and border points in the dataset.

3.2.2 Algorithm SMOTE(T, N, *k*)

Input: Number of minority class samples T; amount of SMOTE N%; Number of Nearest Neighbors, *k*
Output: (N/100) *T synthetic minority class samples. This algorithm was implemented in java codes as presented in Appendix C.

1. (* if N is less than 100%, randomize the minority class samples as only a random percent of them will be SMOTEd.*)
2. If N <100
3. Then Randomize the T minority class samples
4. T = (N/100) * T
5. N = 100

6. End if
7. N = (int) (N/100) (* The amount of SMOTE is assumed to be in integral multiples of 100. *)
8. *k* = Number of Nearest Neighbhour
9. numattrs = number of attributes
10. Sample [ ] [ ]: array for original minority class samples
11. newindex:keeps a count of number of synthetic samples generated, initialized to 0
12. Synthetic [ ] [ ]: array of synthetic samples

    (* compute *k* nearest neighbors for each minority class sample only *)
13. for *i* ← 1 to T
14. compute *k* nearest neighbors for *i*, and save the indices in the nnarray
15. Populate (N, *i*, nnarray)
16. endfor

    Populate (N, *i*, nnarray) (*Function to generate the synthetic samples.*)
17. While N ≠ 0 do
18. Choose a random number between 1 and *k*, call it nn. This step chooses one of the *k* nearest neighbor of *i*.
19. for attr← 1 to numattrs
20. Ifattr = continuous feature
21. Compute: *di f* = Sample [nnarray [nn]] [attr] – Sample [*i*] [attr]
22. Compute: *gap* = random number between 0 and 1
23. Synthetic[newindex] [attr] = Sample [*i*] [attr] + gap * *dif*
24. else
25. attr_ value = majority vote for the attr values between i and nn. If no majority then choose at random.
26. synthetic [newindex] [attr] = attr_value
27. endfor
28. newindex ++
29. N = N – 1
30. end while
31 return (* End of pseudo-code for SMOTE*)

3.2.3 Algorithm ENN (T, θ, k)

Input: Total number of all class samples T; Number of Nearest Neighbors, *k;* class θ

This algorithm was implemented in java codes as presented in Appendix D and is based on the idea that if a sample is erroneously classififed using $k - NN$, it has to be removed from T

Initialisation: $T \leftarrow X$

For each $x_i \in X$;

i. Find the k –Nearest Neighbours of $\{x_i\}$ inside $X - \{x_i\}$, ties are randomly broken when they occur.

ii. If $\delta_{k-NN}(x_i) \neq \theta_i \; then \; T \leftarrow T - \{x_i\}$

3.2.4 Algorithm NCL (T, C, k)

This algorithm was implemented in java codes as presented in Appendix E.

i. Split the dataset T into the minority class C and the rest of the data to O

ii. Identify noisy data A_1 in O with the Algorithm ENN

iii. For each class $C_i \; in \; O$

If $(x \in C_i \; in \; 3 - NN \; of \; misclassified \; y \in C)$

and $(|C_i| \geq 0.5 \; of \; |C|) \; then \; A_2 = \{x\} \cup A_2$

iv. Reduced data $S = T - (A_1 \cup A_2)$

3.2.5 Algorithm RUS (T, N, C)

This algorithm was implemented in java codes as presented in Appendix F.

Input: Total number of instances in a dataset T; Number of minority class samples *C*;

a $_TT_t = \frac{t}{T} * \frac{(t-1)}{(T-1)} * \frac{(t-2)}{(T-1)} * \ldots * \frac{1}{(T-t+1)} = \frac{t!(T-t)!}{(T)!} = \frac{1}{_TT_t}$

i.e $Pr(X_1) = Pr(X_2) = \cdots Pr(X_T) = \frac{1}{T}$

b Repeat (a) above

Until $C = O$

$T = (C \cup O)$

3.3 Implementation of models in WEKA

The data mining tool used is called Waikato Environment for Knowledge Analysis (WEKA) version 3.6.8 (Bouckeart *et al.*, 2010). The standard WEKA GUI with its filters

is shown in Figure 3.2. The SpreadSubSample which depict RUS and SMOTE data sampling schemes were already in a typical standard WEKA filter library as shown in Figure 3.2.

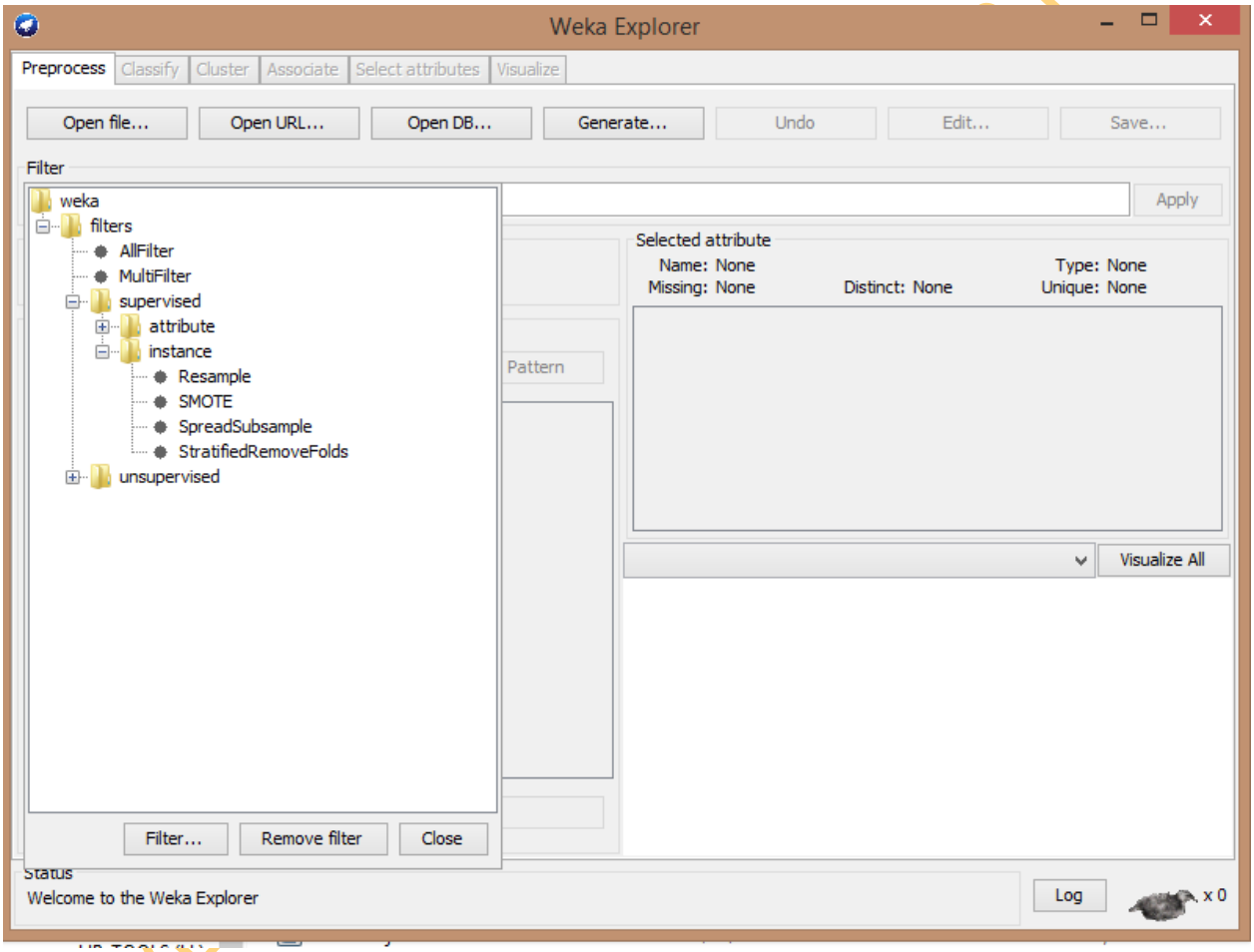


Figure 3.2: The standard WEKA's GUI with filters

The existing data sampling schemes namely: CNN, EditedNN (ENN), NeighborhoodCleaning (NCL) and TomekLink were then added to the WEKA filter library as shown in Figure 3.3 and implemented in java codes in Appendix C, D, E, F and G. The enhanced data sampling schemes were combinations of SMOTE and these existing schemes. For instance, to implement SMOTE300 data sampling scheme, the user will select SMOTE and oversample rate of 300%.

3.3.1 Basic Functionality of WEKA

The following are the basic functionalities offered by WEKA and which were also employed in this study

a. **Data pre-processing**

WEKA supports various other formats (for instance CSV, Matlab ASCII files) as well native file format (ARFF) and database connectivity through JDBC. Data can be filtered by a large number of methods (over 75), ranging from removing particular attributes to advanced operations such as principal component analysis.

All the datasets collected are represented in a spreadsheet. The spreadsheet allows us to export data into a file in Comma – Separated Value (CSV) format as a list of records with commas between items. So, the spreadsheet are converted to ARFF files.

b. Classification

One of WEKA's drawing cards is that it contains more than 100 classification methods. Classifiers are divided into "Bayesian" methods (Naive Bayes, Bayesian nets, etc.), lazy methods (nearest neighbour and variants), rule-based methods (decision tables, OneR, RIPPER), tree learners (C4.5, Naive Bayes trees, M5, Random trees), function-based learners (linear regression, SVMs, Gaussian processes, MLP), and other miscellaneous methods.

Furthermore, WEKA includes meta-classifiers like bagging, boosting, stacking; multiple instance classifiers.

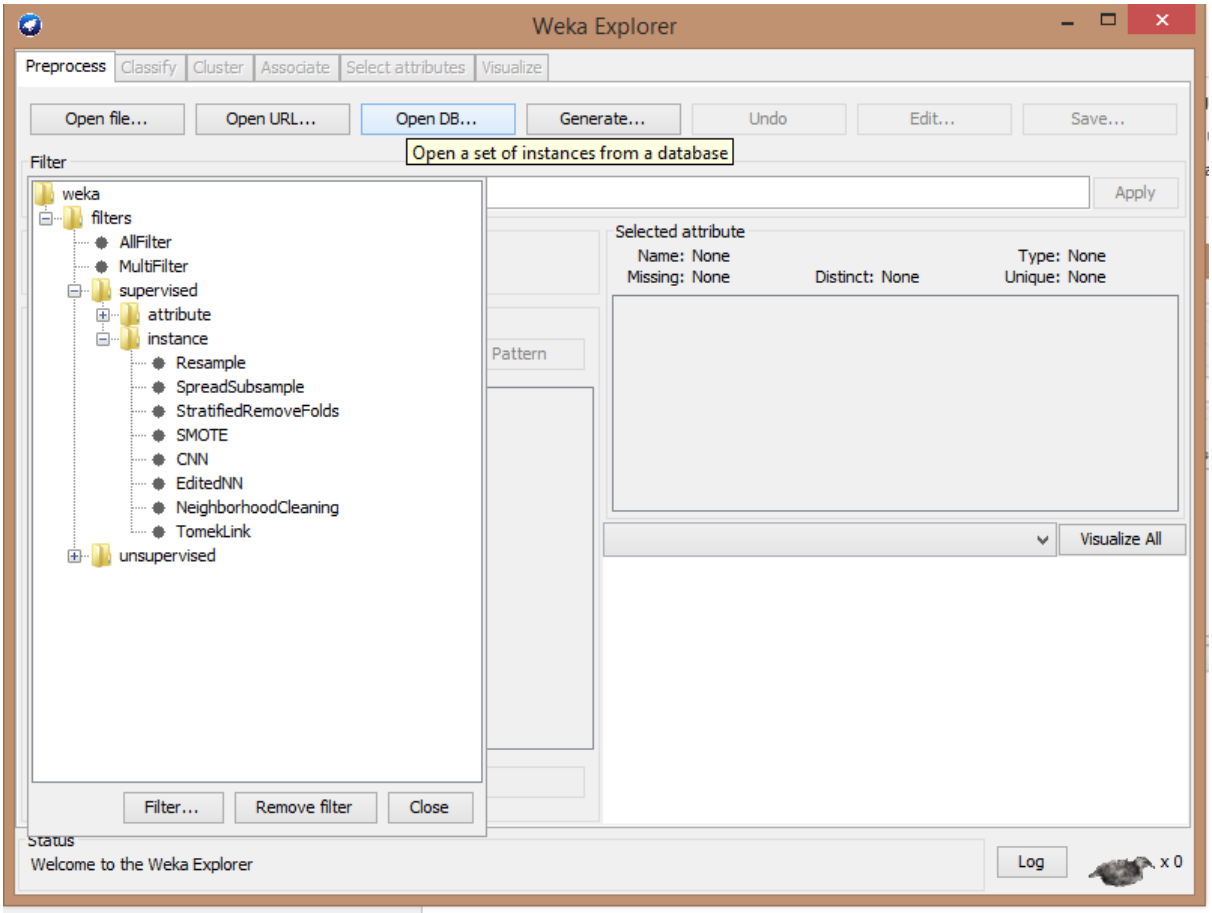


Figure 3.3: The enhanced data sampling schemes implemented in WEKA

c. Clustering

Unsupervised learning is supported by several clustering schemes, including EMbased mixture models, k-means, and various hierarchical clustering algorithms. Although not many methods of clustering are available as for classification, but most of the classic algorithms are included.

d. Attribute selection

The set of attributes used is essential for classification performance. Various attribute selection criteria and search methods are available.

e. Data visualization

Data can be inspected visually by plotting attribute values against the class, or against other attribute values. Classifier output can be compared to training data in order to detect outliers and observe classifier characteristics and decision boundaries. For specific methods there are specialized tools for visualization, such as a tree viewer for any method that produces classification trees.

WEKA also support association rule mining, comparing classifiers, data set generation, facilities for annotated documentation generation for source code, distribution estimation and data conversion.

3.3.2 Graphical User Interfaces

WEKA's functionality can be accessed through various graphical user interfaces, principally the Explorer, Experimenter and the Knowledge Flow interface.

The most popular interface, the Explorer, allows quick exploration of data and supports data loading and filtering, classification, clustering, attributes selection and various forms of visualization; in an interactive fashion.

The Experimenter is a tool for setting up machine learning experiments that evaluate classification and regression methods. It allows easy comparison of performance, and can tabulate summaries in ways that are easy to incorporate into publications. Experiments can be set up to run in parallel over different computers in a network so that multiple repetitions

of cross validation (the default method of performance analysis) can be distributed over multiple machines.

The Knowledge Flow interface is a Java Beans application that allows the same kind of data exploration, processing and visualization as the Explorer (along with some extras), but in a workflow oriented system. The user can define a workflow specifying how data is loaded, pre-processed, evaluated and visualized, which can be repeated multiple times. This makes it easy to optimize the workflow by tweaking parameters of algorithms, or to apply it to other data sources.

WEKA also includes some specialized graphical interfaces, such as a Bayes network editor that focuses on Bayes network learning and inference, an SQL viewer for interaction with databases, and an ARFF data file viewer and editor.

All functionality and some more specialized functions can be accessed from a command line interface, so WEKA can be used without a windowing system.

3.3.3 Extending WEKA

One of WEKA's major strengths is that it is easily extended with customized or new classifiers, Filters, Clusterers, attribute selection methods, and other components. To add a new Filter or classifier, all that is needed is a class that derives from the Classifier class and implements the buildFilter or buildClassifier method for learning, and a FilterInstance or ClassifyInstance method for testing/predicting the value for a data point.

Any new class is picked up by the Graphical User Interfaces (GUI) through Java introspection: no further coding is needed to deploy it from WEKA's GUIs. This makes it easy to evaluate how new algorithms perform compared to any of the existing ones, which explains WEKA's popularity among machine learning researchers.

3.3.3.1 Writing a new Filter

The enhanced schemes were added to WEKA filter library. Filters perform many tasks, from resampling data, to deleting and standardizing attributes.

The following methods are of importance for the implementation of a filter. These methods are declared in the weka.filters.Filter class. These are:

a. getCapabilities()
b. setInputFormat(Instances)
c. getInputFormat()
d. setOutputFormat(Instances)
e. getOutputFormat()
f. input(Instance)
g. bufferInput(Instance)
h. push(Instance)
i. output()
j. batchFinished()
k. flushInput()
l. getRevision()

But only the following methods were modified in this study. In order to include the enhanced data sampling scheme in WEKA

i. getCapabilities()
ii. setInputFormat(Instances)
iii. input(Instance)
iv. batchFinished()
v. getRevision()

setInputFormat(Instances)

With this call, the user tells the filter what structure, i.e., attributes, the input data has. This method also tests, whether the filter can actually process this data, according to the capabilities specified in the getCapabilities()method. If the output format of the filter, i.e., the new Instances header, can be determined based alone on this information, then the method should set the output format via setOutputFormat(Instances) and return true, otherwise it has to return false.

getInputFormat()

This method returns an Instances object containing all currently buffered Instance objects from the input queue.

setOutputFormat(Instances)

This method defines the new Instances header for the output data. For filters that work on a row-basis, there should not be any changes between the input and output format. But filters that work on attributes, e.g. removing, adding, modifying, will affect this format. This method must be called with the appropriate Instances object as parameter, since all Instance objects being processed will rely on the output format (they use it as dataset that they belong to).

getOutputFormat()

This method returns the currently set Instances object that defines the output format. In case setOutputFormat(Instances) has not been called yet, this method will return null.

input(Instance)

This method returns true if the given Instance can be processed straight away and can be collected immediately via the output() method (after adding it to the output queue via push(Instance), of course). This is also the case if the first batch of data has been processed and the Instance belongs to the second batch. Via isFirstBatchDone() one can query whether this Instance is still part of the first batch or of the second.

If the Instance cannot be processed immediately, e.g., the filter needs to collect all the data first before doing some calculations, then it needs to be buffered with bufferInput(Instance) until batchFinished() is called. In this case, the method needs to return false.

bufferInput(Instance)

In case an Instance cannot be processed immediately, one can use this method to buffer them in the input queue. All buffered Instance objects are available via the getInputFormat() method.

push(Instance)

This method adds the given Instance to the output queue.

Output()

This method returns the next Instance object from the output queue and removes it from there. In case there is no Instance available this method returns null.

batchFinished()

This method signals the end of a dataset being pushed through the filter. In case of a filter that could not process the data of the first batch immediately, this is the place to determine what the output format will be (and set if via setOutputFormat(Instances)) and finally process the input data. The currently available data can be retrieved with the getInputFormat() method.

After processing the data, one needs to call flushInput() to remove all the pending input data.

flushInput()

This method removes all buffered Instance objects from the input queue. This method must be called after all the Instance objects have been processed in the batchFinished() method.

Option handling

If the filter should be able to handle command-line options, then the interface weka.core.OptionHandler needs to be implemented. In addition to that, the following code should be added at the end of the setOptions(String[]) method:

```
if (getInputFormat() != null) {
setInputFormat(getInputFormat());
}
```

This will inform the filter about changes in the options and therefore reset it.

The following examples, covering batch and stream filters, illustrate the filter framework and how to use it. Unseeded random number generators like Math.random() should never be used since they will produce different results in each run and repeatable experiments are essential in machine learning.

BatchFilter

This simple batch filter adds a new attribute called blah at the end of the dataset. The rows of this attribute contain only the row's index in the data. Since the batch-filter does not have to see all the data before creating the output format, the setInputFormat(Instances) sets the output format and returns true (indicating that the output format can be queried immediately). The batchFinished() method performs the processing of all the data.

```
import weka.core.*;
import weka.core.Capabilities.*;
public class BatchFilter extends Filter {
public String globalInfo() {
return "A batch filter that adds an additional attribute 'blah' at the end "
+ "containing the index of the processed instance. The output format "
+ "can be collected immediately.";
}
public Capabilities getCapabilities() {
Capabilities result = super.getCapabilities();
result.enableAllAttributes();
result.enableAllClasses();
result.enable(Capability.NO_CLASS); // filter doesn't need class to be set
return result;
}
public boolean setInputFormat(Instances instanceInfo) throws Exception {
super.setInputFormat(instanceInfo);
Instances outFormat = new Instances(instanceInfo, 0);
outFormat.insertAttributeAt(new Attribute("blah"),
outFormat.numAttributes());
setOutputFormat(outFormat);
return true; // output format is immediately available
}
public boolean batchFinished() throws Exception {
if (getInputFormat() = null)
```

```
throw new NullPointerException("No input instance format defined");
Instances inst = getInputFormat();
Instances outFormat = getOutputFormat();
for (int i = 0; i < inst.numInstances(); i++) {
double[] newValues = new double[outFormat.numAttributes()];
double[] oldValues = inst.instance(i).toDoubleArray();
System.arraycopy(oldValues, 0, newValues, 0, oldValues.length);
newValues[newValues.length - 1] = i;
push(new Instance(1.0, newValues));
}
flushInput();
m_NewBatch = true;
m_FirstBatchDone = true;
return (numPendingOutput() != 0);
}
public static void main(String[] args) {
runFilter(new BatchFilter(), args);
}
}
```

3.4 Evaluation metrics and Statistical analysis tool

Statistical significance is a measure from statistics which attempts to determine how unlikely a result is to have occurred by chance. After performing cross validation over a wide variety of RAW DATA and 13 data sampling schemes to produce results, then there is need to determine which classifier is better or which data sampling scheme is the best. In this study, ROC_AUC, RECALL of the minority class, RMSE, Kappa Statistics and performance Loss/gain metric were used for evaluation. ROC_AUC, RMSE and Kappa Statistics are used to measure the general ability of the classifier to separate the positive and negative classes while RECALL utilized the threshold of 0.5 (if the posterior probability of positive class membership is greater than 0.5, the example is classified as belonging to positive class). The results generated were analysed using both non-parametric and parametric statistical methods. Friedman Test, ANOVA with Tukey Post Hoc and box and whisker plot at statistical significance level of 0.05% with confidence

level of 95% in Statistical Package for Social Sciences (SPSS) software (SPSS, 2007) were used to determine which classifier is better and on which data sampling scheme.

3.4.1 Hypothesis Testing

A statistical hypothesis test involves using statistical inference to test the validity of postulated values for the population parameter. A test's result is said to be statistically significant if it has been predicted as unlikely to have been due to sampling error alone, according to a threshold probability—the significance level. Hypothesis tests are used in determining what outcomes of a study would lead to a rejection of the null hypothesis for a pre-specified level of significance.

H_0 $(null\ hypothesis)$: All μ_i are equal.

H_1 $(alternative\ hypothesis)$: At least one μ_i are different.

where

$\mu\ is\ the\ mean$

Significance Level used is $\alpha = 0.05$.

The rejection region is a region at which the null hypothesis is being rejected and is set at p-value = 0.05.

3.4.2 Friedman Test

The Friedman test is a non-parametric test that compares three or more matched or paired groups. It is used to detect differences in treatments across multiple tests attempts. The Friedman test first ranks the values in each matched set (each row) from low to high. Each row is ranked separately. It then sums the ranks in each group (column). If the sums are very different, the ρ value will be small. The mean rank value of the Friedman statistic is calculated from the sums of ranks and the sample sizes.

3.4.3 Analysis Of Variance (ANOVA)

ANOVA is a method of multiple comparisons of means of several variables with more than two independent variables. It employ tests based on variance ratios to determine whether or not significant differences exist among the means of several groups of observations, where each group follows a normal distribution (Olatayo *et al.* 2011). A one-

way ANOVA is used to determine the effect of one independent variable (CLASSIFIER) on a dependent variable (VALUES) as presented in equation 3.1. A two-way ANOVA is used to determine the effects of two independent vatiables (SCHEMES and CLASSIFIER) on a dependent variable (VALUES) as presentesbin equation 3.2.

A one-way ANOVA model is presented in equation 3.1.

$$y_{ij} = \mu + \alpha_i + \varepsilon_{ij} \quad (3.1)$$

Where

$y_{ij}\ is\ the\ ith\ response\ in\ the\ jth\ column$

$\mu\ is\ the\ overall\ mean$

$\alpha_i\ is\ the\ measure\ of\ ith\ CLASSIFIER\ effects$

$\varepsilon_{ij}\ is\ the\ random\ error\ component$

Testing for CLASSIFIER effects

H_0: All α_i are equal.

H_1: At least one α_i are different.

A two-way ANOVA model is presented in equation 3.2.

$$y_{ij} = \mu + \alpha_i + \beta_j + \varepsilon_{ij} \quad (3.2)$$

Where

$y_{ij}\ is\ the\ ith\ response\ in\ the\ jth\ column$

$\mu\ is\ the\ overall\ mean$

$\alpha_i\ is\ the\ measure\ of\ i^{th}\ CLASSIFIER\ effects$

$\beta_j is\ the\ measure\ of\ j^{th}\ SCHEMES\ effect$

$\varepsilon_{ij}\ is\ the\ random\ error\ component$

Testing for CLASSIFIER effects

H_0: All $\alpha_i$are equal.

H_1: At least one α_i are different.

Testing for SCHEMES effects

H_0: All β_j are equal.

H_1: At least one β_j are different.

3.4.4 Tukey–Kramer method

This is a single-step multiple comparison procedure and statistical test. It can be used on raw data or in conjunction with an ANOVA (Post-hoc analysis) to find means that are significantly different from each other. It compares all possible pairs of means, and is based on a *studentized range distribution* (q).

3.4.5 Box and Whisker Plots

These plots offer a pictorial summary of important dataset characteristics including the central tendency, dispersion, asymmetry and extremes arrived at through percentile rank analysis and the plotting of maximum and minimum dataset values. Its graphically compact nature facilitates side by side comparison of multiple datasets, which can otherwise be difficult to interpret using more complete representations such as the histogram (Banaco, 2011). Ordered data are divided into lower and upper half by the median. The median of the lower half is the lower quartile. The median of the upper half is the upper quartile. The lower extreme is the least data value. The upper extreme is the greatest value. Important characteristics of each scheme: central tendency, skewness, dispersion and extremes are easy to interpret and visualise. Each box in the box plots represents a data sampling scheme. The whiskers at the end of the box plots show the minimum and maximum values, while the bar shows the median. If the median bar is above zero or higher, the data sampling scheme represented by the box plot is doing better on average than the data sampling scheme that is being compared with. And if the complete box, including the whiskers, is above zero, then that data sampling scheme is better than the other data sampling schemes.

3.5 The Datasets

When conducting research on classification, the norm is to test algorithm on, and draw conclusion from, a number of different datasets from different problem domains. If some of the conclusions drawn holds true for different tasks, then the conclusion will most likely hold true for all in general. In this study, the datasets used were collected from different domains in South Western Nigeria.

3.5.1 **Diabetes Mellitus (DM) dataset**

Diabetes mellitus or simply diabetes is a group of metabolic diseases in which a person has high blood sugar content, either because the pancreas does not produce enough insulin, or because cells do not respond to the insulin that is produced. This high blood sugar content produces the classical symptoms of polyuria (frequent urination), polydipsia (increased thirst) and polyphagia (increased hunger).

Three main types DM considered were:

a. Type1 DM which is the outcome of the body's failure to produce insulin, and requires the person to inject insulin or wear an insulin pump. This form was previously referred to as "insulin-dependent diabetes mellitus" (IDDM) or "juvenile diabetes".

b. Type2 DM which results from insulin resistance, a condition in which cells fail to use insulin properly, sometimes this is combined with an absolute insulin deficiency. This form was previously referred to as non-insulin-dependent diabetes mellitus (NIDDM) or "adult-onset diabetes".

c. The third main form, gestational diabetes (GDM) occurs when pregnant women without a previous diagnosis of diabetes develop a high blood glucose level. It may preceed development of type 2 DM and is the class of interest (minority class) in this study.

Other forms of DM include congenital diabetes, which is due to genetic defects of insulin secretion, cystic fibrosis-related diabetes, steroid diabetes induced by high doses of glucocorticoids, and several forms of monogenic diabetes (Sarwar *et al*., 2010).

The raw data for this disease condition used in this study was obtained from the records department of the Family Medicine Clinic of Wesley Guild Unit of Obafemi Awolowo University Teaching Hospital Complex, Ilesha, Osun State, Nigeria. The dataset of outgoing patients suffering from DM was extracted, reviewed and processed. The dataset contained 886 instances of complete record of DM patients from January 2009 to May 2010. This dataset was collected by Awokola (2010) for research purpose. It contained information about patients with three types of diabetes. The dataset contained 886 instances, had 18 attributes and three different classes namely: TYPE1, TYPE2 and Gestational Diabetes Mellitus (GDM). The dataset class distribution was 807:62:17 where

TYPE2 had 807 instances, TYPE1 had 62 instances and GDM had only 17 instances (the minority class and also the class of interest). The dataset is highly skewed.

3.5.2 Senior Secondary School Certificate Examination Result (SSS Result) dataset

The data collected comprised of results of students in five secondary schools in Ibadan, Nigeria. The records of students who sat for SSS Result consisting of both West Africa Examination Council (WAEC) and Nigeria Examination Council (NECO) examinations results in the schools were used for analysis. These results were for both public and private secondary schools within Ibadan metropolis, Oyo state, Nigeria for a period of five years (2005-2009). This dataset was collected by Agboola (2010) for research purpose. For the purpose of this study, only data on English Language and Mathematics were used for the analysis because they were compulsory for all students. Any student that passes both English Language and Mathematics in both WAEC and NECO was regarded as PASSBOTH, students that failed English Language and Mathematics both in WAEC and NECO examination was regarded as FAILBOTH. Students that passed English Language and Mathematics in WAEC alone was regarded as PASSWAEC while students that passed both English Language in NECO alone was regarded as PASSNECO. The dataset contains 1163 instances consisting of 8 different attributes with four different classes namely: FAILBOTH with 775 instances, PASSNECO with 248 instances, PASSWAEC with 45 instances and PASSBOTH with 95 instances. PASSWAEC is the class of interest and also the minority class in this study. The dataset class distribution was 775:248:45:95.

3.5.3 Tuberculosis (TB) dataset

Tuberculosis (TB) is a disease caused by bacteria called *Mycobacterium tuberculosis*. It is usually spread through the air and attacks low immune bodies such as patients with Human Immuno-deficiency Virus (HIV) (Asha *et al*,. 2011). It is a disease which can affect virtually all organs, not sparing even the relatively inaccessible sites. The microorganisms usually enter the body by inhalation through the lungs. They spread from the initial location in the lungs to other parts of the body via the blood stream. It presents a diagnostic dilemma even for physicians with a great deal of experience with this disease.

Hence Tuberculosis (TB) is a contagious bacterial disease caused by mycobacterium which affects usually lungs and is often co-infected with HIV/AIDS (Asha *et al.*, 2012).

Nigeria has the tenth highest burden of TB among the 22 TB high –burden countries in the world (Lawson *et al*., 2012)

The medical dataset that were classified included 768 real records of patients suffering from tuberculosis (TB) obtained during the cause of this study from Ijaye State Hospital, Ogun State. The entire dataset was put in one file having many records. Each record corresponds to most relevant information of one patient. Initial queries by Doctors for symptoms and required test result details of patients were considered as main attributes. On the aggregate, there were 12 attributes (symptoms) and four classes namely: Pulmonary TB (PTB), Extra PTB (EPTB), Retroviral PTB (RPTB) and Retroviral EPTB (REPTB) which is the minority class and also the class of interest. The dataset class distribution was 589:124:37:6 where PTB had 589 instances, RPTB had 124 instances, EPTB had 37 instances and REPTB had only 6 instances (the minority class and also the class of interest).

3.5.4 Contraceptive Method (CM) dataset

This dataset was collected during this study from the Government Health Centre clinic at Ibadan North East Local Government, Ibadan, Oyo state. The dataset was collected for a period of seven (7) years (2008–2014) for this research purpose. The dataset contained 775 instances, 20 attributes with 5 different classes namely: NONE, SECONDARY^{+}, SECONDARY, PRIMARY and PRIMARY^{+}. NONE represented patients without any education that is illiterate, PRIMARY represented patients that attended primary school but did not complete their education, PRIMARY^{+} represented patients that had elementary primary education with certificate, SECONDARY represented patients that went to secondary school but did not complete the senior secondary school but could have or have not completed the junior secondary school while SECONDARY^{+} stood for patients that had a complete secondary education and/or with either College of Education, Polytechnic or University education. The dataset class distribution was 414:247:45:53:16 where SECONDARY^{+} had 414 instances, SECONDARY had 247 instances, PRIMARY had 45 instances, PRIMARY^{+} had 53 instances and NONE had only 16 instances (the minority class and also the class of interest).

The summary of datasets used in this study is presented in Table 3.1. It showed the datasets with the number of attributes, the number of classes they contain and the percentage of minority class.

3.6 EXPERIMENTAL DESIGN

This section describes the specific configuration used by all the classification algorithms in this study. For simplicity, WEKA default values were used for all algorithm configuration.

3.6.1 Classification algorithms' configuration

This section give the details of specific configuration and factors used with the classification algorithm for their implementation in WEKA. The classification task was carried out by a set

of n input instances $X_1, X_2, \ldots, X_n \; with \; j \; features \; a_1, a_2, \ldots, a_j \in T$ which can either be nominal or numerical values, whose desired output class labels $Y_i \in C = \{C_1, C_2, \ldots, C_k\}$. Hence, the classifier or learner generates a mapping function that is defined over the pattern $T^i \rightarrow C$

3.6.1.1 Random Tree

This is a classifier for constructing a tree that considers K randomly chosen attributes at each node. It performed no pruning. Also, allowed estimation of class probabilities based on a hold-out set. The number of randomly chosen attributes was set to 0 log 2 (number_of_attributes) +1). The maximum depth of the tree was also set to 0 for unlimited. The number of folds which determines the amount of data used for back fitting was set to 0 that is no back fitting and the random number seed used for selecting attributes was set to 1.

Table 3.1 Summary of datasets

Datasets	Attributes	Number of classes	%Minority class
DM	19	3	2
SSS Result	8	4	4
TB	13	4	0.79
CM	20	5	7

3.6.1.2 RIPPER

This class implements a propositional rule learner, Repeated Incremental Pruning to Produce Error Reduction (RIPPER), which was proposed by Cohen (1995) as an optimized version of Incremental Reduced Error Prunning (IREP). The number of folds used was 3. The minimum number total weight of the instances in a rule was 2, the number of optimization runs was 2, the seed used for randomizing the data was set to 1 and it was un-pruned.

3.6.1.3 Decision Tree

This was used for generating a pruned or un-pruned C4.5 decision tree with a Confidence Factor of 0.25. One of the reasons to avoid pruning is that most pruning scheme attempt to minimize the overall error rate. These pruning schemes can be detrimental to the minority class, since reducing the error rate in the majority class, which stands for most of the examples, would result in a greater impact over the overall error rate (Batista *et al.*, 2004, Zadrozny and Elkan, 2001, Chawla, 2003). The minimum number of instances per leaf was 2 and the number of folds was 3. The number of seed used for randomizing the data when reduced-error pruning was used was 1. The tree was not pruned, Laplace smoothing and MDL correction was used. Hence, for this configuration, the decision tree used was C4.4, a variant of C4.5.

3.6.1.4 K-Nearest Neighbours classifier (1B3)

K-Nearest Neighbours classifier where k = 3 is the number of nearest neighbour (Aha *et al.*, 1991). Hold-one-out cross-validation was used to select this k value. The nearest neighbour search algorithm used was neighboursearch.Linear NNSearch based on distance weighting method.

3.6.1.5 REPTree

This is a fast decision tree learner that built a decision/regression tree using information gain/variance and pruneed it using reduced-error pruning (with backfitting). Missing values are dealt with by splitting the corresponding instances into pieces. The maximum tree depth was set to -1 for no restriction, the minimum total weight of the instances in a leaf was 2 with no Pruning, and the number of folds was set to 3 while the seed used for randomizing the data was set to 1.

3.6.1.6 Support Vector Machine (SVM)

This implementation globally replaced all missing values and transformed nominal attributes into binary ones (Platt, 1998). It also normalized all attributes by default. (The coefficients in the output are based on the normalized data, not the RAW DATA data as this is important for interpreting the classifier.) Multi-class problems are solved using pairwise classification (one-vs-one). The complexity parameter was set to 1. The epsilon for round-off error was set to 1.0E-12. The kernel used was kernel Polykernel and the number of folds for the cross-validation that used to generate the training data for logistic models was set to 1.

3.6.1.7 MultiLayerPerceptron (MLP)

This classifier used back-propagation to classify instances. The nodes in this network are all sigmoid. This divided the starting learning rate by the epoch number, to determine what the current learning rate should be. The number of hidden layers of the neural network used here was 1. The learning Rate was set to 0.3 while the momentum was set to 0.2. The seed used to initialize the random number was set to 0. The TrainingTime, which is the number of epochs to train through, was set to 500, the percentage size of the validation set was set to 10 and the validation Threshold used to terminate validation testing was set to 20.

3.6.1.8 Multiple Class Classifier

This is a meta classifier for handling multiple class datasets with 2-class classifiers. This classifier is also capable of applying error correcting output codes for increased accuracy. The random number seed used is 1. The base classifier used is an unpruned decision tree with Laplace Smoothing and Minimum (MDL) correction. The decomposition method used for transforming the multi-class problem into several 2-class ones was one – against – all (OVA).

3.6.1.9 RandomCommittee

This an ensemble of randomizable base classifiers (Random Tree). Each base classifier was built using a different random number seed (but based on the same data). The final prediction was a straight average of the predictions generated by the individual base classifiers. The base classifier used was random tree. The number of iterations was 10 on 1 and the random number seed.

3.6.1.10 Random Forest

This is an ensemble of random trees for constructing a forest trees by using bootstrap samples of training data. This algorithm was used with all its default value set. The maximum depth of the trees set to 0 for unlimited. The number of trees to be generated was set to 10 and the random number seed to be used was set to 1.

3.6.1.11 Random Subspace (Decision Forest)

This ensemble method constructs a decision tree based classifier that maintained highest accuracy on training data and improved on generalization accuracy as it grows in complexity. The classifier consists of multiple trees constructed systematically by pseudo randomly selecting subsets of components of the feature vector, that is, trees constructed in randomly chosen subspaces. The base classifier used is decision tree (C4.4). The number of iterations performed was 10 and the random number seed used was 1. The size of each subspace was 0.5.

3.6.1.12 Stacking

This is an ensemble of classifiers that combine four different classifiers using the stacking method. The base classifiers were arranged to form a heterogeneous ensemble in this order: RIPPER, Decision tree, 1B3, Support Vector Machine and MultilayerPerceptron. The meta classifier used was the decision tree, un-pruned and Laplace smoothing with a seed number of 1 and 10 folds for cross-validation.

3.6.1.13 Bagging

This is an ensemble method for bagging a classifier to reduce variance. The size of each bag (as a percentage of the training set size) was set to 100 and the base classifier was decision tree (C4.4). 10 iterations were performed on the dataset with 1 random number seed.

3.6.1.14 Boosting (AdaBoostM1)

This is an ensemble for boosting a nominal class classifier using the AdaboostM1 method. Only nominal class problems can be tackled. Often dramatically improves performance, but sometimes over fits. Decision tree (C4.4) was used as the base classifier. 10 iterations

were performed with 1 random number seed. Weight threshold for weight pruning was set to 100.

3.6.2 Ten - fold Cross Validation

Ten-fold cross validation was used for training the datasets. All the datasets were all divided randomly into ten parts in which the class is represented in approximately the same proportion as in the full dataset. Each part is held out in turn and the learning scheme trained on the remaining nine – tenths; then its error rate is calculated on the holdout set. Thus, the learning procedure is executed a total of 10 times on different training sets (each set has a lot in common with the others). Finally, the ten error estimates are averaged to yield an overall error estimate.

3.7 Percentage Reduction/Increment in the dataset

The formula used to calculate the percentage reduction/ increment in the entire dataset after applying the enhanced schemes and the existing class imbalance schemes presented in Equation 3.2.

$$\%\text{Reduction/Increment} = \frac{\text{OriginalSize - NewSize}}{\text{OriginalSize}} \times 100 \qquad (3.2)$$

3.8 Percentage number of the minority class in the dataset

The procedure used to calculate the percentage number of the minority class in the total number of instances in the entire dataset is presented in Equation 3.3

$$\%\text{minority class} = \frac{\text{total number of instances of minority class}}{\text{total number of instances in the dataset}} \times 100 \qquad (3.3)$$

3.9 Measuring the impact of class distribution on classifier performance

This section focused on identifying and explaining any differences in classification performance between the minority and majority class. It was observed that there was a large error rate for minority class. The analysis of this section was to show that the minority class predictions and test samples both had larger error rate than their majority class counter parts. Though the three datasets used in this study were multiple class problems,

the classes were collapsed to two class problems where the positive class represented the minority class while the negative class represented the majority class. Equations 3.4-3.7 was used to calculate the errors. $\overline{PPV}$ is the minority class prediction error. $\overline{NPV}$ is the majority class prediction error. False Negative Rate (FNR) is the likelihood that a positive example is classified as a negative example. FNR is the error rate associated with the positive examples. False Positive Rate (FPR) is the likelihood that a negative example is classified as a positive example. FPR is the error associated with the negative examples.

$$\overline{\text{PPV}} = \frac{\text{FP}}{\text{TP+FP}} \tag{3.4}$$

$$\overline{\text{NPV}} = \frac{\text{FN}}{\text{TN+FN}} \tag{3.5}$$

$$\text{FNR} = \frac{\text{FN}}{\text{TP+FN}} \tag{3.6}$$

$$\text{FPR} = \frac{\text{FP}}{\text{TN+FP}} \tag{3.7}$$

3.10 Performance Loss/Gain on classifiers

The performance loss or gain was calculated relative to the original distribution of each dataset. The formula is presented in equation 3.8

$$\text{PerformanceLoss/gain} = \frac{ROCOriginal - ROCNew}{ROCOriginal} \tag{3.8}$$

where ROCOriginal is the performance obtained from the RAW DATA dataset on a classifier measured in ROC_AUC and ROCNew is the performance obtained with the new datasets generated using all data sampling schemes.

CHAPTER FOUR

RESULTS AND DISCUSSION

4.1 Introduction

This chapter presents the results obtained from the implementation of both the enhanced and existing data sampling schemes in WEKA. A total of thirteen (13) balanced datasets which were created from the 13 different data sampling schemes (both existing and enhanced) were trained on fourteen (14) different classifiers. The results obtained were analyzed statistically using Friedman test, Analysis Of Variance (ANOVA) test and Box and whisker plot. This chapter also presented the discussion of the result obtained.

4.1.1 Analysis of error rates of the minority and majority class distributions

A typical classification analysis using a Decision Tree classifier illustrates the motivation for this study. The four test datasets namely Diabetes Mellitus disease (DM), Senior Secondary School Result (SSS Result), Contraceptive Methods (CM) and Tuberculosis (TB) used in the study were converted to binary class problems and Decision Tree classifier was used to train them. For the binary class, the minority is the positive class while the majority class is the negative class. The confusion matrices obtained for the binary class datasets are presented Table 4.1. The error rates from both classes are presented in Table 4.2. This table showed the percentage of the minority class examples in their natural class

Table 4.1: Confusion matrix of the bi-class from Decision Tree classifier on study dataset

Dataset	DM		SSS Result		CM		TB	
	Positive Prediction	Negative Prediction	Positive Prediction	Negative Prediction	Positive Prediction	Negative Prediction	Positive Prediction	Negative Prediction
Actual Positive	3	14	0	45	2	14	6	0
Actual Negative	1	869	0	1118	1	758	0	751

Table 4.2: Error rates of the study datasets with Decision Tree Classifier

		Prediction error		Actual error	
Dataset	% minority examples	Minority $\overline{PPV}$	Majority $\overline{NPV}$	Minority (FNR)	Majority (FPR)
DM	2	25	1.6	82	0.11
SSS Result	4	100	3.87	100	0
CM	7	33	1.8	88	0.13
TB	0.79	0	0	0	0
Average		39.5	1.82	67.5	0.06

distribution of each dataset. The prediction error columns show the error rates obtained for the minority and majority class after training the test datasets with Decision Tree classifier. The actual error column presents the actual classification error rates for the minority and majority class samples.

The percentage of minority class to the total number of examples in DM dataset was 2%. The minority labelled predictions had an error rate of 25% while that of the majority was 1.6%. The minority class test samples had a classification error of 82% while that of majority class test samples was 0.11%.

The percentage of minority class to the total number of examples in SSS Result dataset was 4%. The minority labelled predictions had an error rate of 100% while that of the majority was 3.87%. The minority class test samples had a classification error of 100% while that of the majority class test samples was 0%.

The percentage of minority class to the total number of examples in CM dataset was 2%. The minority labelled predictions had an error rate of 33% while that of the majority was 1.8%. The minority class test samples had a classification error of 88% while that of the majority class test samples was 0.13%.

TB disease dataset had the percentage of minority class to the total number of examples to be 0.79%. The minority labelled predictions had an error rate of 0% while that of majority labelled prediction was 0%. The minority and majority class test samples zero classification error.

The minority samples predictions had average error rate of 39.5% while that of majority samples predictions was 1.82% on all four datasets. It can also be observed that the average error rate for minority class test samples (FNR) was 67.5% while the average error rate for the majority class test samples was 0.06% from Table 4.2. It can be inferred from the results in Table 4.2 that the minority class predictions performance were worse than the majority class predictions and that the minority class examples were mis-classified more frequently than the majority class. In all the datasets, the minority class test examples had a higher error rate than the majority class test examples except for TB dataset. The TB did not show that it was affected by the class distribution of samples in the classification result.

Hence, it cannot benefit from the application of any data sampling schemes intended to further improve the classification result.

4.1.1.1 Discussion on the error rates

The minority class predictions of datasets (DM, SSS Result and CM) had a higher error rate $\overline{PPV}$ than their majority class predictions $\overline{NPV}$. One of the reasons why the minority class predictions are so error prone is that from the results obtained from Table 4.2, it showed that naturally imbalanced dataset yielded classifiers with higher error concentrations than their balanced versions (datasets treated with data sampling schemes) of the same datasets. The explanation for this behaviour is that the test distribution effect makes the minority class harder to train especially when allied with class disjuncts. The explanation why minority class test examples are misclassified much more often than majority class test examples (FNR>FPR) according to Japkowicz and Stephen, (2002) is that since the fraction of positive examples in the test set is very small, the true negative rate is weighted more than the true positive rate. This means that a plan to maximize accuracy will place higher emphasis on maximizing the number of true negatives (TN) than maximizing the number of true positives (TP), and also more emphasis on minimizing the number of false negatives. A classifier tendered towards maximizing accuracy would prefer false negative errors to false positive errors. Another reason why minority class examples are mis-classified more often is that fewer minority class are likely to be sampled from the class distribution. Therefore, the training data are less likely to include (enough) instances of all the minority class and the classifier may not have the opportunity to represent all truly positive regions. For this reason, some minority class samples will be mistakenly classified as belonging to the majority class. The reason why a lower error rate is generally observed for majority class test samples (FN>FP) is because the majority class is predicted far more often than the minority class as agreed by Weiss and Provost, (2003).

4.1.2 Steps involved in evaluation of result

The datasets used in the study were pre-processed with both the enhanced and existing data sampling schemes. Thirteen (13) different datasets were created from the pre-processing. These datasets were then trained on 14 different classification algorithms. Figure 4.1 presents the steps taken in the pre-processing, evaluation, statistical analysis and reporting of the result. The classification algorithms were chosen from different

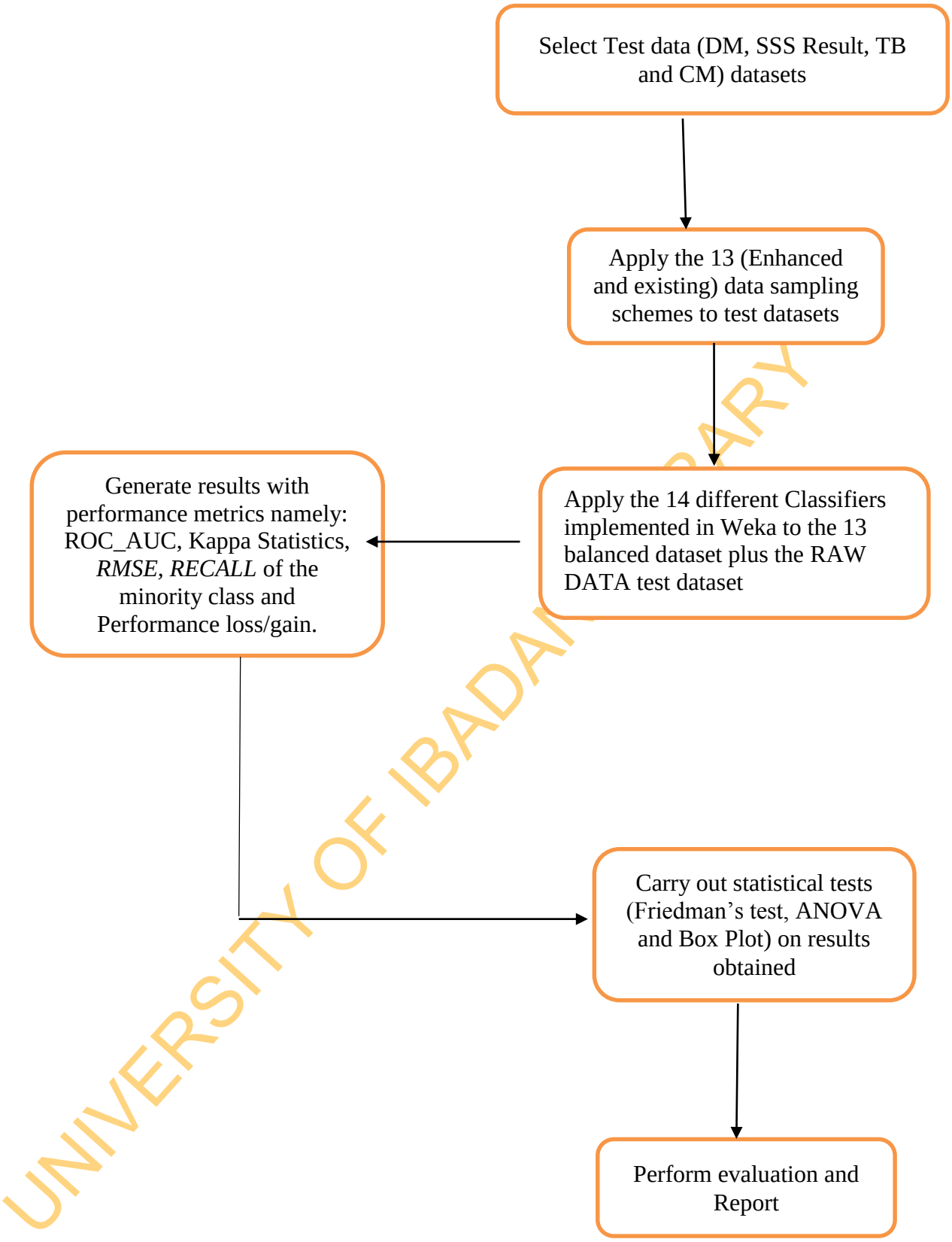


Figure 4.1: Steps used for the pre-processing and analysis of result

categories namely: base, homogeneous and heterogeneous ensemble classification algorithms. The results obtained with these performance measures (Kappa Statistics, *RMSE*, *RECALL* of the minority class, *ROC_AUC* and Performance loss/gain) were analysed using *ANOVA*, Friedman's test and Box plot.

4.1.3 Dataset Distribution

The class distributions generated for all the datasets used in the study after the application of the 13 data sampling schemes to the RAW DATA were presented in Tables 4.3, 4.4, 4.5 and 4.6 and also plotted in Figure 4.2, 4.3, 4.4 and 4.5. These tables and figures showed all the data sampling schemes used in the study, the various class distribution obtained after the application of the 13 data sampling schemes, the total number of instances of the resultant datasets, the percent reduction or increment in the size of the dataset compared to the RAW DATAsets and the percentage of the minority class in each dataset. The results presented in these tables form the basis for the analysis presented in this chapter.

Table 4.3: DM Dataset class distribution

S/N	Data Sampling Schemes	Class distribution			Number of Instances	%Reduction/ Increment	% Minority Class
		TYPE2	TYPE1	GDM			
1	RAW DATA	807	62	17	886		2
2	CNN	164	50	15	229	74	7
3	ENN	778	11	4	793	10	1
4	RUS	17	17	17	51	94	33
5	NCL	698	62	17	777	12	2
6	5ENN	784	12	4	800	10	1
7	SMOTE	807	62	34	903	-2	4
8	SMOTE300	807	62	68	937	-6	7
9	SMOTEENN	770	14	21	805	9	3
10	SMOTENCL	700	62	34	796	10	4
11	SMOTERUS	34	34	34	102	88	33
12	SMOTE300ENN	766	15	58	839	5	7
13	SMOTE300NCL	693	62	68	823	7	8
14	SMOTE300RUS	62	62	62	186	79	33

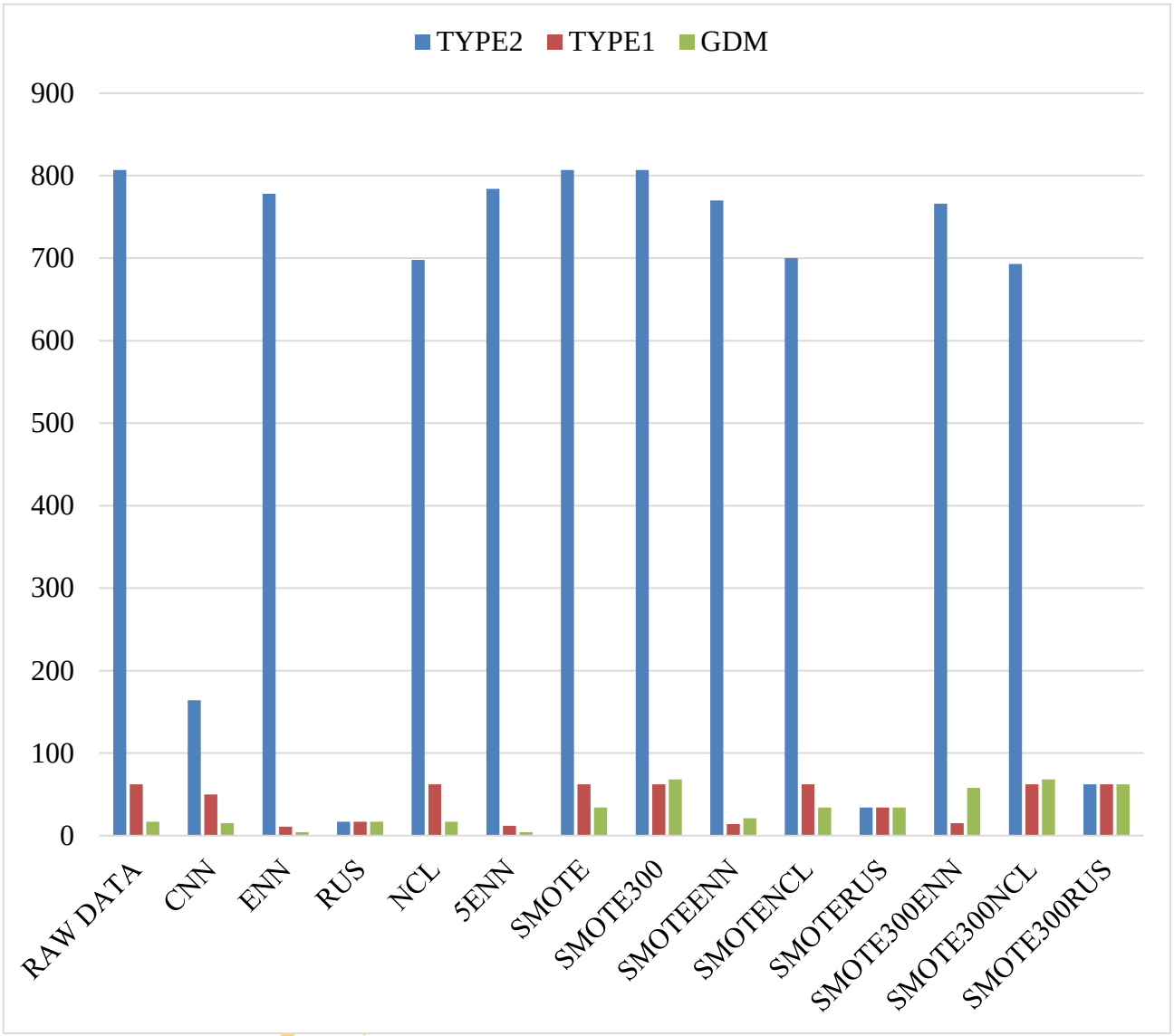


Figure 4.2: Chart showing the DM Dataset class distribution

Table 4.4: SSS Result Dataset class distribution

S/N	Data Sampling Schemes	Class distribution				Number of Instances	%Reduction/ Increment	%Minority Class
		FAILBOTH	PASSNECO	PASSWAEC	PASSBOTH			
1	RAW DATA	775	248	45	95	1163		4
2	CNN	195	211	45	77	528	55	9
3	ENN	699	86	1	60	846	27	0
4	RUS	45	45	45	45	180	85	25
5	NCL	657	248	45	95	1045	10	4
6	5ENN	699	86	1	60	846	27	0
7	SMOTE	775	248	90	95	1208	-4	7
8	SMOTE300	775	248	180	95	1298	-12	14
9	SMOTEENN	699	83	10	56	848	27	1
10	SMOTENCL	661	248	90	95	1094	6	8
11	SMOTERUS	90	90	90	90	360	69	25
12	SMOTE300ENN	699	82	103	16	900	23	11
13	SMOTE300NCL	661	248	180	95	1184	-2	15
14	SMOTE300RUS	95	95	95	95	380	67	25

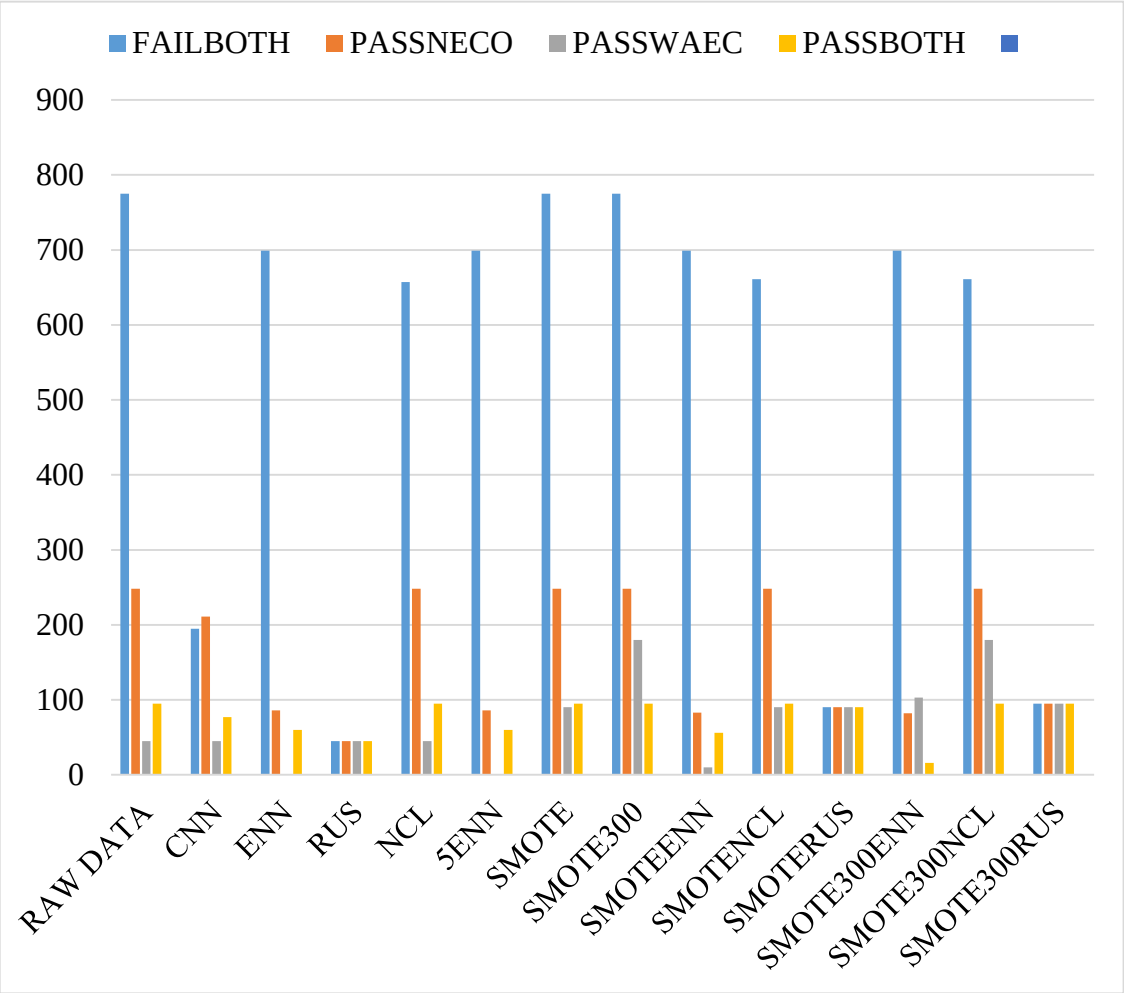


Figure 4.3: Chart showing the SSS Result Dataset class distribution

Table 4.5: TB Dataset class distribution

Schemes	Class distribution				Number of Instances	% Minority Class
	PTB	RPTB	EPTB	REPTB		
RAW DATA	589	124	37	6	758	0.79

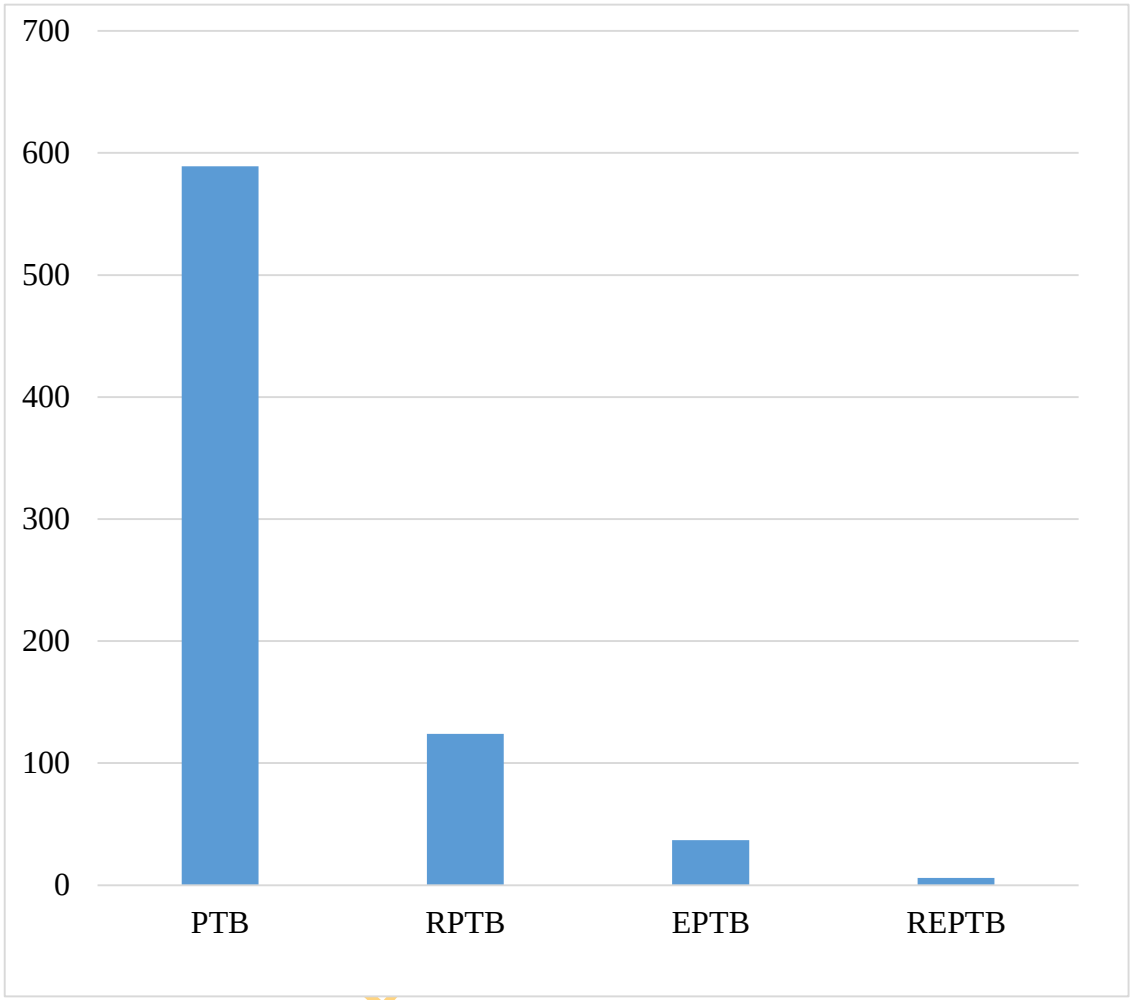


Figure 4.4: Chart showing the TB Dataset class distribution

Table 4.6: CM Dataset class distribution

S/N	Data Sampling Schemes	Class distribution					Number of Instances	%Reduction /Increment	%Minority Class
		SECONDARY+	SECONDARY	PRIMARY	PRIMARY+	NONE			
1	RAW DATA	414	247	45	53	16	247		7
2	CNN	275	201	44	50	15	585	25	3
3	ENN	386	209	36	43	14	688	11	2
4	RUS	16	16	16	16	16	80	90	20
5	NCL	337	247	45	53	16	698	10	2
6	5ENN	384	196	26	27	9	642	17	1
7	SMOTE	414	247	45	53	32	791	-2	4
8	SMOTE300	414	247	45	53	64	823	-6	8
9	SMOTEENN	386	210	35	35	27	702	9	4
10	SMOTENCL	337	247	45	53	32	714	8	5
11	SMOTERUS	32	32	32	32	32	160	79	20
12	SMOTE300ENN	385	207	34	45	58	729	6	8
13	SMOTE300NCL	335	247	45	53	64	744	4	9
14	SMOTE300RUS	45	45	45	45	45	225	71	20

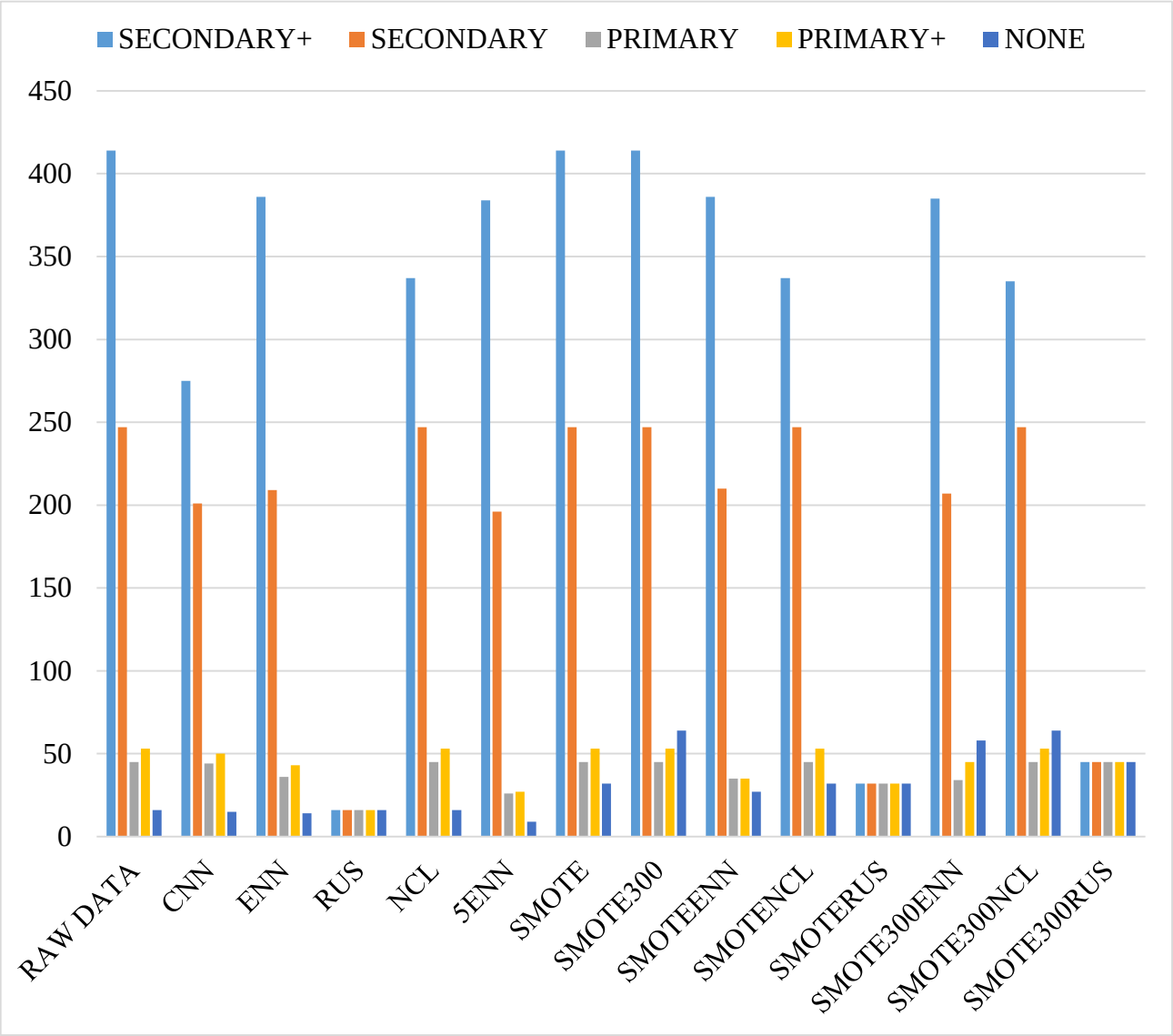


Figure 4.5: Chart showing the CM Dataset class distribution

4.1.4 Analysis of classification results of performance metrics on all datasets

This sub section presents performance metrics used for the classification results obtained from training the 13 datasets created from the data sampling schemes and the RAW DATA on the 14 classifiers. The performance metrics are ROC_AUC, Kappa Statistics, RMSE, RECALL of the minority class, and Performance loss/gain. The values for the best performing scheme are in bold. The screen shots for all results obtained in WEKA's API are presented in Appendix A.

4.1.4.1 Analysis of ROC_AUC metrics

The results obtained using the ROC_AUC metric on all data sampling schemes including RAW DATA on all the 14 classifiers on DM dataset is presented in Table 4.7 and displayed in Figure 4.6. It was observed that SMOTE300ENN, one of the enhanced data sampling schemes consistently gave the best performance with the majority of the classifiers. SMOTEENN, one the existing data sampling schemes gave a comparable performance to SMOTE300ENN. The class boundary diagram for visualising the class probability estimates of this dataset with all data sampling schemes is presented in Appendix B

The ROC_AUC metric values for SSS Result dataset is presented in Table 4.8 and charted in Figure 4.7. It was observed that SMOTE300ENN, one of the enhanced data sampling schemes gave the best performance. 5ENN, ENN and SMOTEENN which are all existing data sampling schemes also had comparable performance in many instances.

The result obtained for the CM dataset is presented in Table 4.9 and plotted in Figure 4.8. It was observed that SMOTE300RUS had the best performance followed by SMOTE300ENN.

Table 4.7: ROC_AUC metric values for DM dataset

Scheme Learner	SMOTE 300ENN	5ENN	CNN	RAW DATA	ENN	NCL	RUS	SMOTE	SMOTE 300	SMOTE ENN	SMOTE NCL	SMOTE RUS	SMOTE 300NCL	SMOTE 300RUS
RIPPER	**0.90**	0.61	0.72	0.63	0.65	0.84	0.80	0.76	0.79	0.88	0.80	0.82	0.79	0.79
Decision Tree	0.95	0.87	0.67	0.82	0.87	0.86	0.80	0.86	0.87	**0.97**	0.88	0.89	0.91	0.86
Random Forest	**0.98**	0.86	0.70	0.83	0.85	0.88	0.77	0.88	0.87	0.97	0.9	0.84	0.92	0.87
Random Tree	**0.88**	0.68	0.60	0.63	0.73	0.80	0.69	0.72	0.75	0.83	0.81	0.69	0.81	0.77
REPTree	0.96	0.80	0.71	0.74	0.85	0.94	0.83	0.73	0.81	**0.97**	0.81	0.87	0.87	0.84
MLP	**0.96**	0.74	0.67	0.82	0.64	0.86	0.61	0.85	0.88	0.94	0.90	0.73	0.89	0.75
SVM	**0.86**	0.50	0.50	0.50	0.50	0.54	0.59	0.51	0.69	0.51	0.60	0.76	0.74	0.80
1B3	**0.94**	0.80	0.23	0.74	0.77	0.77	0.59	0.79	0.82	0.88	0.76	0.76	0.85	0.83
Boosting	**0.99**	0.83	0.65	0.83	0.89	0.78	0.78	0.84	0.87	0.93	0.88	0.84	0.90	0.86
Bagging	**0.98**	0.88	0.67	0.85	0.89	0.87	0.81	0.88	0.89	0.97	0.89	0.88	0.91	0.87
MulticlassClassifier	0.96	0.87	0.70	0.83	0.86	0.86	0.80	0.86	0.88	**0.97**	0.88	0.87	0.87	0.84
RandomCommittee	**1.00**	0.89	0.67	0.83	0.89	0.90	0.79	0.86	0.87	0.95	0.91	0.86	0.92	0.88
Decision Forest	**0.98**	0.89	0.7	0.84	0.93	0.89	0.80	0.87	0.87	**0.98**	0.9	0.86	0.90	0.86
Stacking	**1.00**	0.86	0.80	0.77	0.87	0.84	0.70	0.83	0.87	0.95	0.88	0.82	0.88	0.86

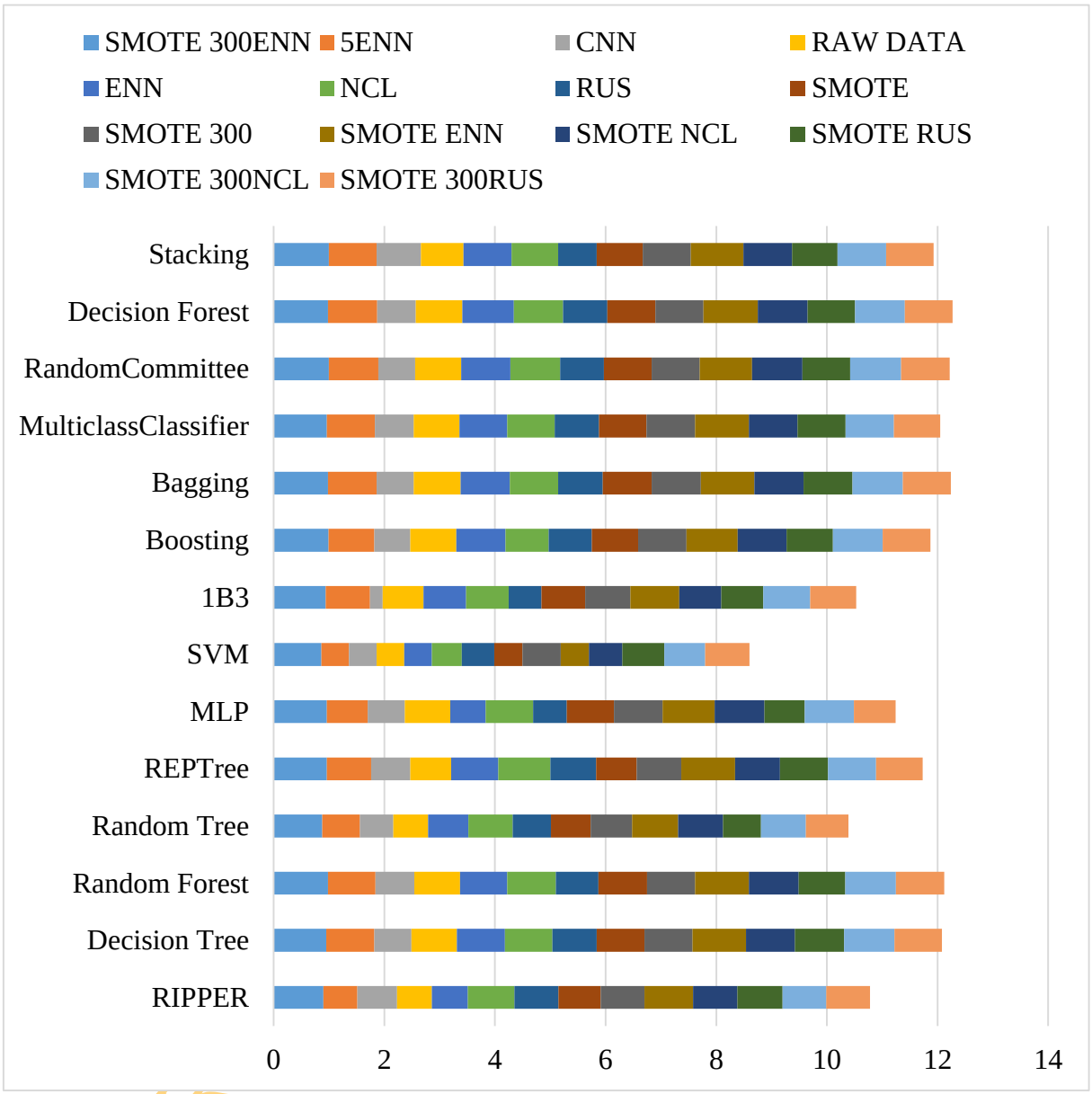


Figure 4.6: Chart showing the ROC_AUC metric values for DM dataset

Table 4.8: ROC_AUC metric values for SSS Result dataset

Scheme Learner	SMOTE 300ENN	5ENN	CNN	RAW DATA	ENN	NCL	RUS	SMOTE	SMOTE 300	SMOTE ENN	SMOTE NCL	SMOTE RUS	SMOTE 300NCL	SMOTE 300RUS
RIPPER	**0.99**	**0.99**	0.51	0.59	**0.99**	0.71	0.69	0.61	0.64	0.98	0.68	0.73	0.70	0.80
Decision Tree	**1.00**	**1.00**	0.56	0.81	**1.00**	0.86	0.70	0.82	0.84	**1.00**	0.86	0.77	0.88	0.81
Random Forest	**1.00**	**1.00**	0.53	0.81	**1.00**	0.86	0.69	0.82	0.84	**1.00**	0.87	0.76	0.88	0.80
Random Tree	**1.00**	**1.00**	0.52	0.81	**1.00**	0.86	0.68	0.82	0.83	**1.00**	0.87	0.75	0.88	0.80
REPTree	**1.00**	**1.00**	0.54	0.81	**1.00**	0.86	0.73	0.82	0.84	**1.00**	0.87	0.76	0.88	0.80
MLP	**0.96**	0.82	0.63	0.77	0.95	0.80	0.73	0.77	0.79	0.95	0.82	0.74	0.81	0.74
SVM	**0.95**	0.86	0.58	0.68	0.94	0.75	0.75	0.69	0.72	0.93	0.74	0.76	0.77	0.77
1B3	**1.00**	**1.00**	0.53	0.81	**1.00**	0.86	0.69	0.82	0.84	**1.00**	0.87	0.76	0.88	0.80
Boosting	**1.00**	**1.00**	0.55	0.79	**1.00**	0.86	0.68	0.82	0.83	**1.00**	0.87	0.72	0.88	0.75
Bagging	**1.00**	**1.00**	0.56	0.82	**1.00**	0.87	0.72	0.82	0.84	**1.00**	0.87	0.77	0.88	0.81
MulticlassClassifier	**1.00**	**1.00**	0.57	0.81	**1.00**	0.86	0.72	0.82	0.83	**1.00**	0.86	0.76	0.87	0.79
RandomCommittee	**1.00**	**1.00**	0.52	0.81	**1.00**	0.86	0.68	0.82	0.83	**1.00**	0.87	0.75	0.88	0.79
Decision Forest	**0.99**	**0.99**	0.60	0.82	**0.99**	0.87	0.74	0.82	0.84	**0.99**	0.87	0.77	0.87	0.81
Stacking	**0.99**	**0.99**	0.60	0.78	**0.99**	0.84	0.71	0.80	0.82	**0.99**	0.84	0.74	0.87	0.78

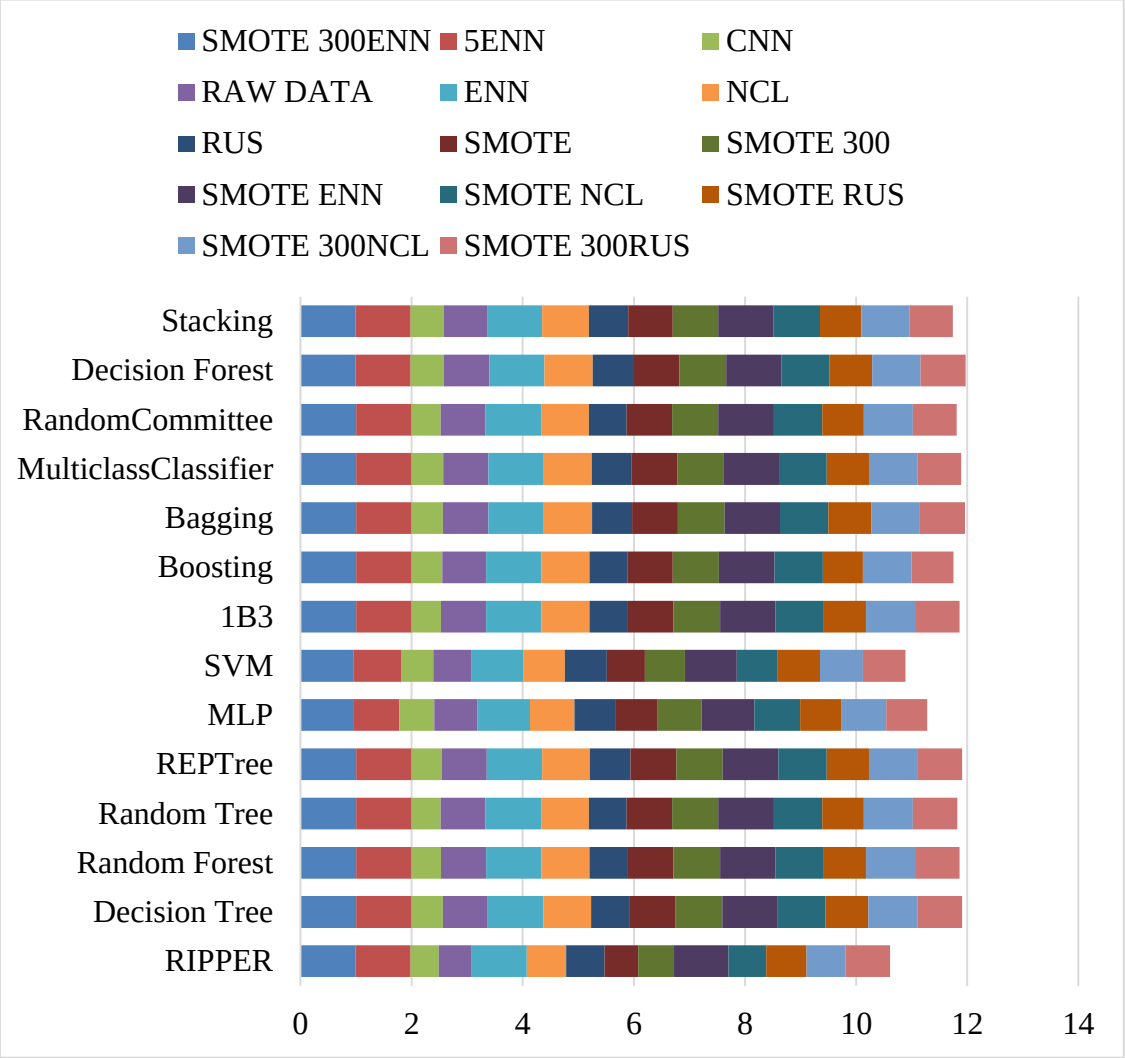


Figure 4.7: Chart showing the ROC_AUC metric values for SSS Result dataset

Table 4.9: ROC_AUC metric values for CM dataset

Scheme Learner	SMOTE 300ENN	5ENN	CNN	RAW DATA	ENN	NCL	RUS	SMOTE	SMOTE 300	SMOTE ENN	SMOTE NCL	SMOTE RUS	SMOTE 300NCL	SMOTE 300RUS
RIPPER	0.59	0.56	0.50	0.52	0.54	0.50	0.59	0.53	0.58	0.56	0.54	**0.65**	0.59	0.64
Decision Tree	**0.70**	0.58	0.52	0.57	0.61	0.57	0.48	0.58	0.61	0.59	0.59	0.62	0.64	0.67
Random Forest	0.70	0.67	0.52	0.61	0.64	0.64	0.53	0.62	0.65	0.66	0.62	0.64	0.68	**0.71**
Random Tree	0.65	0.59	0.48	0.56	0.57	0.58	0.52	0.54	0.61	0.61	0.62	0.62	0.63	**0.70**
REPTree	0.62	0.53	0.47	0.53	0.49	0.50	0.43	0.49	0.61	0.50	0.50	0.62	0.60	**0.64**
MLP	0.55	0.59	0.51	0.52	0.59	0.54	0.48	0.55	0.56	0.59	0.56	0.63	0.57	**0.67**
SVM	0.68	0.64	0.53	0.58	0.63	0.62	0.64	0.60	0.64	0.65	0.62	0.67	0.66	**0.71**
1B3	0.67	0.67	0.42	0.58	0.61	0.59	0.59	0.58	0.63	0.64	0.61	0.65	0.64	**0.72**
Boosting	**0.69**	0.61	0.53	0.57	0.64	0.61	0.50	0.60	0.62	0.62	0.61	0.68	0.65	0.65
Bagging	0.64	0.54	0.51	0.52	0.55	0.55	0.49	0.55	0.58	0.58	0.56	0.65	0.62	**0.69**
MulticlassClassifier	**0.66**	0.55	0.52	0.55	0.56	0.55	0.50	0.56	0.61	0.58	0.57	0.63	0.63	0.65
RandomCommittee	**0.69**	0.64	0.48	0.58	0.61	0.63	0.60	0.61	0.63	0.64	0.61	0.67	0.67	0.68
Decision Forest	0.65	0.57	0.52	0.56	0.59	0.59	0.51	0.57	0.63	0.60	0.61	0.66	0.66	**0.69**
Stacking	0.66	0.64	0.59	0.52	0.59	0.56	0.54	0.58	0.61	0.61	0.57	0.62	0.64	**0.68**

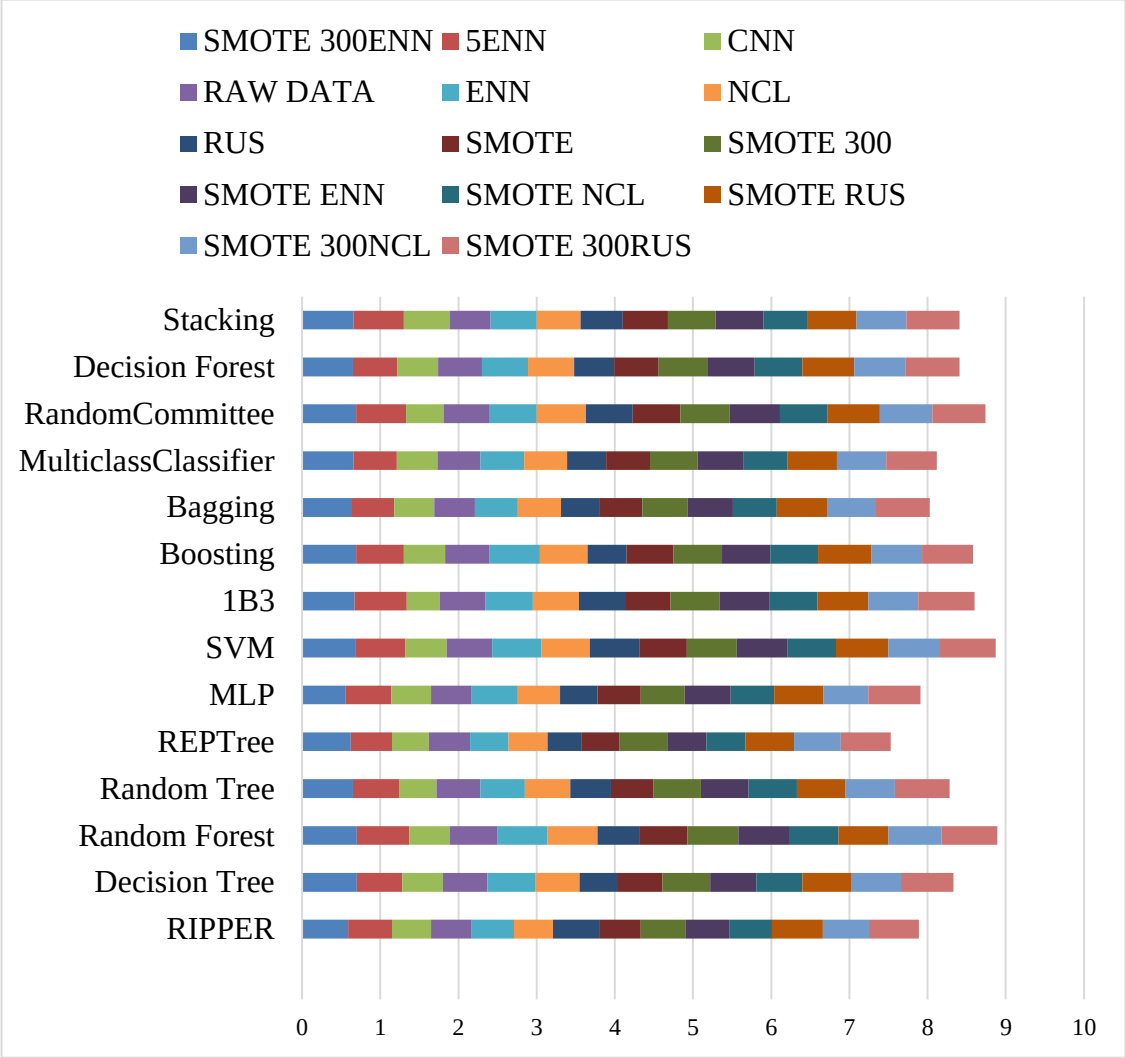


Figure 4.8: Chart showing the ROC_AUC metric values for CM dataset

4.1.4.2 Analysis of Kappa statistics metrics

Kappa statistics metric values are presented in Table 4.10 and also plotted in Figure 4.9 for all results obtained on all data sampling schemes on all the 14 classifiers on DM dataset. It was also observed that SMOTE300ENN, one of the enhanced data sampling schemes consistently had the best performance.

The report on Kappa statistics metric for SSS Result dataset is presented in Table 4.11 and charted in Figure 4.10. It was observed that SMOTE300ENN, one of the enhanced data sampling schemes had the best performance

The results for CM dataset is presented in Table 4.12 and also plotted in Figure 4.11. The values showed that SMOTE300ENN and SMOTE300RUS which are two of the enhanced data sampling schemes had the best performance.

Table 4.10: Kappa Statistic metric values for DM dataset

Scheme Learner	SMOTE 300ENN	5ENN	CNN	RAW DATA	ENN	NCL	RUS	SMOTE	SMOTE 300	SMOTE ENN	SMOTE NCL	SMOTE RUS	SMOTE 300NCL	SMOTE 300RUS
RIPPER	**0.84**	0.25	0.35	0.23	0.41	0.69	0.56	0.53	0.60	0.74	0.65	0.63	0.65	0.54
Decision Tree	**0.81**	0.30	0.25	0.31	0.26	0.59	0.50	0.48	0.55	0.75	0.65	0.68	0.68	0.53
Random Forest	**0.85**	0.21	0.17	0.32	0.23	0.64	0.44	0.46	0.60	0.75	0.70	0.59	0.70	0.58
Random Tree	**0.80**	0.34	0.21	0.22	0.38	0.54	0.38	0.42	0.49	0.61	0.59	0.38	0.56	0.52
REPTree	**0.84**	0.53	0.31	0.44	0.54	0.60	0.59	0.52	0.59	0.76	0.68	0.68	0.70	0.64
MLP	**0.77**	0.21	0.10	0.25	0.12	0.52	0.12	0.27	0.55	0.45	0.52	0.34	0.61	0.42
SVM	**0.77**	0.00	0.00	0.00	0.00	0.12	0.15	0.00	0.50	0.00	0.27	0.47	0.55	0.54
1B3	**0.72**	-0.00	-0.13	0.19	-0.00	0.28	0.15	0.31	0.49	0.44	0.32	0.35	0.56	0.44
Boosting	**0.88**	0.36	0.23	0.33	0.38	0.60	0.47	0.50	0.54	0.80	0.67	0.62	0.67	0.53
Bagging	**0.84**	0.33	0.25	0.36	0.14	0.65	0.62	0.52	0.62	0.76	0.71	0.69	0.71	0.59
MulticlassClassifier	**0.82**	0.10	0.65	0.29	0.03	0.61	0.56	0.50	0.59	0.76	0.68	0.63	0.70	0.58
RandomCommittee	**0.88**	0.20	0.18	0.31	0.30	0.63	0.53	0.46	0.56	0.76	0.69	0.60	0.71	0.60
Decision Forest	**0.57**	0.00	0.132	0.07	0.00	0.12	0.53	0.26	0.33	0.38	0.48	0.60	0.49	0.58
Stacking	**0.81**	0.40	0.25	0.27	0.25	0.51	0.24	0.38	0.53	0.65	0.65	0.49	0.66	0.57

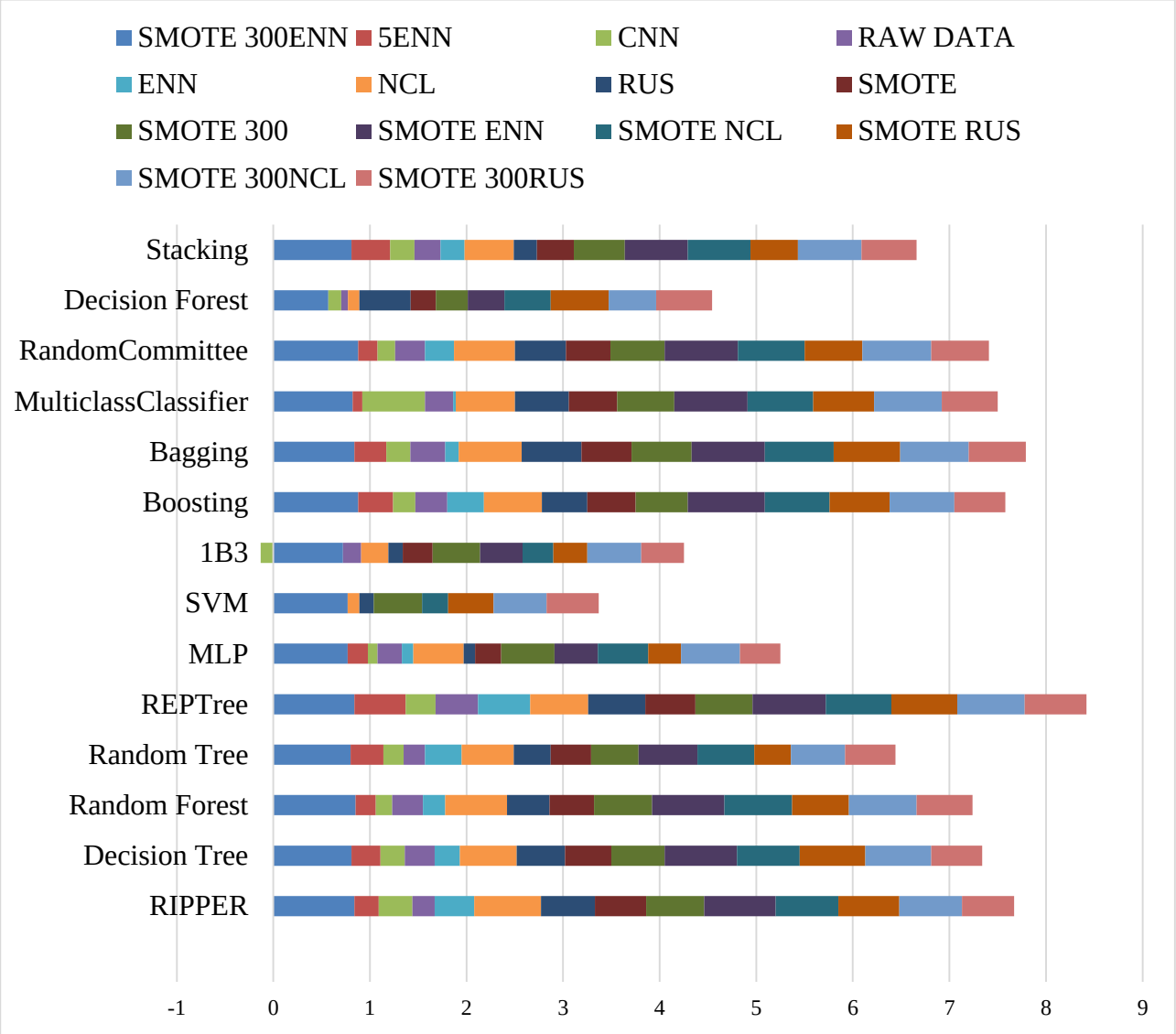


Figure 4.9: Chart showing the Kappa Statistics metric values for DM dataset

Table 4.11: Kappa Statistic metric for SSS Result dataset

Scheme Learner	SMOTE 300ENN	5ENN	CNN	RAW DATA	ENN	NCL	RUS	SMOTE	SMOTE 300	SMOTE ENN	SMOTE NCL	SMOTE RUS	SMOTE 300NCL	SMOTE 300RUS
RIPPER	**0.99**	0.98	-0.11	0.20	**0.99**	0.42	0.21	0.24	0.29	0.97	0.35	0.31	0.38	0.47
Decision Tree	**0.99**	0.98	-0.14	0.39	**0.99**	0.53	0.24	0.38	0.42	0.98	0.52	0.42	0.54	0.45
Random Forest	0.99	**1.00**	-0.16	0.40	0.99	0.76	0.29	0.39	0.43	0.99	0.51	0.41	0.52	0.44
Random Tree	**0.99**	**0.99**	-0.21	0.38	**0.99**	0.51	0.20	0.38	0.42	**0.99**	0.50	0.42	0.53	0.46
REPTree	**0.99**	0.98	-0.16	0.38	**0.99**	0.51	0.35	0.37	0.42	**0.99**	0.52	0.43	0.53	0.46
MLP	**0.74**	0.62	0.03	0.29	0.71	0.42	0.44	0.24	0.32	0.68	0.41	0.29	0.35	0.25
SVM	**0.91**	0.80	0.14	0.37	0.90	0.48	0.36	0.37	0.37	0.86	0.47	0.43	0.45	0.44
1B3	**0.99**	0.97	-0.21	0.38	0.98	0.50	0.2	0.37	0.69	0.98	0.50	0.41	0.52	0.46
Boosting	**0.99**	0.98	-0.05	0.37	**0.99**	0.51	0.27	0.38	0.43	**0.99**	0.50	0.40	0.51	0.41
Bagging	**0.98**	**0.98**	-0.12	0.39	**0.98**	0.53	0.34	0.38	0.42	**0.98**	0.52	0.43	0.52	0.45
MulticlassClassifier	**0.99**	0.98	-0.16	0.35	**0.99**	0.53	0.27	0.38	0.41	0.98	0.52	0.43	0.51	0.45
RandomCommittee	**0.99**	0.98	-0.21	0.38	**0.99**	0.51	0.21	0.38	0.41	**0.99**	0.51	0.43	0.53	0.47
Decision Forest	0.79	0.76	-0.06	0.34	0.78	0.49	0.35	0.34	0.38	**0.80**	0.47	0.44	0.48	0.44
Stacking	**0.99**	**0.99**	0.11	0.31	0.98	0.51	0.27	0.35	0.41	0.98	0.48	0.32	0.50	0.36

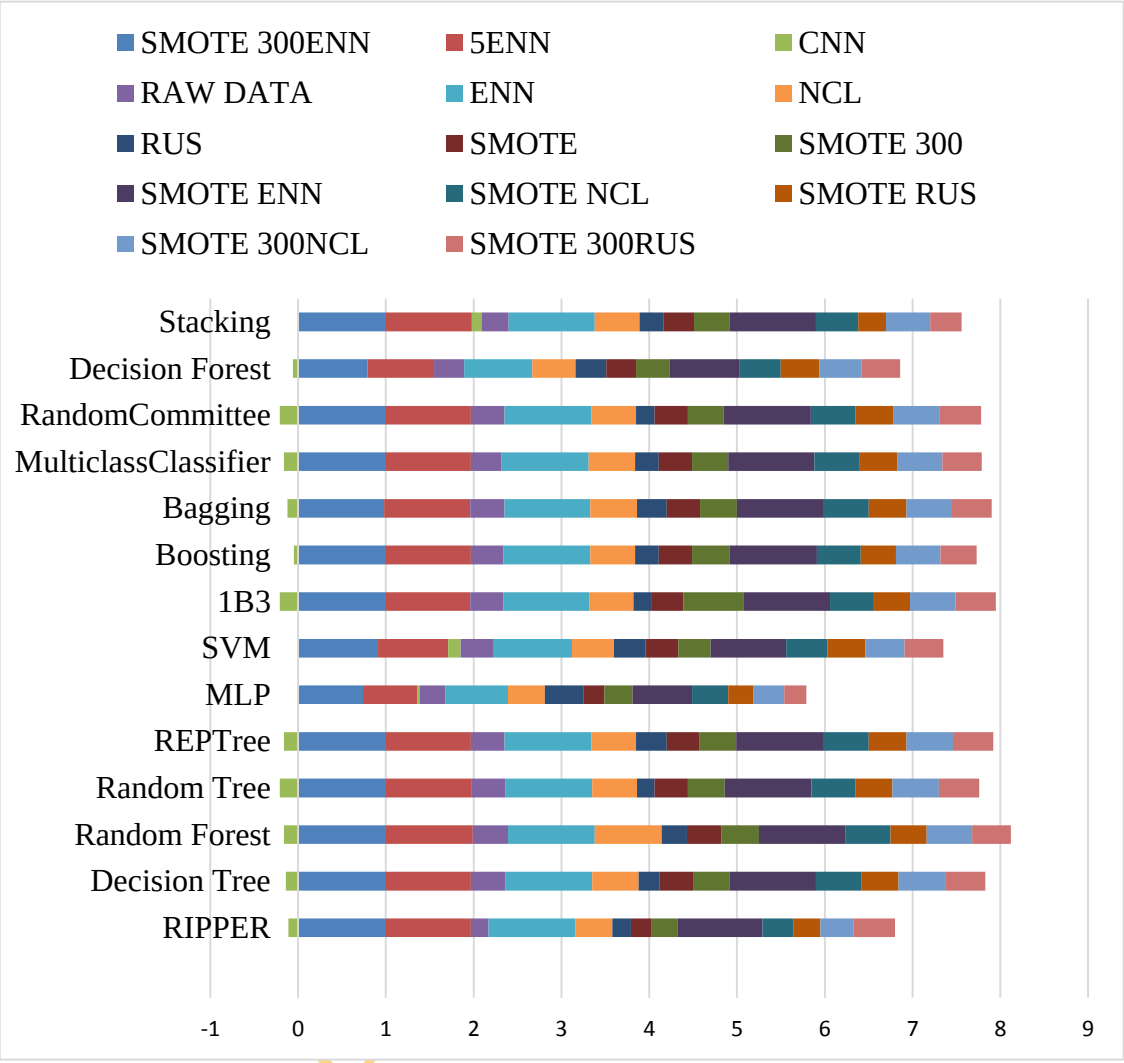


Figure 4.10: Chart showing the Kappa Statistics metric values for SSS Result dataset

Table 4.12: Kappa Statistics metric values for CM dataset

Scheme Learner	SMOTE 300ENN	5ENN	CNN	RAW DATA	ENN	NCL	RUS	SMOTE	SMOTE 300	SMOTE ENN	SMOTE NCL	SMOTE RUS	SMOTE 300NCL	SMOTE 300RUS
RIPPER	0.21	0.14	0.03	0.05	0.10	0.05	0.11	0.10	0.19	0.14	0.11	0.25	0.18	**0.23**
Decision Tree	**0.25**	0.04	0.03	0.05	0.08	0.08	-0.03	0.08	0.19	0.15	0.11	0.09	0.20	0.21
Random Forest	**0.23**	0.09	-0.05	0.07	0.05	0.11	0.02	0.10	0.19	0.14	0.10	0.13	0.22	**0.23**
Random Tree	0.21	0.11	-0.06	0.04	0.04	0.09	0.02	0.03	0.15	0.11	0.13	0.14	0.17	**0.26**
REPTree	0.17	0.01	-0.11	-0.02	0.00	0.01	-0.08	0.00	0.14	0.00	0.01	0.12	0.16	**0.21**
MLP	0.05	0.06	-0.02	-0.01	0.07	0.05	-0.06	0.01	0.11	0.03	0.01	0.13	0.09	**0.19**
SVM	**0.33**	0.24	0.03	0.13	0.22	0.20	0.19	0.17	0.25	0.29	0.20	0.23	0.29	0.32
1B3	0.25	0.24	-0.11	0.11	0.13	0.12	0.06	0.12	0.19	0.21	0.16	0.22	0.22	**0.28**
Boosting	**0.24**	0.10	0.03	0.03	0.12	0.11	0.05	0.10	0.17	0.15	0.12	0.18	0.20	0.18
Bagging	**0.26**	0.02	0.00	0.03	0.04	0.07	-0.06	0.08	0.16	0.14	0.11	0.14	0.18	0.16
MulticlassClassifier	**0.25**	0.03	0.02	0.06	0.07	0.07	0.02	0.09	0.19	0.13	0.11	0.15	0.23	0.22
RandomCommittee	0.20	0.07	-0.07	0.01	0.05	0.12	0.11	0.09	0.18	0.13	0.12	0.16	0.22	**0.23**
Decision Forest	**0.23**	0.06	0.00	0.03	0.07	0.13	0.03	0.07	0.17	0.15	0.11	0.20	0.21	0.21
Stacking	0.20	0.17	0.07	0.01	0.10	0.06	0.08	0.13	0.17	0.15	0.08	0.17	0.19	**0.24**

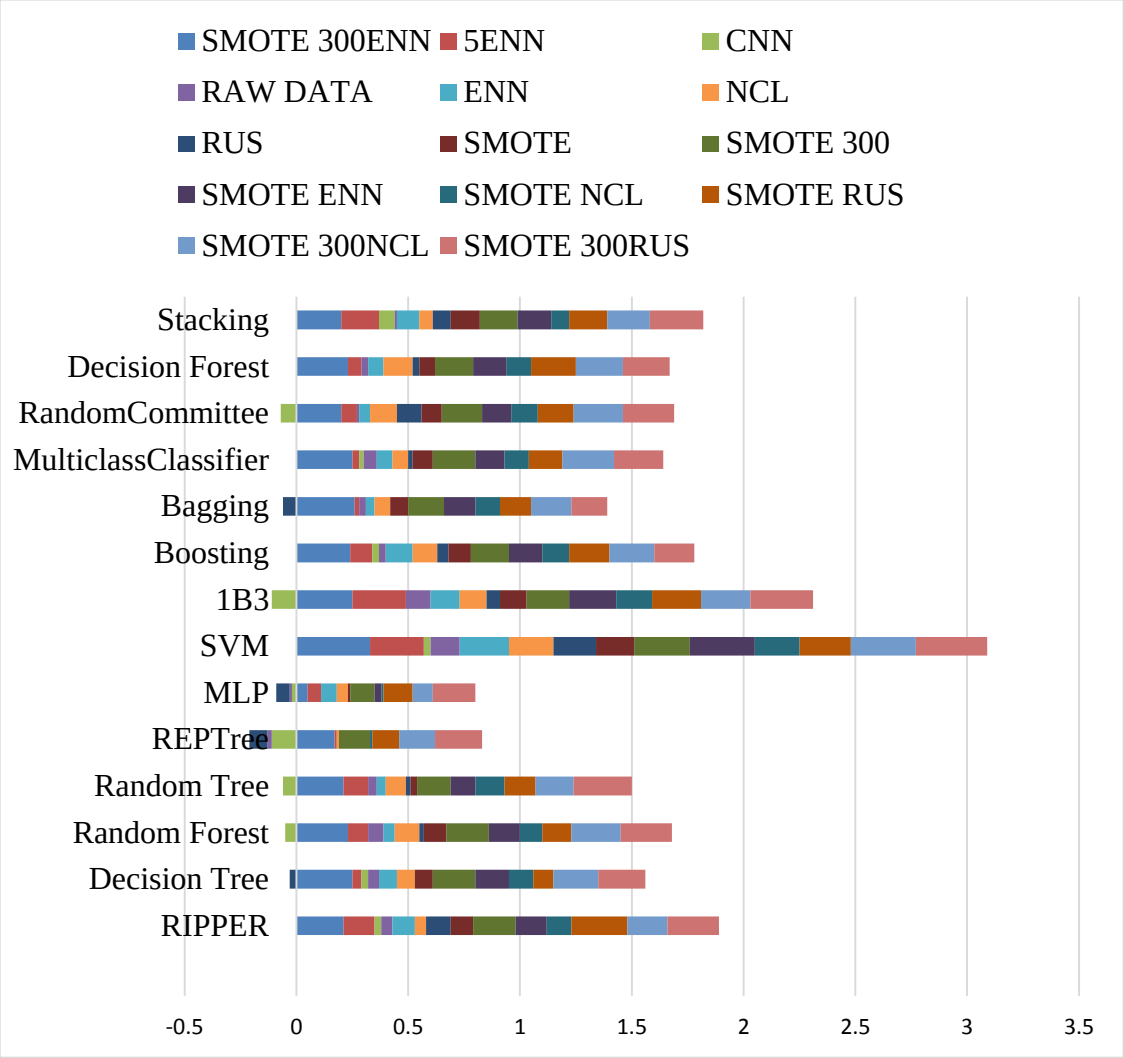


Figure 4.11: Chart showing the Kappa Statistics metric values for CM dataset

4.1.4.3 Analysis of RMSE metrics

A low RMSE value indicates a better performance. This means that the lower the RMSE value for a data sampling scheme, the better the classification performance of the data sampling scheme.

The results obtained for the RMSE metric on all data sampling schemes on all the 14 classifiers used to train the DM dataset is presented in Table 4.13 and plotted in Figure 4.12. It was observed that ENN, SMOTEENN and 5ENN, all part of the existing data sampling schemes had the lowest values.

The RMSE values for SSS Result dataset is presented in Table 4.14 and charted in Figure 4.13. It was observed that ENN, SMOTE300ENN, 5ENN and SMOTEENN data sampling schemes had the lowest RMSE values respectively.

The RMSE values obtained for CM dataset is presented in Table 4.15 and plotted in Figure 4.14. It was observed that 5ENN data sampling schemes had the best classification performance for the majority of the classifiers.

Table 4.13: RMSE metric values for DM dataset

Scheme Learner	SMOTE 300ENN	5ENN	CNN	RAW DATA	ENN	NCL	RUS	SMOTE	SMOTE 300	SMOTE ENN	SMOTE NCL	SMOTE RUS	SMOTE 300NCL	SMOTE 300RUS
RIPPER	0.13	0.13	0.35	0.23	**0.11**	0.18	0.39	0.22	0.23	**0.11**	0.22	0.40	0.24	0.44
Decision Tree	0.13	0.11	0.38	0.22	**0.10**	0.20	0.40	0.22	0.23	**0.10**	0.22	0.33	0.22	0.37
Random Forest	0.11	0.11	0.38	0.21	**0.10**	0.19	0.42	0.21	0.21	**0.10**	0.19	0.37	0.20	0.24
Random Tree	0.14	0.14	0.46	0.29	**0.13**	0.24	0.52	0.28	0.29	0.15	0.24	0.52	0.29	0.46
REPTree	0.13	0.11	0.39	0.22	**0.10**	0.19	0.39	0.23	0.23	**0.10**	0.19	0.35	0.22	0.36
MLP	0.14	**0.11**	0.38	0.22	**0.11**	0.19	0.47	0.22	0.23	0.13	0.21	0.43	0.22	0.41
SVM	0.29	**0.28**	0.39	0.31	**0.28**	0.32	0.50	0.32	0.32	0.29	0.32	0.41	0.32	0.40
1B3	0.15	**0.12**	0.49	0.24	**0.12**	0.25	0.58	0.25	0.25	0.15	0.25	0.46	0.26	0.41
Boosting	0.11	0.12	0.44	0.25	0.11	0.21	0.46	0.24	0.26	**0.10**	0.20	0.40	0.23	0.43
Bagging	0.12	**0.10**	0.37	0.21	**0.10**	0.18	0.38	0.21	0.22	**0.10**	0.19	0.34	0.20	0.36
MulticlassClassifier	**0.30**	**0.30**	0.41	0.33	**0.30**	0.32	0.41	0.33	0.33	**0.30**	0.32	0.40	0.33	0.41
RandomCommittee	**0.09**	0.10	0.39	0.22	0.10	0.18	0.41	0.22	0.22	**0.09**	0.19	0.37	0.20	0.35
Decision Forest	0.16	**0.03**	0.36	0.21	**0.10**	0.20	0.42	0.21	0.24	0.13	0.21	0.39	0.23	0.38
Stacking	0.13	**0.10**	0.38	0.23	0.11	0.20	0.47	0.25	0.24	0.12	0.19	0.40	0.22	0.37

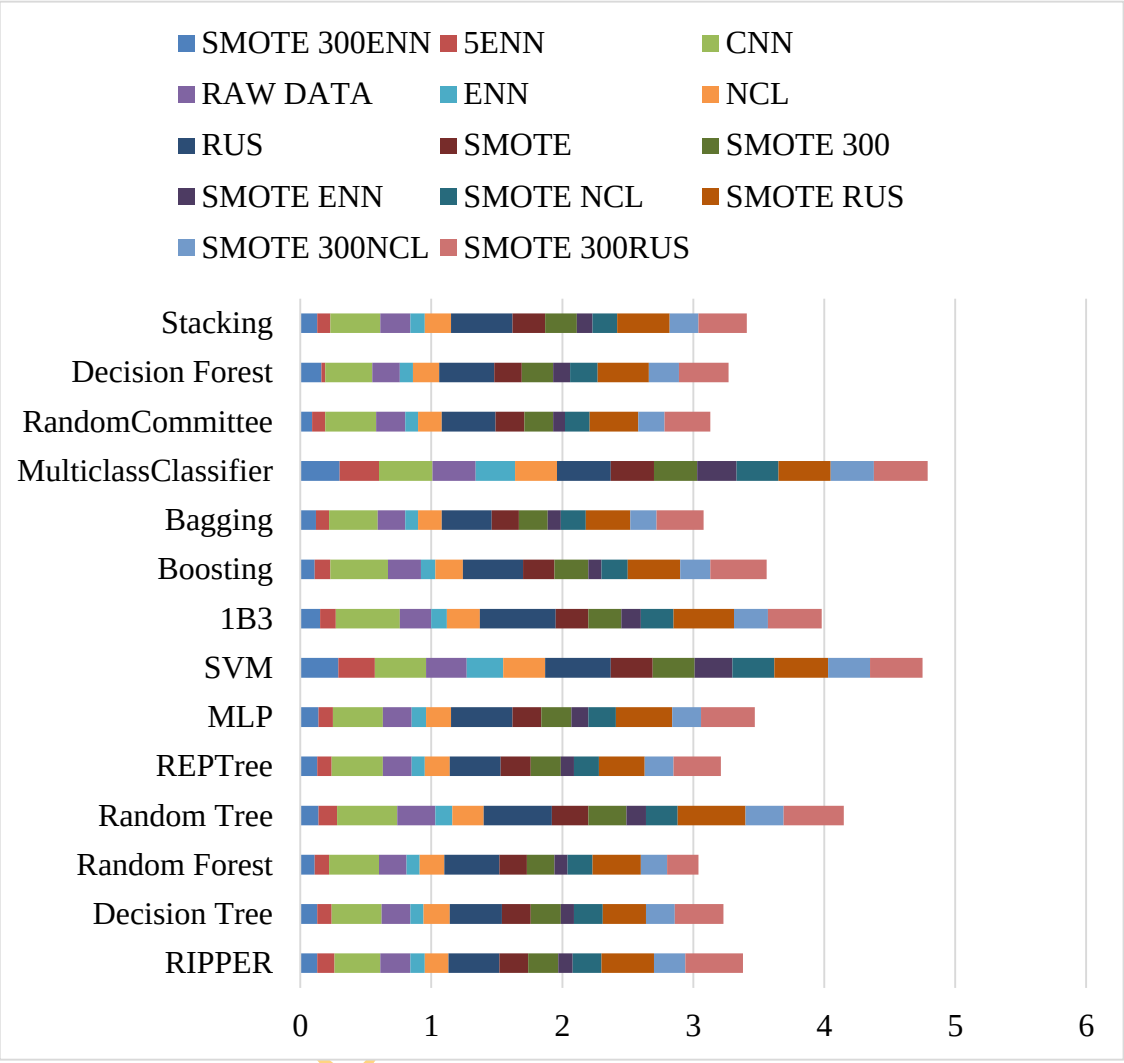


Figure 4.12: Chart showing the RMSE metric values for DM dataset

Table 4.14: RMSE metric for SSS Result dataset

Scheme Learner	SMOTE 300ENN	5ENN	CNN	RAW DATA	ENN	NCL	RUS	SMOTE	SMOTE 300	SMOTE ENN	SMOTE NCL	SMOTE RUS	SMOTE 300NCL	SMOTE 300RUS
RIPPER	**0.04**	0.05	0.42	0.34	**0.04**	0.32	0.42	0.35	0.36	0.06	0.34	0.40	0.35	0.37
Decision Tree	**0.07**	0.06	0.41	0.31	**0.07**	0.29	0.41	0.32	0.32	0.07	0.30	0.38	0.30	0.37
Random Forest	**0.03**	**0.03**	0.43	0.31	0.04	0.29	0.43	0.32	0.32	0.04	0.30	0.39	0.30	0.38
Random Tree	0.04	**0.03**	0.44	0.32	0.04	0.29	0.44	0.32	0.32	0.04	0.30	0.40	0.30	0.38
REPTree	0.05	**0.04**	0.43	0.31	**0.04**	0.29	0.40	0.32	0.32	0.05	0.30	0.39	0.30	0.37
MLP	0.19	0.19	0.40	0.32	**0.17**	0.32	0.40	0.34	0.34	0.18	0.33	0.39	0.34	0.40
SVM	**0.32**	**0.32**	0.41	0.36	**0.32**	0.36	0.40	0.37	0.37	**0.32**	0.36	0.39	0.37	0.39
1B3	**0.04**	0.06	0.43	0.31	**0.04**	0.29	0.44	0.32	0.32	**0.04**	0.30	0.39	0.30	0.38
Boosting	**0.04**	0.05	0.42	0.34	0.04	0.31	0.42	0.35	0.34	0.05	0.32	0.41	0.32	0.40
Bagging	**0.07**	**0.07**	0.41	0.31	**0.07**	0.29	0.40	0.32	0.32	0.08	0.30	0.38	0.30	0.37
MulticlassClassifier	**0.35**	**0.35**	0.42	0.39	**0.35**	0.39	0.42	0.39	0.40	**0.35**	0.39	0.41	0.39	0.41
RandomCommittee	0.04	0.05	0.43	0.31	0.04	0.29	0.44	0.32	0.32	**0.03**	0.30	0.39	0.30	0.38
Decision Forest	0.15	0.15	0.40	0.31	**0.14**	0.30	0.40	0.32	0.32	0.15	0.31	0.38	0.31	0.37
Stacking	0.06	**0.05**	0.42	0.33	**0.05**	0.30	0.42	0.33	0.33	0.06	0.31	0.40	0.31	0.39

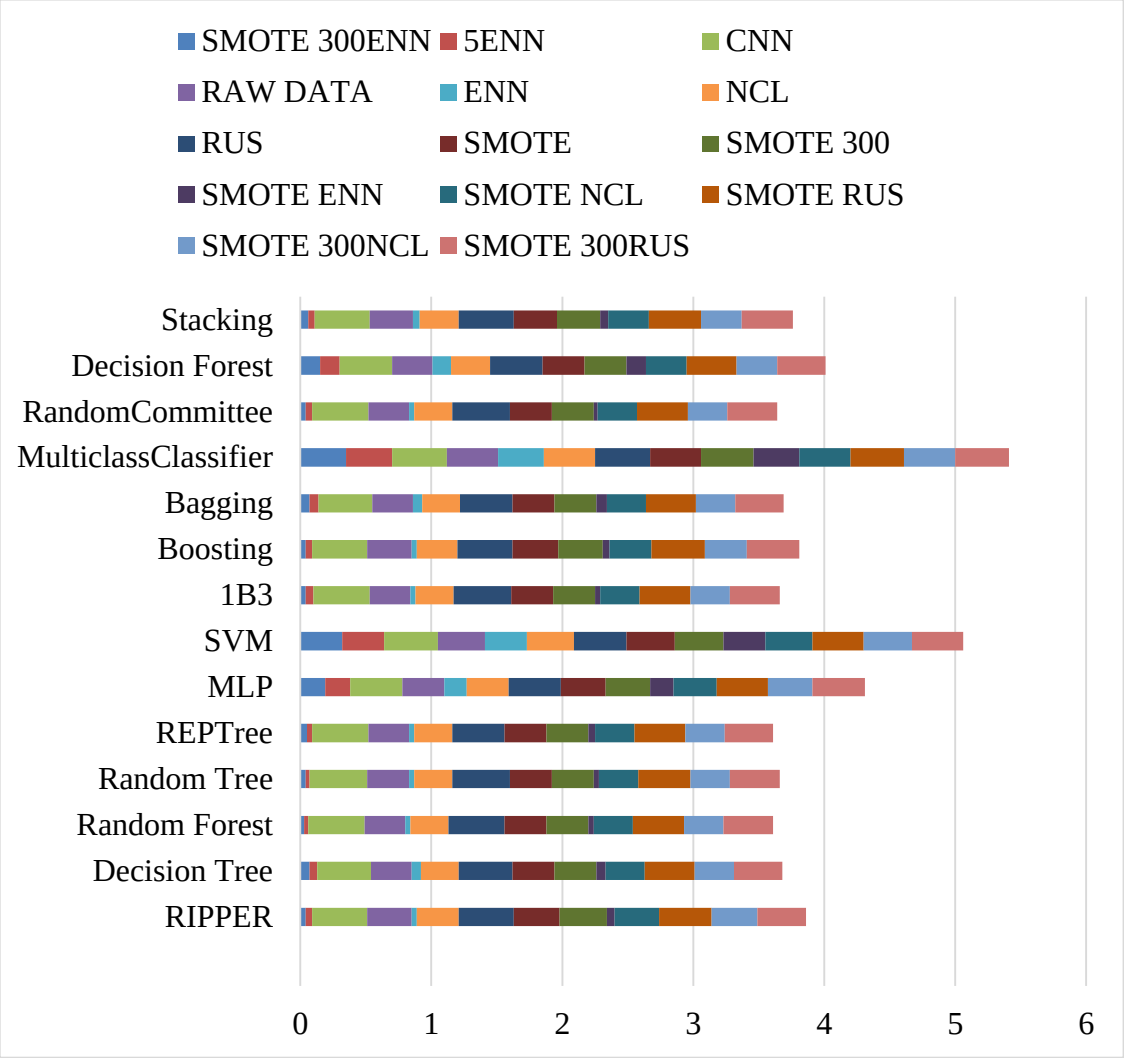


Figure 4.13: Chart showing the RMSE metric values for SSS Result dataset

Table 4.15: RMSE metric values for CM dataset

Scheme Learner	SMOTE 300ENN	5ENN	CNN	RAW DATA	ENN	NCL	RUS	SMOTE	SMOTE 300	SMOTE ENN	SMOTE NCL	SMOTE RUS	SMOTE 300NCL	SMOTE 300RUS
RIPPER	**0.34**	**0.34**	0.38	0.37	0.37	0.38	0.46	0.36	0.35	0.36	0.38	0.44	0.36	0.43
Decision Tree	**0.34**	**0.34**	0.38	0.36	0.35	0.37	0.43	0.36	**0.34**	0.35	0.36	0.41	0.36	0.42
Random Forest	0.33	**0.32**	0.36	0.34	0.34	0.35	0.40	0.34	**0.32**	0.33	0.35	0.38	0.34	0.36
Random Tree	0.36	**0.35**	0.41	0.38	0.37	0.38	0.44	0.39	0.36	0.36	0.38	0.41	0.38	0.40
REPTree	0.36	0.37	0.41	0.38	**0.34**	0.36	0.43	0.35	0.36	0.35	0.36	0.41	0.38	0.42
MLP	0.35	**0.33**	0.36	0.35	0.34	0.36	0.40	0.35	0.34	0.35	0.36	0.38	0.35	0.36
SVM	**0.35**	**0.35**	0.37	0.36	0.36	0.36	0.40	0.36	0.36	**0.35**	0.36	0.39	0.36	0.38
1B3	0.36	**0.34**	0.43	0.38	0.37	0.39	0.50	0.38	0.36	0.36	0.39	0.45	0.38	0.41
Boosting	**0.35**	0.37	0.38	0.38	0.36	0.37	0.41	0.37	0.36	0.37	0.37	0.39	0.37	0.40
Bagging	0.35	0.35	0.37	0.37	0.36	0.37	0.40	0.36	**0.34**	0.35	0.37	0.38	0.36	0.37
MulticlassClassifier	**0.39**	**0.39**	**0.39**	**0.39**	**0.39**	**0.39**	0.40	**0.39**	**0.39**	**0.39**	**0.39**	0.40	**0.39**	**0.39**
RandomCommittee	**0.33**	**0.33**	0.38	0.35	0.34	0.35	0.40	0.35	**0.33**	0.34	0.35	0.38	0.34	0.37
Decision Forest	0.35	0.35	0.37	0.36	0.36	0.37	0.40	0.36	**0.34**	0.35	0.36	0.38	0.36	0.37
Stacking	0.36	**0.35**	0.39	0.39	0.37	0.39	0.42	0.38	0.36	0.36	0.39	0.40	0.37	0.38

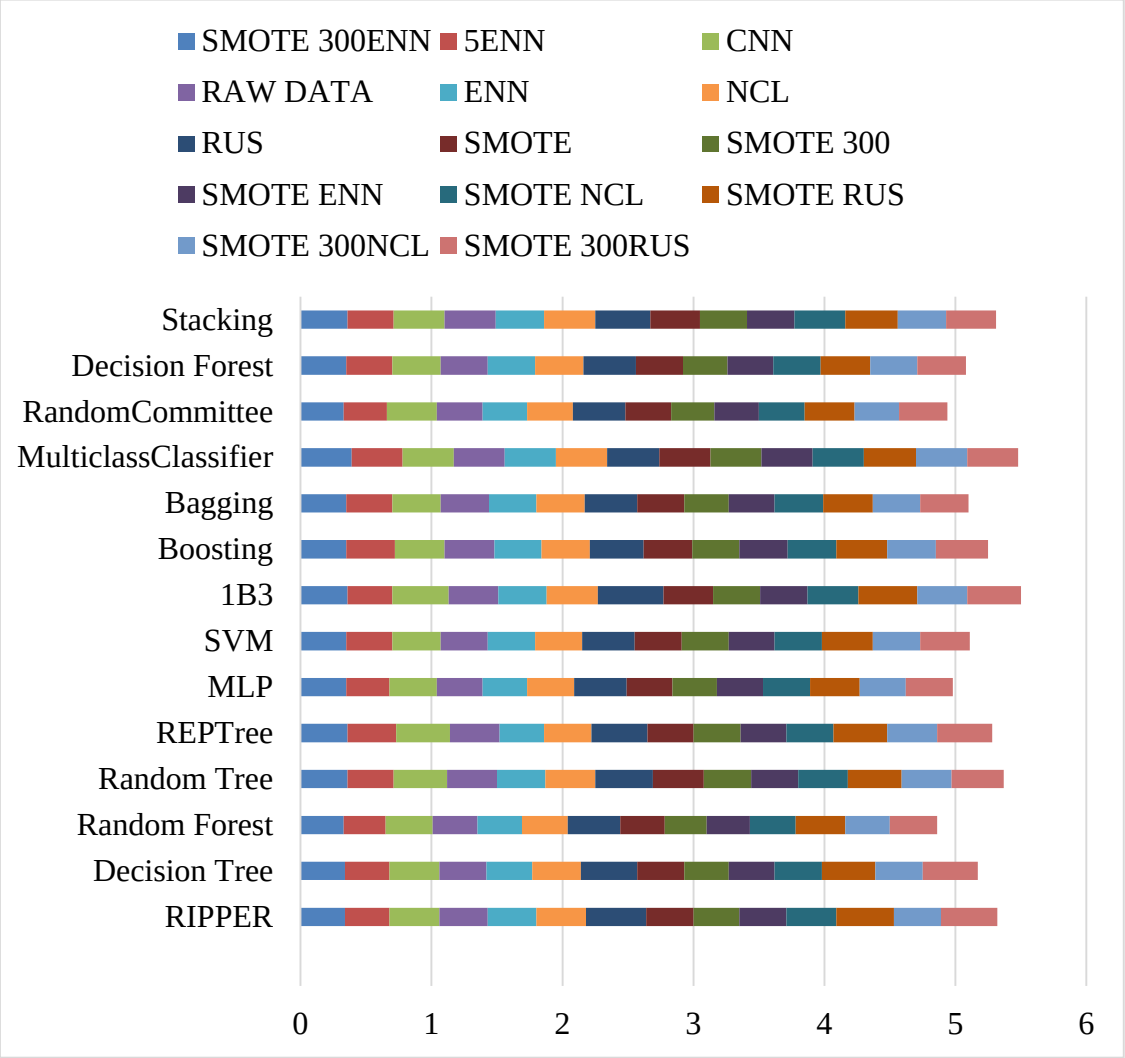


Figure 4.14: Chart showing the RMSE metric values for CM dataset

4.1.4.4 Analysis of RECALL of minority class metrics

RECALL of minority class (GDM) metric results obtained for DM datasets are presented in Table 4.16 and plotted in Figure 4.15 on all data sampling schemes. It was observed that SMOTE300ENN, one of the enhanced data sampling schemes consistently had the best performance from all classifiers trained on the dataset while SMOTEENN had a comparable performance to SMOTE300ENN. MLP and SVM classifiers did not detect any of the minority class for 5ENN, CNN, RAW DATA, ENN and NCL data sampling schemes respectively.

The result of the RECALL of the minority class (PASSWAEC) metric for the SSS Result dataset is presented in Table 4.17 and charted in Figure 4.16. It was observed that SMOTE300ENN, one of the enhanced data sampling schemes had all the best performance. Classifiers trained on 5ENN, CNN, RAW DATA and ENN data sampling schemes failed to detect the minority class.

The result of the RECALL of the minority class (NONE) metric for CM dataset is presented in Table 4.18 and plotted in Figure 4.17. SMOTE300ENN, one of the enhanced data sampling schemes had the best performance for all classifiers. CNN and NCL data sampling did not detect any instance of the minority class for the dataset.
REP Tree and MLP classifier did not detect any instance of the minority class on 5ENN, CNN, RAW DATA, ENN and NCL data sampling schemes.

Tables 4.16: RECALL of minority class (GDM) metric values for DM dataset

Scheme Learner	SMOTE 300ENN	5ENN	CNN	RAW DATA	ENN	NCL	RUS	SMOTE	SMOTE 300	SMOTE ENN	SMOTE NCL	SMOTE RUS	SMOTE 300NCL	SMOTE 300RUS
RIPPER	**0.88**	0.25	0.33	0.12	0.25	0.53	0.71	0.50	0.68	0.76	0.71	0.71	0.72	0.81
Decision Tree	0.85	0.50	0.33	0.29	0.25	0.65	0.71	0.65	0.68	**0.86**	0.77	0.77	0.75	0.79
Random Forest	0.90	0.00	0.07	0.18	0.00	0.35	0.65	0.65	0.71	**0.95**	0.77	0.79	0.77	0.81
Random Tree	**0.78**	0.50	0.33	0.12	0.25	0.41	0.59	0.53	0.66	0.68	0.65	0.68	0.69	0.79
REPTree	0.81	0.25	0.13	0.29	0.25	0.53	0.35	0.74	0.75	**0.95**	0.82	0.85	0.78	0.73
MLP	**0.91**	0.00	0.00	0.00	0.00	0.00	0.65	0.21	0.82	0.52	0.32	0.65	0.87	0.82
SVM	**0.86**	0.00	0.00	0.00	0.00	0.00	0.59	0.00	0.01	0.00	0.27	0.79	0.74	0.82
1B3	**0.81**	0.00	0.00	0.24	0.00	0.24	0.53	0.38	0.69	0.57	0.50	0.65	0.75	0.74
Boosting	**0.95**	0.25	0.40	0.16	0.25	0.71	0.77	0.59	0.72	0.86	0.74	0.77	0.78	0.79
Bagging	**0.91**	0.00	0.20	0.24	0.00	0.71	0.65	0.68	0.74	0.91	0.79	0.79	0.78	0.79
MulticlassClassifier	0.86	0.00	0.40	0.18	0.00	0.65	0.65	0.71	0.75	**0.91**	0.77	0.82	0.81	0.84
RandomCommittee	0.91	0.25	0.07	0.18	0.00	0.47	0.59	0.59	0.69	**0.95**	0.71	0.71	0.75	0.79
Decision Forest	0.52	0.00	0.07	0.00	0.00	0.01	0.65	0.24	0.24	0.38	0.62	**0.80**	0.59	0.77
Stacking	**0.90**	0.00	0.27	0.06	0.00	0.59	0.15	0.53	0.68	0.86	0.74	0.74	0.81	0.73

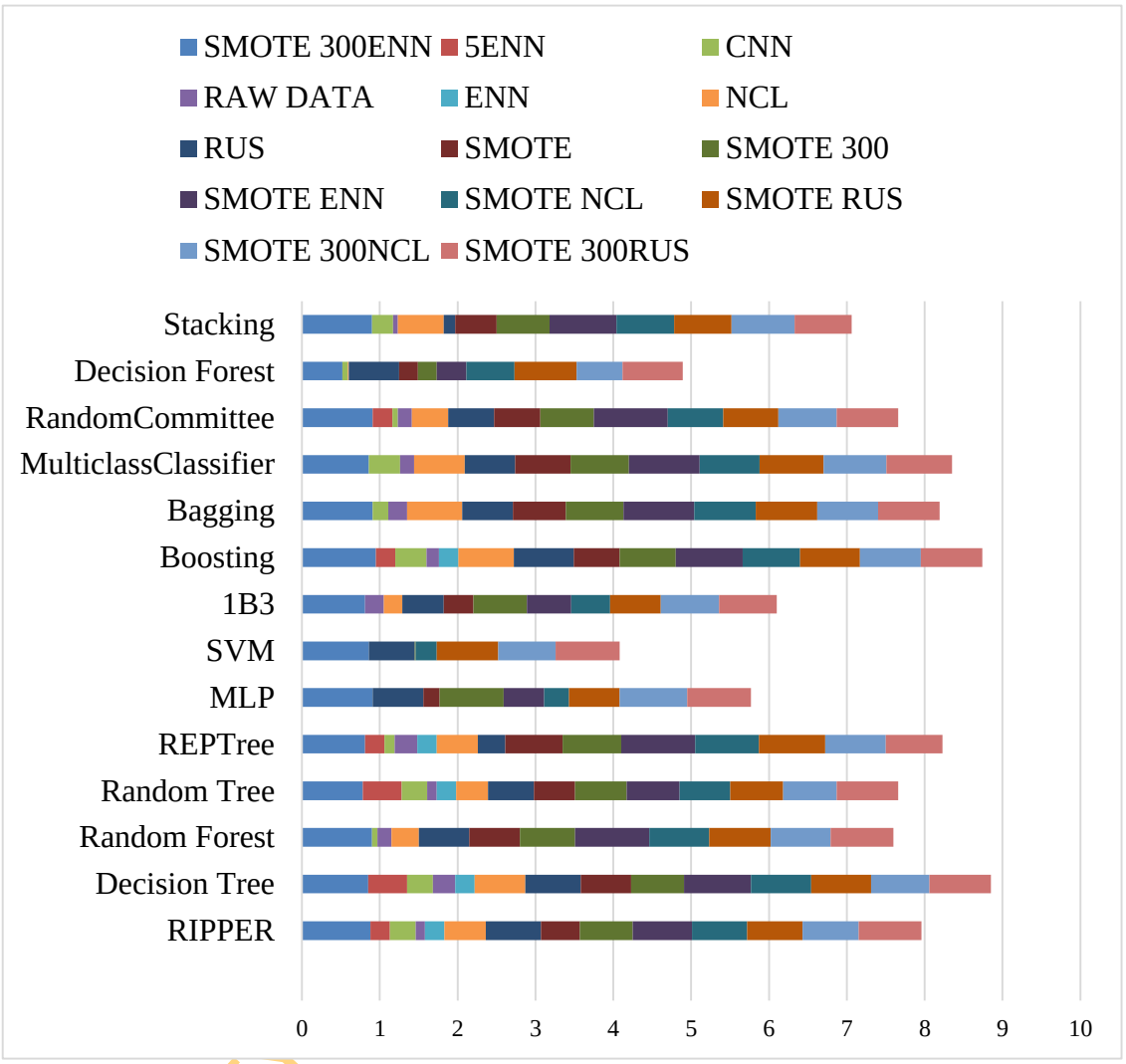


Figure 4.15: Chart showing the RECALL of the minority class (GDM) metric values for DM dataset

Table 4.17: RECALL of the minority class (PASSWAEC) metric values for SSS Result dataset

Scheme Learner	SMOTE 300ENN	5ENN	CNN	RAW DATA	ENN	NCL	RUS	SMOTE	SMOTE 300	SMOTE ENN	SMOTE NCL	SMOTE RUS	SMOTE 300NCL	SMOTE 300RUS
RIPPER	**0.99**	0.00	0.00	0.00	0.00	0.00	0.40	0.16	0.33	0.50	0.21	0.56	0.35	0.70
Decision Tree	**0.97**	0.00	0.00	0.00	0.00	0.04	0.44	0.22	0.54	0.60	0.24	0.54	0.54	0.62
Random Forest	**0.99**	0.00	0.04	0.00	0.00	0.04	0.58	0.22	0.54	0.90	031	0.58	0.55	0.63
Random Tree	**0.98**	0.00	0.00	0.00	0.00	0.04	0.42	0.23	0.51	0.90	0.28	0.60	0.53	0.63
REPTree	**0.97**	0.00	0.00	0.00	0.00	0.02	0.44	0.18	0.49	0.50	0.14	0.44	0.52	0.59
MLP	**0.99**	0.00	0.00	0.00	0.00	0.00	0.44	0.00	0.66	0.00	0.00	0.61	0.46	0.60
SVM	**0.99**	0.00	0.00	0.00	0.00	0.02	0.56	0.26	0.56	0.70	0.27	0.67	0.50	0.73
1B3	**0.99**	0.00	0.00	0.00	0.00	0.07	0.42	0.23	0.51	0.90	0.07	0.58	0.53	0.58
Boosting	**0.99**	0.00	0.11	0.04	0.00	0.07	0.51	0.24	0.53	0.80	0.07	0.51	0.58	0.71
Bagging	**0.97**	0.00	0.04	0.00	0.00	0.04	0.53	0.22	0.52	0.70	0.28	0.58	0.52	0.63
MulticlassClassifier	**0.97**	0.00	0.00	0.00	0.00	0.02	0.51	0.26	0.50	0.60	0.23	0.54	0.48	0.63
RandomCommittee	**0.99**	0.00	0.00	0.00	0.00	0.04	0.44	0.23	0.51	0.90	0.29	0.59	0.53	0.63
Decision Forest	**0.99**	0.00	0.00	0.00	0.00	0.00	0.53	0.23	0.58	0.60	0.19	0.66	0.58	0.68
Stacking	**0.99**	0.00	0.13	0.00	0.00	0.11	0.47	0.21	0.56	0.90	0.30	0.54	0.52	0.72

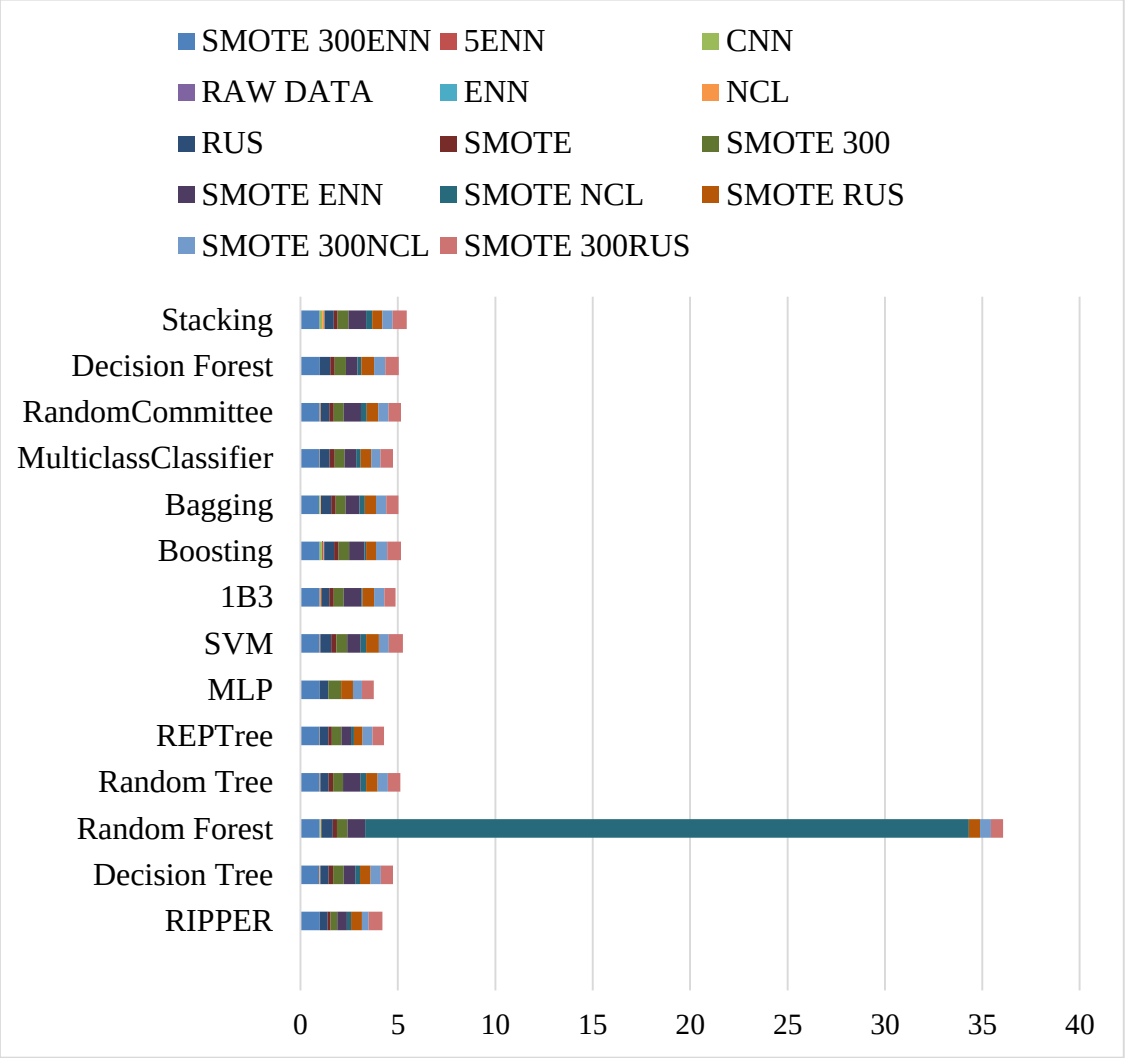


Figure 4.16: Chart showing the RECALL of the minority class (PASSWAEC) metric values for SSS Result dataset

Table 4.18: RECALL of the minority class (NONE) metric values for CM dataset

Scheme Learner	SMOTE 300ENN	5ENN	CNN	RAW DATA	ENN	NCL	RUS	SMOTE	SMOTE 300	SMOTE ENN	SMOTE NCL	SMOTE RUS	SMOTE 300NCL	SMOTE 300RUS
RIPPER	0.81	0.00	0.00	0.00	0.00	0.00	0.25	0.50	0.77	0.56	0.50	0.53	0.75	**0.86**
Decision Tree	**0.88**	0.11	0.00	0.13	0.07	0.06	0.19	0.53	0.77	0.67	0.50	0.53	0.81	0.82
Random Forest	**0.86**	0.22	0.00	0.13	0.07	0.00	0.19	0.53	0.77	0.67	0.53	0.59	0.78	0.82
Random Tree	**0.85**	0.22	0.00	0.13	0.14	0.00	0.19	0.45	0.75	0.56	0.53	0.63	0.73	0.82
REPTree	**0.83**	0.00	0.00	0.00	0.00	0.00	0.13	0.00	0.75	0.00	0.06	0.56	0.75	0.82
MLP	0.22	0.00	0.00	0.00	0.00	0.00	0.19	0.06	0.50	0.00	0.00	0.59	0.50	**0.82**
SVM	**0.86**	0.33	0.00	0.13	0.14	0.00	0.31	0.56	0.78	0.67	0.56	0.66	0.78	0.82
1B3	**0.86**	0.33	0.07	0.19	0.14	0.00	0.31	0.59	0.80	0.67	0.56	0.69	0.78	0.82
Boosting	**0.86**	0.22	0.00	0.13	0.14	0.00	0.25	0.53	0.01	0.67	0.56	0.56	0.78	0.82
Bagging	**0.86**	0.22	0.00	0.00	0.07	0.00	0.25	0.53	0.77	0.63	0.53	0.63	0.77	0.82
MulticlassClassifier	**0.83**	0.00	0.07	0.00	0.00	0.00	0.31	0.47	0.75	0.59	0.47	0.66	0.80	0.82
RandomCommittee	**0.86**	0.22	0.00	0.13	0.14	0.00	0.19	0.56	0.78	0.67	0.56	0.56	0.78	0.82
Decision Forest	**0.86**	0.11	0.00	0.13	0.14	0.00	0.25	0.53	0.77	0.67	0.50	0.66	0.78	0.84
Stacking	**0.85**	0.00	0.00	0.63	0.14	0.00	0.38	0.56	0.77	0.63	0.50	0.63	0.78	0.80

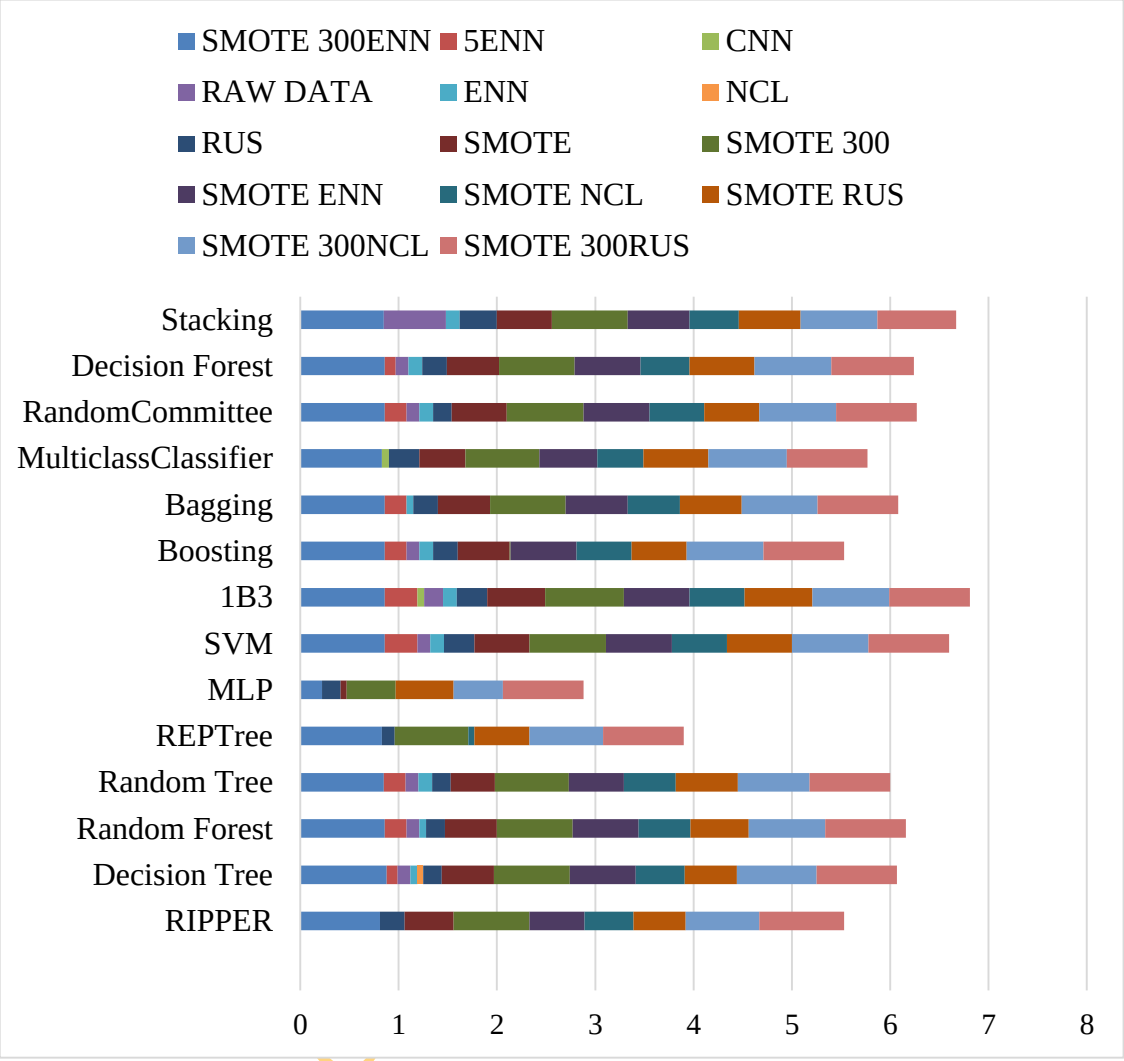


Figure 4.17: Chart showing the RECALL of the minority class (NONE) metric values for CM dataset

4.1.4.5 Analysis of Performance Loss/gain metrics

A low Performance loss/gain value indicates improvement in performance. The result of the performance loss/gain metric on all the data sampling schemes on DM dataset are presented in Table 4.19 and plotted in Figure 4.18. SMOTE300ENN, one the enhanced data sampling schemes had the lowest values hence it had the best improvement in performance.

The performance loss/gain result for SSS Result dataset is presented in Table 4.20 and charted in Figure 4.19. It can be observed that datasets created from SMOTE300ENN, one of the enhanced data sampling schemes had the best performance over the RAW DATA when trained on all the 14 classifiers. 5ENN, ENN and SMOTEENN data sampling schemes also showed great improvement in the performance of the classification result.

The result of performance loss/gain for CM dataset is presented in Table 4.21 and plotted in Figure 4.20. It was observed that SMOTE300RUS showed the greatest improvement on classification performance and closely followed by SMOTE300ENN.

Table 4.19: Performance Loss/gain values for on DM dataset against RAW DATA using ROC_AUC metric

	SMOTE 300ENN	5ENN	CNN	ENN	NCL	RUS	SMOTE 300	SMOTE ENN	SMOTE NCL	SMOTE RUS	SMOTE 300NCL	SMOTE 300RUS
RIPPER	**-0.43**	0.03	-0.14	-0.03	-0.33	-0.27	-0.21	-0.25	-0.40	-0.27	-0.30	-0.25	-0.25
Decision Tree	-0.16	-0.06	0.18	-0.06	-0.05	0.02	-0.05	-0.06	**-0.18**	-0.07	-0.09	-0.11	-0.05
Random Forest	**-0.18**	-0.04	0.16	-0.02	-0.06	0.07	-0.06	-0.05	-0.17	-0.08	-0.01	-0.11	-0.05
Random Tree	**-0.40**	-0.08	0.05	-0.16	-0.27	-0.10	-0.14	-0.19	-0.32	-0.29	-0.10	-0.29	-0.22
REPTree	-0.30	-0.08	0.04	-0.15	-0.27	-0.12	0.01	-0.09	**-0.31**	-0.09	-0.18	-0.18	-0.14
MLP	**-0.17**	0.10	0.18	0.22	-0.05	0.26	-0.04	-0.07	-0.15	-0.10	0.11	-0.09	0.09
SVM	**-0.72**	0.00	0.00	0.00	-0.08	-0.18	-0.02	-0.38	-0.02	-0.20	-0.52	-0.48	-0.60
1B3	**-0.27**	-0.08	0.69	-0.04	-0.04	0.20	-0.07	-0.11	-0.19	-0.03	-0.03	-0.15	-0.12
Boosting	**-0.19**	0.00	0.22	-0.07	0.06	0.06	-0.01	-0.05	-0.12	-0.06	-0.01	-0.08	-0.04
Bagging	**-0.15**	-0.04	0.21	-0.05	-0.02	0.05	-0.04	-0.05	-0.14	-0.05	-0.04	-0.07	-0.02
MulticlassClassifier	-0.16	-0.05	0.16	-0.04	-0.04	0.04	-0.04	-0.06	**-0.17**	-0.06	-0.05	-0.05	-0.01
RandomCommittee	**-0.20**	-0.07	0.19	-0.07	-0.08	0.05	-0.04	-0.05	-0.14	-0.10	-0.04	-0.11	-0.06
Decision Forest	**-0.17**	-0.06	0.17	-0.11	-0.06	0.05	-0.04	-0.04	**-0.17**	-0.07	-0.02	-0.07	-0.02
Stacking	**-0.30**	-0.12	-0.04	-0.13	-0.09	0.09	-0.08	-0.13	-0.23	-0.14	-0.06	-0.14	-0.12

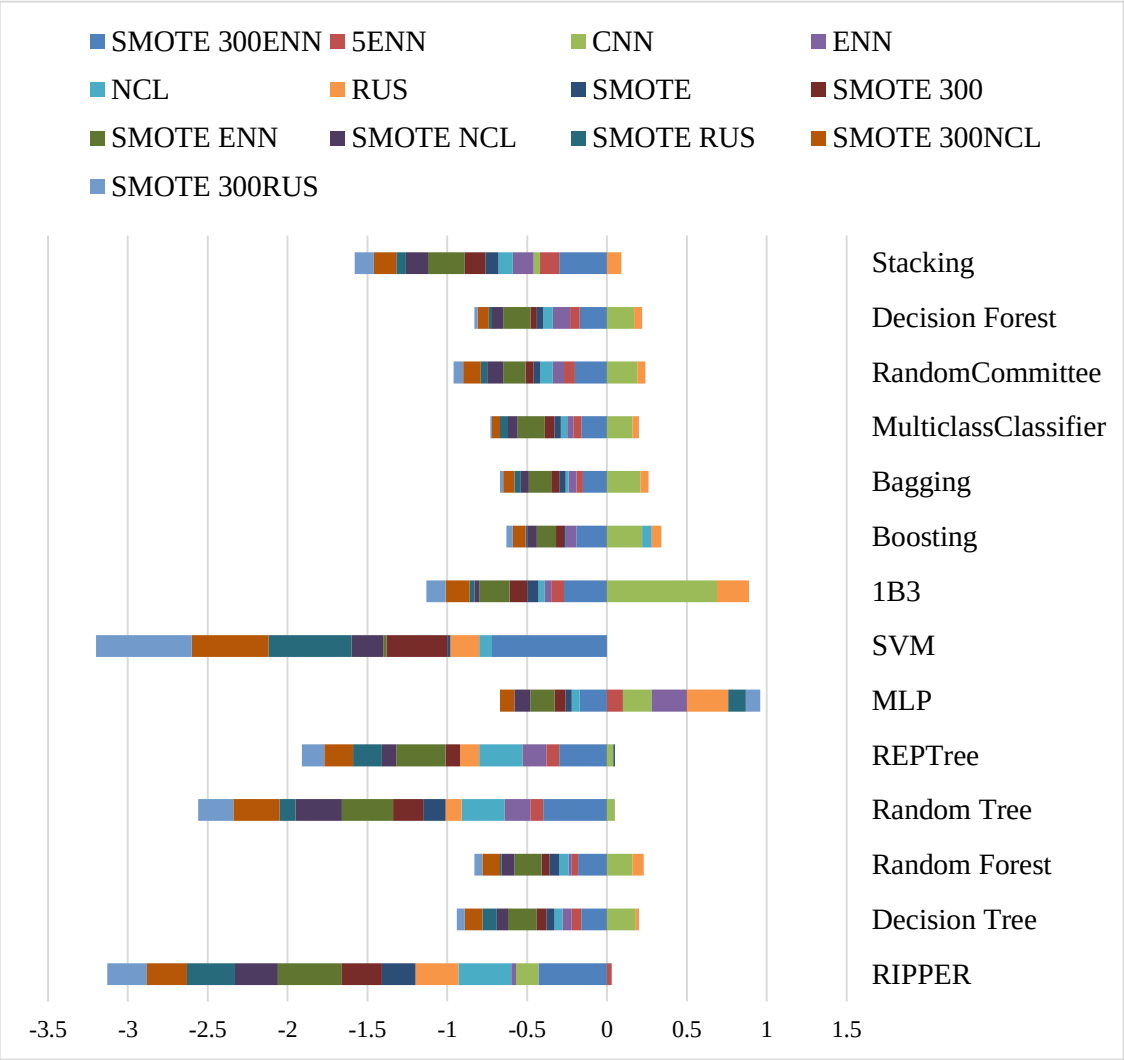


Figure 4.18: Chart showing the Performance Loss/gain values for on DM dataset against RAW DATA using ROC_AUC metric

Table 4.20: Performance Loss/gain values for SSS Result dataset against RAW DATA using ROC_AUC metric

Scheme Learner	SMOTE 300ENN	5ENN	CNN	ENN	NCL	RUS	SMOTE	SMOTE 300	SMOTE ENN	SMOTE NCL	SMOTE RUS	SMOTE 300NCL	SMOTE 300RUS
RIPPER	**-0.68**	**-0.68**	0.14	**-0.68**	-0.20	-0.17	-0.03	-0.08	-0.66	-0.15	-0.24	-0.19	-0.36
Decision Tree	**-0.23**	**-0.23**	0.31	**-0.23**	-0.06	0.14	-0.01	-0.04	**-0.23**	-0.06	0.05	-0.09	0.00
Random Forest	**-0.23**	**-0.23**	0.35	**-0.23**	-0.06	0.15	-0.01	-0.04	**-0.23**	-0.07	0.06	-0.09	0.01
Random Tree	**-0.23**	**-0.23**	0.36	**-0.23**	-0.06	0.16	-0.01	-0.02	**-0.23**	-0.07	0.07	-0.09	0.01
REPTree	**-0.23**	**-0.23**	0.33	**-0.23**	-0.06	0.10	-0.01	-0.04	**-0.23**	-0.07	0.06	-0.09	0.01
MLP	**-0.25**	-0.06	0.18	-0.23	-0.04	0.05	0.00	-0.03	-0.23	-0.06	0.04	-0.05	0.04
SVM	**-0.40**	-0.26	0.15	-0.38	-0.10	-0.10	-0.01	-0.06	-0.37	-0.09	-0.12	-0.13	-0.13
1B3	**-0.23**	**-0.23**	0.35	**-0.23**	-0.06	0.15	-0.01	-0.04	**-0.23**	-0.07	0.06	-0.09	0.01
Boosting	**-0.27**	**-0.27**	0.30	**-0.27**	-0.09	0.14	-0.04	-0.05	**-0.27**	-0.10	0.09	-0.11	0.05
Bagging	**-0.22**	**-0.22**	0.32	**-0.22**	-0.06	0.12	0.00	-0.02	**-0.22**	-0.06	0.06	-0.07	0.01
MulticlassClassifier	**-0.23**	**-0.23**	0.30	**-0.23**	-0.06	0.11	-0.01	-0.02	**-0.23**	-0.06	0.06	-0.07	0.02
RandomCommittee	**-0.23**	**-0.23**	0.36	**-0.23**	-0.06	0.16	-0.01	-0.02	**-0.23**	-0.07	0.07	-0.09	0.02
Decision Forest	**-0.21**	**-0.21**	0.27	**-0.21**	-0.06	0.10	0.00	-0.02	**-0.21**	-0.06	0.06	-0.06	0.01
Stacking	**-0.27**	**-0.27**	0.23	**-0.27**	-0.08	0.09	-0.03	-0.05	**-0.27**	-0.08	0.05	-0.12	0.00

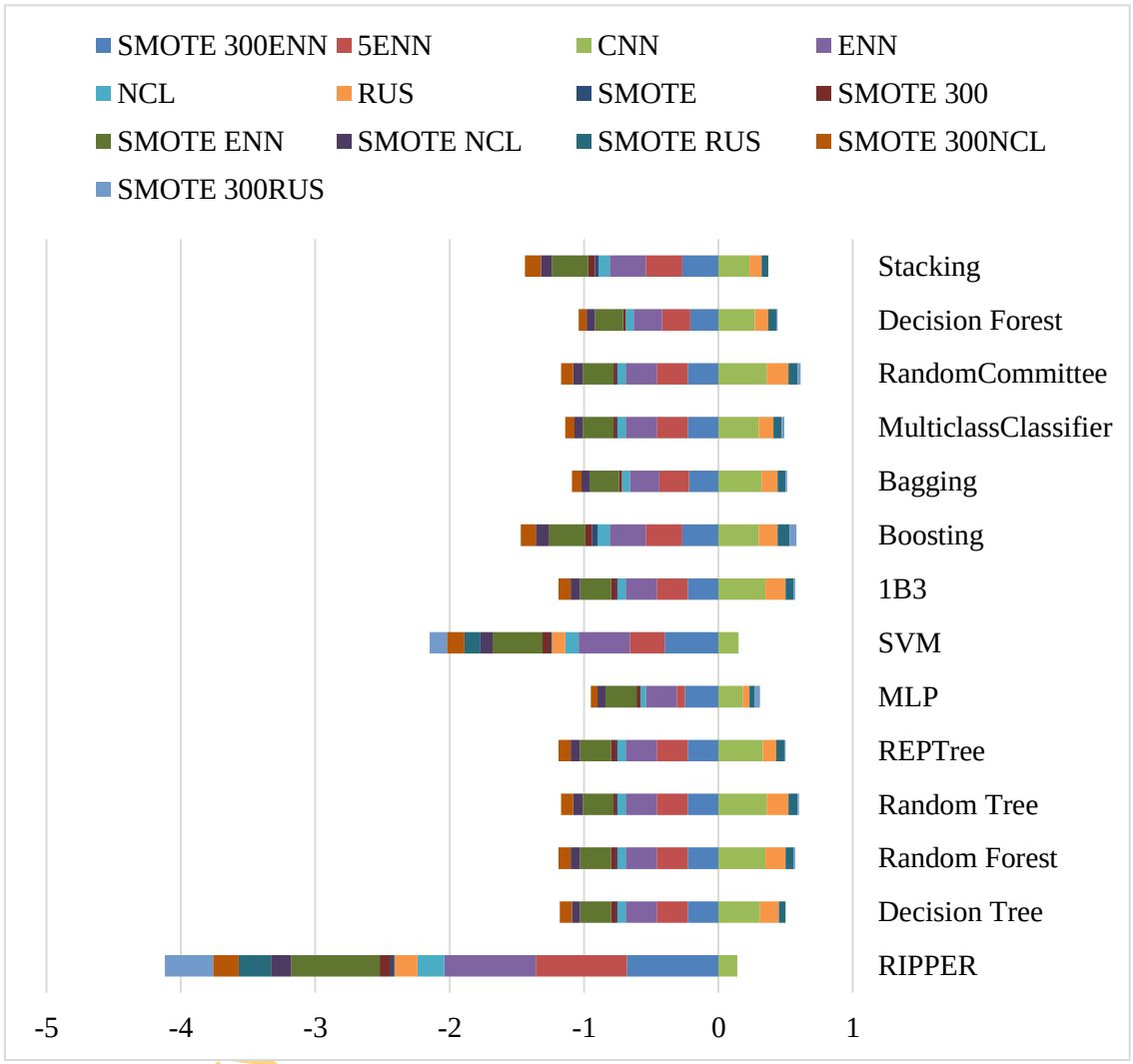


Figure 4.19: Chart showing the Performance Loss/gain values for on SSS Result dataset against RAW DATA using ROC_AUC metric

Table 4.21: Performance Loss/gain values for CM dataset against RAW DATA using ROC_AUC metric

Scheme Learner	SMOTE 300ENN	5ENN	CNN	ENN	NCL	RUS	SMOTE	SMOTE 300	SMOTE ENN	SMOTE NCL	SMOTE RUS	SMOTE 300NCL	SMOTE 300RUS
RIPPER	-0.13	-0.08	0.04	-0.04	0.04	-0.13	-0.02	-0.12	-0.08	-0.04	**-0.25**	-0.13	-0.23
Decision Tree	**-0.23**	-0.02	0.09	-0.07	0.00	0.16	-0.02	-0.07	-0.04	-0.04	-0.09	-0.12	-0.18
Random Forest	-0.15	-0.10	0.15	-0.05	-0.05	0.13	-0.02	-0.07	-0.08	-0.02	-0.05	-0.11	**-0.16**
Random Tree	-0.16	-0.05	0.14	-0.02	-0.04	0.07	0.04	-0.09	-0.09	-0.11	-0.11	-0.13	**-0.25**
REPTree	-0.17	0.00	0.11	0.08	0.06	0.19	0.08	-0.15	0.06	0.06	-0.17	-0.13	**-0.21**
MLP	-0.06	-0.13	0.02	-0.13	-0.04	0.08	-0.06	-0.08	-0.13	-0.08	-0.21	-0.10	**-0.29**
SVM	-0.17	-0.10	0.09	-0.09	-0.07	-0.10	-0.03	-0.10	-0.12	-0.07	-0.16	-0.14	**-0.22**
1B3	-0.16	-0.16	0.28	-0.05	-0.02	-0.02	0.00	-0.09	-0.10	-0.05	-0.12	-0.10	**-0.24**
Boosting	**-0.21**	-0.07	0.07	-0.12	-0.07	0.12	-0.05	-0.09	-0.09	-0.07	-0.19	-0.14	-0.14
Bagging	-0.23	-0.04	0.02	-0.06	-0.06	0.06	-0.06	-0.12	-0.12	-0.08	-0.25	-0.19	**-0.33**
MulticlassClassifier	**-0.20**	0.00	0.05	-0.02	0.00	0.09	-0.02	-0.11	-0.05	-0.04	-0.15	-0.15	-0.18
RandomCommittee	**-0.19**	-0.10	0.17	-0.05	-0.09	-0.03	-0.05	-0.09	-0.10	-0.05	-0.16	-0.16	-0.17
Decision Forest	-0.16	-0.02	0.07	-0.05	-0.05	0.09	-0.02	-0.13	-0.07	-0.09	-0.18	-0.18	**-0.23**
Stacking	-0.27	-0.23	-0.13	-0.13	-0.08	-0.04	-0.12	-0.17	-0.17	-0.10	-0.19	-0.23	**-0.31**

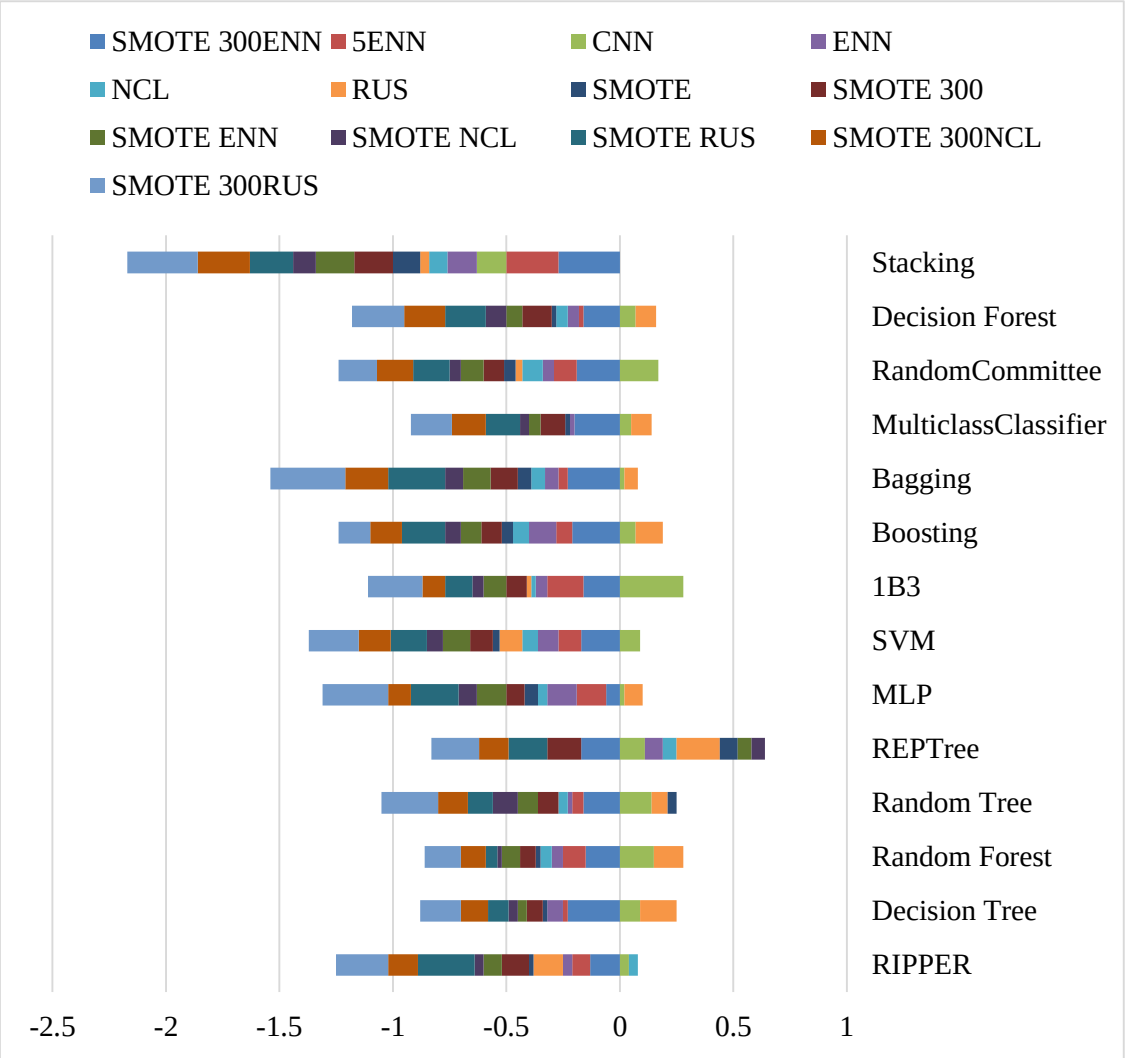


Figure 4.20: Chart showing the Performance Loss/gain values for on CM dataset against RAW DATA using ROC_AUC metric

4.1.4.6 Analysis of all metrics on Tuberculosis (TB) dataset

The analysis of all performing metrics on TB dataset is presented in Table 4.22 and also plotted in Figure 4.21. It was observed that there was no sub optimal classification performance on all classifiers, so there was no need to apply data sampling scheme to the dataset.

Table 4.22: All metrics values for Tuberculosis dataset

Scheme Learner	ROC_AUC	Kappa Statistics	RMSE	RECALL
RIPPER	1.00	1.00	0.03	1.00
Decision Tree	1.00	1.00	0.03	1.00
Random Forest	1.00	1.00	0.04	1.00
Random Tree	1.00	1.00	0.03	1.00
REPTree	1.00	1.00	0.03	1.00
MLP	1.00	0.97	0.11	0.00
SVM	1.00	1.00	0.31	1.00
1B3	1.00	0.99	0.04	0.83
Boosting	1.00	0.99	0.04	0.83
Bagging	1.00	1.00	0.04	1.00
MulticlassClassifier	1.00	1.00	0.35	1.00
Random Committee	1.00	1.00	0.04	1.00
Decision Forest	1.00	0.74	0.17	0.00
Stacking	1.00	1.00	0.03	1.00

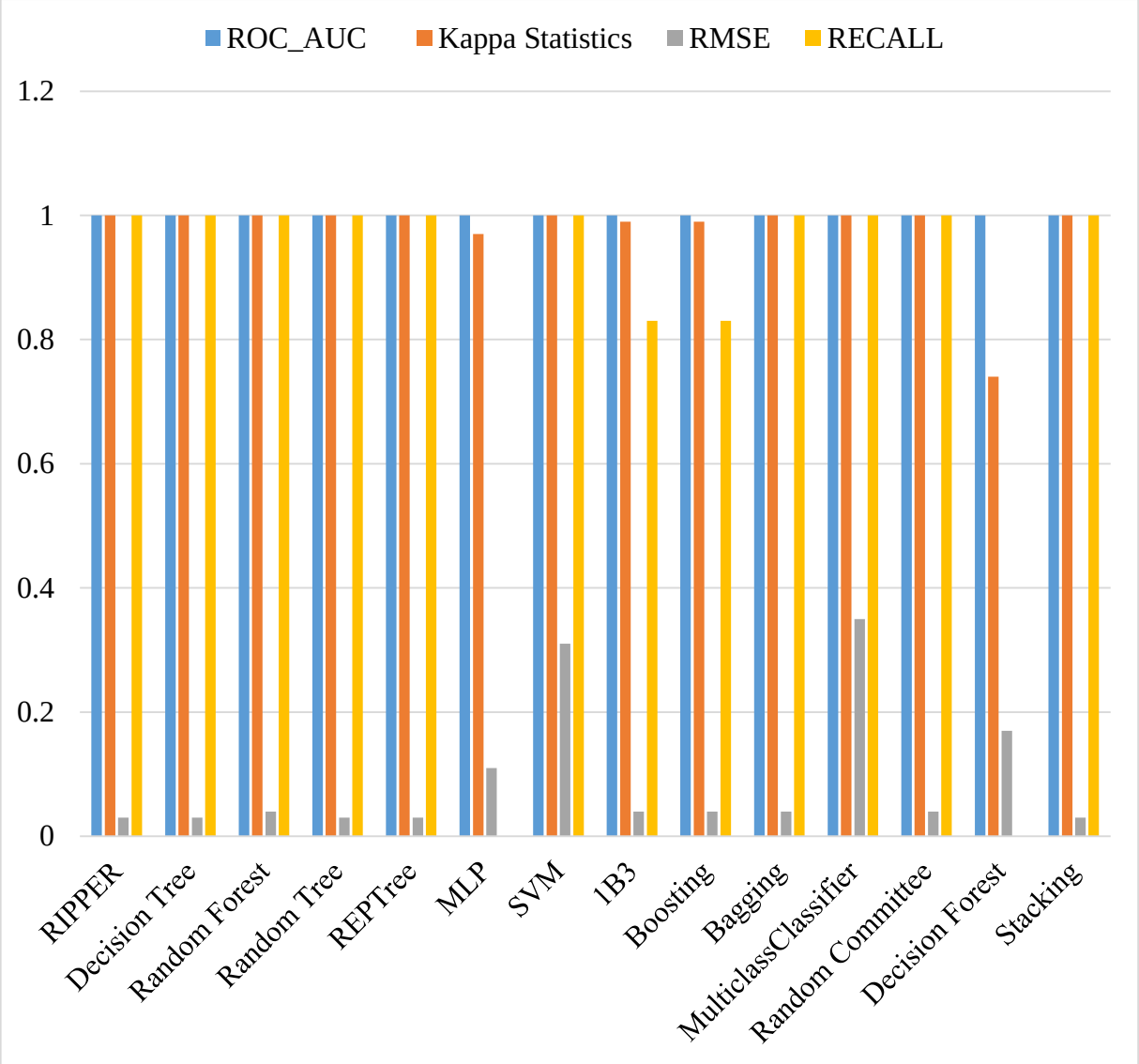


Figure 4.21: Chart showing all metrics values for Tuberculosis (TB) dataset

4.1.5 Statistical Analysis of classification results on performance metrics

This sub section presents the statistical analysis of the performance metrics obtained from the classification results for the data sampling schemes on all datasets. The results of performance metrics were subjected to Friedman's Test, ANOVA with Tukey post hoc test and Box and Whisker plots at statistical significance level of 0.05%.

4.1.5.1 Report of Friedman test on ROC_AUC metric for all dataset

The results obtained with Friedman's test on ROC_AUC metric on all datasets with their mean rank values is presented in Table 4.23 and plotted in Figure 4.22. The data sampling schemes with the highest mean rank value is the best performing scheme.

SMOTE300ENN with mean rank values of 13.75 and 12.75 gave the best performance for the DM and SSR Result dataset respectively while SMOTE300RUS with a mean rank value of 13.54, gave the best classification performance for CM dataset.

CNN performed least on DM, SSS Result and CM datasets, with mean rank values of 1.61, 1.00 and 1.93 respectively

Therefore, Friedman analysis established that two of the enhanced data sampling schemes (SMOTE300ENN and SMOTE300NCL) were ranked among the best seven data sampling schemes across all datasets based on Friedman analysis.

Table 4.23: Result of Friedman analysis on ROC_AUC metric for all datasets

S/N	Data sampling schemes	DM	SSS Result	CM
1	SMOTERUS	6.93	3.82	11.61
2	SMOTENCL	9.82	8.39	6.61
3	SMOTEENN	12.71	12.36	8.79
4	SMOTE300RUS	7.29	4.86	**13.54**
5	SMOTE300NCL	11.00	9.61	11.14
6	SMOTE300ENN	**13.75**	**12.75**	12.32
7	SMOTE300	8.46	6.57	8.89
8	SMOTE	6.04	5.46	4.46
9	RUS	3.39	2.61	2.82
10	RAW DATA	3.14	4.61	3.25
11	NCL	7.75	8.11	5.00
12	ENN	7.14	12.57	6.75
13	CNN	**1.61**	**1.00**	**1.93**
14	5ENN	5.96	12.29	7.89

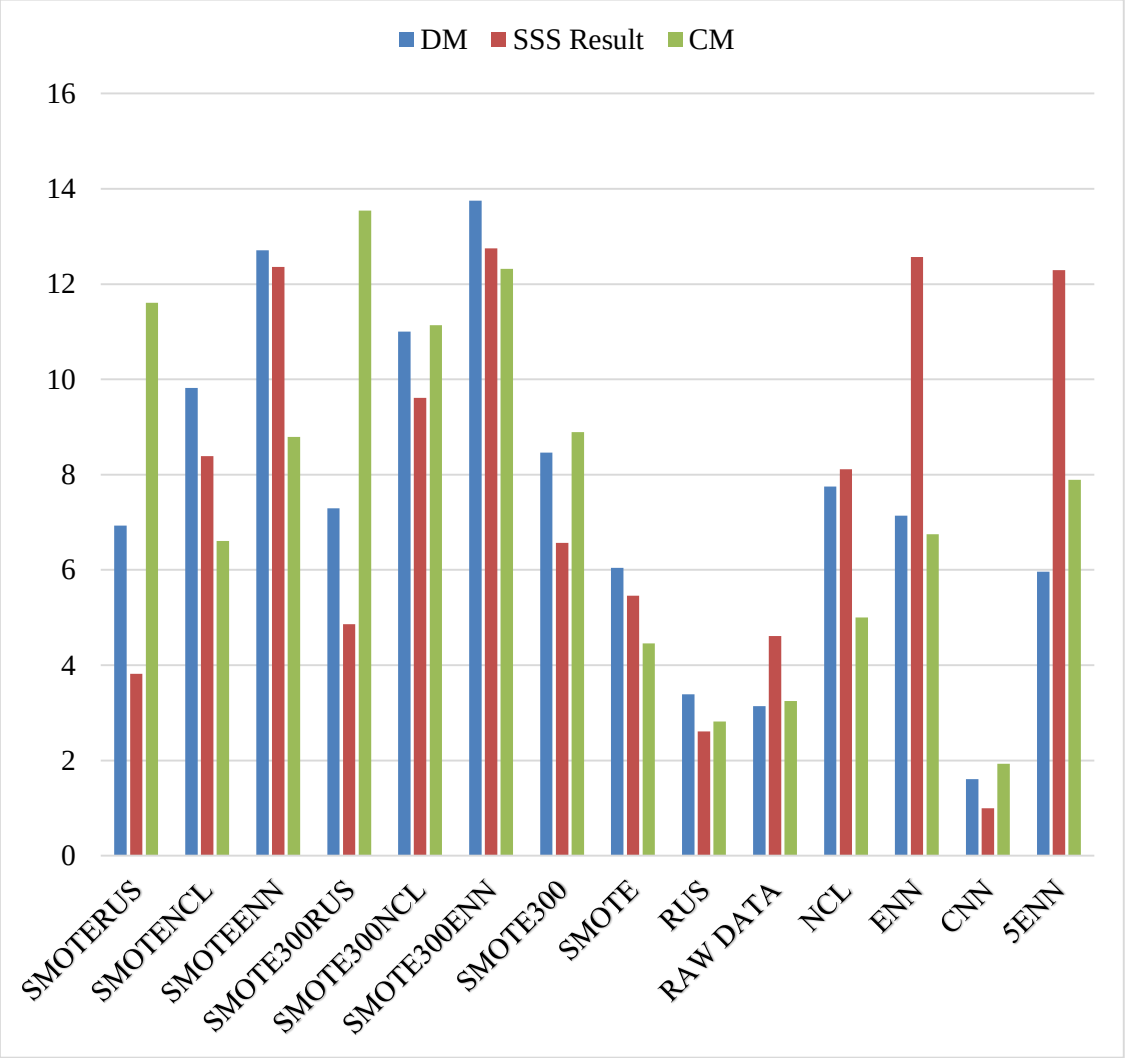


Figure 4.22: Chart showing the Friedman analysis on ROC_AUC metric for all datasets

4.1.5.2 Report of Friedman test on Kappa statistics metric for all datasets

The results obtained with Friedman's test using Kappa statistics metric on all dataset with their mean rank value is presented in Table 4.24 and plotted in Figure 4.23.

SMOTE300ENN, one of the enhanced data sampling schemes with mean rank value of 13.86 and 13.21 gave the best performance of all other data sampling schemes on DM and SSS Result dataset respectively while SMOTE300RUS, also one of the enhanced data sampling schemes with a mean rank value of 13.14, outperformed all other data sampling schemes on CM dataset.

CNN, the least performing data sampling scheme on DM, SSS Result and CM dataset had its mean rank values of 2.43, 1.00 and 1.43 respectively.

Similarly, the Friedman analysis revealed that three of the enhanced data sampling schemes (SMOTE300ENN, SMOTE300NCL and SMOTENCL) were ranked amongst the best seven data sampling schemes across all datasets.

Table 4.24: Result of Friedman analysis on Kappa Statistics metric for all datasets

S/N	Data sampling schemes	DM	SSS Result	CM
1	SMOTERUS	9.18	5.50	10.32
2	SMOTENCL	10.54	8.18	7.04
3	SMOTEENN	11.39	12.29	8.64
4	SMOTE300RUS	8.50	6.68	**13.14**
5	SMOTE300NCL	11.82	8.93	11.96
6	SMOTE300ENN	**13.86**	**13.21**	12.93
7	SMOTE300	8.57	5.79	10.61
8	SMOTE	5.25	3.39	5.21
9	RUS	5.68	2.79	2.79
10	RAW DATA	3.32	3.50	2.93
11	NCL	8.61	9.25	5.93
12	ENN	2.89	12.75	5.61
13	CNN	**2.43**	**1.00**	**1.43**
14	5ENN	2.96	11.75	6.46

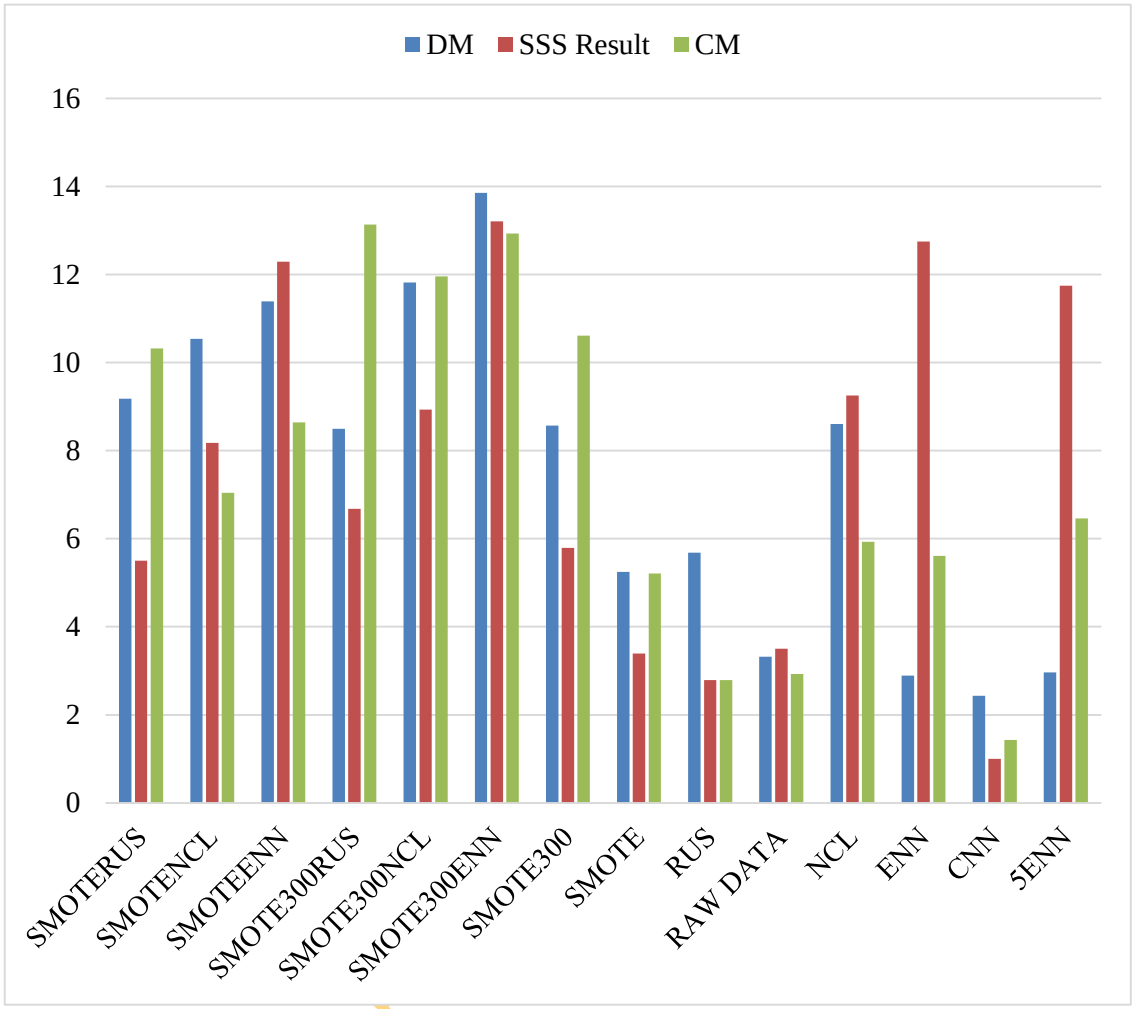


Figure 4.23: Chart showing the Friedman analysis on Kappa Statistics metric for all datasets

4.1.5.3 Report of Friedman's test on RMSE metric for all datasets

The results obtained with Friedman's test using RMSE metric with all the dataset used with their mean rank value is presented in Table 4.25 and plotted in Figure 4.24. The lower the mean rank value, the better the data sampling scheme.

ENN scheme, with the least mean value of 1.86 and 2.07 performed best of all data sampling schemes on DM and SSS Result dataset rspectively while 5ENN with the least mean rank value of 2.57 was the best performing data sampling scheme on CM dataset.

RUS data sampling scheme performed poorly on DM and CM datasets with mean rank values of 13.71 and 13.96 respectively. CNN data sampling scheme gave the least performance on SSS Result dataset with a mean rank value of 13.50.

Hence, the Friedman's analysis indicated that two of the enhanced data sampling schemes (SMOTE300ENN and SMOTE300NCL) were ranked amongst the best seven data sampling schemes across all datasets.

Table 4.25: Result of Friedman analysis on RMSE metric for all datasets

S/N	Data sampling schemes	DM	SSS Result	CM
1	SMOTERUS	12.11	11.86	12.64
2	SMOTENCL	6.18	6.57	8.11
3	SMOTEENN	2.36	2.93	3.5
4	SMOTE300RUS	11.89	11.21	10.93
5	SMOTE300NCL	8.04	7.07	6.25
6	SMOTE300ENN	3.36	2.54	2.79
7	SMOTE300	9.14	9.36	5.71
8	SMOTE	8.14	9.14	6.43
9	RUS	**13.71**	13.43	**13.96**
10	RAW DATA	7.82	7.61	7.75
11	NCL	5.68	5.25	8.64
12	ENN	**1.86**	**2.07**	4.68
13	CNN	12.29	**13.50**	11.04
14	5ENN	2.43	2.46	**2.57**

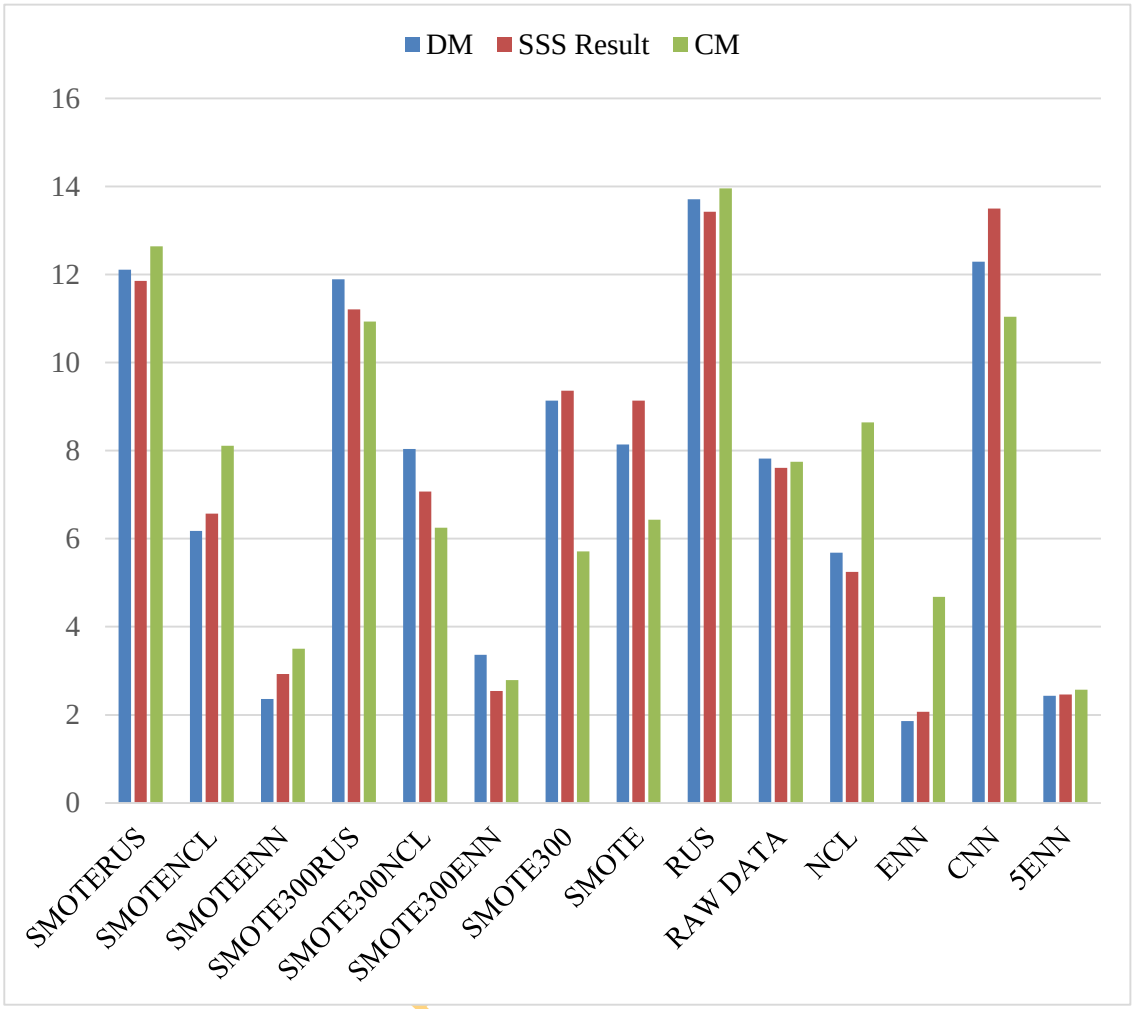


Figure 4.24: Chart showing the Friedman analysis on RMSE metric for all datasets

4.1.5.4 Report of Friedman test on RECALL of the minority class metric for all datasets

The results obtained with Friedman's test using RECALL of the minority class metric on all dataset with their mean rank values is presented in Table 4.26 and plotted in Figure 4.25.

The analysis of result generated from the RECALL of the minority class had SMOTE300ENN with a mean rank values of 13.04, 14.00 and 13.64 performed best on DM, SSS Result and CM dataset respectively.

ENN, 5ENN and NCL with mean rank values of 2.04, 2.54 and 2.11 had zero detection of the minority class on DM, SSS Result and CM datasets respectively.

Thus, Friedman analysis of the RECALL of the minority class established that SMOTE300ENN, one of the enhanced data sampling schemes gave the best RECALL/ detection of the minority class across all datasets.

Four of the enhanced data sampling schemes (SMOTE300ENN, SMOTE300RUS, SMOTE300NCL and SMOTERUS) were amongst the best seven ranked out of all data sampling schemes.

Table 4.26: Report of Friedman analysis on RECALL of Minority class metric for all datasets

S/N	Data sampling schemes	DM	SSS Result	CM
1	SMOTERUS	10.79	10.71	9.54
2	SMOTENCL	9.36	6.32	7.29
3	SMOTEENN	11.5	11.75	8.89
4	SMOTE300RUS	11.68	12.32	13.14
5	SMOTE300NCL	10.82	9.57	11.61
6	SMOTE300ENN	**13.04**	**14.00**	**13.64**
7	SMOTE300	8.29	9.54	10.82
8	SMOTE	6.00	6.25	7.43
9	RUS	7.54	8.86	6.07
10	RAW DATA	2.82	2.64	3.89
11	NCL	5.18	4.54	**2.11**
12	ENN	**2.04**	**2.54**	3.75
13	CNN	3.29	3.43	2.29
14	5ENN	2.68	**2.54**	4.54

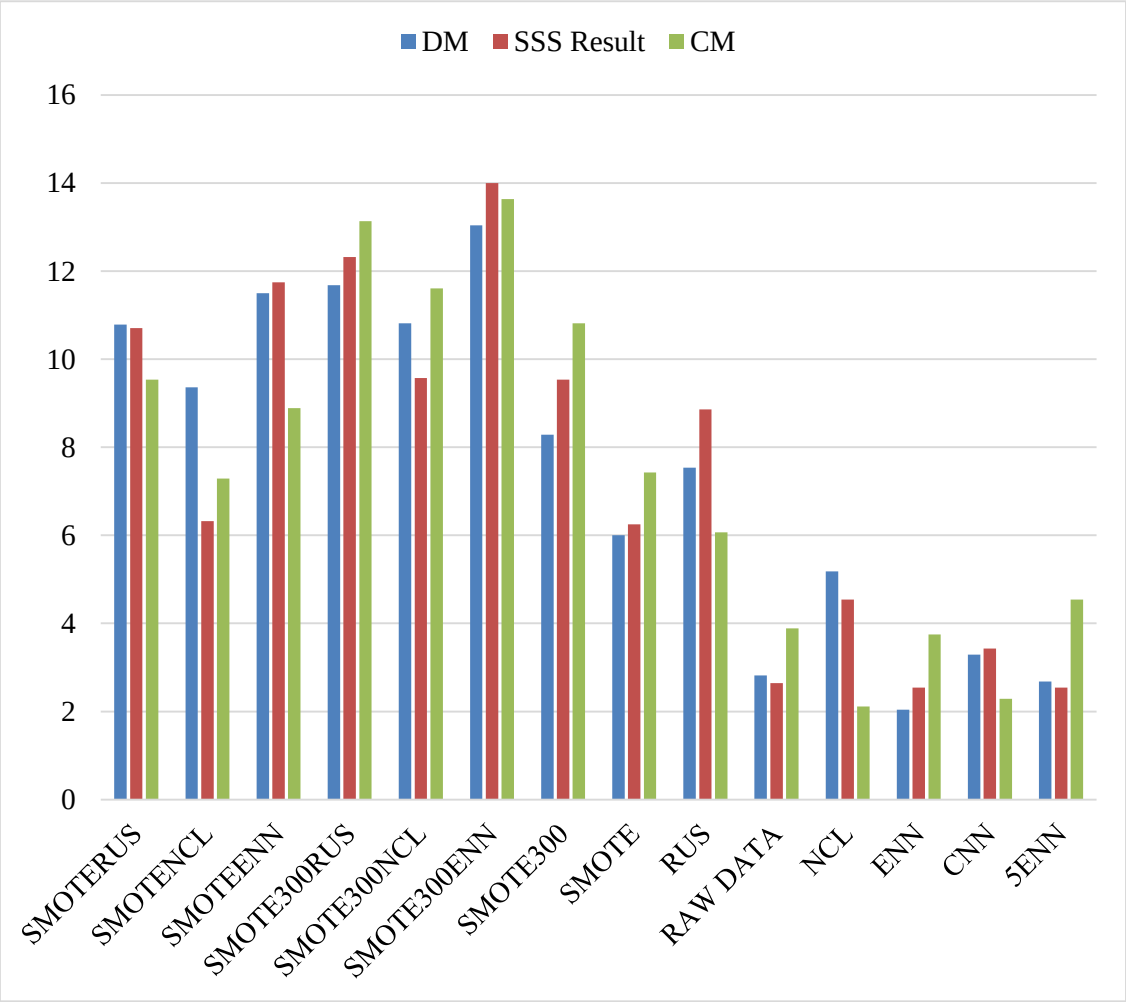


Figure 4.25: Chart showing the Friedman analysis on RECALL of minority class metric for all datasets

4.1.5.6 Report of Friedman's test on ROC_AUC metric for all classifiers

The results obtained with Friedman's test using ROC_AUC metric on 14 different classifier for the datasets with their mean rank values is presented in Table 4.27 and plotted in Figure 4.26. The analysis revealed that:

BAGGING, with a mean rank value of 11.64 and 10.93 respectively outperformed all other classifiers on DM and SSS Result dataset while RANDOM FOREST, with a mean rank value of 12.54 outperformed all other classifiers on CM dataset.

Considering base classifiers, Decision Tree gave the best classification performance on all data sampling schemes on DM and SSS Result dataset while SVM surpassed the other data sampling schemes with best classification ability on CM dataset.

For ensemble classifiers, BAGGING (homogeneous ensemble) with Decision Tree as the base classifier gave the best classification performance on DM and SSS Result dataset while RANDOM FOREST (homogeneous ensemble) gave the best classification ability with CM dataset. BOOSTING (homogeneous ensemble) with Decision Tree as the base classifier performed least on DM dataset, STACKING (a heterogeneous ensemble) performed least on SSS Result dataset while BAGGING performed least on CM dataset. SVM, RIPPER and REPTREE classifiers performed least across all three dataset respectively.

Table 4.27: Report of Friedman's analysis of on ROC_AUC metric for all classifiers

S/N	Classifiers	DM	SSS Result	CM
1	1B3	3.68	9.07	9.89
2	BAGGING	**11.64**	**10.93**	4.50
3	BOOSTING	7.86	7.32	10.00
4	DECISION FOREST	11.29	9.39	8.39
5	DECISIONTREE	9.25	9.71	7.25
6	MLP	6.25	3.93	4.04
7	MULTICLASSCLASSIFIER	9.21	8.43	5.39
8	RANDOM COMMITTEE	11.00	7.86	10.86
9	RANDOM FOREST	10.07	9.07	**12.54**
10	RANDOMTREE	2.79	8.11	6.75
11	REPTREE	7.82	9.64	**2.18**
12	RIPPER	4.54	**2.71**	3.54
13	STACKING	8.11	4.96	7.50
14	SVM	**1.50**	3.86	12.18

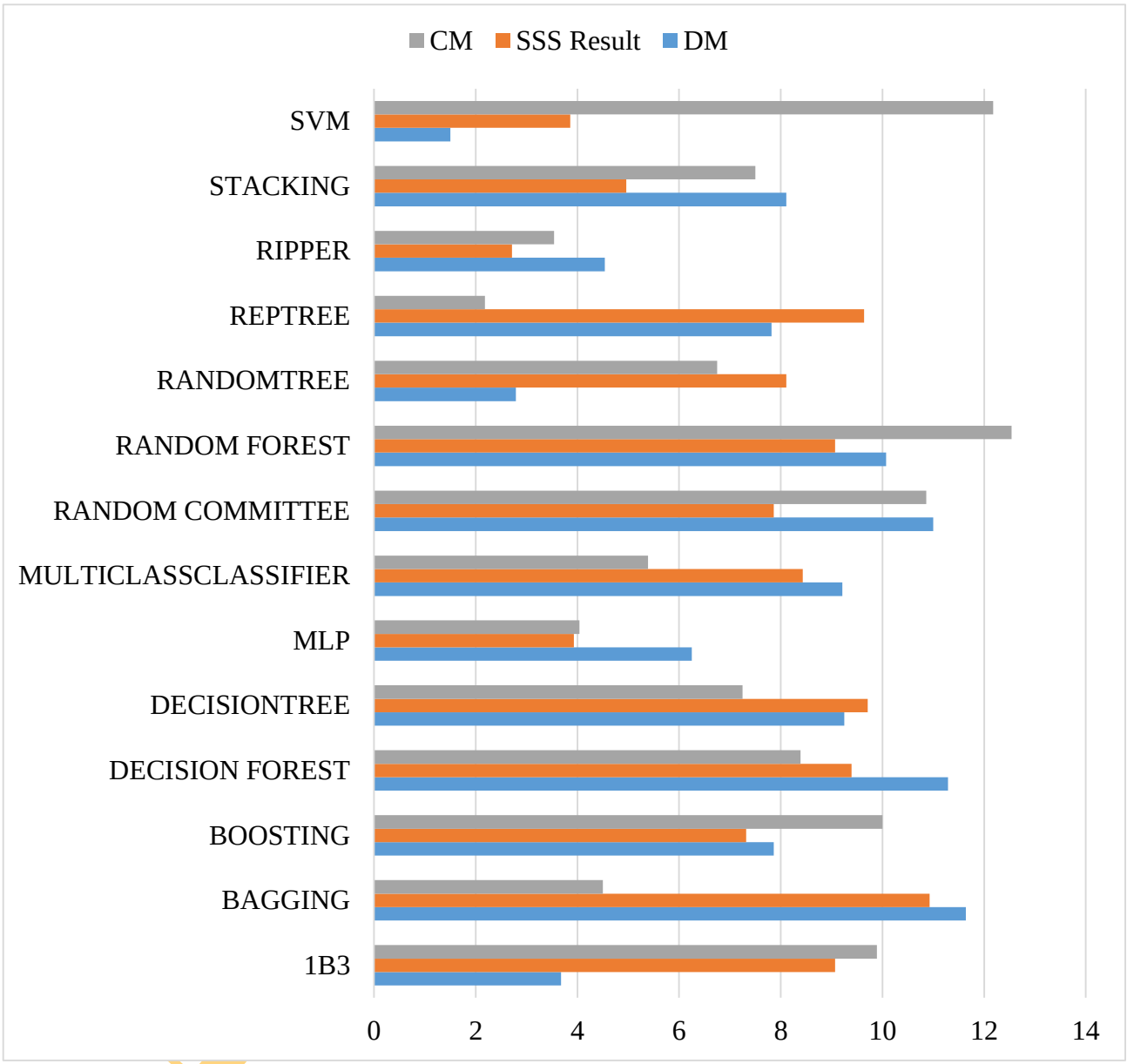


Figure 4.26: Chart showing the Report of Friedman analysis of on ROC_AUC metric for all classifiers

4.1.5.6 Report of Friedman test on Performance Loss/gain metric for on all datasets

The results obtained with Friedman's test on performance loss/gain metric on all three dataset with their mean rank value is presented in Table 4.28 and plotted in Figure 4.27. The lower the mean rank value of the data sampling scheme, the better.

SMOTE300ENN with a mean rank value of 1.25 and 2.25 gave the best performance on DM and SSS Result datasets respectively.

The CM dataset had SMOTE300RUS with a mean rank value of 1.46; one of enhanced data sampling schemes outperformed all other data sampling schemes.

Thus, Friedman analysis on performance loss/gain metric on all dataset revealed consistently that two of the enhanced data sampling schemes (SMOTE300ENN and SMOTE300NCL) were highly ranked out the best seven of the fourteen data sampling schemes across all datasets. CNN scheme incurred the greatest loss of performance across all datasets.

Table 4.28: Result of Friedman analysis on performance loss/gain metric for all datasets

S/N	Data sampling schemes	DM	SSS Result	CM
1	SMOTERUS	8.00	10.32	3.39
2	SMOTENCL	5.18	6.61	8.32
3	SMOTEENN	2.29	2.64	6.14
4	SMOTE300RUS	7.64	9.36	**1.46**
5	SMOTE300NCL	4.00	5.39	3.86
6	SMOTE300ENN	**1.25**	**2.25**	2.68
7	SMOTE300	6.54	8.43	6.11
8	SMOTE	8.89	9.43	10.36
9	RUS	10.89	11.54	11.54
10	NCL	7.18	6.89	9.79
11	5ENN	8.82	2.71	7.04
12	ENN	7.75	2.43	8.18
13	CNN	**12.57**	**13.00**	**12.14**

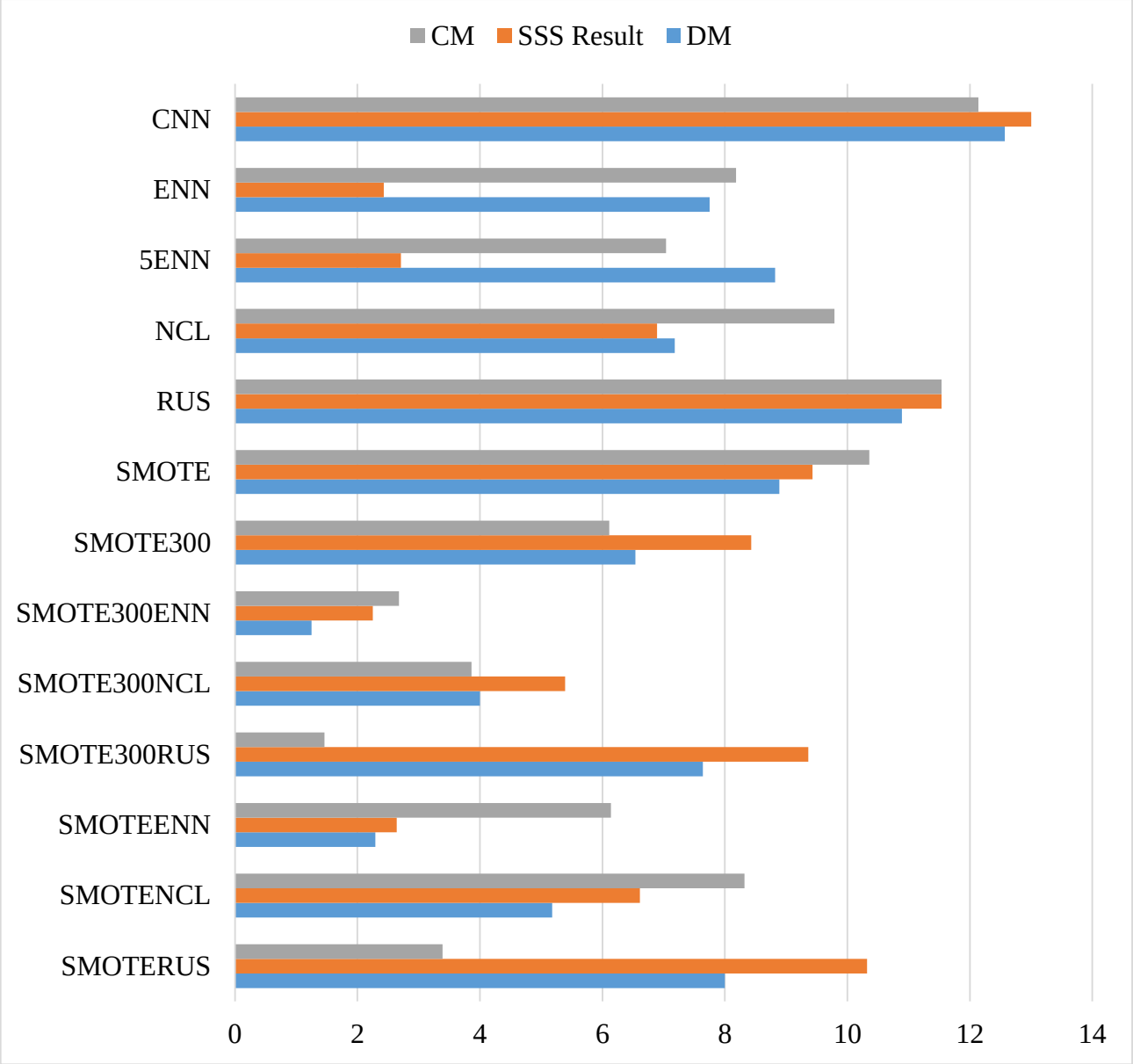


Figure 4.27: Chart showing the Result of Friedman analysis on Performance Loss/gain metric for all datasets

4.1.6 Result of Analysis of Variance (ANOVA)

This section presents the result from a two-way ANOVA test on all datasets in this study. The hypothesis used is that if $p - value \leq 0.05$, then the null hypothesis H_0 is rejected else the alternative hypothesis H_1 is accepted. If the null hypothesis is rejected, then there is enough evidence to conclude that at least one of the 14 data sampling schemes differ. So, a post hoc analysis using Tukey-Kramer (Tukey's W) multiple comparison analysis was be performed to determine which data sampling schemes were similar and by how much. Hence, homogeneous subsets was created by averaging the means of all data sampling schemes and displayed how data sampling schemes were grouped. The null hypothesis were all rejected in all the cases considered. The data sampling schemes in the same subset had the same block letter. They are statistically similar in characteristics and not significantly different from each other but from data sampling schemes with a different block letter. Data sampling schemes appearing in more than one subset have characteristics across all the data sampling schemes in those subsets.

4.1.6.1 ANOVA test on ROC_AUC metric on all datasets.

The result of ANOVA test result on ROC_AUC metric on all three datasets is presented in Table 4.29. Eight subsets were created where all the data sampling schemes were grouped separately for DM dataset. One of the enhanced data sampling schemes, SMOTE300ENN gave the best performance. It is in the same subset 'A' as SMOTENN, one of the existing data sampling schemes. CNN data sampling scheme performed least and was the only data sampling scheme in its subset 'G'

Similarly, SSS Result dataset had seven subsets created. One of the enhanced data sampling schemes, SMOTE300ENN outperformed all other data sampling schemes. It is in the same subset 'A' as three of existing data sampling schemes namely ENN, SMOTEENN and 5ENN data sampling schemes. CNN data sampling scheme performed least and was alone in its subset 'F'.

Also, nine subsets were created from the multiple comparisons of all data sampling schemes with the CM dataset. SMOTE300RUS, one of the enhanced data sampling schemes performed best of all data sampling schemes and in the same subset 'A' with

SMOTE300ENN and SMOTERUS. CNN data sampling scheme was the least performing scheme and alone in its subset 'I'.

Hence, the results from ANOVA test as confirmed by Friedman test bared that two of the enhanced data sampling schemes (SMOTE300ENN and SMOTE300NCL) were ranked among the best seven data sampling schemes across all three datasets.

Table 4.29: ANOVA on all datasets with all data sampling schemes using ROC_AUC metric

		DM		SSS Result		CM	
S/N	Data sampling schemes	Mean	Subset	Mean	Subset	Mean	Subset
1	SMOTERUS	0.8207	C,D,E	0.7529	D	0.6436	B,C
2	SMOTENCL	0.8429	B,C,D	0.8400	B	0.5850	E,F,G
3	SMOTEENN	0.9071	A,B	0.9886	A	0.6021	D,EF
4	SMOTE300RUS	0.8343	B,C,D,E	0.7893	C,D	0.6786	A
5	SMOTE300NCL	0.8686	B,C	0.8521	B	0.6343	B,C,D
6	SMOTE300ENN	0.9529	A	0.9914	A	0.6536	A
7	SMOTE300	0.8379	B,C,D	0.8093	B,C	0.6121	C,D,E
8	SMOTE	0.8029	C,D,E,F	0.7907	C,D	0.5686	F,G
9	RUS	0.7400	F	0.7079	E	0.5286	H,I
10	RAW DATA	0.7614	E,F	0.7800	C,D	0.5550	G,H
11	NCL	0.8307	C,D,E	0.8371	B	0.5736	F,G
12	ENN	0.7993	C,D,E,F	0.9900	A	0.5871	E,F,G
13	CNN	0.6421	G	0.5571	F	0.5071	I
14	5ENN	0.7914	D,E,F	0.9750	A	0.5986	E,F

4.1.6.2 ANOVA test on all datasets using Kappa statistics metric

The result of ANOVA on Kappa statistics metric on all the three datasets presented in Table 4.30.

From the analysis of the DM dataset, the six homogeneous subset were created. SMOTE300ENN, one of the enhanced data sampling schemes having the highest mean value is alone in its subset 'A'. The next subset 'B' had the remaining of the enhanced data sampling schemes: SMOTE300NCL, SMOTENCL, SMOTERUS and SMOTE300RUS alongside with SMOTEENN and SMOTE300, two of the existing data sampling schemes in the same subset. The least performing scheme, CNN was in the same subset 'F' as ENN, 5ENN and RAW DATA with similar characteristics of low mean value and poor performance.

The homogeneous subset created for SSS Result dataset were seven in number. The first subset 'A' had four components namely SMOTE300ENN, the best performing data sampling schemes and ENN, SMOTEENN and 5ENN. The least performing scheme, CNN is alone in its subset 'G'.

Similarly, for CM dataset, eight homogenous subsets were created. Three of the enhanced data sampling schemes namely SMOTE300RUS, SMOTE300ENN and SMOTE300NCL are components of the leading subset 'A'. Again, CNN performed least of all the data sampling schemes is alone in its subset 'H'.

The results from ANOVA test as validated by Friedman test revealed that three of the enhanced data sampling schemes (SMOTE300ENN, SMOTE300NCL and SMOTENCL) were ranked amongst the best seven data sampling schemes across all datasets.

Table 4.30: ANOVA on all datasets with all data sampling schemes using Kappa Statistics metric

		DM		SSS Result		CM	
S/N	Data sampling schemes	Mean	Subset	Mean	Subset	Mean	Subset
1	SMOTERUS	0.5536	B,C	0.3979	D,E	0.1650	B,C
2	SMOTENCL	0.5900	B,C	0.4843	B,C,D	0.1057	D,E
3	SMOTEENN	0.6150	B,C	0.9400	A	0.1371	C,D
4	SMOTE300RUS	0.5471	B,C	0.4293	B,C,D,E	0.2264	A
5	SMOTE300NCL	0.6393	B	0.4907	B,C	0.1971	A,B
6	SMOTE300ENN	0.8000	A	0.9514	A	0.2200	A
7	SMOTE300	0.5386	B,C,D	0.4157	C,D,E	0.1750	B,C
8	SMOTE	0.4007	E	0.3536	E,F	0.0836	H
9	RUS	0.4171	D,E	0.2857	F	0.0329	G
10	RAW DATA	0.2564	F	0.3521	E,F	0.0421	F,G
11	NCL	0.5071	C,D,E	0.5150	B	0.0907	E
12	ENN	0.2171	F	0.9464	A	0.0814	E,F
13	CNN	0.2107	F	-0.0936	G	-0.0150	H
14	5ENN	0.2307	F	0.9279	A	0.0986	D,E

4.1.6.3 ANOVA test on RMSE metric all datasets

The report of the ANOVA test on RMSE metric on all three datasets in this study is presented in Table 4.31.

Four subsets were created for DM dataset. 5ENN, ENN, SMOTEENN, all existing data sampling schemes and SMOTE300ENN, one of the enhanced data sampling schemes are in the same subset 'D'. The RUS data sampling scheme performed least and is alone in its subset 'A'.

Five homogeneous subsets were created on SSS Result dataset. Subset 'E' with components ENN, 5ENN, SMOTEENN and SMOTE300ENN performed best. The least performing data sampling scheme, CNN, is in the same subset 'A' with RUS data sampling scheme.

Five homogeneous subsets were created for CM dataset. 5ENN, SMOTEENN, ENN, SMOTE300, SMOTE, SMOTE300ENN and SMOTE300NCL, are all in the same subset 'E' with low RMSE value. The least performing data sampling scheme, RUS is alone in its subset 'A'.

Also, results from ANOVA test as confirmed by Friedman test showed that ENN data sampling scheme performed best on DM and SSS Result dataset while 5ENN gave the best performance on the CM dataset. Two of the enhanced data sampling schemes (SMOTE300ENN and SMOTE300NCL) were ranked amongst the best seven data sampling schemes across all datasets.

Table 4.31: ANOVA on all datasets with all data sampling schemes using RMSE metric

		DM		SSS Result		CM	
S/N	Data sampling Schemes	Mean	Subset	Mean	Subset	Mean	Subset
1	SMOTERUS	0.3979	B	0.3929	B	0.4000	B
2	SMOTENCL	0.2243	C	0.3186	D	0.3693	D
3	SMOTEENN	0.1407	D	0.1086	E	0.3507	E
4	SMOTE300RUS	0.385	B	0.3829	A,B,C	0.3900	B
5	SMOTE300NCL	0.2414	C	0.3207	D	0.3643	D,E
6	SMOTE300ENN	0.1521	D	0.1064	E	0.3514	E
7	SMOTE300	0.2500	C	0.3357	B,C,D	0.3621	D,E
8	SMOTE	0.2436	C	0.335	B,C,D	0.3643	D,E
9	RUS	0.4443	A	0.4171	A	0.4207	A
10	RAW DATA	0.2421	C	0.3264	C,D	0.3686	D
11	NCL	0.2179	C	0.3093	D	0.3707	C,D
12	ENN	0.1336	D	0.1036	E	0.3586	D,E
13	CNN	0.3979	B	0.4193	A	0.3864	B,C
14	5ENN	0.1329	D	0.1071	E	0.3486	E

4.1.6.4 ANOVA test on RECALL of minority class metric on all datasets

The report of the ANOVA test on RECALL of the minority class metric on all three datasets is presented in Table 4.32.

From the report of the analysis on DM dataset, six homogeneous subsets were created. SMOTE300ENN, one of the enhanced data sampling schemes gave the best RECALL of the minority class (GDM). The components of the first subset 'A' are SMOTE300ENN, SMOTE300RUS, SMOTE300NCL, SMOTERUS and SMOTEENN. The least performing data sampling scheme, ENN and 5ENN, CNN, RAW DATA are all in the same subset 'F'. They gave zero RECALL of the minority class.

Analysis on SSS Result dataset had six homogeneous subsets created. SMOTE300ENN gave the best RECALL of the minority class, PASSWAEC and also alone in its subset 'A'. Again, the least performing data sampling scheme, ENN and 5ENN, CNN, NCL and the RAW DATA are all in the same subset 'F'. They gave very low RECALL of the minority class, PASSWAEC.

The result of ANOVA test which were also confirmed by Friedman's test on the RECALL of the minority class established that four of the enhanced data sampling schemes namely SMOTE300ENN, SMOTE300RUS, SMOTE300NCL and SMOTERUS were amongst the seven best ranked of all data sampling schemes.

Table 4.32: ANOVA on all datasets with all data sampling schemes using RECALL of minority class metric

		DM		SSS Result		CM	
S/N	Data sampling schemes	Mean	Subset	Mean	Subset	Mean	Subset
1	SMOTERUS	0.7514	A,B,C	0.5714	C,D	0.6057	B,C
2	SMOTENCL	0.6557	B,C,D	0.2057	E	0.4543	D
3	SMOTEENN	0.7257	A,B,C	0.6786	B	0.5471	C,D
4	SMOTE300RUS	0.7871	A,B	0.6486	B,C	0.8229	A
5	SMOTE300NCL	0.7564	A,B,C	0.5136	D	0.7550	A
6	SMOTE300ENN	0.8464	A	0.9836	A	0.8064	A
7	SMOTE300	0.6300	B,C,D	0.5243	D	0.6957	A,B
8	SMOTE	0.5000	D,E	0.2064	E	0.4571	D
9	RUS	0.5886	C,D,E	0.4779	D	0.2421	E
10	RAW DATA	0.1471	F	0.0029	F	0.1236	E,F
11	NCL	0.4179	E	0.0364	F	0.0043	F
12	ENN	0.0893	F	0.0000	F	0.0850	F
13	CNN	0.1857	F	0.0229	F	0.0100	F
14	5ENN	0.1429	F	0.0000	F	0.1414	E,F

4.1.6.5 ANOVA test on all classifiers

Analysis of the performance of classifiers used in the study on all the datasets was carried out using the ROC_AUC metric. The results are presented in Table 4.33.

It can be observed from Table 4.33 that SVM was the least performing classifier on the DM dataset and was alone in its subset out of the four subsets created. DECISION FOREST gave the best performance of all the classifiers and it shared similar classification characteristics with BAGGING, RANDOM COMMITTEE, RANDOM FOREST, DECISION TREE, MULTICLASS CLASSIFIER, STACKING, BOOSTING, REP TREE, MLP and RIPPER in the same subset 'A' respectively.

The *p*–value = Sig. (Significance) ≈ 0.675 > 0.05, so both the null and alternative hypothesis were retained for SSS Result dataset. The mean values do not differ amongst the 14 classifiers so there was no need to perform a post hoc test. However, the post hoc Tukey-Kramer (Tukey's W) multiple comparison analysis was still carried out. One homogeneous subset table was created as shown in the result displayed in Table 4.33. All the 14 classifiers were grouped together in one subset 'A'. DECISION FOREST classifier gave the best performance while RIPPER classifier gave the least performance.

Four homogeneous subsets were created for the CM dataset. RANDOM FOREST classifier gave the best performance out of all classifiers and is in the same subset 'A' with SVM, RANDOM COMMITTEE, 1B3, BOOSTING, STACKING, DECION FOREST, DECISION TREE, RANDOM TREE, MULTICLASSCLASSIFIER and BAGGING classifiers. REPTREE gave the least performance of all the classifiers and is in the same subset 'D' as RIPPER, MLP, BAGGING, MULTICLASS CLASSIFIER, RANDOM TREE, DECISION TREE and STACKING.

Therefore, the ANOVA test result which was also validated by Friedman's test showed that all the classifiers behaved similarly on all the 14 different data sampling schemes with SSS Result dataset. Any of the classifiers can be used for classification of this dataset.

Considering base classifiers, Decision Tree classifier gave the best performance on DM and SSS Result dataset. The SVM classifier surpassed other classifiers on CM dataset.

For ensembles, BAGGING, a homogeneous ensemble with Decision Tree classifier as the base classifier gave the best performance on DM and SSS Result dataset while RANDOM FOREST, also a homogeneous ensemble gave the best performance CM dataset. BOOSTING, a homogeneous ensemble with Decision Tree classifier as the base classifier gave the least performance on DM dataset. STACKING, a heterogeneous ensemble had the least performance on SSS Result dataset while BAGGING had the least performance on CM dataset. SVM, RIPPER and REPTREE classifiers had the poorest performance on all three dataset respectively.

Table 4.33: ANOVA on all classifiers with all data sampling schemes using ROC_AUC metric

		DM		SSS Result		CM	
S/N	Classifiers	Mean	Subset	Mean	Subset	Mean	Subset
1	DECISION FOREST	0.8764	A	0.8550	A	0.6007	A,B,C,D
2	BAGGING	0.8743	A	0.8543	A	0.5736	A,B,C,D
3	RANDOM COMMITTEE	0.8729	A	0.8436	A	0.6243	A,B,C
4	RANDOM FOREST	0.8657	A,B	0.8471	A	0.6350	A
5	DECISION TREE	0.8629	A,B	0.8507	A	0.5950	A,B,C,D
6	MULTICLASSCLASSIFIER	0.8607	A,B,C	0.8493	A	0.5800	A,B,C,D
7	STACKING	0.8521	A,B,C	0.8386	A	0.6007	A,B,C,D
8	BOOSTING	0.8479	A,B,C	0.8393	A	0.6129	A,B,C
9	REP TREE	0.8379	A,B,C	0.8507	A	0.5379	D
10	MLP	0.8029	A,B,C	0.8057	A	0.5650	B,C,D
11	RIPPER	0.7700	A,B,C	0.7579	A	0.5636	C,D
12	1B3	0.7521	B,C	0.8471	A	0.6143	A,B,C
13	RANDOM TREE	0.7421	C	0.8443	A	0.5914	A,B,C,D
14	SVM	0.6143	D	0.7779	A	0.6336	A,B

4.1.6.6 ANOVA on all datasets with all data sampling schemes using performance loss/gain metric

The result of ANOVA test on performance loss/gain on both enhanced and existing data sampling schemes is presented in Table 4.34.

Five homogenous subsets were created for the DM dataset. One of the enhanced data sampling schemes, SMOTE300ENN performed best. This established an improvement (gain) on performance with respect to the RAW DATA. SMOTE300ENN, SMOTEENN and SMOTE300NCL are all members of the same subset 'E'. The CNN data sampling performed least and is alone in its subset 'A'.

Similarly, six homogeneous subsets were created for SSS Result dataset. SMOTE300ENN gave the best improvement and in the same subset 'E' as ENN, SMOTEENN, 5ENN, SMOTE300NCL and SMOTENCL data sampling schemes. CNN data sampling scheme performed least and alone in its subset 'E'.

Seven homogeneous subsets were created for CM dataset. SMOTE300RUS and SMOTE300ENN are in the same subset 'G' and gave good performance. The data sampling schemes with the least performance were are CNN and RUS respectively and are both in the same subset 'A'.

Therefore, ANOVA test result as confirmed by Friedman test on performance loss/gain metric on all dataset revealed that two (SMOTE300ENN and SMOTE300NCL) of the enhanced data sampling schemes were ranked out of the best seven of the fourteen data sampling schemes across all datasets. CNN data sampling scheme gave the worst performance across all dataset.

The average performance loss/gain metric on all the data sampling schemes across the three datasets is presented in Table 4.35. The results in the table also corroborates the result of analysis.

Table 4.34: ANOVA on all datasets with all data sampling schemes using performance loss/gain metric

		DM		SSS Result		CM	
S/N	Data sampling schemes	Mean	Subset	Mean	Subset	Mean	Subset
1	SMOTERUS	-0.0948	B,C,D	0.0279	B,C	-0.1618	E,F
2	SMOTENCL	-0.1150	C,D	-0.0785	C,D,E	-0.0542	B,C
3	SMOTEENN	-0.1934	D,E	-0.2762	E	-0.0851	B,C,D
4	SMOTE300RUS	-0.1155	C,D	-0.0197	C,D	-0.2245	G
5	SMOTE300NCL	-0.1552	C,D,E	-0.0948	C,D,E	-0.1435	D,E,F
6	SMOTE300ENN	-0.2711	E	-0.2804	E	-0.1776	F,G
7	SMOTE300	-0.1128	C,D	-0.0387	C,D,E	-0.1039	C,D,E
8	SMOTE	-0.0573	B,C	-0.0142	C,D	-0.0248	B
9	RUS	0.0156	B	0.0851	B	0.0470	A
10	NCL	-0.0990	B,C,D	-0.0761	C,D,E	-0.0328	B
11	ENN	-0.0507	B,C	-0.2784	E	-0.0580	B,C
12	CNN	0.1473	A	0.2807	A	0.0836	A
13	5ENN	-0.0387	B,C	-0.2580	E	-0.0786	B,C

Table 4.35: Summary of performance gain/loss on performance of the scheme compared to the RAW DATA in percentage

SCHEMES / DATASETS	SMOTE 300ENN	5ENN	CNN	ENN	NCL	RUS	SMOTE 300	SMOTE ENN	SMOTE NCL	SMOTE RUS	SMOTE 300NCL	SMOTE 300RUS
DM	27	4	-15	5	10	-2	6	11	19	12	9	16	12
SSS Result	28	26	-28	28	8	-9	1	4	28	8	-3	9	2
CM	18	8	-8	6	3	5	2	10	9	5	16	14	22
AVERAGE	24	13	-17	13	7	-2	3	8	19	8	7	13	12

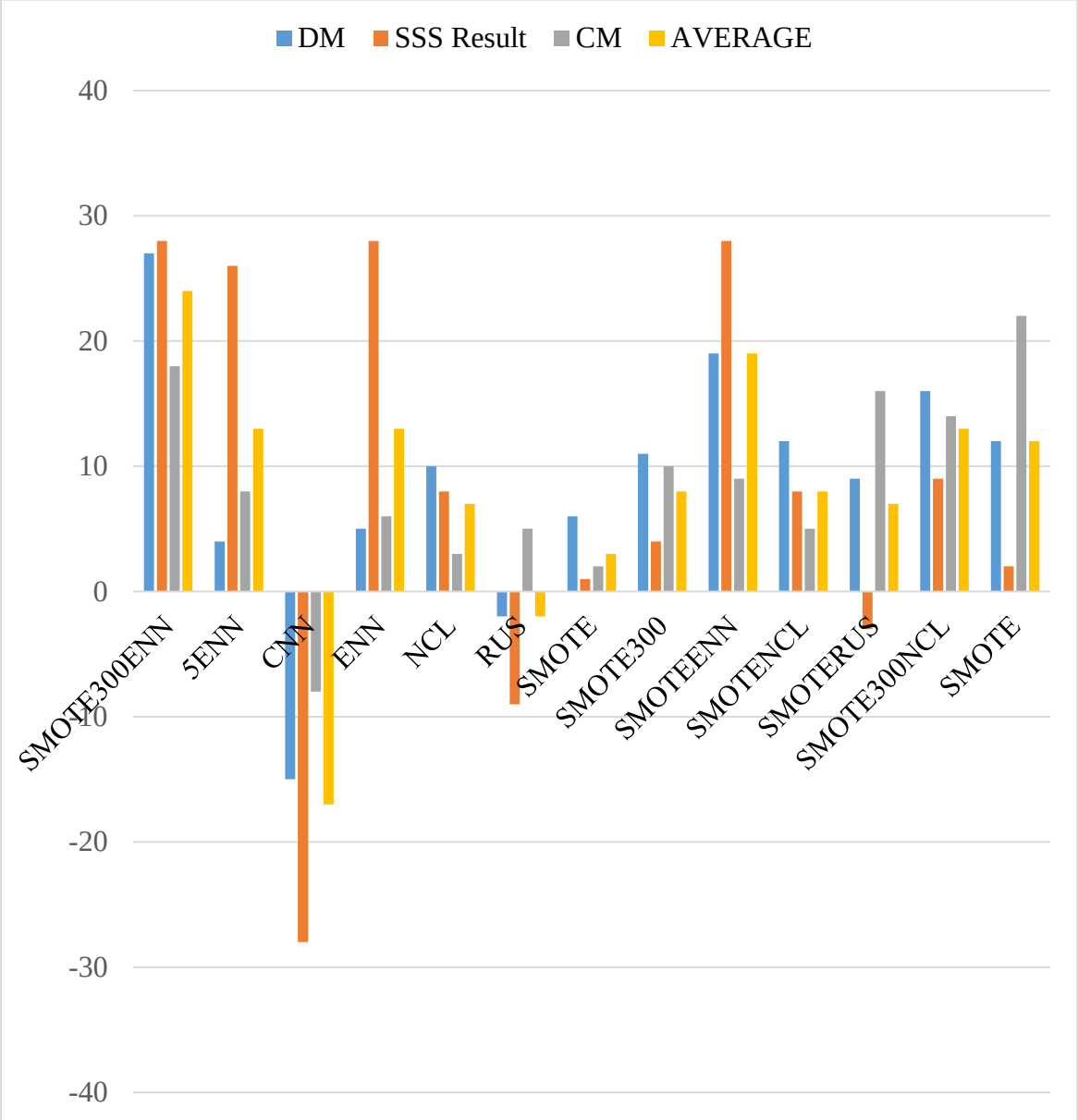


Figure 4.28: Chart showing the summary of performance gain/loss on performance of the scheme compared to the RAW DATA in percentage

4.1.7 Box and whisker Plot

The box and whisker plots were plotted from values of all the metrics used in the study. The values of metric under consideration were plotted on the Y-axis while the data sampling schemes and classifiers were plotted on the X-axis.

Each box in the plots represents a data sampling scheme. It allows for easy comparison of data sampling schemes at various points in the distribution. The whiskers at the end of the box plots show the minimum and maximum values, while the bar shows the median. If the median bar is above zero or higher, the data sampling scheme represented by the box plot is doing better on average than the data sampling scheme is being compared with. And if the complete box, including the whiskers, is above zero, then that scheme is consistently better than the other data sampling schemes. The findings from the box and whisker plots also conform to that of Friedman and ANOVA's test. Figure 4.29 to Figure 4.46 presents the box and whisker plots analysis for this study.

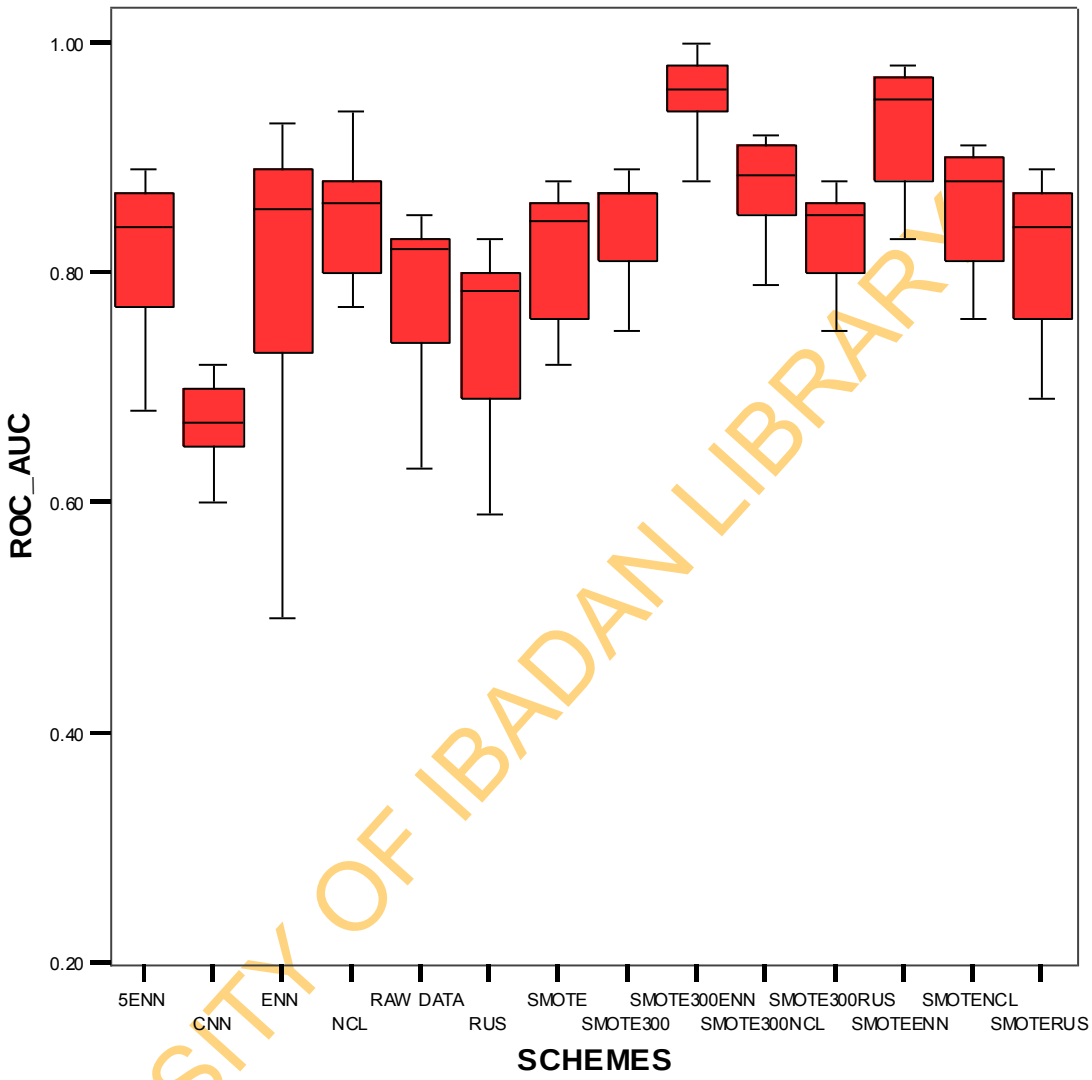


Figure 4.29: Box and whisker plots for ROC_AUC metric for DM dataset

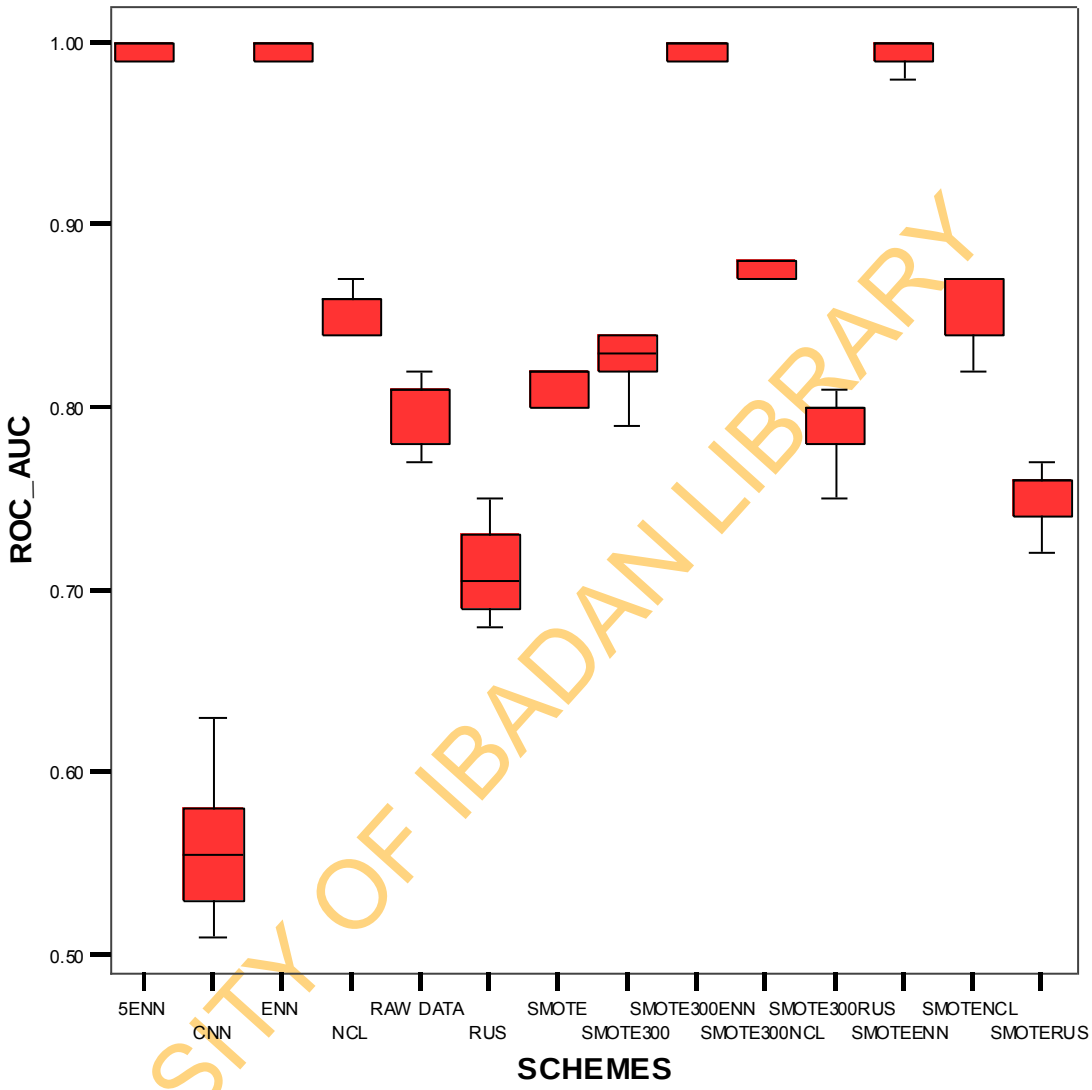


Figure 4.30: Box and whisker plots for ROC_AUC metric for SSS Result dataset

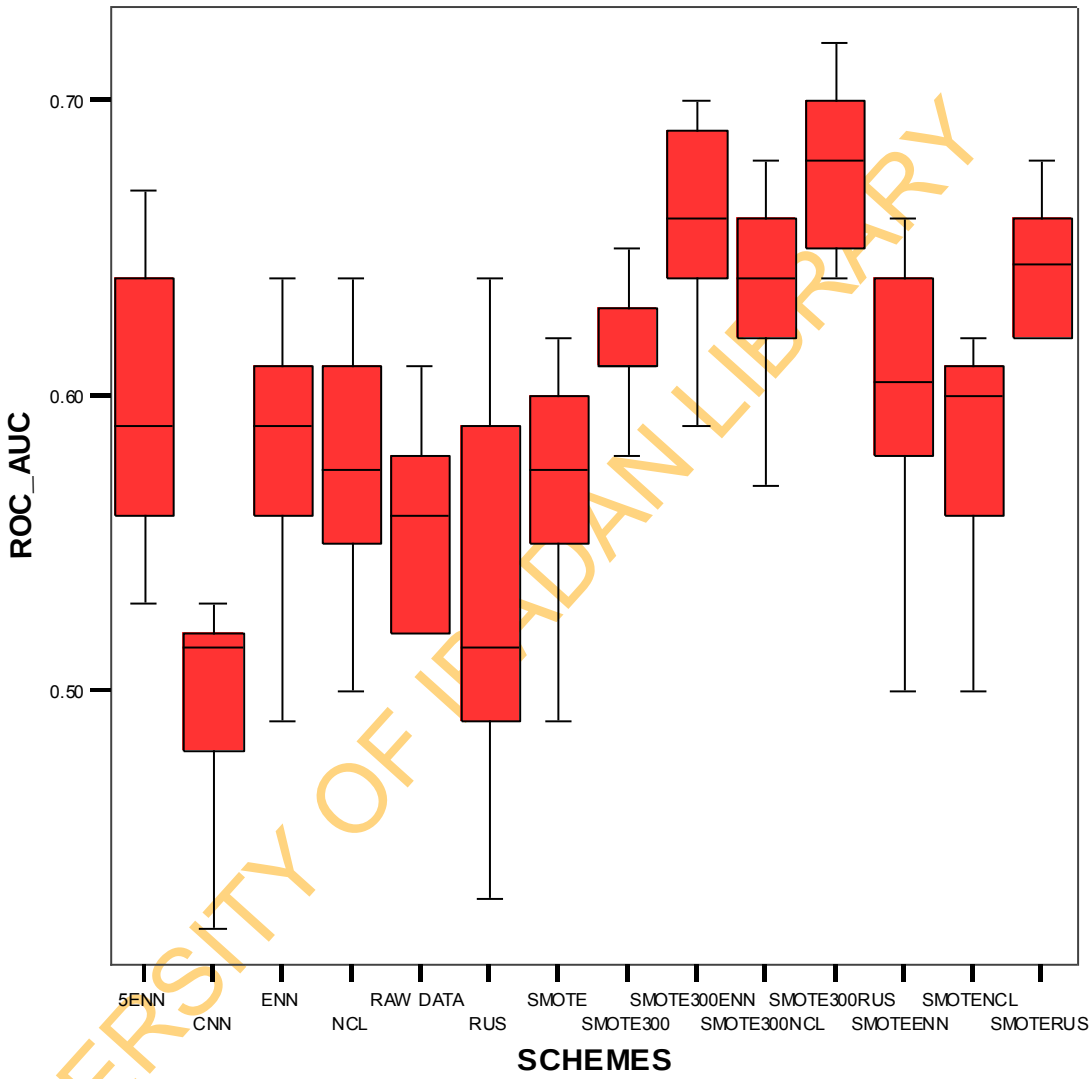


Figure 4.31: Box and whisker plots for ROC_AUC metric for CM dataset

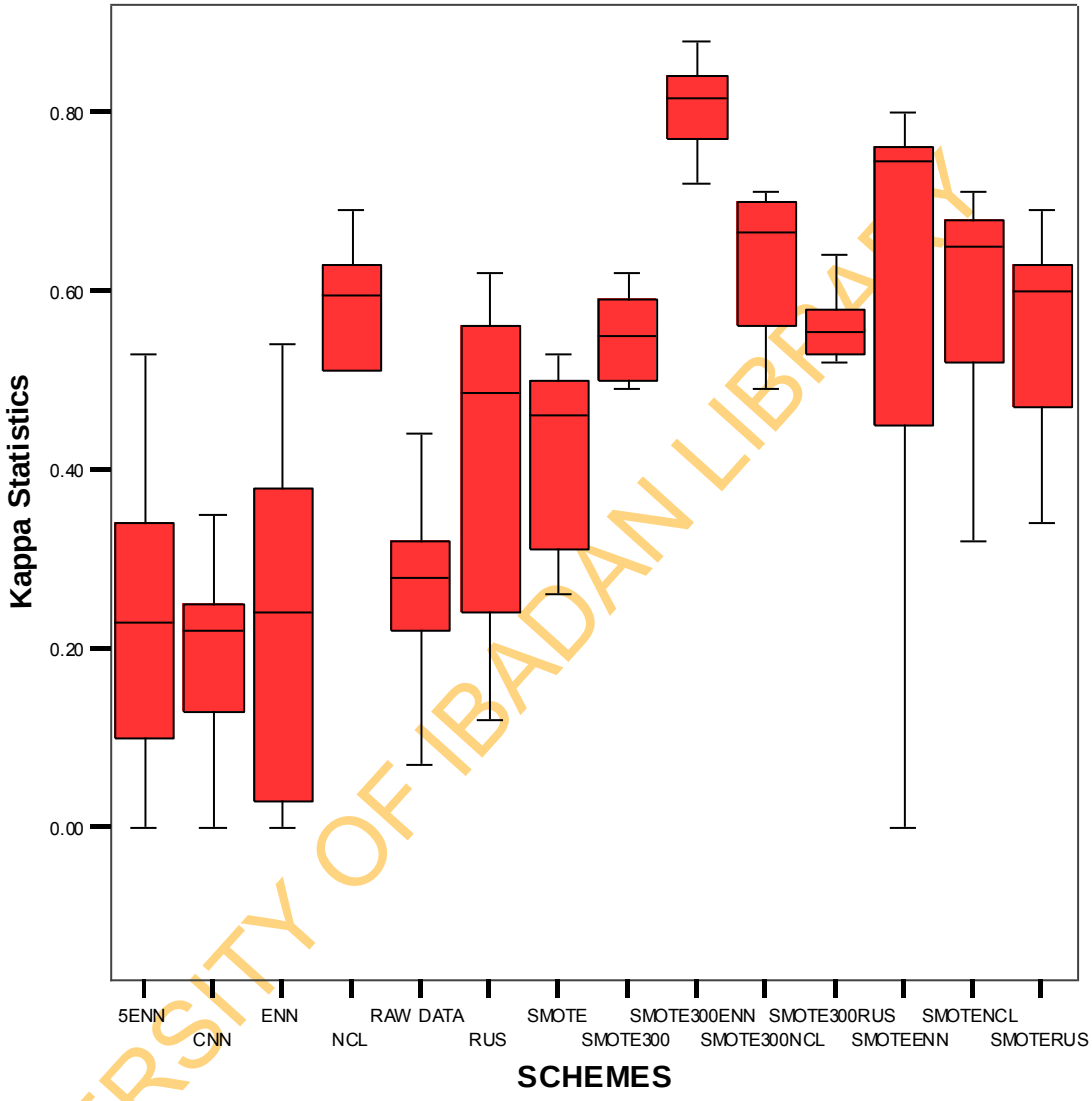


Figure 4.32: Box and whisker plots for Kappa Statistics metric for DM dataset

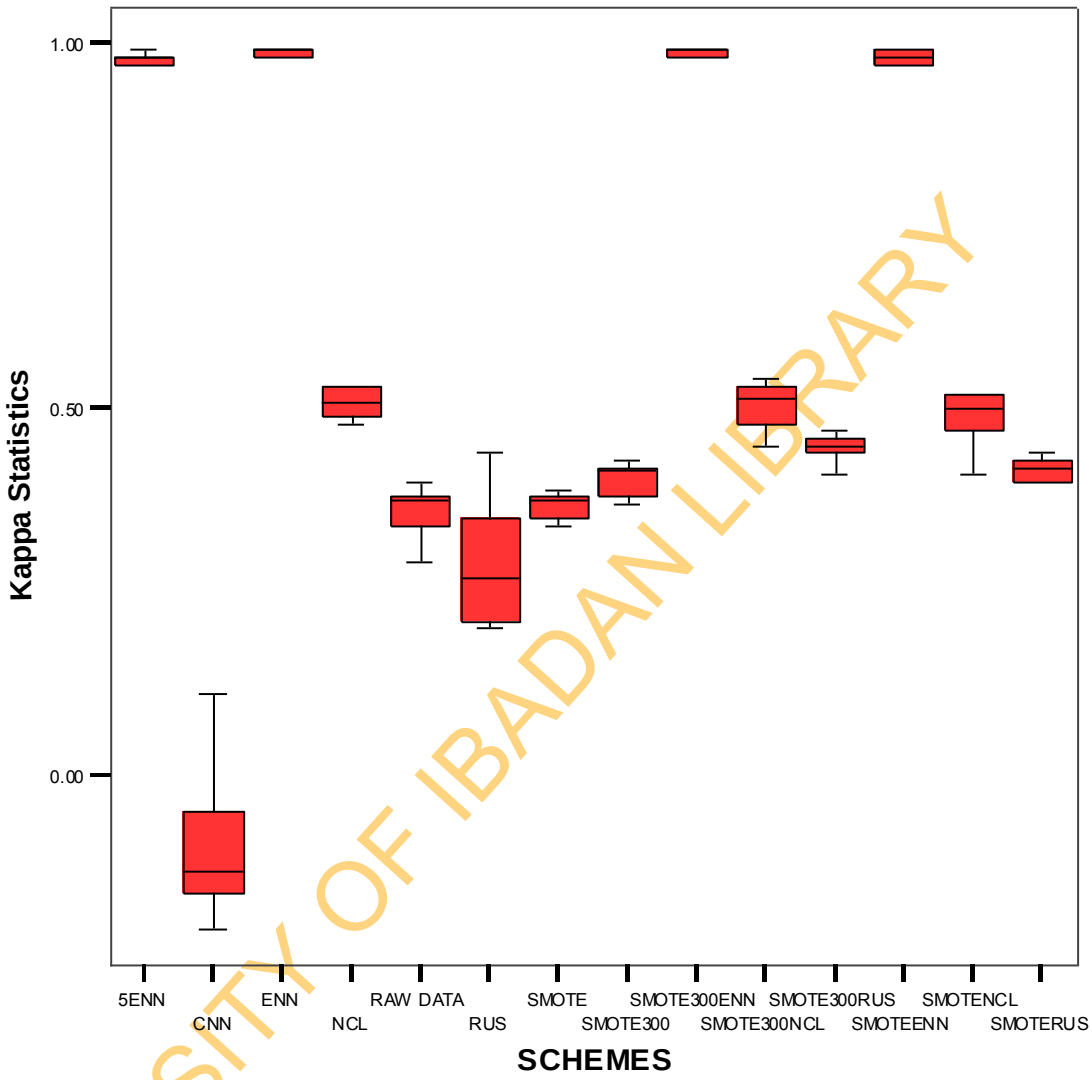


Figure 4.33: Box and whisker plots for Kappa Statistics metric for SSS Result dataset

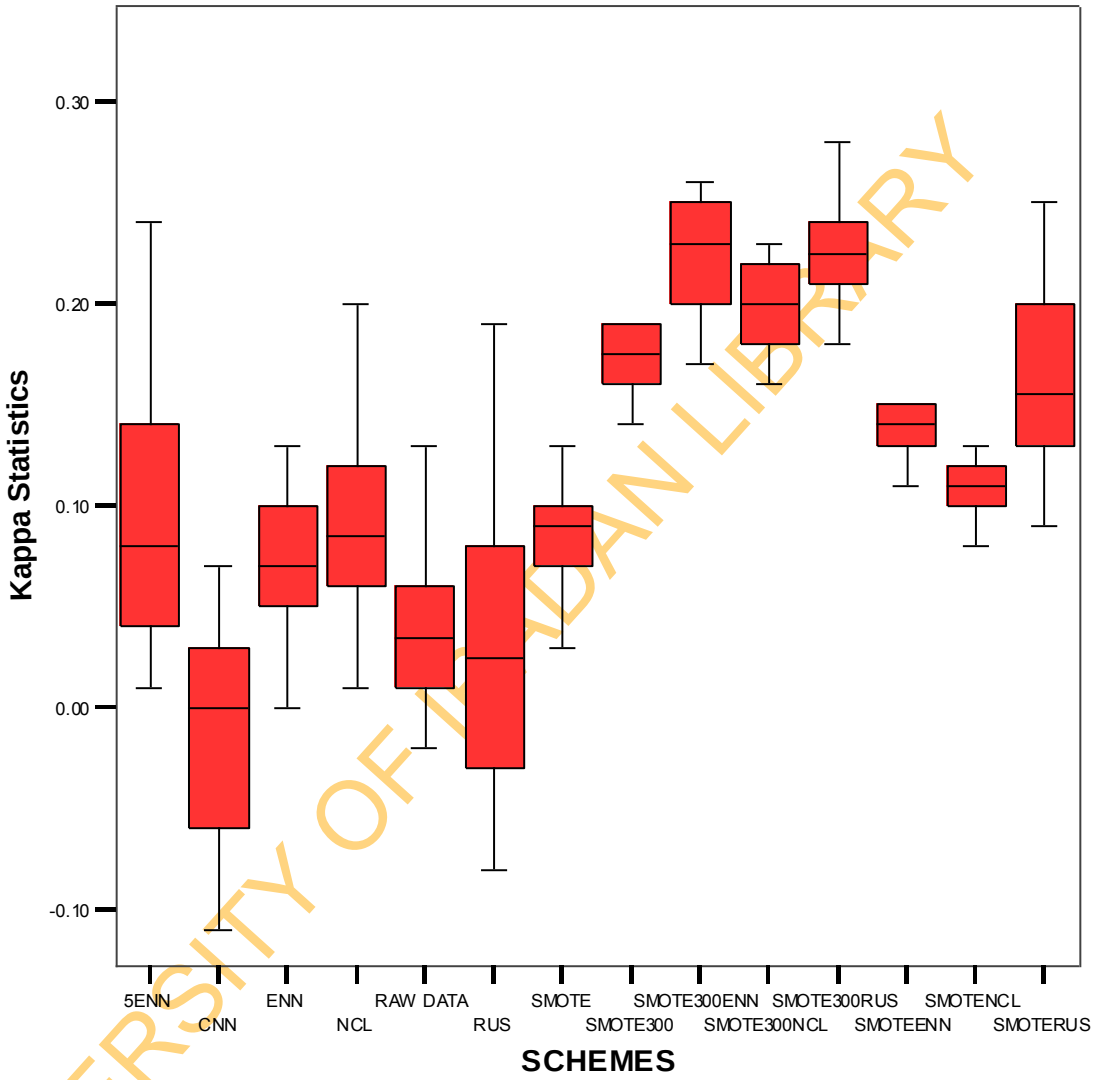


Figure 4.34: Box and whisker plots for Kappa statistics metric for CM dataset

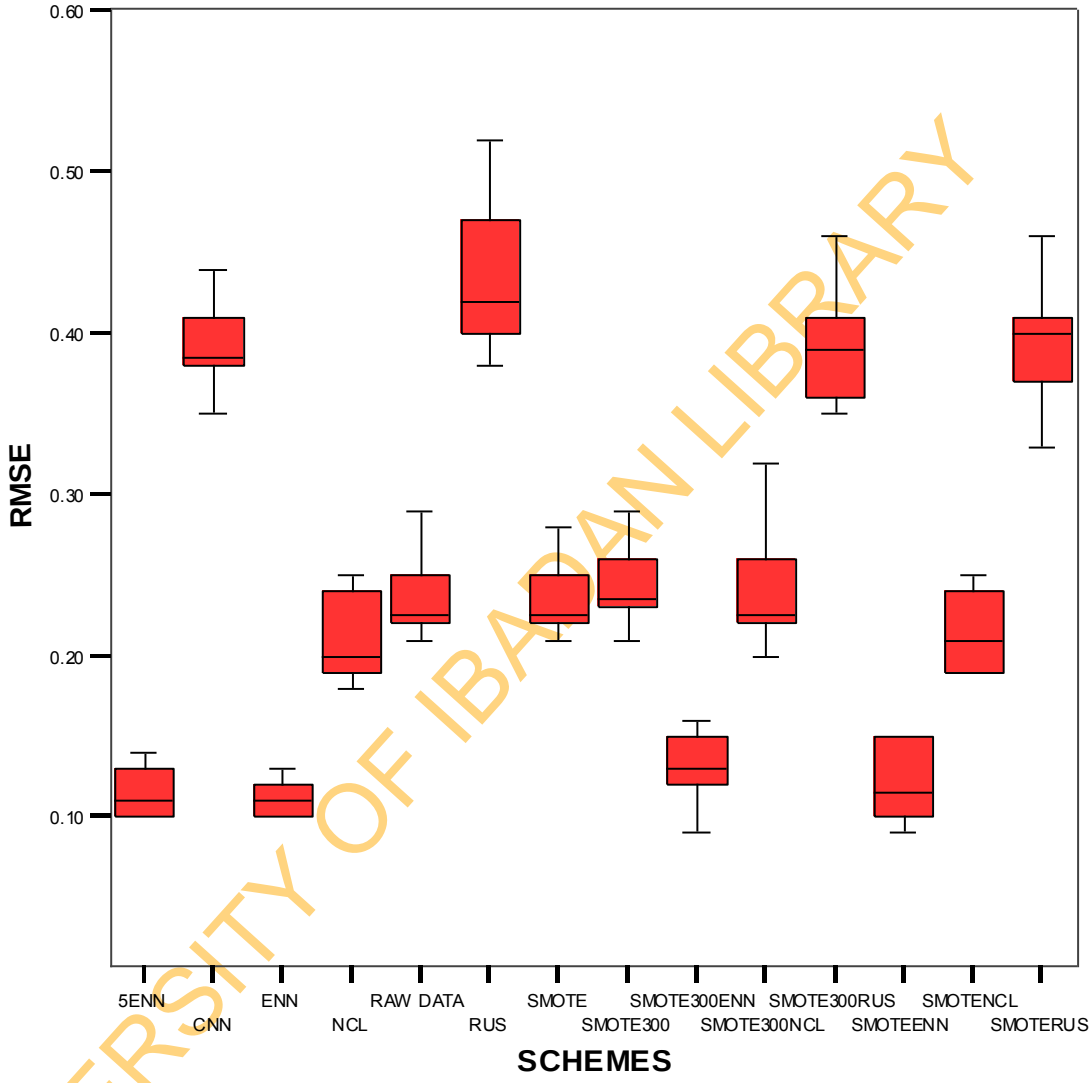


Figure 4.35: Box and whisker plots for RMSE metric for DM dataset

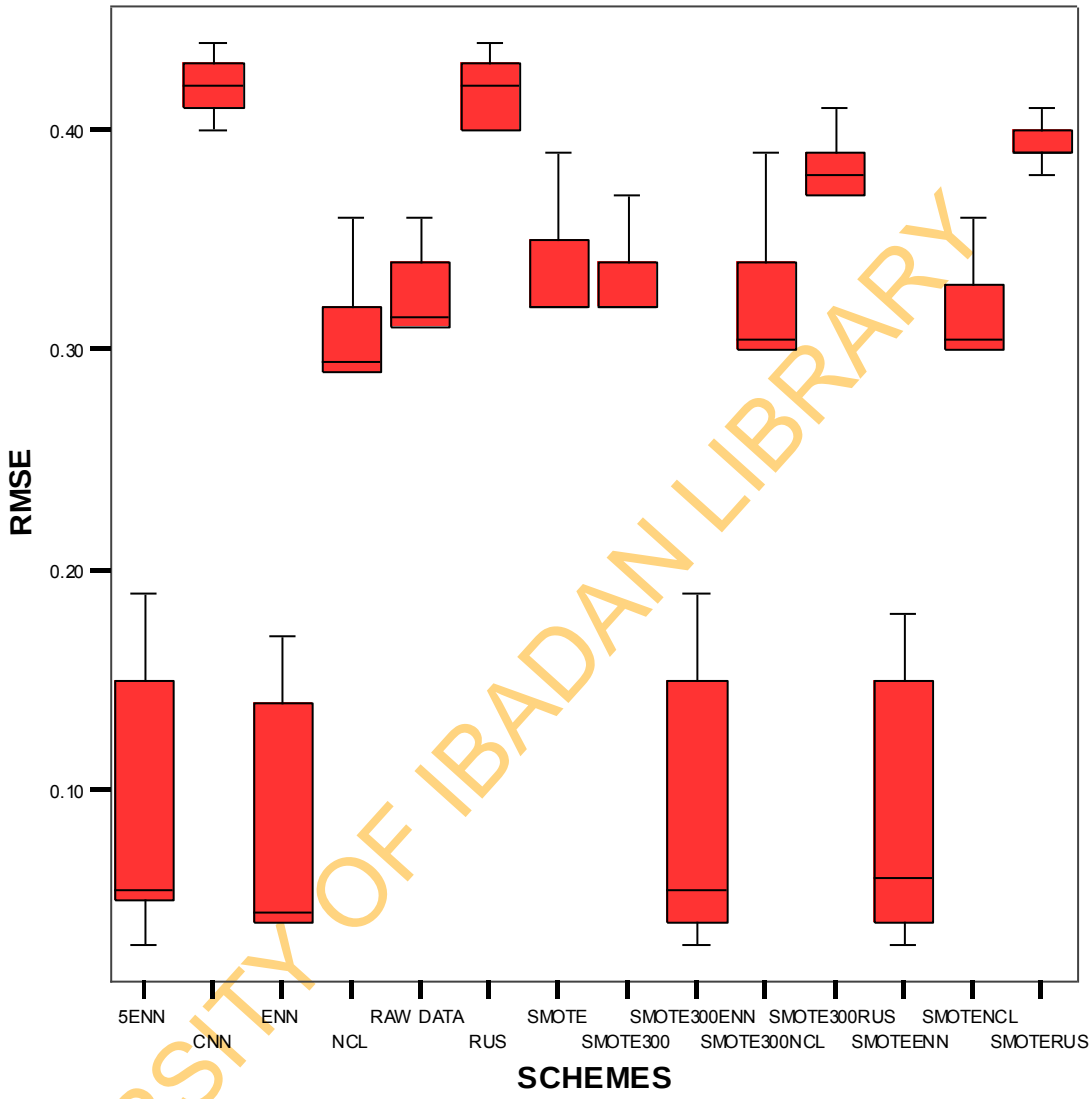


Figure 4.36: Box and whisker plots for RMSE metric for SSS Result dataset

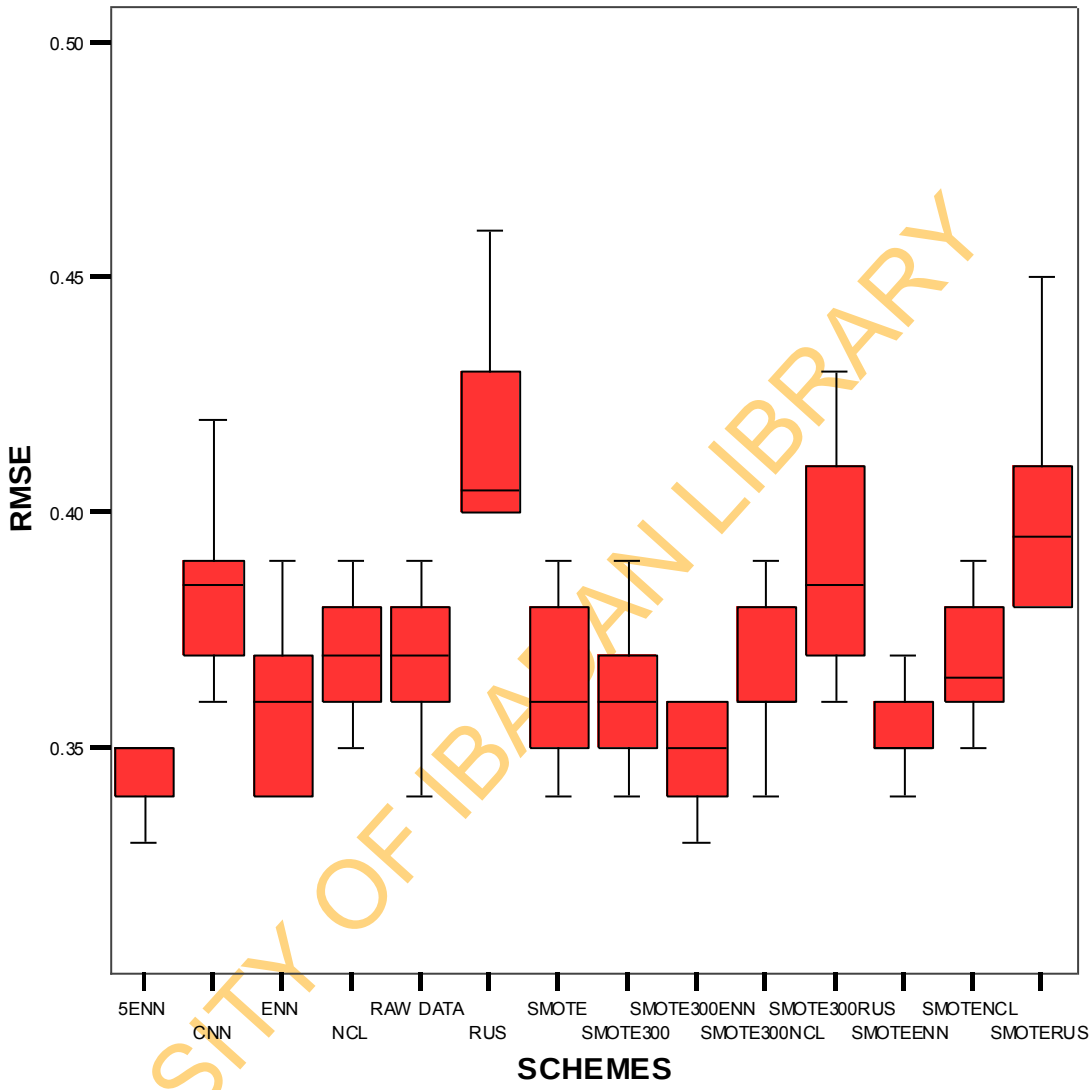


Figure 4.37: Box and whisker plots for RMSE metric for CM dataset

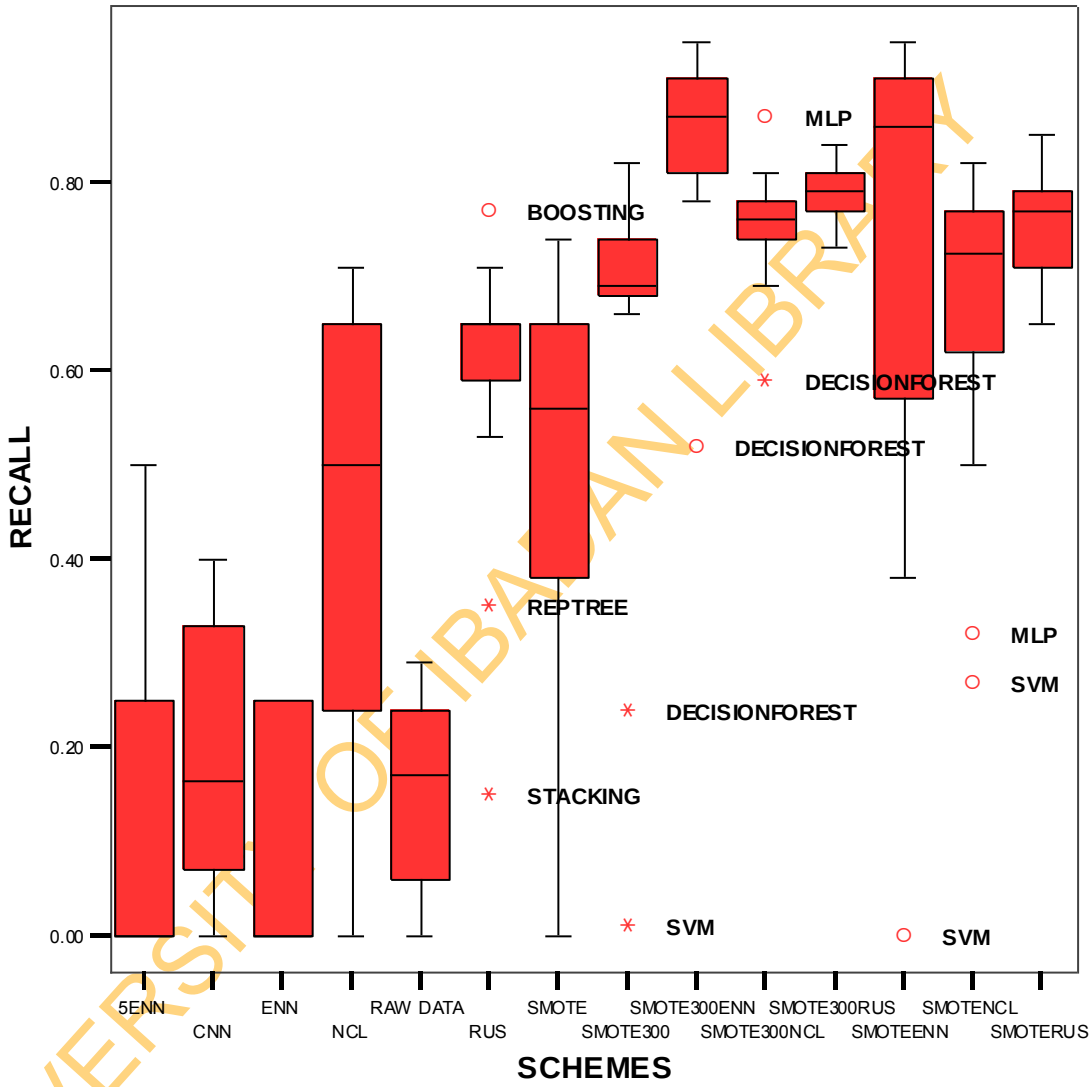


Figure 4.38: Box and whisker plots for RECALL metric for DM dataset

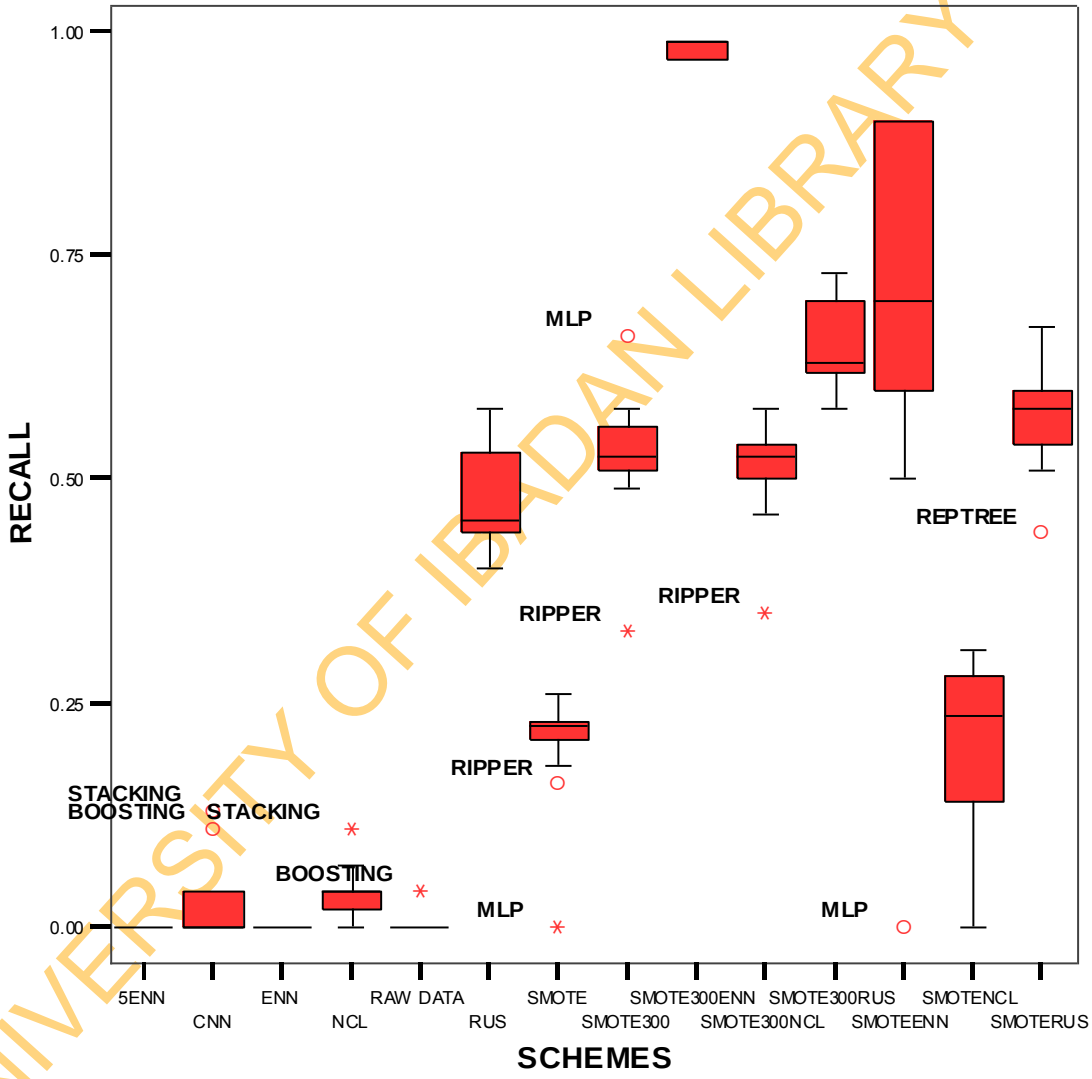


Figure 4.39: Box and whisker plots for RECALL metric for SSS Result dataset

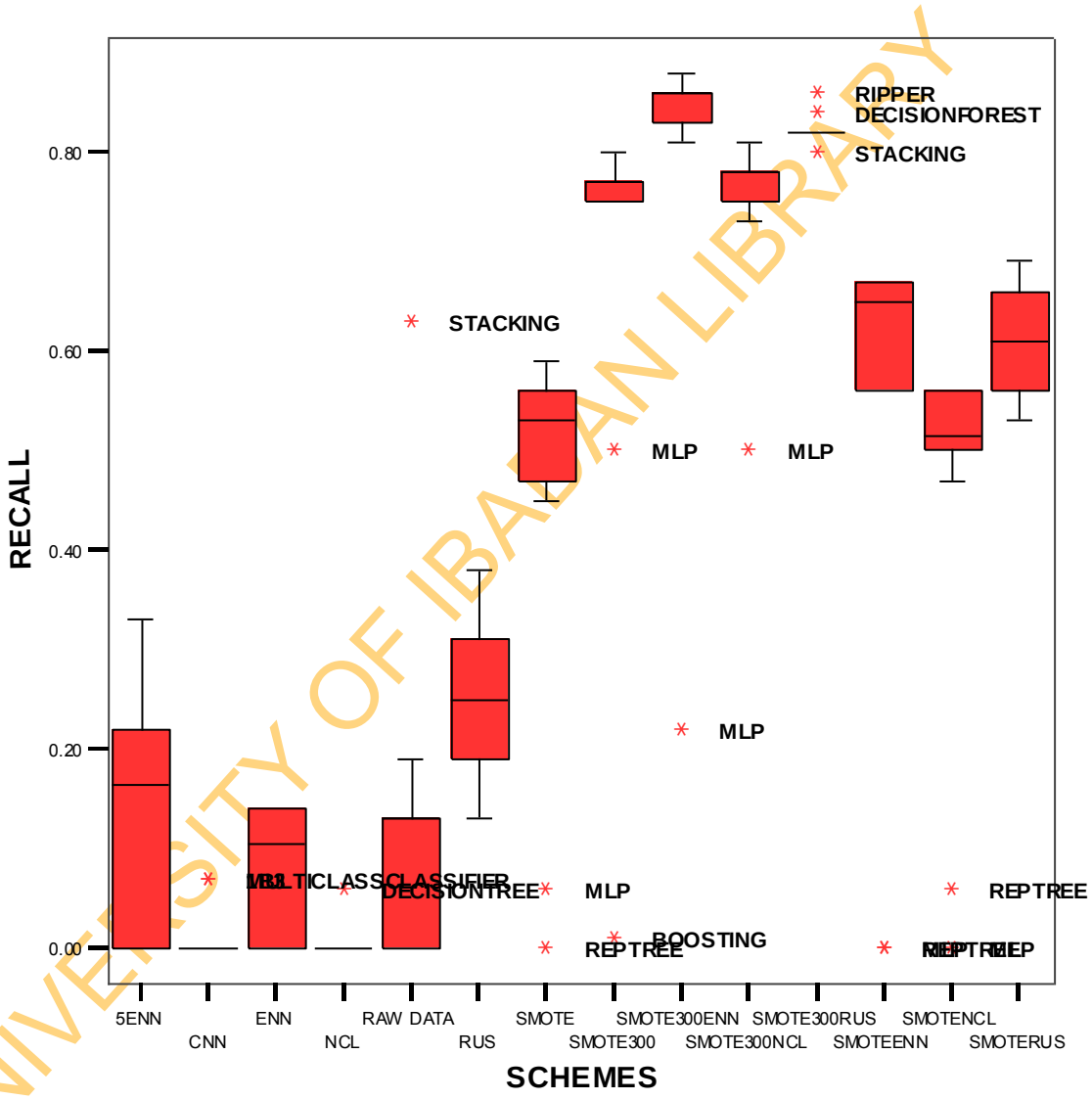


Figure 4.40: Box and whisker plots for RECALL metric for CM dataset

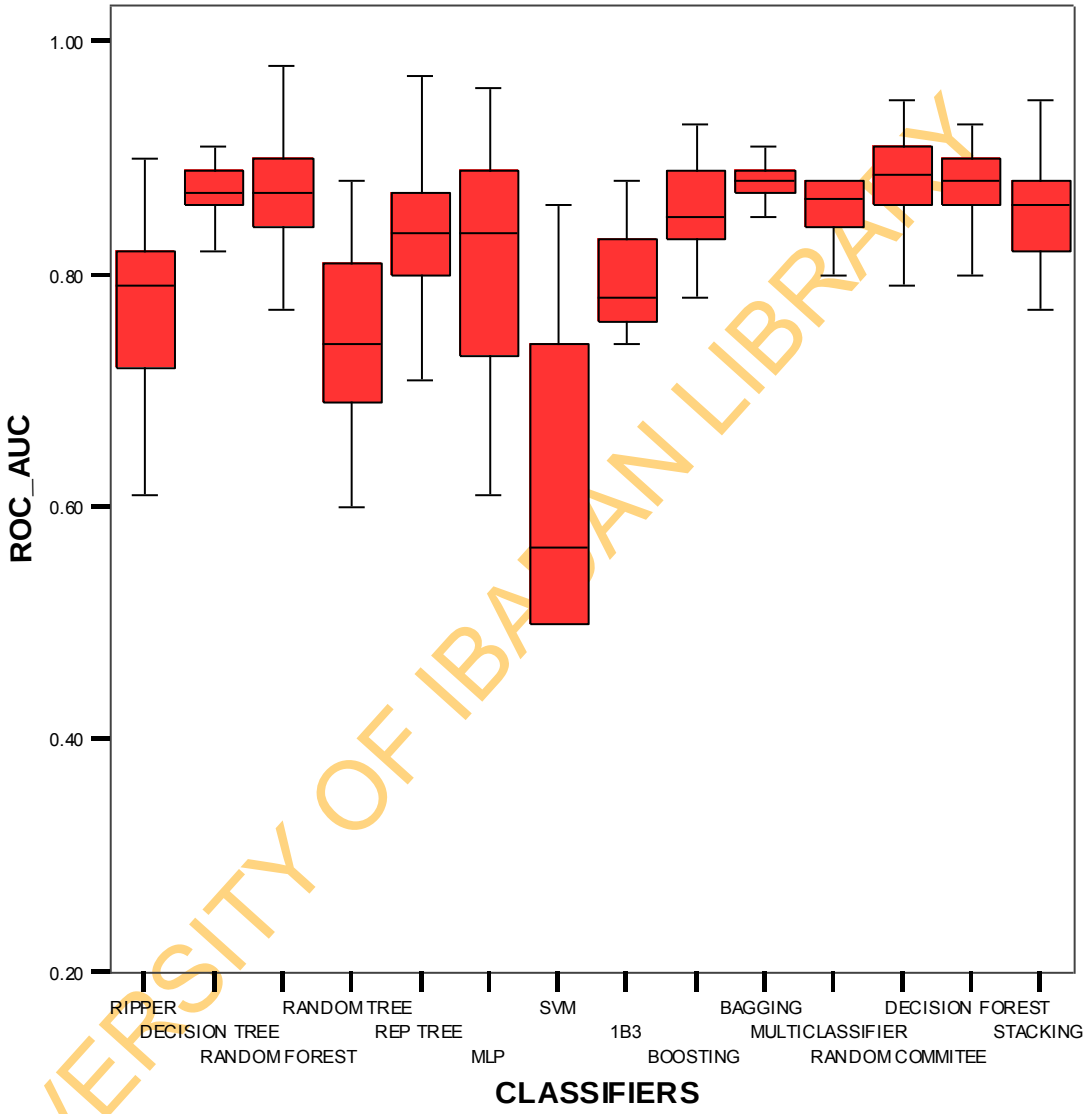


Figure 4.41: Box and whisker plots for classifiers on DM dataset

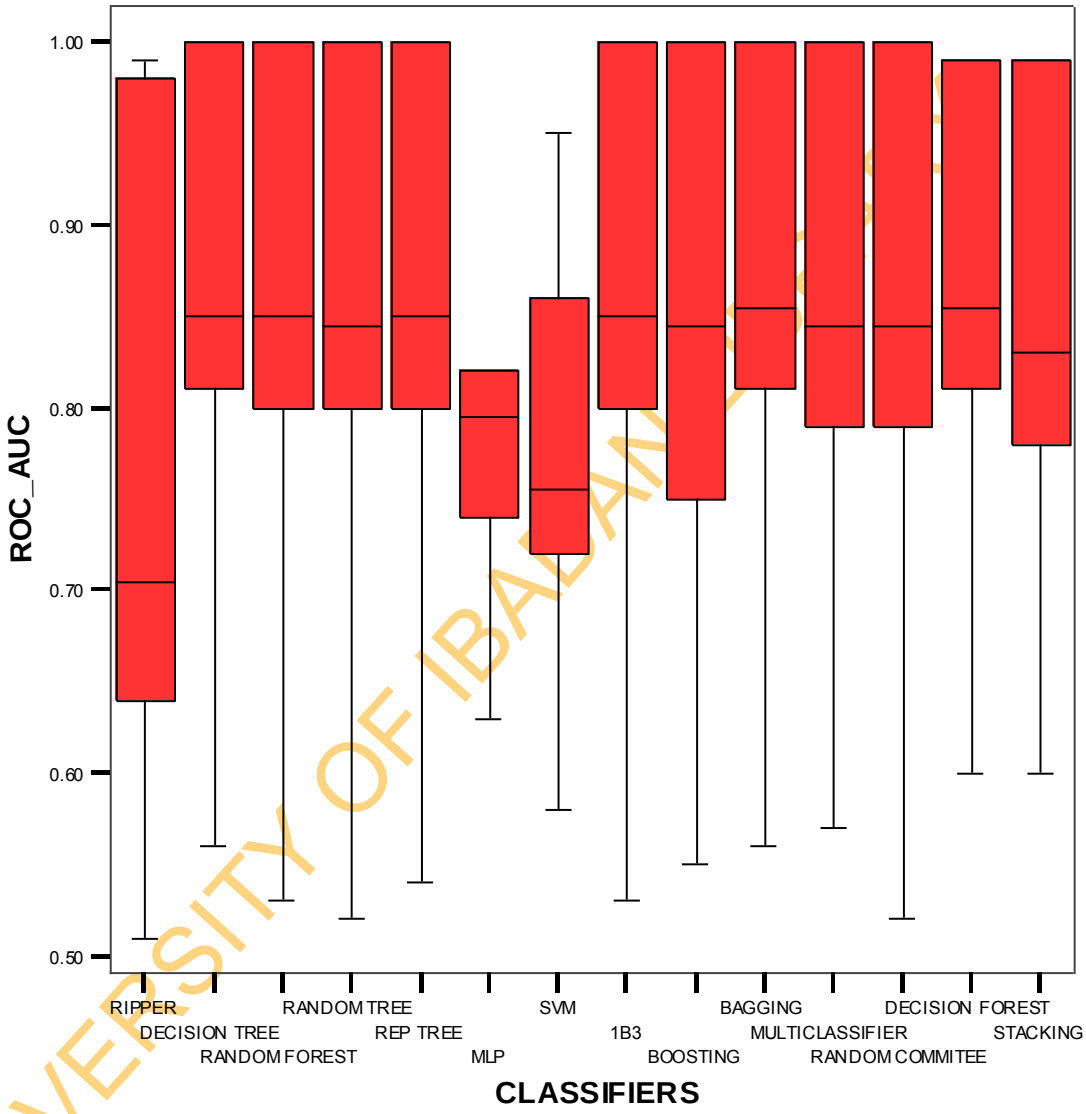


Figure 4.42: Box and whisker plots for classifiers on SSS Result dataset

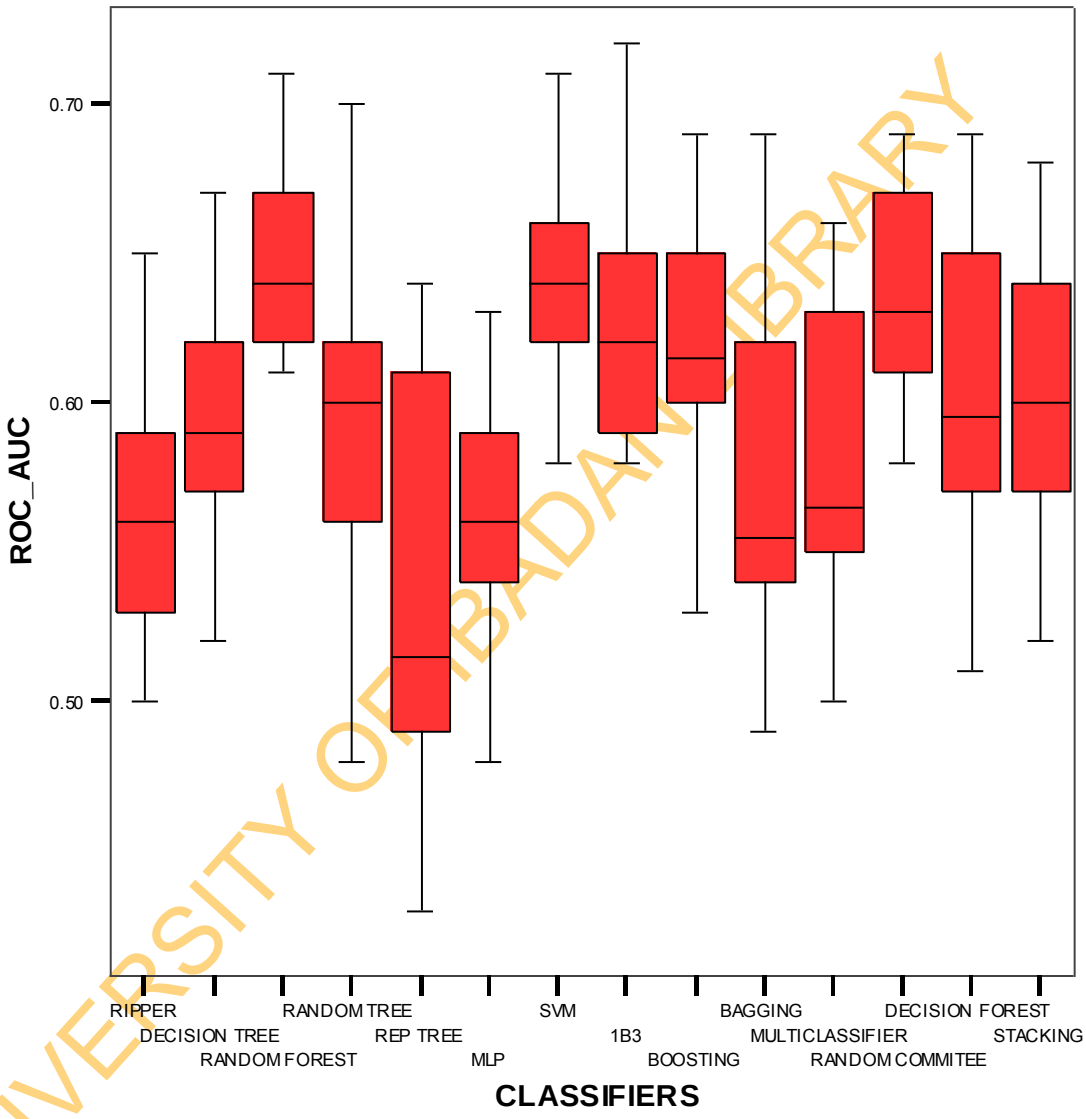


Figure 4.43: Box and whisker plots for classifiers on CM dataset

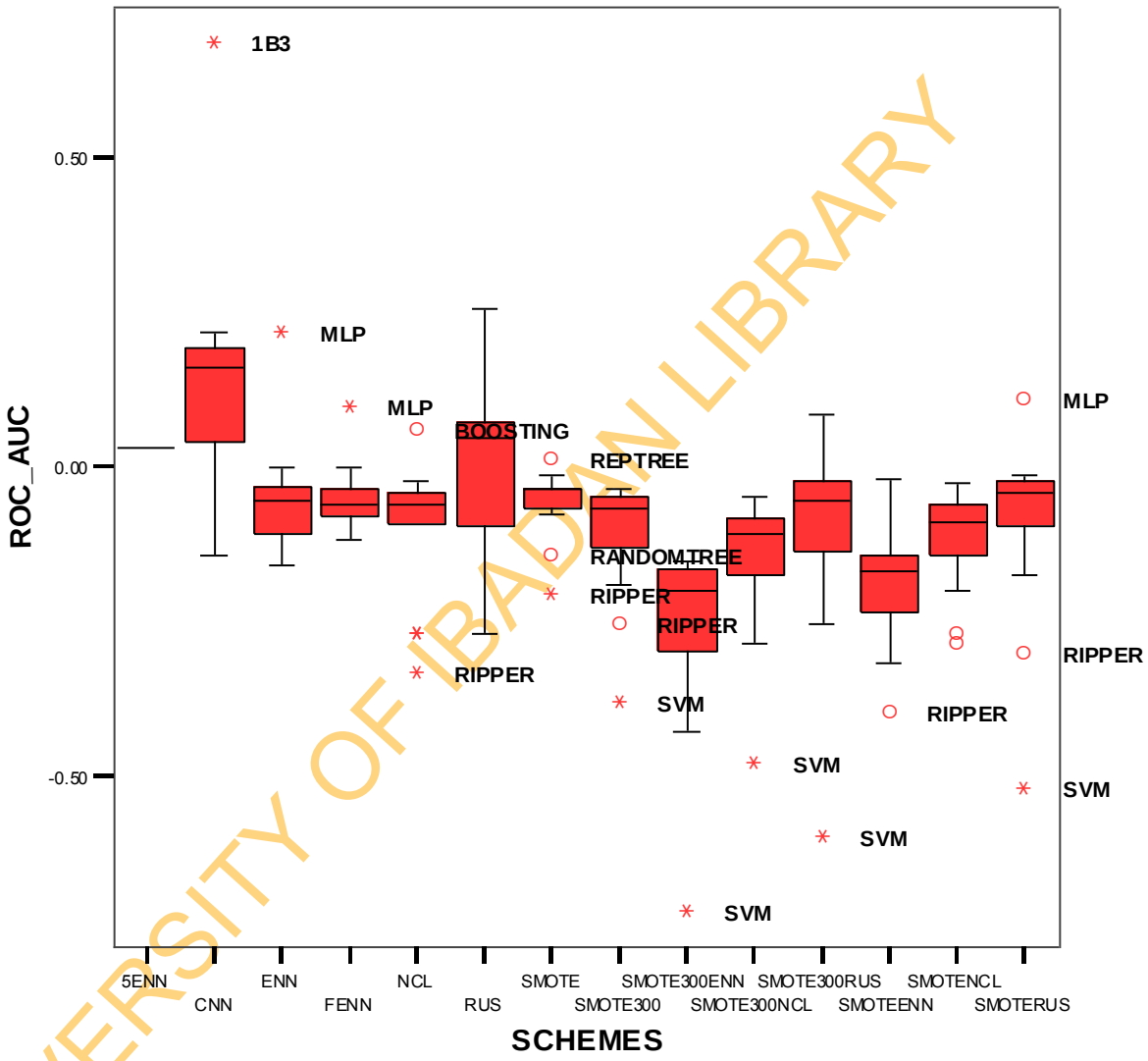


Figure 4.44: Box and whisker plots on Performance Loss/Gain on DM dataset

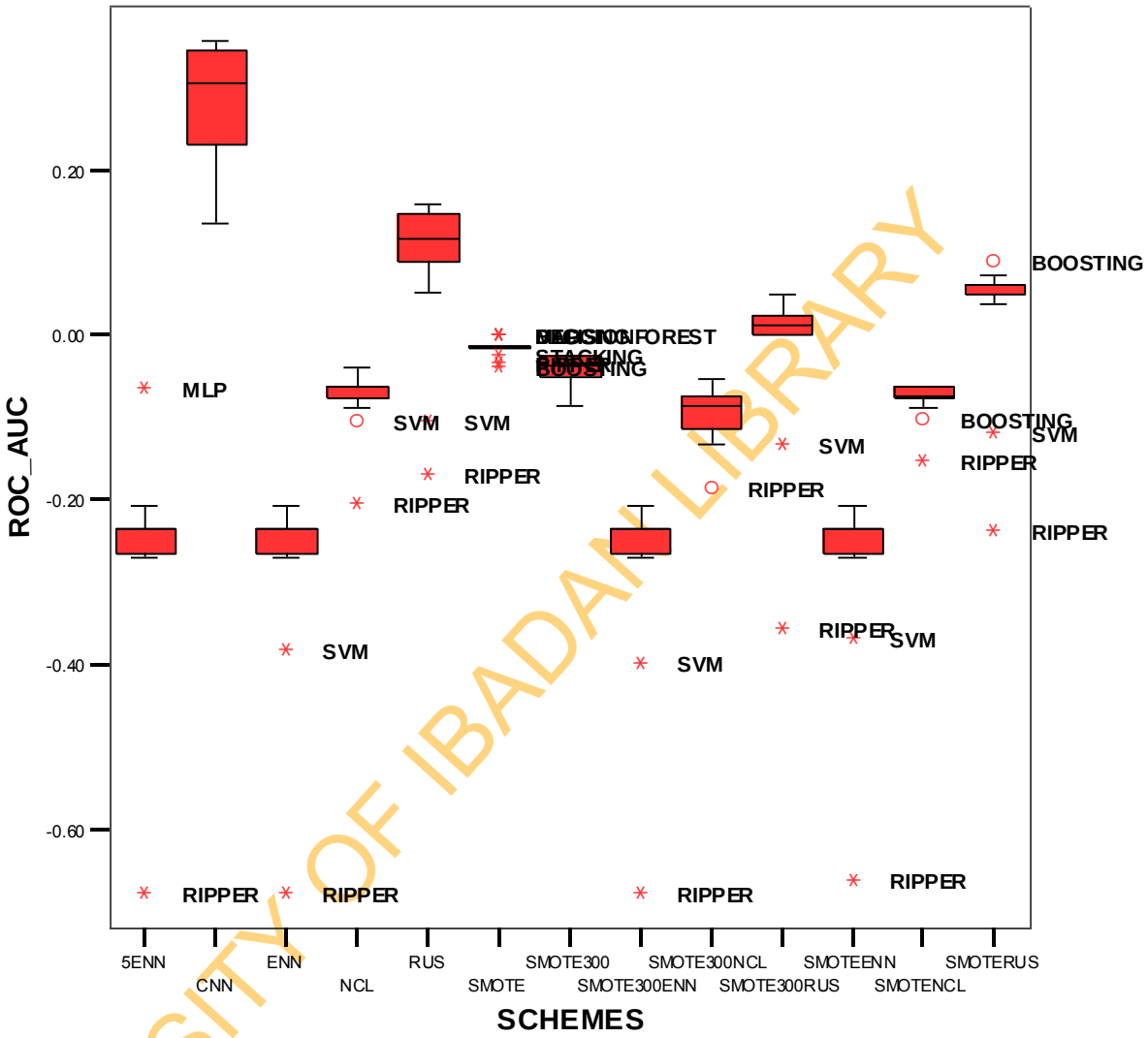


Figure 4.45: Box and whisker plots on Performance Loss/Gain on SSS Result dataset

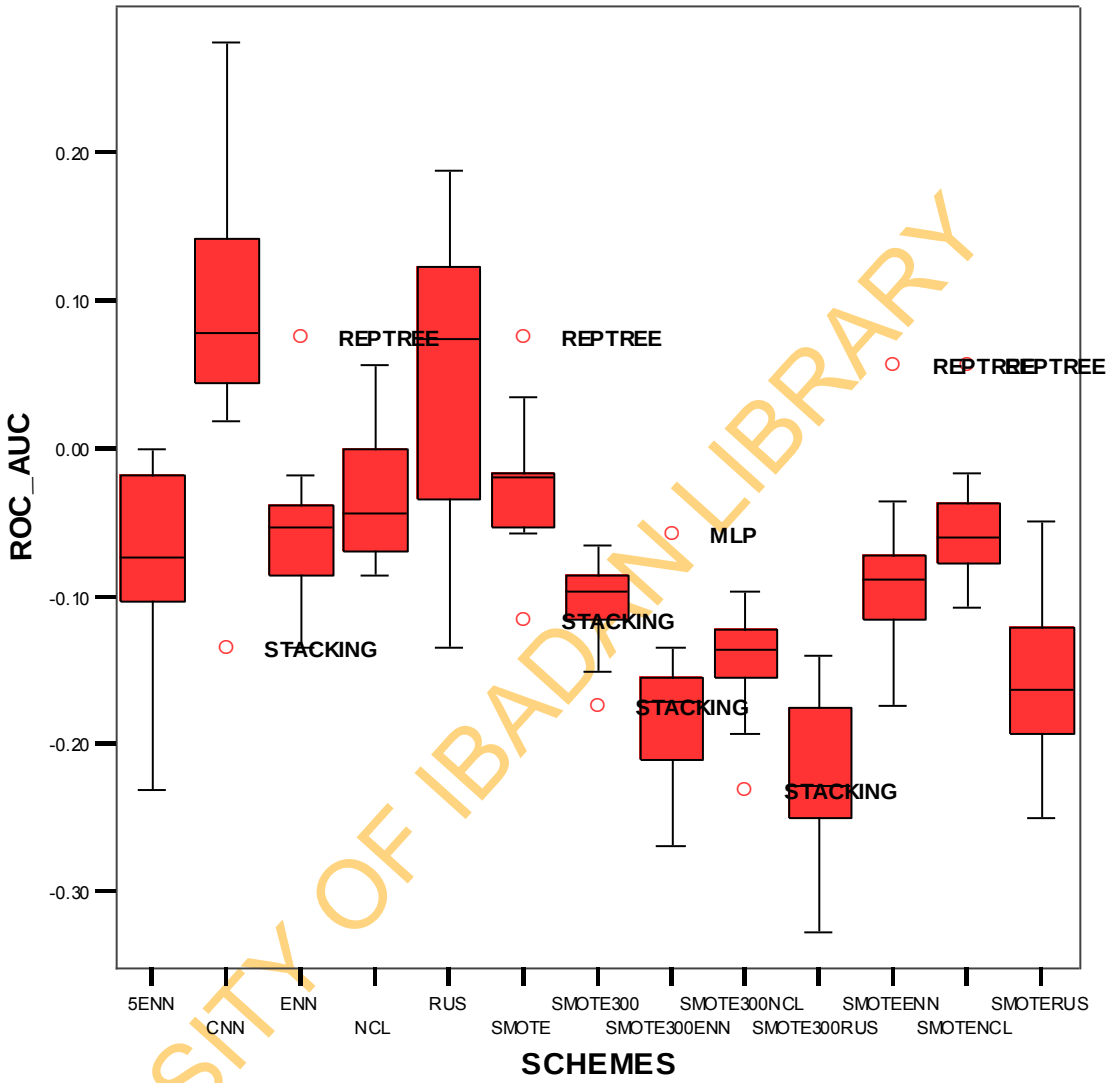


Figure 4.46: Box and whisker plots on Performance Loss/Gain on CM dataset

4.2 Remarks

In this study, five enhanced data sampling schemes were developed namely SMOTERUS, SMOTENCL, SMOTE300ENN, SMOTE300RUS and SMOTE300NCL. Eight existing data sampling schemes were also studied and implemented namely CNN, ENN, RUS, NCL, 5ENN, SMOTE, SMOTE300 and SMOTEENN. These data sampling schemes were applied to the imbalanced datasets for improved classification ability, prevention of data loss, over fitting of dataset and to allow for better detection of minority classes that are also difficult to identify. The effect of data reduction/increment on the classification ability of the RAW DATA were also studied with 14 different classifiers namely SVM, MLP, C4.4, JRIP, 1B3, REPTree, RandonTree, AdaBoost.M1, BAGGING, Stacking, Decision forest, RandomForest, RandomCommittee and MultiClass classifier. Classification results were obtained from the 13 sampled and RAW DATA datasets. These datasets were trained on 14 classifiers and were evaluated using ROC_AUC, Kappa Statistics, RECALL of the minority class, RMSE and performance loss/gain metrics. Further statistical test using both parametric and non-parametric methods such as ANOVA and Friedman Test were carried out.

4.2.1 Analysis of performance of datasets generated from existing data sampling schemes

The results presented in Tables 4.22, 4.23, 4.24 and 4.25 showed that on ROC_AUC, Kappa Statistics, RMSE and performance loss/gain metrics:

CNN: This data sampling scheme performed least across all the three datasets. One of the reasons for this could be that the scheme does not guarantee a minimal subset as an under sampling scheme (Wilson and Martinez, 2000). The reduction capability of condensation methods is normally high due to the fact that there are fewer border points than internal points but can often result in marginally poorer classification/recognition performance. From the result obtained, CNN data sampling scheme dropped over 50% of the data. This could lead to loss of information for a classifier to work with. Moreover, it is especially sensitive to noise as this data sampling scheme only removes redundant examples that are far from the decision border from the dataset thereby retaining noisy instances. This corroborates the report of Dasarathy *et al.*, (2000). Hence, this data sampling scheme is good when memory requirement (Bhatia and

Vandana, 2010) and computational advantage (Dasarathy *et al*., 2000) is the main concern.

5ENN and ENN: The performances of ENN and 5ENN (5-Nearest Neighbour) data sampling schemes were observed to be similar. This suggests that the additional complexity required to use a larger number of neigbhours than three is not warranted. This is due to the small decrease in the error rate when more than three nearest neighbour are used. This could also be due to the fact that few pre-classified samples were required to approach the asymptotic performance quite closely for ENN, many fewer samples than were required to approach the asymptotic performance for using five or more nearest neighbour (Wilson, 1972).

Since ROC_AUC, RMSE and Kappa statistics were good measure of performance for classifier and are insensitive to the imbalance distribution of the classes in datasets, the resulting 'cleaned' dataset gave higher values for the metrics. This could be attributed to the fact that 5ENN and ENN data sampling schemes eliminates erroneously labelled, common outliers. It also 'clean' the possible overlapping (border) region of the different classes, leaving smoother decision boundaries (Vazquez *et al.*, 2005). However, the minority class will still be ignored and not detected.

RUS: This data sampling scheme often performed better than CNN across all three dataset using these metrics: ROC, RMSE and Kappa statistics. The reason could be that an average of 85% and above of the majority class was removed from the total instances across all three dataset. This will affect the classifiers as they had so little information to work with. Though instances were removed randomly from the dataset to give a balanced distribution i.e. the size of all the classes were the same, the scheme gave a better recognition or increases the class bias of the minority class.

NCL: This is a variant of ENN. Its good performance could be due to the fact that it also 'cleans' the dataset before classification like its predecessor (ENN) but only the majority class. It was able to detect the minority class because it removed 10-12% of only the majority class in the dataset not touching the minority class. This advantage gave the minority class chances to be detected. Another reason is that NCL attempted

to avoid the problems caused by noise by applying the ENN algorithm that is designed for noise filtering. NCL data sampling scheme also 'cleaned' neighbourhood that misclassifies examples belonging to the minority class which is the class of interest (Laurikala, 2001).

SMOTE: This data sampling scheme increases the size of the minority class. So also will the class sub-clusters and boundary points. This scheme synthetically increases the number of the minority class which is also the class of interest. But the detection of the minority class increased and also their decision boundary as supported by (Blagus and Lusa, 2012; Dittman *et al.*, 2014).

4.2.2 Analysis of performance datasets generated from the enhanced data sampling schemes.

Among the five enhanced data sampling schemes developed, SMOTE300ENN consistently gave the best performance than the other four enhanced data sampling schemes. These enhanced data sampling schemes were based on advanced sampling. Though over sampling the minority class examples can treat the imbalance class distribution, but some other problems usually present in datasets with skewed class distributions will not be solved. Frequently, class clusters were not well defined since some majority class examples might be invading the minority class space. The opposite can also be true, since interpolating minority class examples can expand the minority class clusters, introducing artificial minority class examples too deeply in the majority class space. Inducing a classifier under such a situation can lead to over fitting. Thus, the development of the enhanced data sampling schemes.

SMOTE300ENN: SMOTE300 increased the size of the minority by 300% for better recognition. But this will not solve the problem of clusters. In order to create better-defined class clusters, ENN scheme was applied to dataset created from SMOTE300 to remove noisy, erroneous and mis-classified instances from both classes. Hence, this enhanced data sampling scheme provides a set of instances organised in relatively compact and homogeneous subgroup for better detection of both majority and minority classes and optimal classification. This data sampling scheme also solved the problem of class overlapping and sub- class clusters. The new dataset created from this scheme

will be free from noise, errors and class overlapping. The average performance gained by the use of SMOTE300ENN data sampling scheme was 24.33%. SVM and RIPPER classifiers offered the best improvement but MLP did not perform well with this scheme across all datasets.

SMOTE300RUS: This data sampling scheme performed second best on all the datasets. SMOTE300 increased the size of the minority by 300% for better recognition. RUS scheme was applied to randomly remove majority class instances to give a balanced distribution. This reduction was not 'intelligently' carried out as all majority class samples had equal chances of being removed from the dataset. The advantage of SMOTE300RUS data sampling scheme was that all class probability and the sizes of the classes were the same. This data sampling scheme further enhanced the decision region of the minority class and better detection. But the noise level is still as in RAW DATA. This scheme may not be recommended for a dataset with highly overlapped classes. The average performance gained by the use of SMOTE300RUS scheme was 12%. Also SVM and RIPPER classifier performed excellently with this data sampling scheme but did poorly with Stacking classifier across datasets.

SMOTE300NCL: SMOTE300 increased the size of the minority by 300% for better recognition. The NCL data sampling scheme was applied to remove noisy, erroneous and mis-classified instances from only the majority class instances while the size of the minority class remain the same. With the reduced dataset, it was difficult to maintain the original classification accuracy. The new dataset created from SMOTE300NCL data sampling scheme was free from noise from the majority but not from the minority class. The average performance gained by the use of this scheme was 13%. RIPPER and SVM performed excellently by improving performance with this scheme.

SMOTERUS: This scheme is similar to SMOTE300RUS. SMOTE increased the size of the minority by 100% and not 300% for better recognition than the RAW DATA. The RUS data sampling scheme was applied to SMOTE to randomly remove majority class instances to give a balanced distribution. This reduction was also not 'intelligently' carried out as all the majority class samples have equal chances of being removed from the dataset. The advantage of this data sampling scheme was that all

class probability and the sizes of the classes are the same. This data sampling scheme further enhanced the decision region of the minority class and better detection. But the noise level is still as in original. This scheme may not be recommended for a dataset with highly overlapped classes. The average performance gained by the use of this scheme was 7.3%. RIPPER, SVM and Boosting classifier offered very good improvement on performance on this data sampling scheme.

SMOTENCL: This scheme is also similar to SMOTE300NCL. SMOTE increased the size of the minority by 100% and not 300% for better recognition than the RAW DATA. The NCL data sampling scheme was applied to SMOTE to remove noisy, erroneous and mis-classified instances from only the majority class. The size of the minority class remained the same. With the reduced dataset, it was difficult to maintain the original classification accuracy. The new dataset created from the application of SMOTENCL will be free from noise from the majority but not from the minority class. The average performance gained by the use of this scheme was 8.3%. RIPPER, Random Tree and Boosting classifier greatly improved performance but REPTree performed extremely worse on this scheme across all datasets.

It was remarkable to mention that consistently, four out of the enhanced data sampling schemes were generally ranked amongst the best seven out all the data sampling schemes. SMOTE300ENN gave the best performance in all metrics used.

4.2.3 Analysis of performance of the Tuberculosis dataset

There were no errors recorded for both majority and minority class of this dataset as depicted by Table 4.1 and 4.2. The explanation for this is that there were no overlap of class region and no small disjunct of the minority class. The results obtained on all metrics in this study conformed with results obtained previously by (Prati *et al.*, 2004; Batista *et al.*, 2004; Laurikala, 2001; Japkowicz, 2003 and Weiss and Provost, 2003), where they all concluded that class imbalance alone does not seem to be the problem but when allied with overlapped classes, class disjucts (Weiss, 2003) and concept complexity (Japkowicz, 2003). This happened with the Tuberculosis dataset used in this study where although the classes were highly skewed, they were not overlapped.

The class clusters are well defined and their decision boundaries suffered no overlap. The classification performance was not suboptimal.

4.2.4 Analysis of classifiers

ROC_AUC metric was used as it measures the general ability of a classifier. Since the Area Under Curve (AUC) of ROC graph was a portion of the unit square, its value will always be between 0 and 1. Any classifier with ROC_AUC value greater than 0.5, did not make a random guess (prediction) of choosing a positive instance higher than a negative instance. So, no realistic classifier should have AUC less than 0.5 (Fawcett, 2006). For all the classifiers considered in this study, their ROC_AUC values were greater than 0.5. Though, least performances was recorded for SVM and MLP across all datasets, they were still far from random guessing. Japkowicz and Stephen, (2002) and Carvajal *et al.*, (2004) reported that the reason for deficient performance of MLP was due to the fact that minority class was inadequately weighted in networks. Bhatnagar *et al.*, (2010) and Batuwita and Palade, (2012) agreed that though SVM can handle class imbalance problem but get overwhelmed when faced with more severe class imbalance problem.

Akbani *et al.*, (2003) and Wu and Chang (2003) also indicated that SVM can be ineffective at determining class boundary when faced with class imbalance problem. The underlining reason was that as the training data gets more imbalanced, the support vector ratio between the majority class and the minority class also becomes more imbalanced. The small amount of cumulative error on the minority class instances count for very little in the trade-off between maximizing the width of the margin and minimizing the training error. SVMs simply learn to classify all instances as the majority class in order to make the margin the largest and the error the minimum.

Summarily, homogeneous ensemble performed well across all datasets. When the sample size of the minority class is extremely small, MLP will not be able to detect them for lack of information as in the case of Tuberculosis dataset.

CHAPTER FIVE

Summary, Conclusion and Recommendations

5.1 Summary

This study explored domains of imbalanced dataset. It explored datasets from different domains where the imbalanced distribution of classes gave both optimal and sub-optimal performances in classification. It also reviewed solutions both at the data and algorithm level to alleviate the class imbalance problem. It explored the various decomposition methods for transforming a multiple class problem (more than two classes) into several binary problems. The enhanced data sampling schemes were implemented to improve classification performance, removal of loss of data, overfitting of dataset and increase the RECALL of the minority class. The minority class which is also the class of interest for this study were Gestational Diabetes Mellitus (GDM) for DM dataset, PASSWAEC for SSS Result dataset, REPTB for TB dataset and NONE for CM dataset. Datasets created from both the enhanced and existing data sampling schemes were trained on 14 different types of base and ensembles classifiers. The study established that the highest percentage of errors came from the minority class.

5.2 Contribution to the study

The study established enhanced data sampling schemes that can improve classification performance, increase detection of the minority class, remove the problem of information loss and over fitting of dataset during classification on a dataset.

These enhanced data sampling schemes, particularly SMOTE300ENN were general strategies to handle class imbalance problem. The empirical evaluation on variety of imbalanced datasets were carried out to established the superiority of the enhanced data sampling schemes with better results compared to state-of-the-earth baselines.

There was also development of a theoretical taxonomy of the relationship between under-sampling schemes in class imbalance learning and their underlying data reduction techniques. The study gave a robust, statistically valid and dependable experimental work to understand the comparative strengths and weaknesses of different data sampling schemes in real world dataset.

Treatment for countering imbalances in dataset as a pre-processing stage before classification was performed. This gave room for dataset portability and classifiers re-usability.

5.3 Conclusion

From the results obtained in this study and corroborated by previous studies, the natural class distribution is often not the best distribution for training a classifier. The study revealed that the highest percentage of prediction errors came from the minority class. Experimental results on the DM, SSS Result and CM datasets showed that enhanced data sampling schemes: SMOTE300ENN, SMOTERUS, SMOTENCL, SMOTE300NCL and SMOTE300RUS increased the RECALL of the minority class across all datasets compared with the RAW DATA. The enhanced data sampling schemes provided a new approach where the combination of SMOTE + 300% and ENN in particular for datasets with few minority class examples, provided very good results in practice. SMOTE300ENN gave the best performance based its on domination on ROC_AUC, KAPPA Statistics, and RECALL of the minority class, RMSE and performance loss/gain metric across all dataset. SMOTE300ENN forces focused learning and introduced a bias towards the minority class. The reason why synthetic

minority oversampling gave improved performance was its effect on the decision regions in feature space when minority oversampling was introduced. This caused the classifier to build a larger decision region that contain nearby minority points. The data sampling scheme provided more related minority samples to learn from, to the tune of 300%, thus allowing the classifier to carve broader decision regions, leading to more coverage of the minority class. SMOTE300ENN allowed for improved identification of difficult minority classes while keeping the classification ability.

ENN and 5ENN schemes do not work well on their own for the detection of the minority class but they 'clean' datasets for optimal performance on classifiers. These two data sampling schemes had similar performances but they gave better performance when combined with SMOTE.

The TB dataset was imbalanced in nature but all the learning schemes did not have a problem classifying the dataset. The reason for this was due to the fact that all the classes were linearly separable and the regions of the classes were well defined. Therefore, in this case, applying class imbalance data sampling schemes could lead to performance degradation.

5.4 Recommendations

The enhanced data sampling schemes can be applied to highly slewed datasets with a very small number of minority classes as they perform well in the detection of the minority class.

The complete experimental set up depict that the enhanced data sampling schemes can be practically applied to real time dataset.

There are several areas to be considered for further study. These includes:

a. To add Cost Sensitive Learning (CSL) to the enhanced schemes
b. To compare data reduction technique such as Principal Components Analysis (PCA) to data sampling schemes such as RUS, ENN, and NCL.
c. This study used Decision tree as base learners for both homogeneous and heterogeneous ensembles. However, future work could explore the use of bsse learners such as ANN, SVM and RIPPER to create ensembles and application of Repeated ENN (RENN) to datasets which can then be compared with ENN and 5ENN.

References

Agboola, L. A. (2010). Development of A Decision Tree Based Model For Analysing Student Performance in Two National Certificate Examinations (WAEC and NECO). A dissertation submitted for Master's degree in Computer Science in the University of Ibadan, Ibadan, Nigeria.

Aha, D. W., Kibler, D. and Albert, M. K. (1991). Instance-Based Learning Algorithms. *Machine Learning*. Vol. 6, pp. 37-66.

Akbani, R., Kwek, S. and Japkowicz, N. (2003). Applying support vector machines to imbalanced datasets. In the proceedings of European conference on Machine Learning. Pp. 39-50.

Asha, T., Natarajan, S. and Murthy, K. N. B. (2011). A Study of Associative Classifiers with Different Rule Evaluation Measures for Tuberculosis Prediction. *International Journal on Computer science and its Applications (IJCA)* Special Issue on "*Artificial Intelligence Techniques - Novel Approaches and Practical Applications*" AIT.

Asha, T., Natarajan, S. and Murthy, K. N. B. (2012). Data Mining Techniques in the Diagnosis of Tuberculosis, Understanding Tuberculosis - *Global Experiences and Innovative Approaches to the Diagnosis*, Dr. Pere-Joan Cardona (Ed.) Chapter 16. Retrieved from http://www.intechopen.com/books/understanding-tuberculosis-global-experiences-and-innovative-approaches-to-the-iagnosis/data-mining-techniques-in-the-diagnosis-of-tuberculosis downloaded in February, 2014

Awad, M., Khan, L., Thuraisingham, B. and Wang, L. (2009). Design and Implementation of Data Mining Tools. Auerbach Publications, Taylor and Francis Group, 6000 Broken Sound Parkway NW, Suite 300. Boca Raton. pp. 25-26

Awokola, E. O. (2010). Rapid Diagnosis and Treatment of Diabetes Mellitus Using Artificial Neural Networks. Dissertation submitted for Master's degree to the Department of Computer Science, University of Ibadan, Ibadan Oyo State, Nigeria.

Barandela, R, Sanchez, J. S. and Valdovinos, R. M. (2003a). New Application of Ensembles of Classifiers. *Pattern Analytical Application*. Vol. 6, pp. 245–256.

Barandela, R., Sanchez, J. S., Garcia, V. and Rangel, E. (2003b). Strategies for learning in class imbalance problem. T*he Journal of the Pattern Recognition society*, Elsevier. Vol. 36, pp., 849- 851.

Bathia, N and Vandana. A. (2010). Survey of Nearest Neighbour Techniques. *International Journal of Computer Science and Information Security*, (IJCSIS). Vol. 8 No. 2, pp. 302-305.

Banaco, P. C. (2011). Box and Whisker Plots for Local climate Datasets: Interpretation and Creation using Excel 2007/2010. *Eastern Region Technical Attachment*. No. 2011-01

Batista, G.E.A.P.A., Prati, R. C. and Ronard, M. C. (2004). A study of the Behaviour of Several Methods for Balancing Machine Learning Training Data, *SIGKDD Explorations*, Vol. 6, No. 1, pp. 20- 29.

Bekkar, M. and Alitouche, T. A. (2013): Imbalanced data learning approaches review. *International Journal of Data Mining & Knowledge Management Process (IJDKP)*. Vol. 3, No. 4, pp. 15-33.

Bhatnagar, V., Bhardwaj, M. and Mahabal, A. (2010). Comparing SVM Ensembles for Imbalanced Datasets. In the proceedings of 10th *IEEE International conference on Intelligent Systems Design and Applications (ISDA).* Pp. 651-657.

Blagus, R. and Lusa, L. (2012). Evaluation of SMOTE for high- dimensional class imbalance micro- array data. *In proceedings of the 11th Conference in Machine Learning and Applications (ICMLA, 2012)*. Vol. 2, pp. 89-94

Blake, C. L., and Merz, C. J. (1998). UCI repository of machine learning databases, Department of Information and Computer sciences, University of California, Irvine, CA. www.ics.uci.edu/_mlearn/MLRepository.html.

Boontarika, L. and Maythapolnun, A. (2011). An Empirical Study of Multiclass with Class Imbalance Problems, *Proceedigs of Business And Information (BAI)*, Vol. 8, No. 1.

Bouckaert, R. R., Frank, E., Hall, M. A., Holmes, G., Pfahringer, B., Reutemann, P. and Witten, I. A. (2010). WEKA — Experiences with a Java Open-Source Project. *Journal of Machine Learning Research*, Vol. 11, Pp. 2533-2541.

Breinman, L. (2001). Random Forests. *Machine Learning*. Vol. 45, No. 1, pp. 5-32.

Breinman, L. (1996). Bagging Predictors. *Machine Learnin*g. Vol. 24, No. 2, pp. 123–140.

Carletta, Jean. (1996). Assessing agreement on classification tasks: The kappa statistic. *Computational Linguistics*. Vol. 22, No. 2, pp. 249–254.

Carvajal, K., Chacon, M., Mery, D. and Acuna, G. (2004). Neural Network method for failure detection with skewed class distribution. Journal of the British Institute of Non-Destructive Testing. Vol. 46, No. 7, pp 399-402

Chandrahasan, R. K., Angeline, C. Y. and Usha, R. S. (2011). An Empirical Comparison of Boosting and Bagging Algorithms. *In the Proceedings of International Journal of Computer Science and Information Security*. Vol. 9, No. 11, pp. 147-152

Chawla, N. and Sylvester, J. (2007). Exploiting Diversity in Ensembles: Improving the Performance on Unbalanced Datasets. *In the proceedings of 7th International Workshop Multiple Classifier Systems (MCS).* Pp. 397-406

Chawla, N. V., Bowyer, K. W., Hall, L.O. and Kegelmeyer, P.W. (2002). SMOTE: Synthetic Minority Over- sampling Technique, *Journal of Artificial Intelligence research,* Vol. 16, pp. 321- 357.

Chawla, N.V., Japkowicz, N. and Kolcz, A. (2004). Editorial: Special Issue on Learning from Imbalance Data Sets, *SIGKDD Explorations.* Vol. 6, No. 1, pp. 1- 6.

Chawla, N. V., Lazarevic, A., Hall, L. O. and Bowyer, K. W. (2003). SMOTEBoost: Improving Prediction of the Minority Class in Boosting. In the proceedings of the 7th European on Principles and Practice of Knowledge Discovery in Databases (PKDD). pp. 107-119

Chawla, N. V. (2003). C4.5 and Imbalanced Data sets: Investigating the effects of sampling method, probabilistic estimate, and decision tree structure. Workshop on Learning from Imbalanced Datasets II, ICML.

Chawla, N. V. (2005). Data mining for imbalanced datasets: An overview. *The Data mining and knowledge discovery handbook.* Chapter 40, pp. 853-867.

Chen, Y., Zhou, X. and Huang, T. (2001). One class SVM for learning in Image Retrieval. *In the Proceedings of IEEE International Conference on Image Processing (ICIP' 01 oral).*

Cohen, W. C. (1995). Fast Effective Rule Induction. *In the Proceedings of the 12th International Conference on Machine Learning.* Pp. 115-123.

Cover, T. M and Hart, P. E. (1967). Nearest Neighbour Pattern Classification. *IEEE Transaction on Information Theory*. Vol. 13, pp. 21-27.

Dasarathy, B. V., Sanchez, J. S. and Townsend, S. (2000). Nearest Neighbour Editing and Condensing Tools–Synergy Exploitation. *Pattern Analysis and Applications*. Vol. 3, pp. 19 – 30.

Davis, J. and Goadric, M. (2006). The relationship between Precision –Recall and ROC curves, *Proc. 23rd International Conference on Machine learning, Pittsburgh*, PA. pp. 233-240.

Dietterich, T. G. (2000). Ensemble Methods in Machine Learning. *Proceedings of 1st International Workshop Multiple Classifier Systems* (*LNCS)* Vol. 1857. Pp. 1-15

Dietteric, T. G. and Bakiri, G. (1995). Solving multiclass learning problem via error- correcting output codes. *Journal of Artificial Intelligence Research*. Vol. 2, pp. 263-286

Ding, Z. (2011). Diversified Ensemble Classifier for Highly imbalanced Data Learning and their application in Bioinformatics, Ph. D thesis, *College of Arts and science, Department of Computer Science, Georgia State University*, 2011. Retrieved from Http://digitalarchive.gsu.edu/cs_diss/60 in February 2012.

Domigos, P. (1999). MetaCost; A general method for making classifier cost- sensitive. *In the proceedings of the 5th ACM SIGKDD International Conference on Artificial Intelligence.* pp. 155-164.

Dittman, D.J., Taghi, M. K., Randall, W. and Amri, N. (2014). *In the proceedings of the 27th International Florida Artificial Intelligence Research Society Conference.* Pp. 268-271. www.aaai.org

Elkan, C. (2001). The Foundations of Cost-Sensitive Learning. *In the Proceedings of the 17th International Conference on Artificial Intelligence*. pp. 973–978.

Estabrooks A., Jo, T. and Japkowicz, N. (2004). A Multiple Resampling Method for Learning from Imbalanced Data Sets. *Computational Intelligence*. Vol. 20, No. 1, pp. 18-36.

Fawcett, T. (2003). ROC Graphs: Notes and Practical Considerations for Researchers. Technical Report HPL- 2003- 4, *HP Labs.*

Fawcett, T. (2006). An Introduction to ROC analysis. *Pattern Recognition Letters*. Vol. 27, pp. 861- 874.

Fattahi, S., Othman, Z. and Othman, A. Z. (2015). New approach with ensemble method to address class imbalance problem. Journal of theoretical and applied information technology. Vol. 72, No. 1, pp. 23-33. Retrieved from www.iatit.org on 21st July, 2015

Fernandez, A., Garcial, S. and Hererra, F. (2011). Addressing the Classification with Imbalanced Data: Open Problems and New Challenges on Class Distribution. E. Corchado, M. Kurzy´nski, M. Wo´zniak (Eds.): *HAIS, Part I, LNAI* 6678. Pp. 1–10.

Freund, Y. and Schapire, R. E. (1995). A decision- theoretic generalisation of on- line learning and an application to boosting. *Technical Report, AT and T Bell Laboratories, Murray Hill, NJ*.

Freund, Y. and Schapire, R. E. (1996). Experiment with a new boosting algorithm. *In Proceeding of the 13th International Conference on Machine Learning. Pp.* 148-146.

Freund, Y. and Schapire, R. E. (1999). A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence*. Vol. 14, No. 5, pp. 771-780.

Galar M., Fernandez A., Barrenechea E., Bustince H. and Herrera F. (2012). A Review on Ensembles for the Class Imbalance Problem: Bagging-, Boosting and Hybrid Approaches. IEEE Transactions on Systems, Man and Cybernetics - Part C: *Applications and Reviews*. Vol. 42, N0. 4, pp. 463-484.

Garcia S., Joacquin, J., Cano, J. R and Herrera, F. (2012). Prototype Selection for Nearest Neigbhor Classification: Taxonomy and Empirical Study. *IEEE Transactions on Pattern Analysis and Machine intelligence. Vol.* 34, No. 3, pp. 417–435.

Gates, G. W. (1972). Reduced Nearest Neighbour Rule. *IEEE Trans Information Theory*, Vol. 18, No. 3, pp. 431-433.

Georgescu, R., Berger, C. R., Willet, P., Azam, M. and Ghoshal S. (2010). Comparison of Data Reduction Techniques Based on the Performance of SVM- type Classifiers. *Aerospace Conference, IEEE*. Big Sky, MT. Pp. 1–9

Ghanem, A., Venkatesh. S. and West, G. (2010). Multi-Class Pattern Classification in Imbalanced Data, *International Conference on Pattern Recognition*. Pp. 2881- 2884

Gilpin, S. A. and Dunlavy, D. M. (2009). Relationships between Accuracy and Diversity in Heterogeneous Ensemble Classifiers. Technical Report SAND-2009-6940C, Sandia National Laboratories, Albuquerque, NM and Livermore, CA.

Gu, J. (2007). Random Forest Based Imbalance Data Cleaning and Classification. *PAKDD'07*. Retrieved from Http://lamda.nju.edu.cn/conf/pakdd07/dmc07/reports/P251.pdf in March, 2011.

Guo, X., Yin, Y., Dong, C., Yang, G. and Zhou, G. (2008). On the Class Imbalance Problem, *Fourth International Conference on Natural Computation*, IEEE Computer Society. pp.192 – 201

Habibi, S., Ahmadi, M. and Alizadeh, S. (2015). TYPE 2 Diabetes Mellitus screening and Risk Factors using Decision tree: Results of Data Mining. Global Journal of Health Sciences. Vol. 7, No. 5. Pp. 304-310. Retrieved from www.ccsenet.org/gjhs in June, 2015

Han, H., Wang, W. Y., and Mao, B. H. (2005). Borderline-SMOTE: A new over-sampling method in imbalanced data sets learning. *In International Conference on Intelligent Computing (ICIC'05)*. (LNCS) Vol. 3644, pp. 878–887.

Han, J. and Kamber, M. (2001). Data Mining: Concepts and Techniques. *Morgan Kaufmann Publishers.* ISBN: 1-55860-489-8.

Hart, P. E. (1968). The Condensed Nearest Neighbour Rule, *IEEE Transformation Information Theory*. Vol. 16, pp. 515-516.

Hastie, T. and Tibshirani, R. (1998). Classification by Pairwise Coupling. In: *Advances in Neural Information Processing Systems.*

He, H., Bai, Y., Garcia, E. A. and Li, S. (2008). ADASYN: Adaptive Synthetic Sampling Approach for Imbalanced Learning. *In the Proceedings of International Joint Conference on Neural Networks (IJCNN 2008)*. pp. 1322- 1328.

Hido, S. and Kashima, H. (2008). Roughly Balanced Bagging for Imbalanced Data. *In Proceedings of SIAM Conference on Data Mining (SDM2008)*, Atlanta, Georgia, USA, April, 2008.

Ho, T. K. (1998). The Random Subspace Method for Constructing Decision Forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 20, No. 8, pp. 832 - 844.

Hoens, R. T. and Nitesh V. C. 2010. Generating Diverse Ensembles to counter the problem of Class Imbalance. Advances in Knowledge Discovery and Data Mining. *14th Pacific-Asia conference, PAKDD, Part II, LNA* Vol. 6119, Ch. 46, pp. 488- 199.

Hoens, R. T. (2012). Living in an unbalanced world. Ph. D Thesis, University of Notre Dame, Indiana.

Hoens, T. R. and Chawla, N. V. (2010). Generating Diverse Ensembles to Counter the Problem of Class Imbalance. *PAKDD* Vol. 2, No. 10, pp.488 – 499.

Hoens, T. R., Qian, Q., Chawla, N. V. and Zhou, Z. (2012). Building Decision Trees for the Multi-class Imbalance Problem. *PAKDD* Vol. 1, pp. 122–134.

Hulse, J. V., Khoshgoftaar, T. M. and Napolitano, A. (2007). Experimental Perspectives on Learning from Imbalanced Data. In the Proceeding of the 24th International Conference on Machine Learning (ICML), ACM. pp. 935-942.

Jankowski, N. and Grochowski, M. (2004). Comparison of Instances Selection Algorithms I. Algorithm survey. L. Rutkowski et al. (Eds.), *In the Proceedings of International Conference of Artificial Intelligence and Soft Computing* (ICAISC). Pp. 598 - 603.

Japkowicz, N., Myers, C. and Gluck, M. (1995). A Novelty detection approach to classification. *In the Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI – '95).* Pp.518 – 523.

Japkowicz, N. (2003). Class Imbalances: Are we focusing on the right issue? *In the Proceeding of ICML workshop: Learning with Imbalanced Data Sets II*, pp. 17–23

Japkowicz, N. and Stephen, S. (2002). The Class Imbalance problem: A Systemic Study *IDA Journal* 6(5). Pp. 429 – 449.

Jo, T., and Japkowicz, N. (2004). Class imbalances versus small disjuncts. *SIGKDD Explorations*. Vol. 6, pp. 40–49.

Johnson, R. A., Chawla, N. V. and Hellman, J. J. (2012). Species Distribution Modelling and Prediction: A Class Imbalance Problem. NASA Conference on Intelligent Data Understanding (CIDU), Boulder, CO.

Juszczak, P. and Duin, R. P. W. (2003). Uncertainty sampling methods for one- class classifiers. Workshop on Learning from Imbalanced Datasets II, *ICML*, Washington DC.

Karakoulas, G. and Taylor, J. S. (1999). Optimizing classifiers for imbalanced training sets. In *Advances in Neural Information Processing Systems*. pp.253 - 259

Keerthi, S. S., Shevade, S. K., Bhattacharyya, C. and Murthy, K. R. K. (2001). Improvements to Platt's SMO Algorithm for SVM Classifier Design. *Neural Computation.* Vol. 13, No. 3, pp. 637-649.

Kolcz, A., Chowdhury, A. and Alspector, J. (2003). Data duplication: an imbalance problem? Workshop on Learning from Imbalanced Datasets II, *ICML, Washington DC.*

Kubat, M. and Matwin, S. (1997). Addressing the Curse Imbalanced Training Sets: One- Sided Selection. *In the proceedings of the fourteenth conference on machine learning.* Pp. 179- 186.

Kuncheva, L. I. and Whitaker C. J. (2003). Measure of Diversity in Classifier Ensembles and Their Relationship with the Ensemble Accuracy. *Machine Learning*. Vol. 51, pp. 181-207.

Landis, J.R. and Koch, G.G. (1977). "The measurement of observer agreement for categorical data". *Biometrics.* Vol. 33, No. 1, pp. 159–174.

Lawson, L., Zhang J., Gomgnimbou M. K., Abdurrahman ST, Le Moullec S, Mohamed F, Uzoewulu GN, Sogaolu OM, Goh K. S, Emenyonu N, Refregier G, Cuevas L. E and Sola C. (2012). A Molecular Epidemiological and Genetic Diversity Study of Tuberculosis in Ibadan, Nnewi and Abuja, Nigeria. *PLos ONE*. Vol. 7, No. 6. e38409. Mukrousov Igor (Ed).

Laurikala, J. (2001). Improving Identification of Difficult Small Classes by Balancing Class Distribution. In the Proceedings of the International Conference on artificial intelligence in medicine in Europe. Vol. 2101, No. 8, pp. 63-66.

Lessmann, S. (2004). Solving imbalanced classification problems with support vector machines. *In International Conference on Artificial Intelligence.* pp. 214–220.

Lewis, D. and Gale, W. (1994). Training Text Classifiers by Uncertainty Sampling. *In the Proceedings of the 17th ACM SIGIR Conference on Research and Development in Information Retrieval.* Pp. 3-12

Lin, Y., Lee, Y., and Wahba, G. (2002). Support vector machines for classification in nonstandard situations. *Machine Learning*. Vol. *46*, 191–202.

Ling, X. C. and Sheng, V. S. (2008). Cost-Sensitive Learning and the Class Imbalance Problem. *Encyclopedia of Machine Learning*. C. Sammut (Ed.). Springer.

Liu, X. and Zhou, Z. (2006). The Influence of Class Imbalance on Cost-Sensitive Learning: An Empirical Study. *The 6th International Conference on Data Mining* (ICDM'06). pp. 970 – 974.

Liu, X., Wu, J. and Zhou, Z. (2008). Exploratory Under- Sampling for Class-Imbalance Learning. IEEE Transaction on Systems, Man and Cybernetics –Part B.

Liu, Y., Chawla, N. V., Harper, M. P., Shriberg, E. and Stolcke, A. (2006). A study in machine learning from imbalanced data for sentence boundary detection in speech. *Elsevier Journal of Computer Speech and Language*. Vol. 20, pp. 468–494.

Lopez, V., Fernandez, A., Garcia, Palade, V. and Herrera, F. (2013). An insight into classification with imbalanced data: Empirical results and current trends on using data intrinsic characteristics. *Information Sciences*. Vol. 250, pp. 113-141.

Manevitz, L. M. and Yousef, M. (2001). One–Class SVMs for Document Classification. *Journal of Machine Learning Research*. Vol. 2, pp. 139-154.

McCarthy, K., Zabar, B. and Weiss, G. (2005). Does Cost–Sensitive learning Beat Sampling for Classifying Rare Classes? *UBDM '05.* Pp. 69-77.

Miloud-Aouidate, A. and Baba-Ali, A. R. (2011). Survey of Nearest Neighbour Condensing Techniques. *International Journal of Advanced Computer Science and Applications*. Vol. 2, No. 11, pp. 59 – 63.

Nagabhushanam D, Naresh N, Raghunath A, and Praveen K. K. (2013). Prediction of Tuberculosis Using Data Mining Techniques on Indian Patient's Data. *In International Journal of Computer Science and Technology (*IJCST). Vol. 4, No. 4, pp. 262-265.

Nguyen, H. G. (2011). Machine Learning with Informative Samples for Large and Imbalanced Datasets, Ph. D Thesis, *University of Wollongong*, Australia.

Nguyen, H. G., Bouzerdoum, A. and Phung, S. L. (2009). Learning Pattern Classification tasks with imbalanced data sets. *Pattern Recognition*. Chapter 10, pp. 193-208.

Ou, G., Murphy, Y. L., Foldcamp, L. (2004). MultiClass Pattern Classification Using Neural Networks. *Proc. of the 17th International Conference on pattern Recognition.* Vol. 4, pp. 585- 588.

Olatayo, T. O., Amusa, N. A., Aladesida, A. A. and Odunbaku, O. A. (2011). Understanding Biometrics: A manual of statistics methods for biostatistics and agricultural sciences. Chapter 8. Bisibest prints and publishing. ISBN: 978-978-80248-82. pp. 140-161

Paramanto, B., Munro, P. W. and Doyle, H. R. (1996). Improving Committee diagnosis with resampling techniques. In Touretzky *et al*., (Eds.) *Advances in Neural Information Processing Systems*. Vol. 8, pp. 882–888.

Pearson, R. K., Gonye, G. E. and Schwaber, J. S. (2003). Imbalance Clustering of Microarray Time-Series. Workshop on Learning from Imbalanced Datasets II, *ICML*, Washington DC.

Platt, J. (1998). Fast Training of Support Vector Machines using Sequential Minimal Optimization. In B. Schoelkopf and C. Burges and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning.*

Prati, R. C., Batista, G. E. A. P. A., and Monard, M. C. (2004). Class Imbalance versus Class Overlapping: an Analysis of a Learning System Behavior. *In MICAI*, (LNAI). Vol. 2972, pp. 312-321.

Quinonero C. J., Sugiyama, M., Schwaighofer, A. and Lawrence, N.D. (2009). Dataset Shift in *Machine Learning*. The MIT Press, Cambridge.

Rahman, M. M. and Davis, D. N. (2013). Addressing the class imbalance problem in medical datasets. *International Journal of Machine Learning and Computing.* Vol. 3, No 2, pp. 224-228

Rahman, S. Z. and Raju, S. V. P. (2014). Enhanced Classification to Counter the Problem of Cluster Disjuncts. *International Journal of Computer Trends and Technology (IJCTT).* Vol. 18, No 5, pp. 217-223. ISSN: 2231-5381. Retrieved from http://www.ijcttjournal.org on in February 2015

Raskutti, B. and Kowalczyk, A. (2004). Extreme Re- balancing for SVMs: A case study. *In the proceedings of European Conference on Machine learning, ACM SIGKDD Explorations Newsletter*. Vol. 6, No. 1, Pp. 60-69.

Ritters, G. L., Wooddruff, H. B., Lowry, S. R. and Isenhour, T. L. (1975). An Algorithm for a Selective Nearest Neigbhor Decision Rule. *IEEE Transactions on Information Theory*. Vol. 21, No. 6, pp. 665 – 669.

Rodriguez, J. J., Kuncheva, L. I. and Alonso, C. J. (2006). Rotation Forest: A New Classifier Ensemble Method. *IEEE Transaction on Pattern Analysis and Machine Intelligence.* Vol. 28, No. 10, pp. 1619–1630.

Batuwita, R. and Palade, V. (2012). Class imbalance learning methods for support vector machines. In the book titled 'In Imbalanced Learning: Foundations, Algorithms and Applications'. Chapter 6. Haibo H. and Yunqian, M. (Eds) Publisher: John Wiley and Sons, Inc.

Salzberg, C. (1991). A Nearest Hyperrectangle Learning Method. *Machine Learning*. Vol. 6, pp. 277–309.

Sanchez, J. S. (2004). High training set size reduction by space partitioning and prototype abstraction. *Pattern Recognition*. Vol. 37, pp. 1561-1564.

Sarwar N, Gao P, Seshasai SR, Gobin R, Kaptoge S, Di Angelantonio E, Ingelsson E, Lawlor DA, Selvin E, Stampfer M, Stehouwer CD, Lewington S, Pennells L, Thompson A, Sattar N, White IR, Ray KK, Danesh J. (2010). Diabetes mellitus, fasting blood glucose concentration, and risk of vascular disease: a collaborative meta-analysis of 102 prospective studies. *Lancet*. Vol. 375, pp. 2215-2222.

Schapire, R. E. (1997). Using output codes to boost multi class learning problems. *In the Proceeding of the 14th International Conference on Machine Learning*. Pp. 313-321.

Seiffert, C., Khoshgoftaar, T. M., Hulse, J. V. and Napolitano, A. (2010). RUSBoost: A Hybrid Approach to Alleviating Class Imbalance. *IEEE Transactions on Systems, Man, and Cybernetics—Part A: Systems and Humans.* Vol. 40, No. 1, pp. 185–197.

Stefanowski, J. and Wilk, S. (2008). Selective pre-processing of imbalanced data for improving classification performance. 10th International Conference in Data Warehousing and Knowledge Discovery (DaWaK2008) (*LNCS).* Vol. 5182, pp. 283-292.

Stanfill, C. and Waltz, D. (1986). Toward memory–based reasoning. Communication of the *ACM.* Vol. 29, pp. 1213 – 1228.

Singhi, S. K. and Liu, H. (2005). Error–Sensitive Grading for Model Combination. *In Proceedings of 16th European Conference on Machine Learning*. Pp. 724 735.

SPSS Inc. Released 2007. SPSS for Windows, Version 16.0. Chicago, SPSS Inc

Sun Y. (2007). Cost- Sensitive Boosting for Classification of Imbalanced Data. Ph. D Dissertation, Dept. of Electrical and Computer Engineering, *University of Waterloo,* Waterloo, Ontario, Canada.

Sun, Y. Kamel, M. S. and Wang, Y. (2006). Boosting for Learning Multiple Classes with Imbalanced Class Distribution, IEEE *Proc. of the Sixth International Conference on Data Mining (ICDM'*). Vol. 9, pp. 7695-2701.

Thai-Nghe, N., Andre, B., Lars, S. (2009). Improving Academic Performance Prediction by Dealing with Class Imbalance, *Proc. 9th IEEE International Conference on Intelligent Systems Design and Applications IEEE Computer Society, (ISDA).* Pp. 878-883

Thai- Nghe, N., Thanh- Nghi, D. and Lars, T. (2010). Learning Optimal Threshold on Resampling Data to Deal with Class Imbalance. *In the Proceedings Of the 8th IEEE International conference on Computing and Communication Technologies: Research, Innovation and Vision for future RIFV*. Pp.71-76

Thai- Nghe, N., Gantner, Z. and Schmidt- Thieme, L. (2011). A New Evaluation Measure for Learning from Imbalanced Data. *In the Proceeding of the IEEE International Journal Conference on Neural Networks (IJCNN).*pp. 537-542

Tomek I. (1976). Two modification of *CNN*. *IEEE Transactions on Systems, Man and Communications, SMC.* Vol. 6, pp. 769- 772.

Vapnik, V. and Lerner, A. (1963). Pattern recognition using generalised portrait method. *Automation and Remote Control.* Vol. 24, pp. 774-780.

Vazquez, F., Sanchez, J. S. and Pla, F. (2005). Pattern Recognition and Image Analysis," *2nd Iberian Conference ibPRIA* '05, Part II. LNCS 3523, Pp. 35–42.

Vegard, E. (2010). Machine Learning for Network Based Intrusion Detection. Ph. D thesis, *Bounemouth University*, 2010.

Veropoulos, K., Campbell, C., and Cristianini, N. (1999). Controlling the sensitivity of support vector machines. *Proceedings of the International Joint Conference on Artificial Intelligence*. Pp. 55–60.

Visa, S. and Ralescu, A. (2005). Issues in Mining Imbalanced Data Sets - A Review Paper. *In the proceedings of the 16th Midwest Artificial Intelligence and Cognitive Science Conference*, Dayton. Pp 67 - 73

Wang, S. (2011). Ensemble Diversity for Class Imbalance Learning. Ph. D thesis, School of Computer Science, College of Engineering and Physical Sciences, *The University of Birmingham.*

Wang, S., Tang, K. and Yao, X. (2009). Diversity Exploration and Negative Correlation Learning on Imbalanced Data Sets. *In the Proceedings of International Joint Conference on Neural Networks (*IJ*CNN*). Pp. 3259 – 3266.

Wasikowski, M., Chen, X.W. (2010). Combating the small sample class imbalance problem using feature selection. *IEEE Transactions on Knowledge and Data Engineering.* Vol. 22, No. 10, pp. 1388–1400.

Weiss, G.M. and Provost, F.J. (2003). Learning when training data are costly: The effect of class distribution on tree induction. *Journal of Artificial Intelligence Research*. Vol. 19, pp. 315–354.

Weiss, G. M. (2003). The effect of small disjuncts and class distribution on decision tree learning, Ph. D. Dissertation, Deprtment of Computer Science, Rutgers University, New Jersey.

Wilson, D. L. (1972). Asymptotic Properties of Nearest Neighbour Rules Using Edited Data. *IEEE Transactions on Systems, Man, and Cybernetics*. Vol. 2, No. 3, pp. 408–421.

Wilson, D. R. and Martinez, T. R. (1997). Improved Heterogeneous Distance Functions. *Journal of Artificial Intelligence Research*. Vol. 6, pp. 1-34.

Wilson, D. R. and Martinez, T. R. (1997b). Bias and the probability of generalisation. In the proceedings of the *International Conference on Intelligent Information Systems* (IIS'97). Pp. 108-114.

Wilson, D. R. and Martinez, T. R. (1997c). Instance Prunning Techniques. In Fisher ed. *Machine Learning, Proceedings of the fourteenth International Conference.* Pp. 404–411.

Wilson, D. R. and Martinez, T. R. (2000). An Integrated Instance–Based Learning Algorithm. *Computational Intelligence*. Vol. 16, No. 1, pp. 1-28.

Wilson, D. R. and Martinez, T. R. (2000b). Reduction Techniques for Instance–Based Learning Algorithms. *Machine Learning*, Vol. 38, No. 3, pp. 257–286.

Witten, I. H., Frank E. and Hall M. A. (2011). Data Mining: *Practical Machine Learning Tools and Techniques.* San Franscisco, California: Morgan Kaufmann. 3rd Edition.

Wolpert, D. H. (1992). Stacked generalization. *Neural Networks*. Vol. 5, No. 2, pp. 241-259.

Wu, G. and Chang, E. Y. (2003). Class-Boundary Alignment for Imbalanced Dataset Learning. *In the proceedings of ICML Workshop on Learning from Imbalanced Datasets II,* Washington DC.

www.ctec.ufal.br/professor/crfj/.../Model%20evaluation%20methods.doc. Retrieved in September, 2015.

Xu-Ying, T., Jianxin, W. and Zhi-Hua, Z. (2009). Exploratory Under-sampling for Class Imbalance Learning. *IEEE Transactions on Systems, Man and Cybernetics*. Vol. 39, No. 2, pp. 539–550.

Yang, Y. and Ma, G. (2010). Ensemble- based Active Learning for Classification Problem. *J. Biomedical and Engineering.* Vol. 3, pp. 1021- 1028. Retrieved from Http:/www. Scrip.org/journal/jbise in July 2013.

Yang, Q. and Wu, X. (2005). 10 Challenging Problems in Data mining, *International Journal of Information Technology and Decision making.* Vol. 5, No. 4, pp. 597-604.

Yen, S. J. and Lee, Y. S. (2009). Cluster-based under-sampling approaches for imbalanced data distributions. Expert Systems with Applications. Vol. 36, No. 3, pp. 5718-5727.

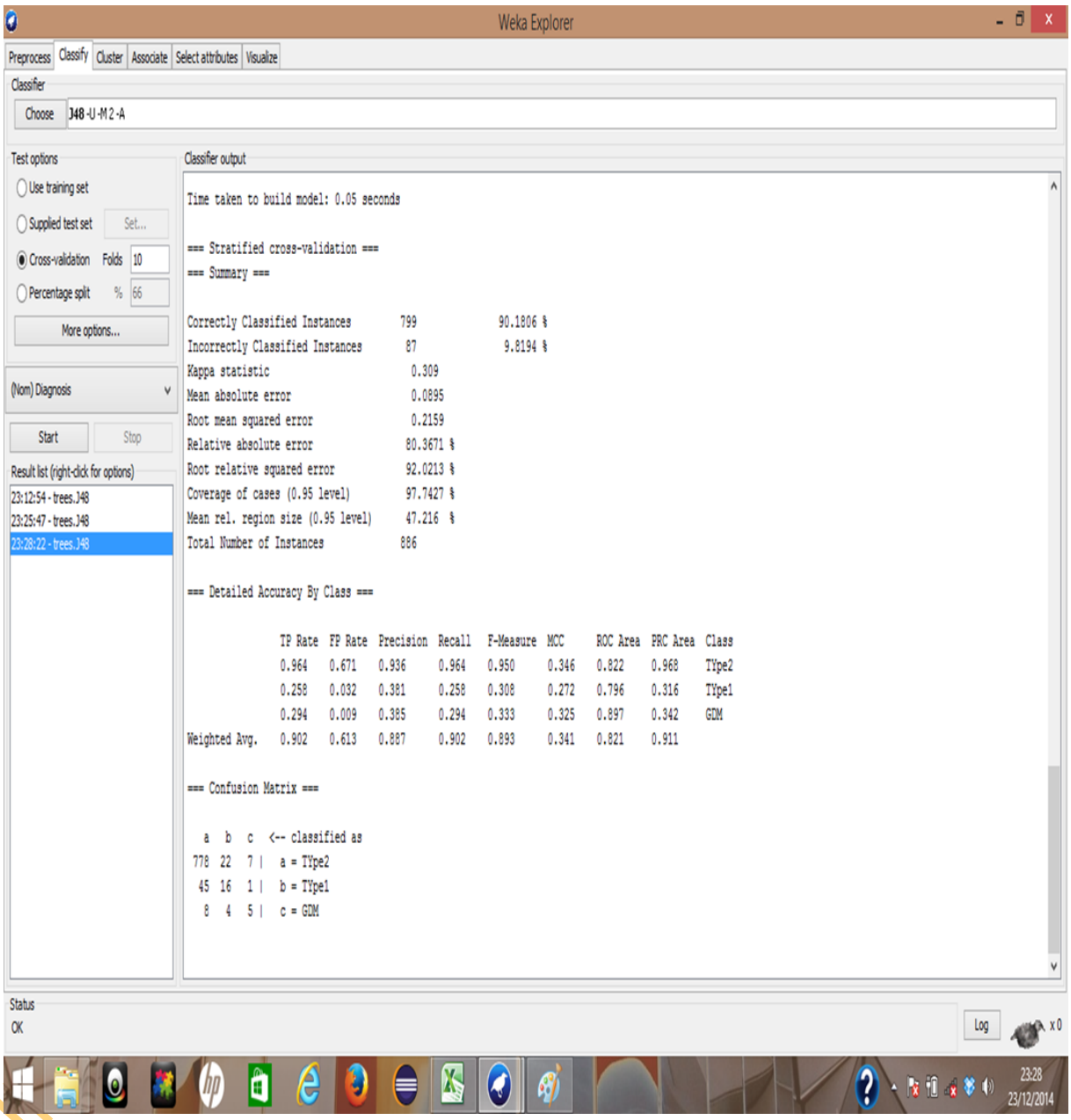
Zadrozny, B., and Elkan, C. (2001). Learning and Making Decisions When Costs and Probabilities are Both Unknown. *In proceedings of the 7th International Conference on Knowledge discovery and data mining, ACM SIGKDD.* pp. 204–213.

Zhang, J. and Mani, I. (2003). *K*NN approach to Unbalanced Data Distribution: A case study involving Information Extraction. *In the proceedings of the ICML Workshop on Learning from Imbalanced Datasets II*, Washington DC.

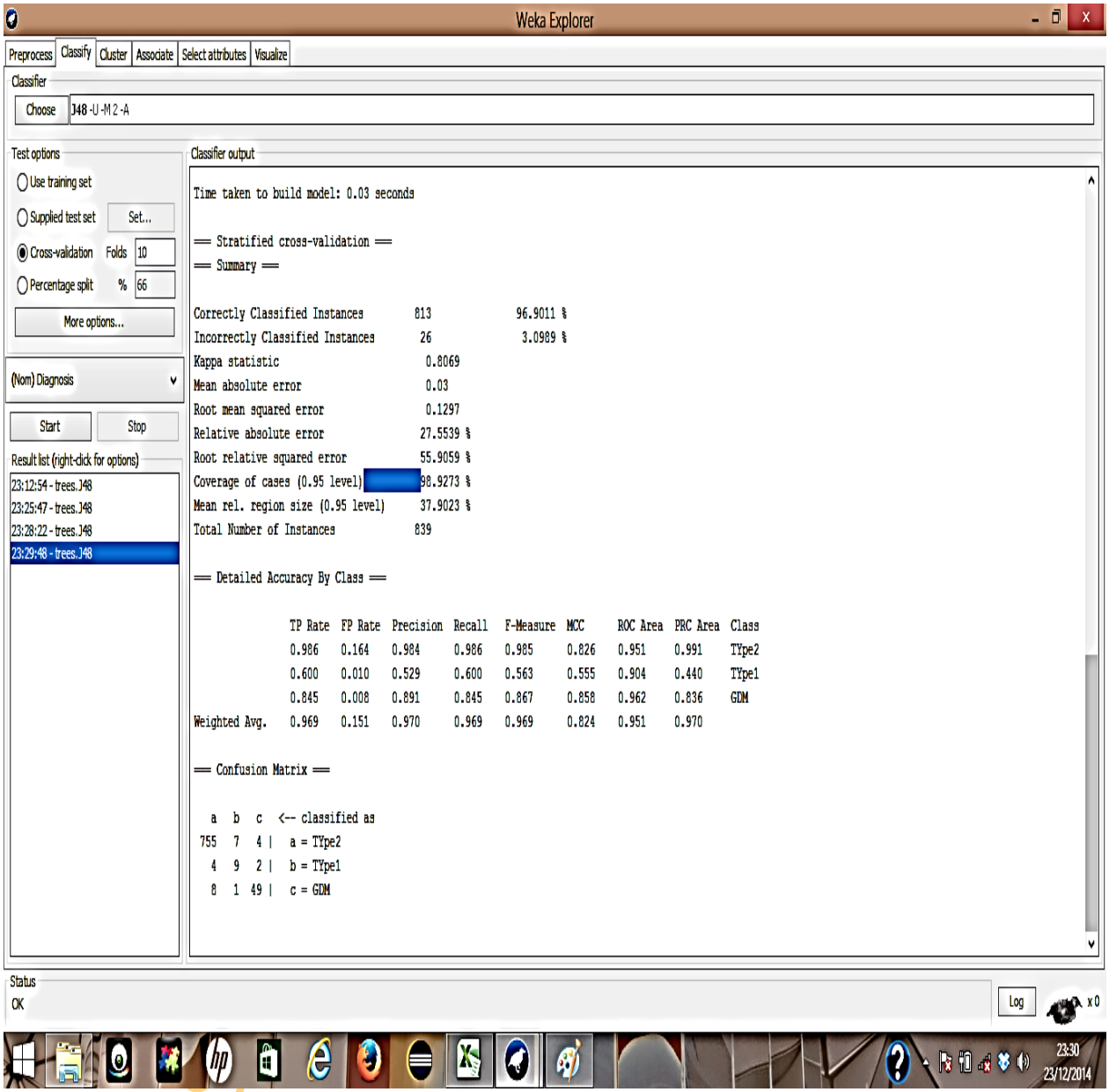
Zhou, X. and Tuck, D. P. (2007). MSVM-RFE: Extension of SVM-RFE for multiclass gene selection on DNA microarray data, Ed. David Rocke, *Oxford journals, BIOINFORAMTICS*. Vol. 23, No. 9, pp. 1106-1114.

APPENDIX A

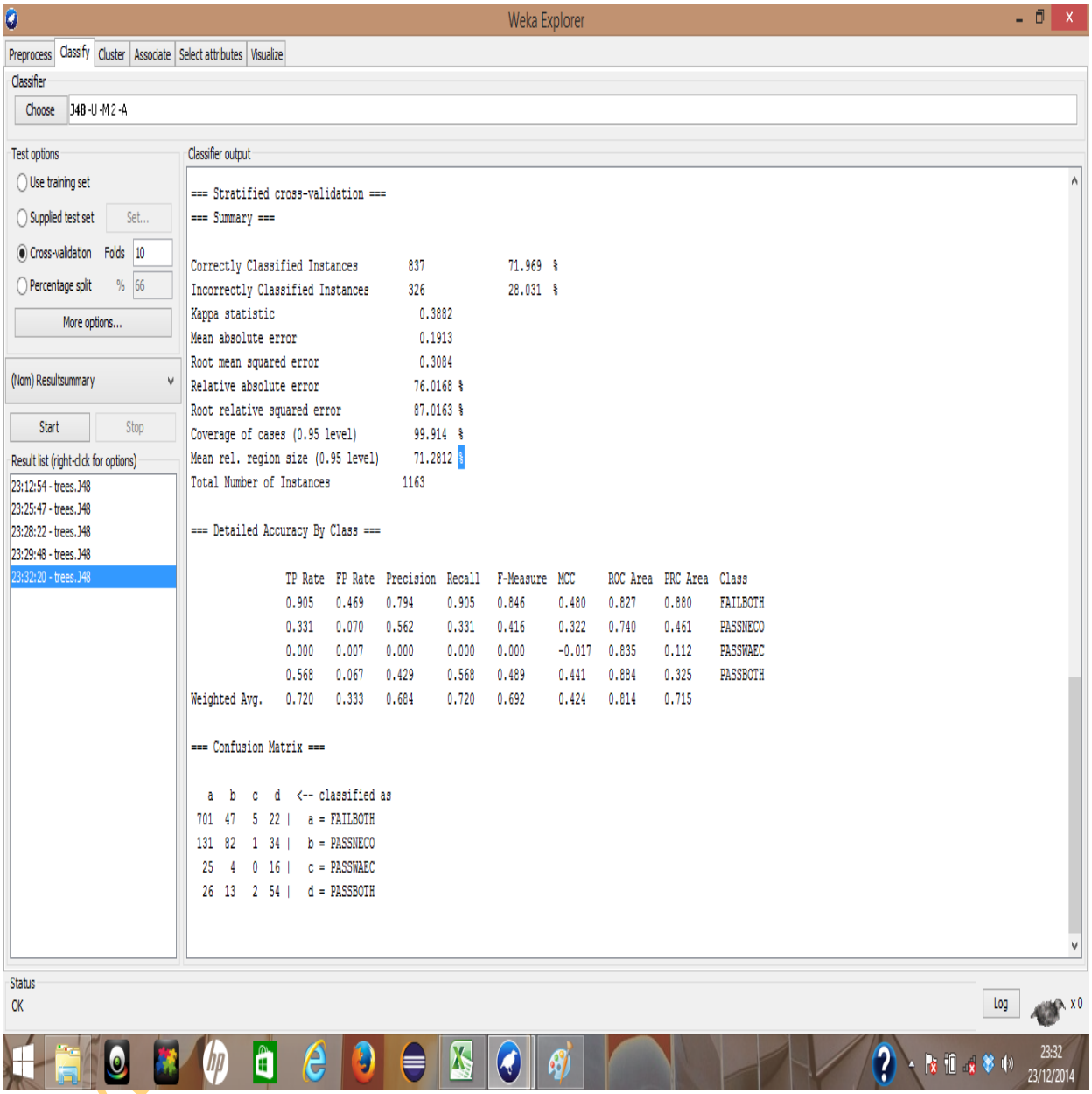
The screenshots of the predictions of datasets on classification algorithm



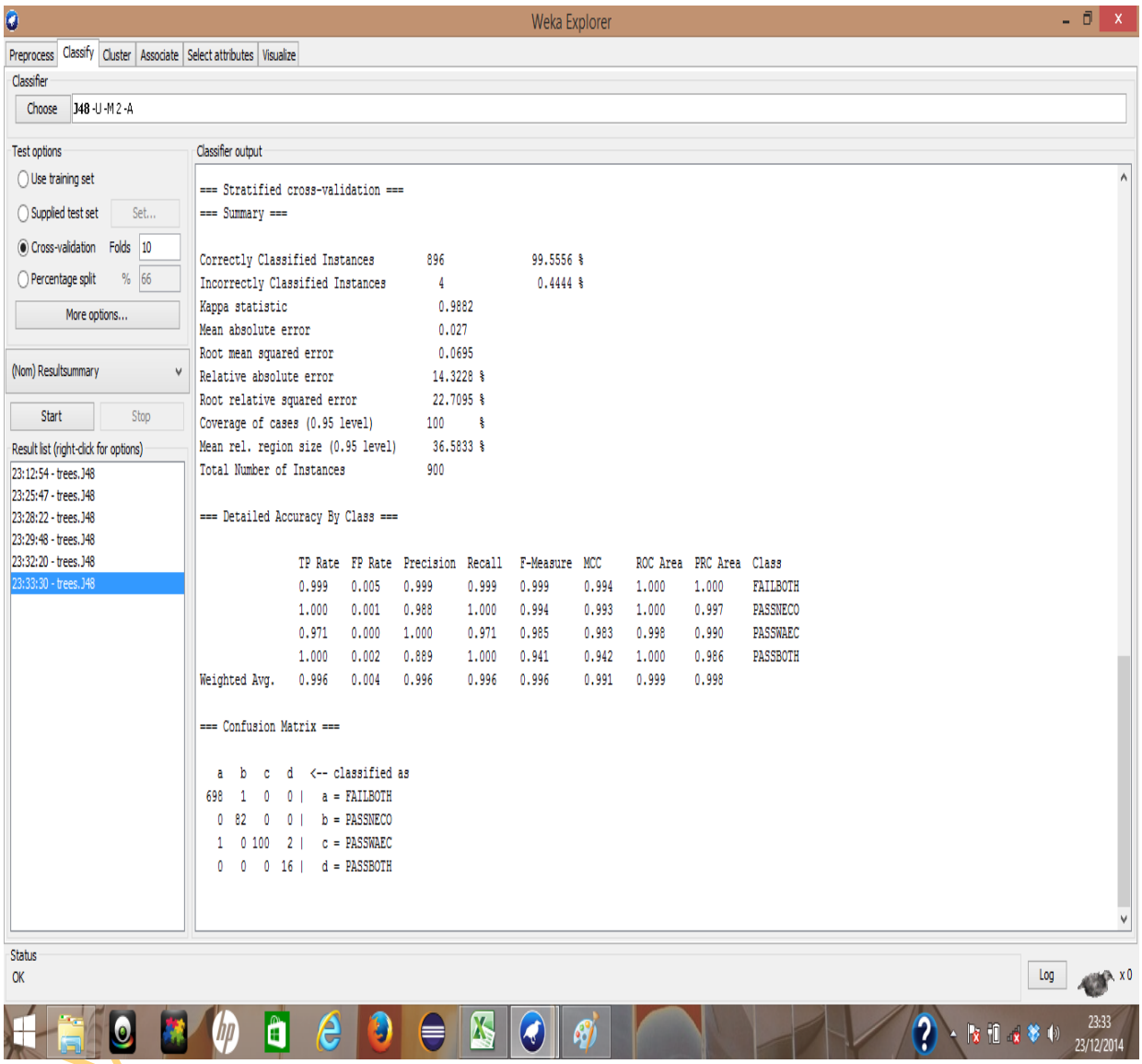
Screenshot of prediction of RAW DATA of Diabetes Mellitus dataset trained on Decision Tree Algorithm



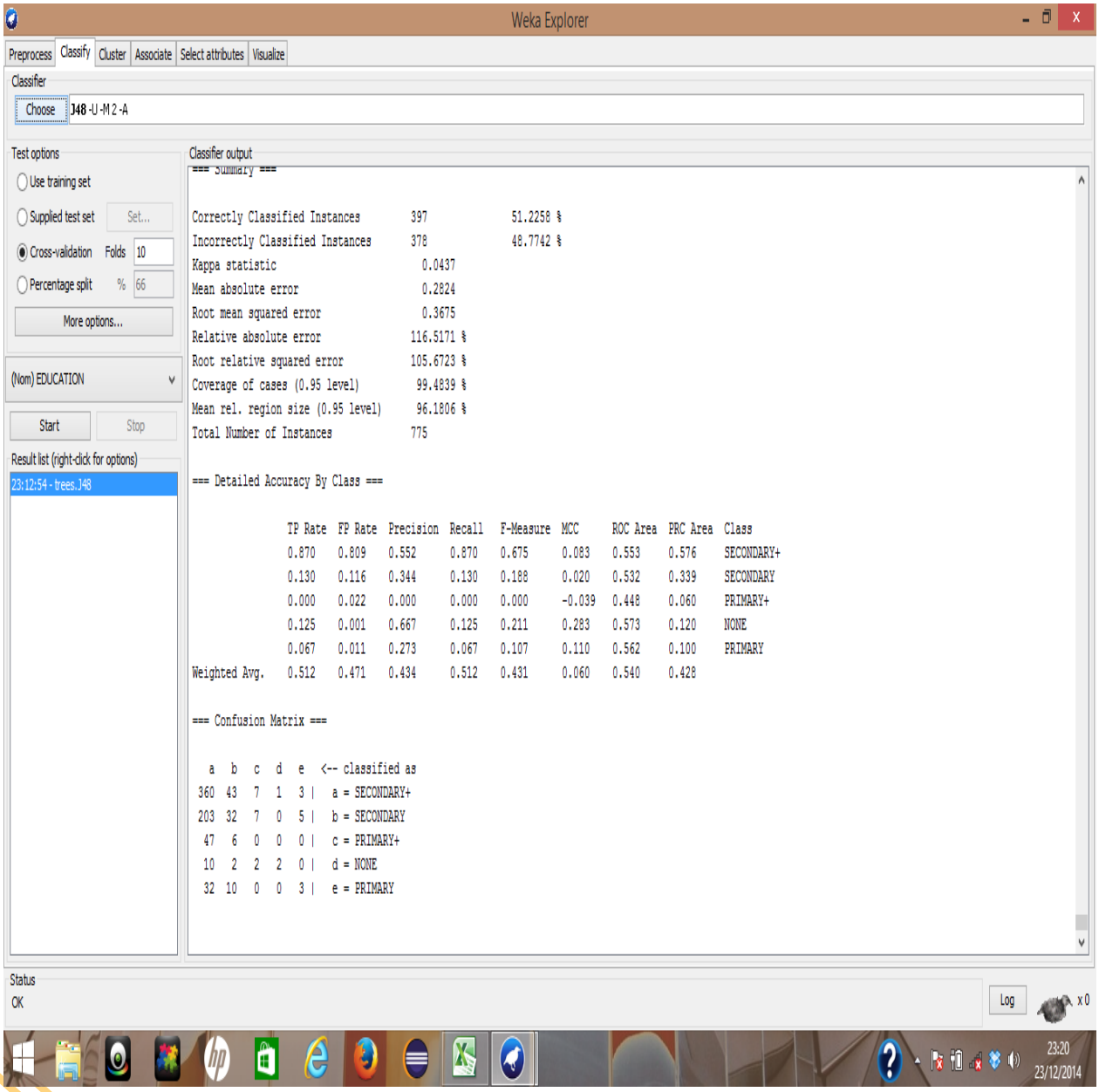
Screenshot of predictions of SMOTE300ENN when used to treat RAW DATA of Diabetes Mellitus dataset trained on Decision Tree Algorithm



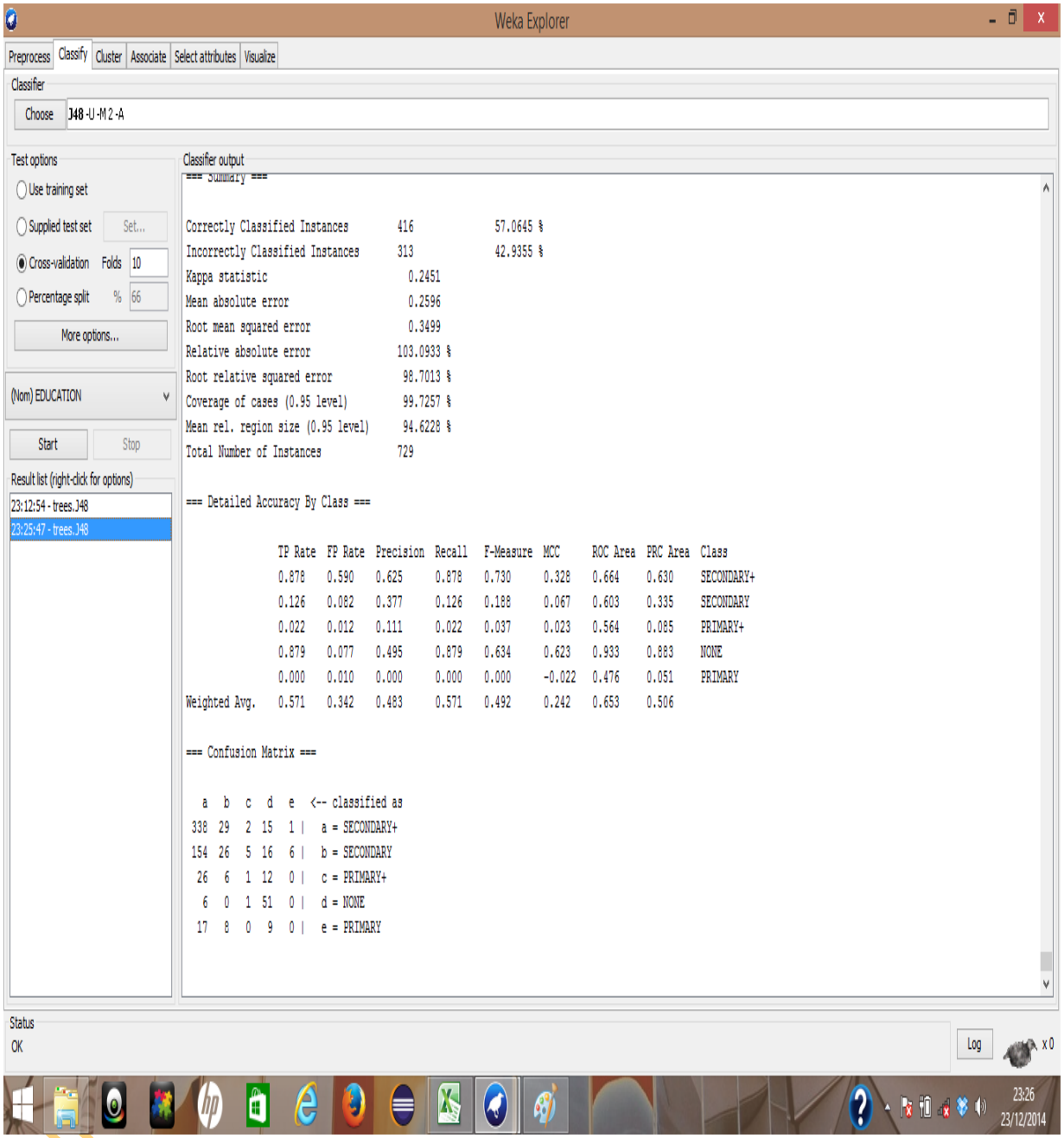
Screenshot of predictions of RAW DATA of SSS Result dataset trained on Decision Tree Algorithm



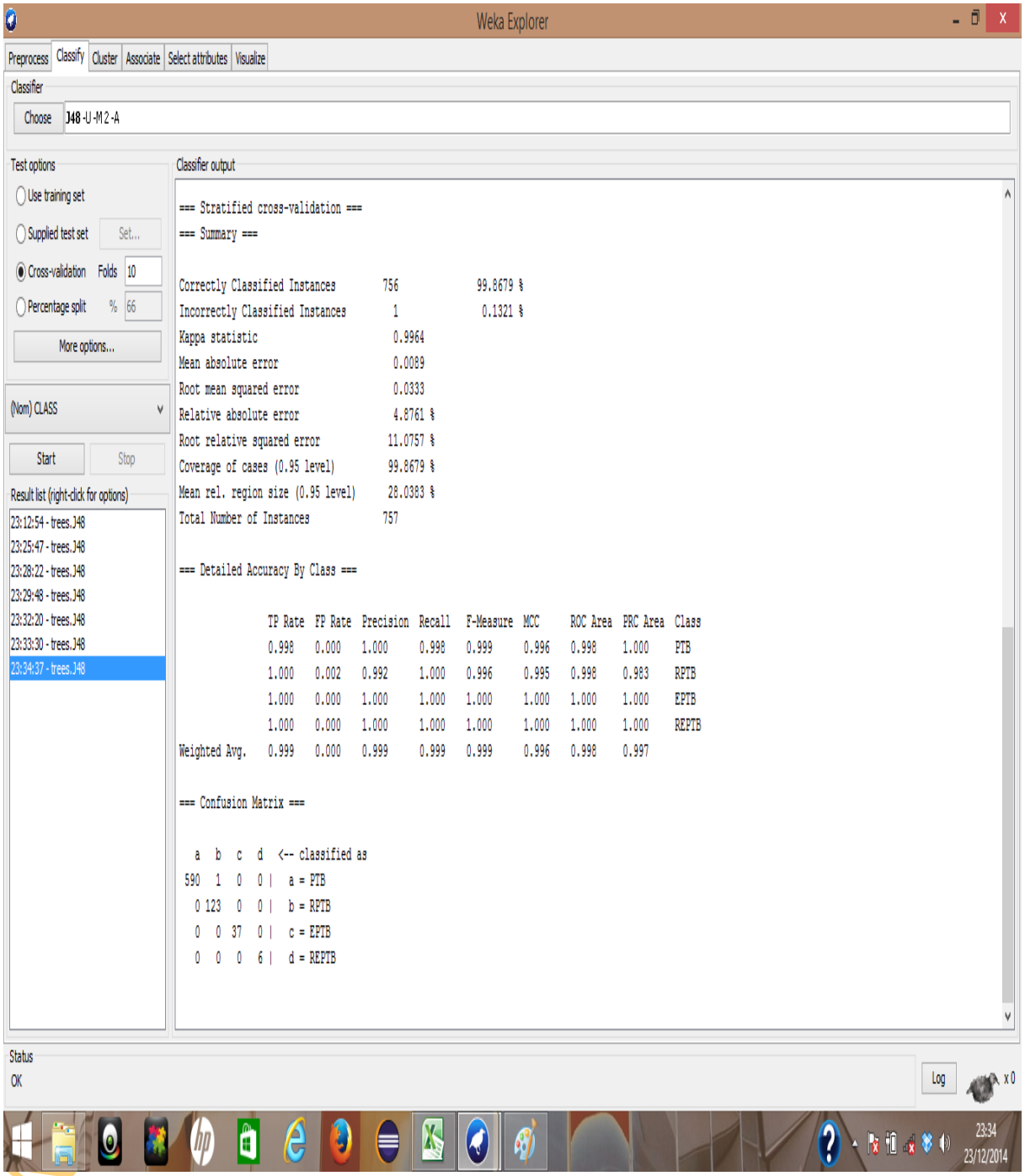
Screenshot of predictions of SMOTE300ENN when used to treat the RAW DATA of SSS Result dataset trained on Decision Tree Algorithm



Screenshot of predictions of RAW DATA of CM dataset trained on Decision Tree Algorithm



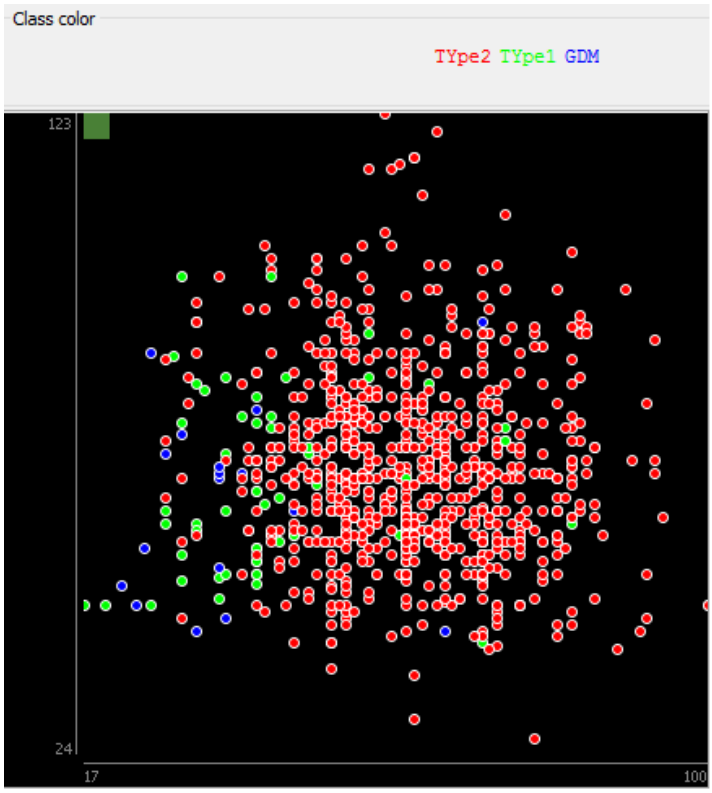
Screenshot of predictions of SMOTE300ENN when used to treat the RAW DATA of CM dataset trained on Decision Tree Algorithm



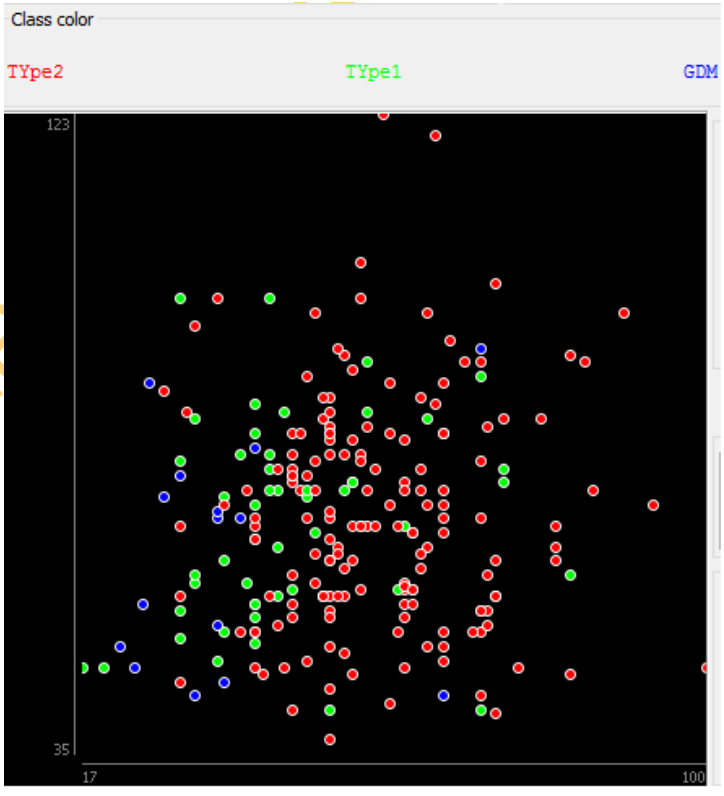
Screenshot of predictions of RAW DATA of Tuberculosis dataset trained on Decision Tree Algorithm

APPENDIX B

The class boundary diagram for DM dataset

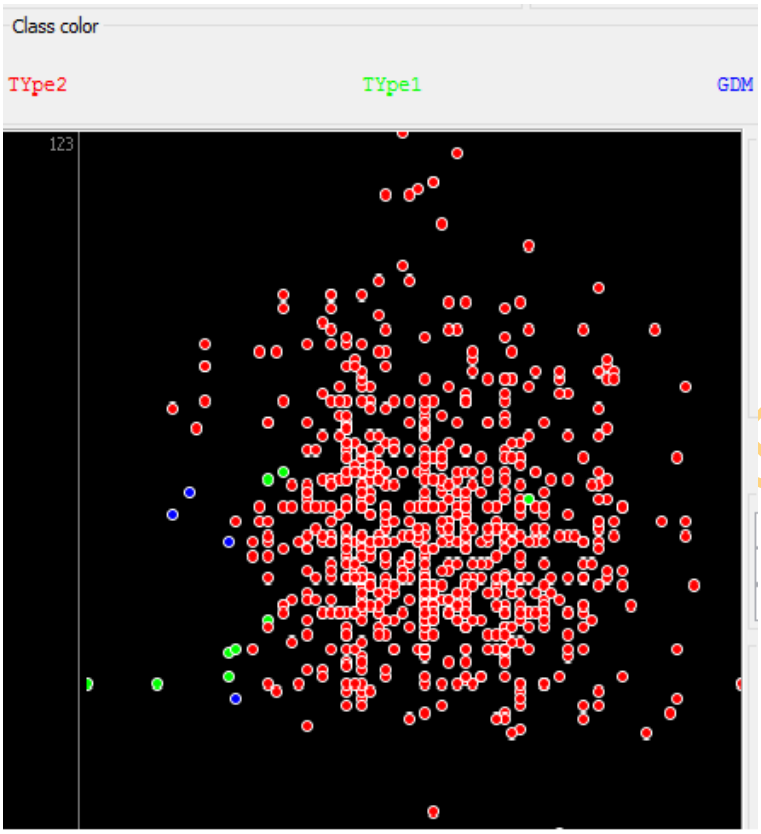


(a) ORIGINAL Diabetes dataset

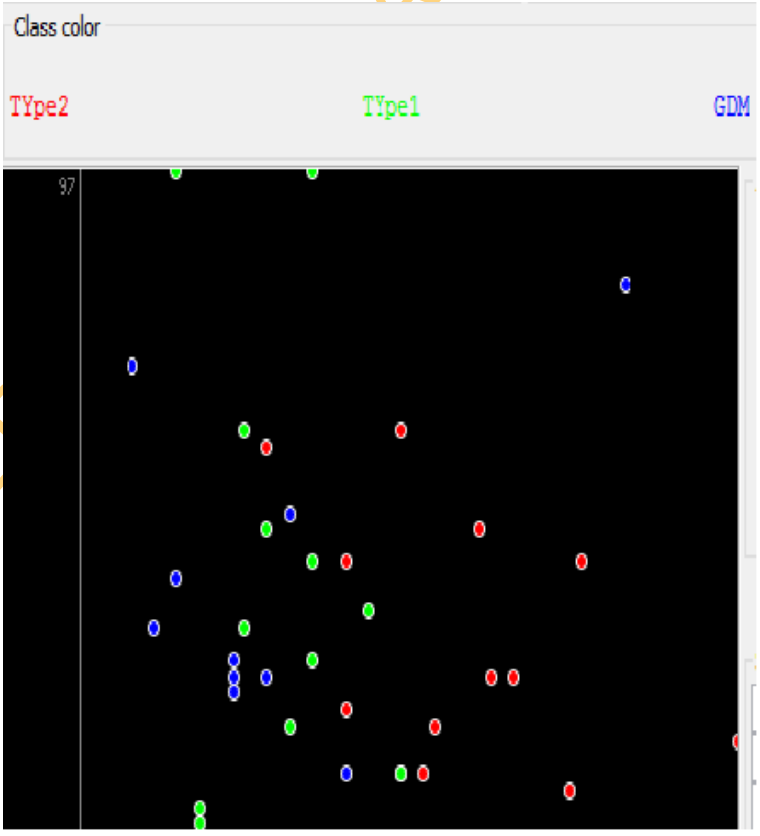


(b) CNN

Figure (a) and (b): Class boundary diagram of Diabetes Mellitus disease with the RAW DATA and *CNN* scheme.

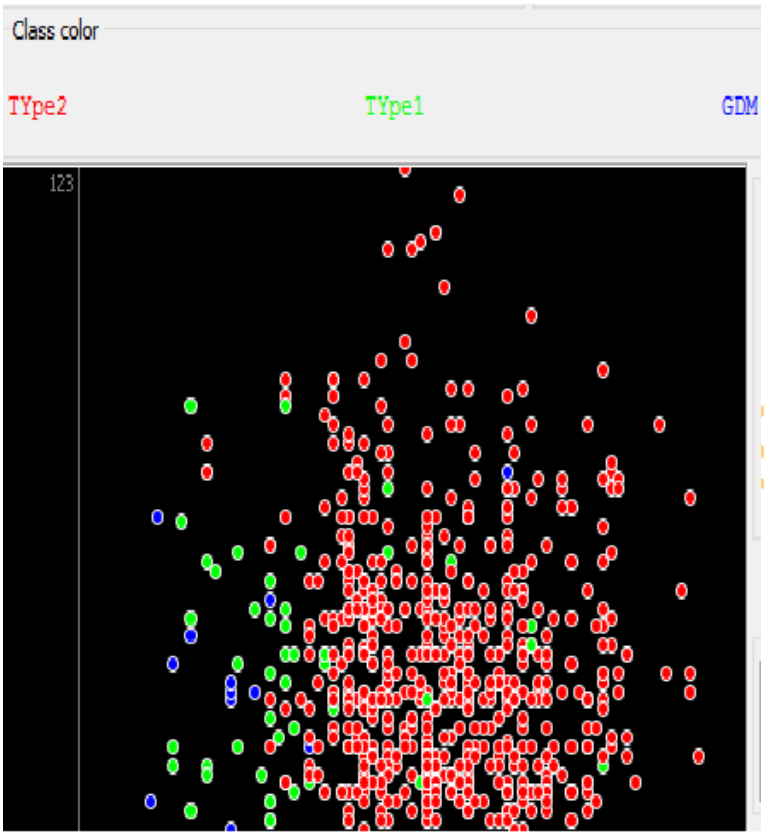


(C) ENN

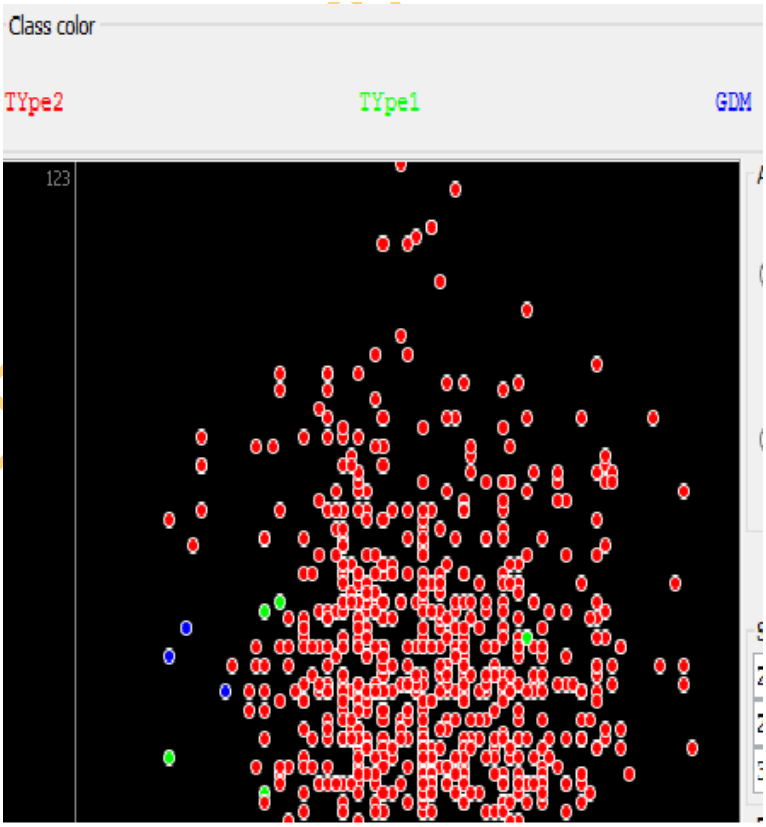


(d) RUS

Figure (c) and (d): Class boundary diagram of Diabetes Mellitus disease with the ENN and RUS scheme.

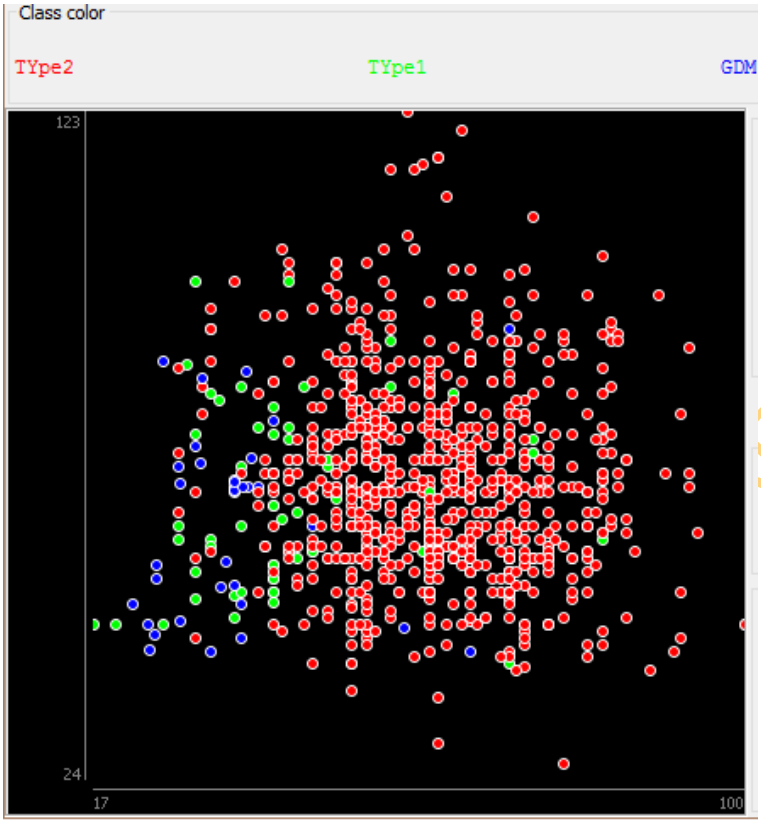


(e) NCL

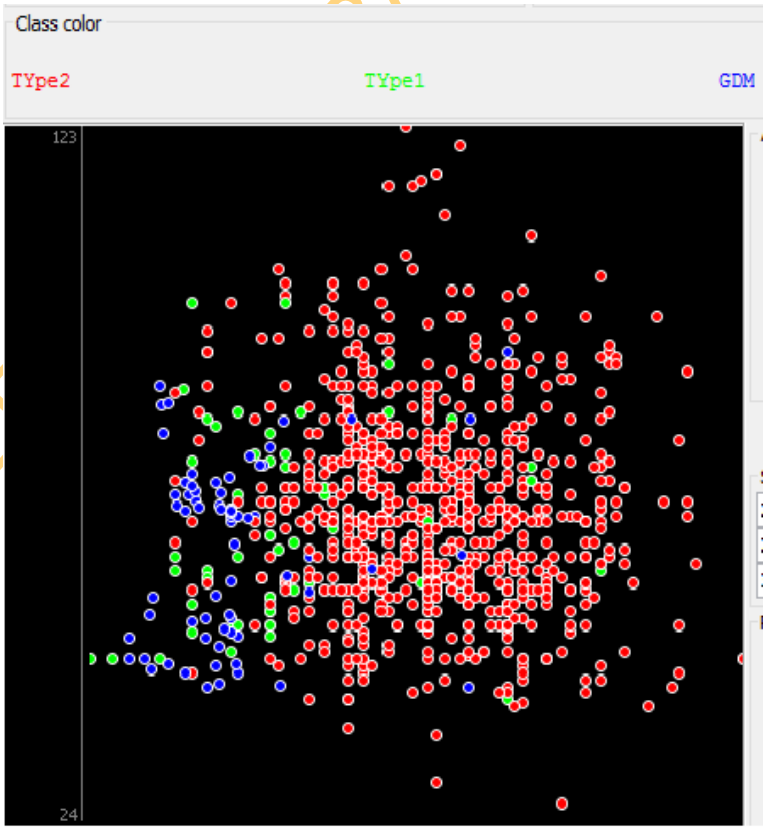


(f) 5ENN

Figure (e) and (f): Class boundary diagram of Diabetes Mellitus disease with the NCL and 5ENN scheme.

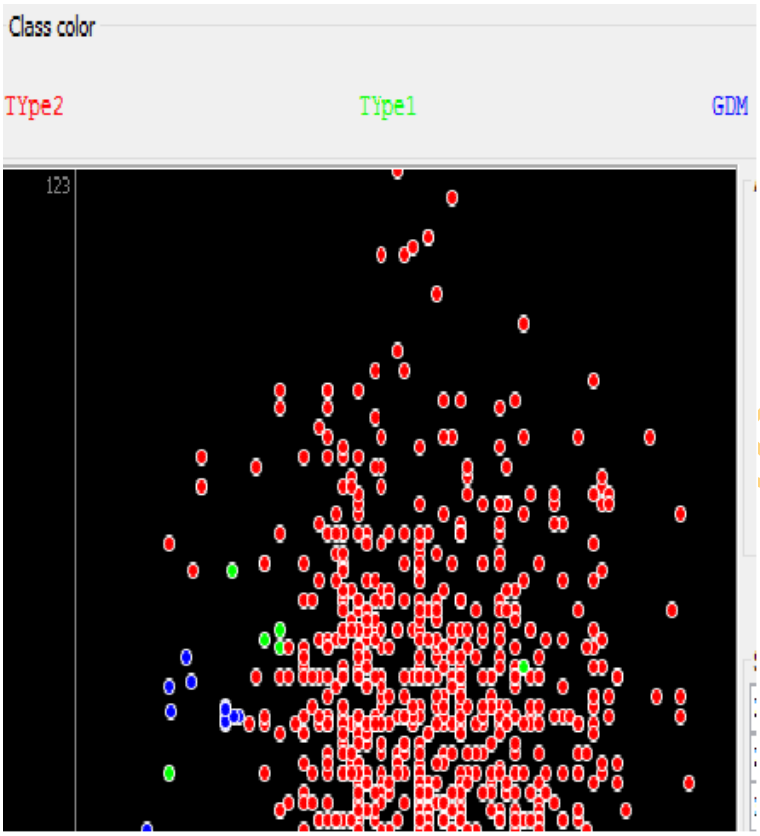


(g) SMOTE

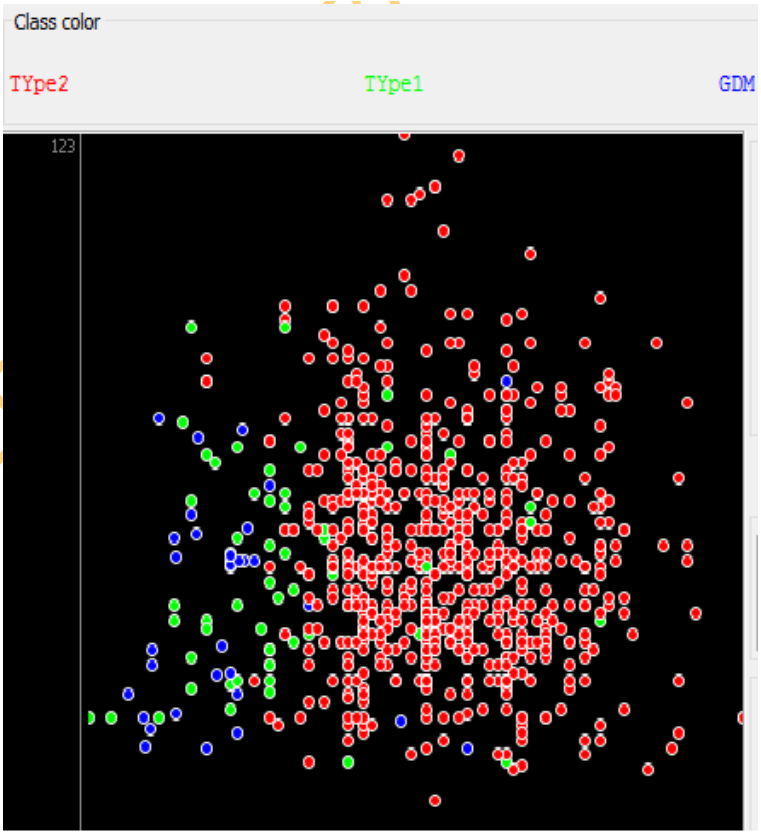


(h) SMOTE300

Figure (g) and (h): Class boundary diagram of Diabetes Mellitus disease with the SMOTE and SMOTE300 scheme.

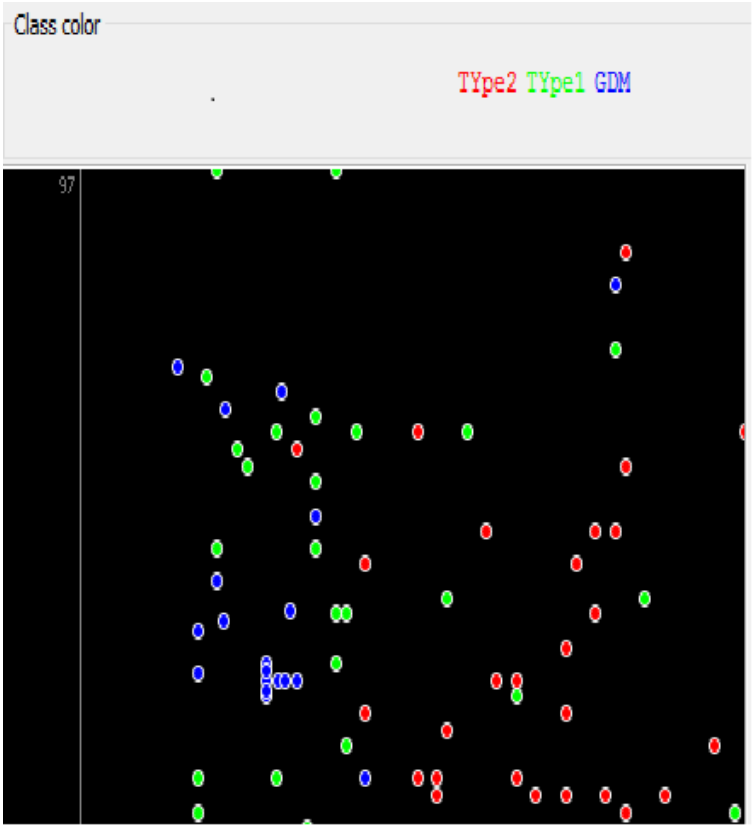


(i) SMOTEENN

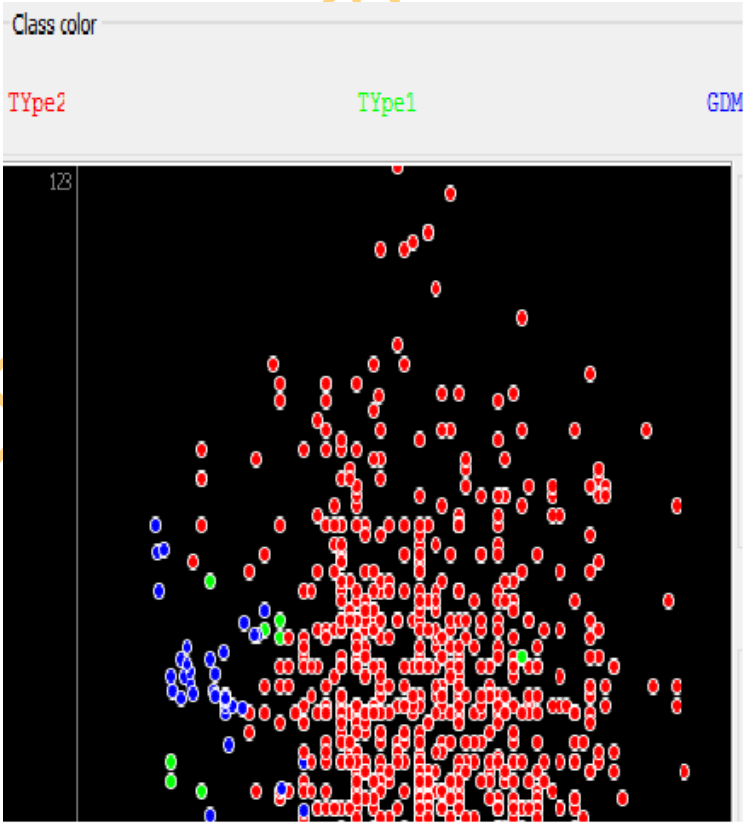


(j) SMOTENCL

Figure (i) and (j): Class boundary diagram of Diabetes Mellitus disease with the SMOTEENN and SMOTENCL scheme.

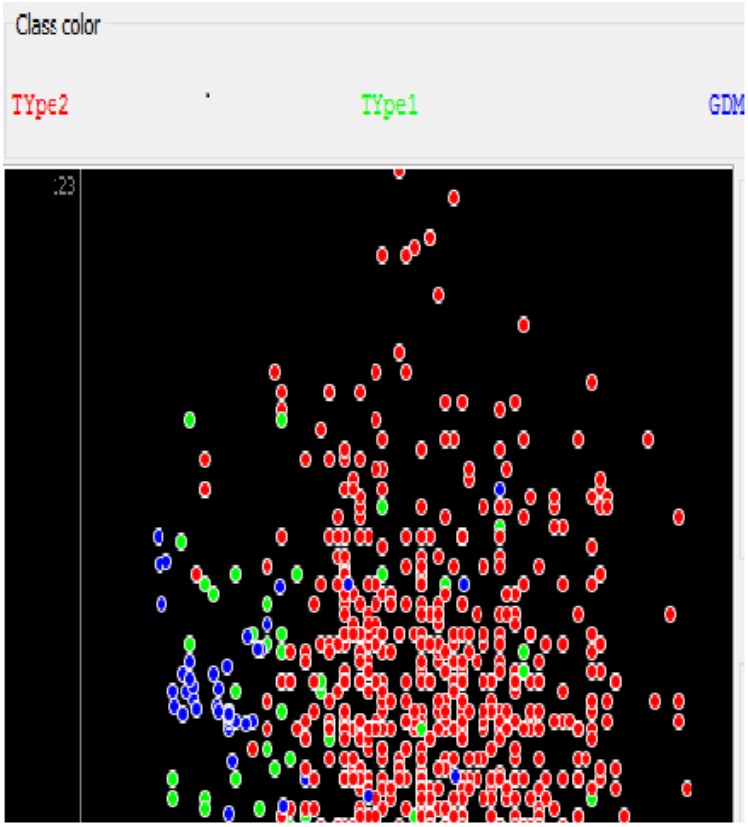


(k) SMOTERUS

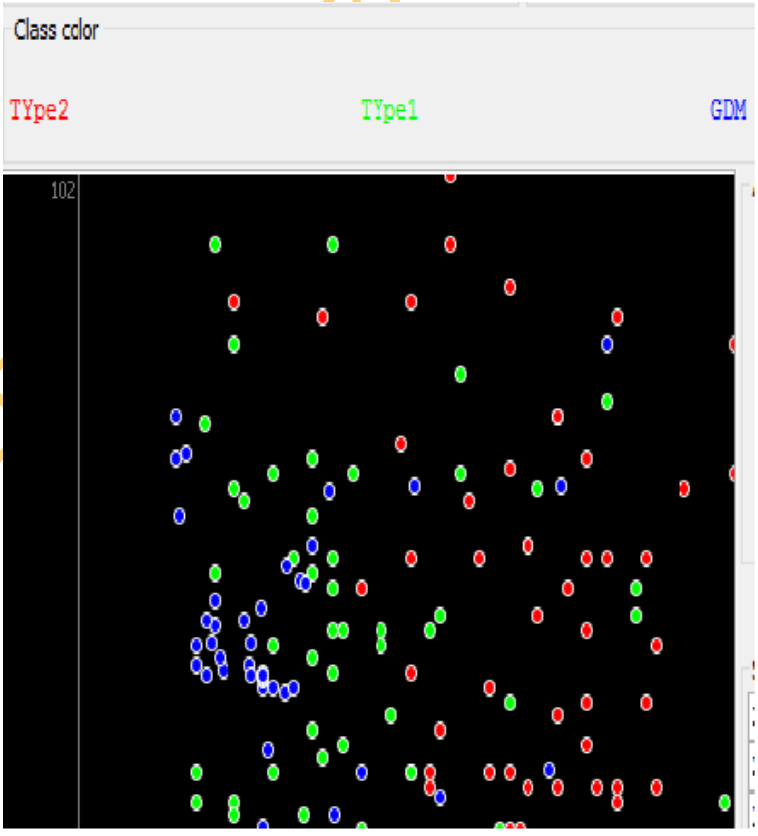


(l) SMOTE300ENN

Figure (k) and (l): Class boundary diagram of Diabetes Mellitus disease with the SMOTERUS and SMOTE300ENN scheme.



(m) SMOTE300NCL



(n) SMOTE300RUS

Figure (m) and (n): Class boundary diagram of Diabetes Mellitus disease with the SMOTE300NCL and SMOTE300RUS scheme.

Appendix C

Java Codes for SMOTE Class

```
/*
 * SMOTE.java
 *
 */

package weka.filters.supervised.instance;

import weka.core.Attribute;
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 java.util.Collections;
import java.util.Comparator;
import java.util.Enumeration;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Random;
import java.util.Set;
import java.util.Vector;
<!-- technical-bibtex-end -->
 *
 <!-- options-start -->
 * Valid options are: <p/>
 *
 * <pre> -S &lt;num&gt;
 *  Specifies the random number seed
 *  (default 1)</pre>
 *
 * <pre> -P &lt;percentage&gt;
 *  Specifies percentage of SMOTE instances to create.
 *  (default 100.0)
 * </pre>
 *
 * <pre> -K &lt;nearest-neighbors&gt;
 *  Specifies the number of nearest neighbors to use.
 *  (default 5)
 * </pre>
 *
 * <pre> -C &lt;value-index&gt;
 *  Specifies the index of the nominal class value to SMOTE
 *  (default 0: auto-detect non-empty minority class))
```

```
 * </pre>
 *
 <!-- options-end -->
  */
public class SMOTE
  extends Filter
  implements SupervisedFilter, OptionHandler,
TechnicalInformationHandler {

  /** for serialization. */
  static final long serialVersionUID = -1653880819059250364L;

  /** the number of neighbors to use. */
  protected int m_NearestNeighbors = 5;

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

  /** the percentage of SMOTE instances to create. */
  protected double m_Percentage = 100.0;

  /** the index of the class value. */
  protected String m_ClassValueIndex = "0";

  /** whether to detect the minority class automatically. */
  protected boolean m_DetectMinorityClass = true;

  /**
   * Returns a string describing this classifier.
   *
   * @return            a description of the classifier suitable for
   *               displaying in the explorer/experimenter gui
   */
  public String globalInfo() {
    return "Resamples a dataset by applying the Synthetic Minority
Oversampling TEchnique (SMOTE)." +
    " The original dataset must fit entirely in memory." +
    " The amount of SMOTE and number of nearest neighbors may be
specified." +
    " For more information, see \n\n"
    + getTechnicalInformation().toString();
  }

    /**
   public String getRevision() {
    return RevisionUtils.extract("$Revision: 5542 $");
  }

  /**
   * 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 percentage of SMOTE instances to create.\n"
      + "\t(default 100.0)\n",
      "P", 1, "-P <percentage>"));

    newVector.addElement(new Option(
      "\tSpecifies the number of nearest neighbors to use.\n"
      + "\t(default 5)\n",
      "K", 1, "-K <nearest-neighbors>"));

    newVector.addElement(new Option(
      "\tSpecifies the index of the nominal class value to SMOTE\n"
      +"\t(default 0: auto-detect non-empty minority class))\n",
      "C", 1, "-C <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;percentage&gt;
   *  Specifies percentage of SMOTE instances to create.
   *  (default 100.0)
   * </pre>
   *
   * <pre> -K &lt;nearest-neighbors&gt;
   *  Specifies the number of nearest neighbors to use.
   *  (default 5)
   * </pre>
   *
   * <pre> -C &lt;value-index&gt;
   *  Specifies the index of the nominal class value to SMOTE
```

```
   *  (default 0: auto-detect non-empty minority class))
   * </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 percentageStr = Utils.getOption('P', options);
    if (percentageStr.length() != 0) {
      setPercentage(new Double(percentageStr).doubleValue());
    } else {
      setPercentage(100.0);
    }

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

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

  /**
   * 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("-K");
    result.add("" + getNearestNeighbors());

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

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

    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 seed used for random sampling.";
}

/**
 * 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 percentageTipText() {
  return "The percentage of SMOTE instances to create.";
}

/**
 * Sets the percentage of SMOTE instances to create.
 *
 * @param value the percentage to use
 */
public void setPercentage(double value) {
  if (value >= 0)
    m_Percentage = value;
  else
    System.err.println("Percentage must be >= 0!");
}

/**
 * Gets the percentage of SMOTE instances to create.
 *
 * @return            the percentage of SMOTE instances to create
 */
public double getPercentage() {
  return m_Percentage;
}
```

```
  /**
   * Returns the tip text for this property.
   *
   * @return            tip text for this property suitable for
   *                displaying in the explorer/experimenter gui
   */
  public String nearestNeighborsTipText() {
    return "The number of nearest neighbors to use.";
  }

  /**
   * Sets the number of nearest neighbors to use.
   *
   * @param value the number of nearest neighbors to use
   */
  public void setNearestNeighbors(int value) {
    if (value >= 1)
      m_NearestNeighbors = value;
    else
      System.err.println("At least 1 neighbor necessary!");
  }

  /**
   * Gets the number of nearest neighbors to use.
   *
   * @return            the number of nearest neighbors to use
   */
  public int getNearestNeighbors() {
    return m_NearestNeighbors;
  }

  /**
   * 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 SMOTE should be
applied. " +
    "Use a value of 0 to auto-detect the non-empty minority class.";
  }

  /**
   * Sets the index of the class value to which SMOTE should be
applied.
   *
   * @param value the class value index
   */
  public void setClassValue(String value) {
    m_ClassValueIndex = value;
    if (m_ClassValueIndex.equals("0")) {
      m_DetectMinorityClass = true;
    } else {
      m_DetectMinorityClass = false;
    }
  }

  /**
   * Gets the index of the class value to which SMOTE should be
applied.
```

```
   *
   * @return            the index of the clas value to which SMOTE
should be applied
   */
  public String getClassValue() {
    return m_ClassValueIndex;
  }

  /**
   * 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 SMOTE, and clear the input instances.
      doSMOTE();
    }
    flushInput();

    m_NewBatch = true;
    m_FirstBatchDone = true;
    return (numPendingOutput() != 0);
  }

  /**
   * The procedure implementing the SMOTE algorithm. The output
   * instances are pushed onto the output queue for collection.
   *
   * @throws Exception  if provided options cannot be executed
   *                    on input instances
   */
  protected void doSMOTE() throws Exception {
    int minIndex = 0;
    int min = Integer.MAX_VALUE;
    if (m_DetectMinorityClass) {
      // find minority class
      int[] classCounts =
getInputFormat().attributeStats(getInputFormat().classIndex()).nomina
lCounts;
      for (int i = 0; i < classCounts.length; i++) {
      if (classCounts[i] != 0 && classCounts[i] < min) {
        min = classCounts[i];
        minIndex = i;
      }
      }
    } else {
      String classVal = getClassValue();
      if (classVal.equalsIgnoreCase("first")) {
      minIndex = 1;
      } else if (classVal.equalsIgnoreCase("last")) {
      minIndex = getInputFormat().numClasses();
      } else {
      minIndex = Integer.parseInt(classVal);
      }
      if (minIndex > getInputFormat().numClasses()) {
      throw new Exception("value index must be <= the number of
classes");
      }
      minIndex--; // make it an index
    }
```

```
    int nearestNeighbors;
    if (min <= getNearestNeighbors()) {
      nearestNeighbors = min - 1;
    } else {
      nearestNeighbors = getNearestNeighbors();
    }
    if (nearestNeighbors < 1)
      throw new Exception("Cannot use 0 neighbors!");

    // compose minority class dataset
    // also push all dataset instances
    Instances sample = getInputFormat().stringFreeStructure();
    Enumeration instanceEnum = getInputFormat().enumerateInstances();
    while(instanceEnum.hasMoreElements()) {
      Instance instance = (Instance) instanceEnum.nextElement();
      push((Instance) instance.copy());
      if ((int) instance.classValue() == minIndex) {
      sample.add(instance);
      }
    }

    // compute Value Distance Metric matrices for nominal features
    Map vdmMap = new HashMap();
    Enumeration attrEnum = getInputFormat().enumerateAttributes();
    while(attrEnum.hasMoreElements()) {
      Attribute attr = (Attribute) attrEnum.nextElement();
      if (!attr.equals(getInputFormat().classAttribute())) {
      if (attr.isNominal() || attr.isString()) {
        double[][] vdm = new
double[attr.numValues()][attr.numValues()];
        vdmMap.put(attr, vdm);
        int[] featureValueCounts = new int[attr.numValues()];
        int[][] featureValueCountsByClass = new
int[getInputFormat().classAttribute().numValues()][attr.numValues()];
        instanceEnum = getInputFormat().enumerateInstances();
        while(instanceEnum.hasMoreElements()) {
          Instance instance = (Instance) instanceEnum.nextElement();
          int value = (int) instance.value(attr);
          int classValue = (int) instance.classValue();
          featureValueCounts[value]++;
          featureValueCountsByClass[classValue][value]++;
        }
        for (int valueIndex1 = 0; valueIndex1 < attr.numValues();
valueIndex1++) {
          for (int valueIndex2 = 0; valueIndex2 < attr.numValues();
valueIndex2++) {
            double sum = 0;
            for (int classValueIndex = 0; classValueIndex <
getInputFormat().numClasses(); classValueIndex++) {
            double c1i = (double)
featureValueCountsByClass[classValueIndex][valueIndex1];
            double c2i = (double)
featureValueCountsByClass[classValueIndex][valueIndex2];
            double c1 = (double) featureValueCounts[valueIndex1];
            double c2 = (double) featureValueCounts[valueIndex2];
            double term1 = c1i / c1;
            double term2 = c2i / c2;
            sum += Math.abs(term1 - term2);
            }
            vdm[valueIndex1][valueIndex2] = sum;
```

```
          }
        }
      }
      }
    }

    // use this random source for all required randomness
    Random rand = new Random(getRandomSeed());

    // find the set of extra indices to use if the percentage is not
evenly divisible by 100
    List extraIndices = new LinkedList();
    double percentageRemainder = (getPercentage() / 100) -
Math.floor(getPercentage() / 100.0);
    int extraIndicesCount = (int) (percentageRemainder *
sample.numInstances());
    if (extraIndicesCount >= 1) {
      for (int i = 0; i < sample.numInstances(); i++) {
      extraIndices.add(i);
      }
    }
    Collections.shuffle(extraIndices, rand);
    extraIndices = extraIndices.subList(0, extraIndicesCount);
    Set extraIndexSet = new HashSet(extraIndices);

    // the main loop to handle computing nearest neighbors and
generating SMOTE
    // examples from each instance in the original minority class
data
    Instance[] nnArray = new Instance[nearestNeighbors];
    for (int i = 0; i < sample.numInstances(); i++) {
      Instance instanceI = sample.instance(i);
      // find k nearest neighbors for each instance
      List distanceToInstance = new LinkedList();
      for (int j = 0; j < sample.numInstances(); j++) {
      Instance instanceJ = sample.instance(j);
      if (i != j) {
        double distance = 0;
        attrEnum = getInputFormat().enumerateAttributes();
        while(attrEnum.hasMoreElements()) {
          Attribute attr = (Attribute) attrEnum.nextElement();
          if (!attr.equals(getInputFormat().classAttribute())) {
            double iVal = instanceI.value(attr);
            double jVal = instanceJ.value(attr);
            if (attr.isNumeric()) {
            distance += Math.pow(iVal - jVal, 2);
            } else {
            distance += ((double[][]) vdmMap.get(attr))[(int)
iVal][(int) jVal];
            }
          }
        }
        distance = Math.pow(distance, .5);
        distanceToInstance.add(new Object[] {distance, instanceJ});
      }
      }

      // sort the neighbors according to distance
      Collections.sort(distanceToInstance, new Comparator() {
      public int compare(Object o1, Object o2) {
        double distance1 = (Double) ((Object[]) o1)[0];
```

```
        double distance2 = (Double) ((Object[]) o2)[0];
        return (int) Math.ceil(distance1 - distance2);
      }
      });

      // populate the actual nearest neighbor instance array
      Iterator entryIterator = distanceToInstance.iterator();
      int j = 0;
      while(entryIterator.hasNext() && j < nearestNeighbors) {
      nnArray[j] = (Instance) ((Object[])entryIterator.next())[1];
      j++;
      }

      // create synthetic examples
      int n = (int) Math.floor(getPercentage() / 100);
      while(n > 0 || extraIndexSet.remove(i)) {
      double[] values = new double[sample.numAttributes()];
      int nn = rand.nextInt(nearestNeighbors);
      attrEnum = getInputFormat().enumerateAttributes();
      while(attrEnum.hasMoreElements()) {
        Attribute attr = (Attribute) attrEnum.nextElement();
        if (!attr.equals(getInputFormat().classAttribute())) {
          if (attr.isNumeric()) {
            double dif = nnArray[nn].value(attr) -
instanceI.value(attr);
            double gap = rand.nextDouble();
            values[attr.index()] = (double) (instanceI.value(attr) +
gap * dif);
          } else if (attr.isDate()) {
            double dif = nnArray[nn].value(attr) -
instanceI.value(attr);
            double gap = rand.nextDouble();
            values[attr.index()] = (long) (instanceI.value(attr) +
gap * dif);
          } else {
            int[] valueCounts = new int[attr.numValues()];
            int iVal = (int) instanceI.value(attr);
            valueCounts[iVal]++;
            for (int nnEx = 0; nnEx < nearestNeighbors; nnEx++) {
            int val = (int) nnArray[nnEx].value(attr);
            valueCounts[val]++;
            }
            int maxIndex = 0;
            int max = Integer.MIN_VALUE;
            for (int index = 0; index < attr.numValues(); index++) {
            if (valueCounts[index] > max) {
              max = valueCounts[index];
              maxIndex = index;
            }
            }
            values[attr.index()] = maxIndex;
          }
        }
      }
      values[sample.classIndex()] = minIndex;
      Instance synthetic = new Instance(1.0, values);
      push(synthetic);
      n--;
      }
    }
  }
```

```
  /**
   * Main method for running this filter.
   *
   * @param args  should contain arguments to the filter:
   *              use -h for help
   */
  public static void main(String[] args) {
    runFilter(new SMOTE(), args);
  }
}
```

Appendix D

Java codes for Wilson's Edited Nearest Neigbhour (ENN) Class

```
/**
 * Class EditedNN
 *
**/

package weka.filters.supervised.instance;

import java.io.Serializable;
import java.util.*;

import weka.filters.*;
import weka.core.*;
import weka.core.Capabilities.Capability;


public class EditedNN extends Filter implements SupervisedFilter,
OptionHandler {


      /**
       * Genereated by Eclipse.
       */
      private static final long serialVersionUID = -
7206839648986893545L;

      /** Returns the revision string. **/
      public String getRevision() {
            return RevisionUtils.extract("$Revision: 1.0 $");
      }

      /** Returns default capabilities of the classifier. **/
      public Capabilities getCapabilities() {

            Capabilities result = super.getCapabilities();

            // attributes
            result.enable(Capability.NOMINAL_ATTRIBUTES);
            result.enable(Capability.NUMERIC_ATTRIBUTES);
            result.enable(Capability.DATE_ATTRIBUTES);
            result.enable(Capability.MISSING_VALUES);

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

            // instances
            result.setMinimumNumberInstances(0);

            return result;

      }

      /** Description of the classifier in Weka's graphical mode. **/
      public String globalInfo() {
            return "";
      }
```

```
	/** Class constructors. **/
	public EditedNN(int k) {
		setKnn(k);
		m_Distance = DIST_HEOM;
		m_Weighting = WEIGHT_INV;
	}

	public EditedNN() {
		setKnn(3);
		m_Distance = DIST_HEOM;
		m_Weighting = WEIGHT_INV;
	}



	/**
	* Stores the maximum and minimum values and standard deviation
	* of each attribute (depending on the metric used).
	**/
	private double[] m_Min, m_Max, m_StdDev;

	/** Structures to store data to calculate the VDM metric. **/
	Vector < double[] > m_VetVDM = new Vector<double[]>();
	Vector < double[][] > m_MatVDM = new Vector<double[][]>();

	/** Auxiliary variable **/
	double m_Bigger;

	/** Number of nearest neighbor (k). **/
	private int m_Knn = 3;
	public void setKnn(int m_Knn) { this.m_Knn = m_Knn; }
	public int getKnn() { return this.m_Knn; }

	/** Distance function to be used in the algorithm. **/
	private int m_Distance = DIST_HEOM;

	public void setDistance(SelectedTag newMethod) {

		if (newMethod.getTags() == TAGS_DISTANCE) {
			this.m_Distance = newMethod.getSelectedTag().getID();
		}

	}

	public SelectedTag getDistance() {
		return new SelectedTag(this.m_Distance, TAGS_DISTANCE);
	}

	public static final int DIST_HEOM = 1;
	public static final int DIST_HVDM = 2;
	public static final int DIST_MANHATTAN = 3;
	public static final Tag [] TAGS_DISTANCE = {
	    new Tag(DIST_HEOM, "Heterogeneous Euclidean-Overlap
Metric"),
	    new Tag(DIST_MANHATTAN, "Heterogeneous Manhattan-Overlap
Metric"),
	    new Tag(DIST_HVDM, "Heterogeneous Value Distance Function")
	};
```

```
	/** Distance weighting. **/
	private int m_Weighting;

	public void setWeighting(SelectedTag newMethod) {

		if (newMethod.getTags() == TAGS_WEIGHTING) {
		  this.m_Weighting = newMethod.getSelectedTag().getID();
		}

	}

	public SelectedTag getWeighting() {
		return new SelectedTag(this.m_Weighting, TAGS_WEIGHTING);
	}

	public static final int WEIGHT_NONE = 1;
	public static final int WEIGHT_INV = 2;
	public static final int WEIGHT_SIM = 3;
	public static final Tag [] TAGS_WEIGHTING = {
	    new Tag(WEIGHT_NONE, "No weight"),
	    new Tag(WEIGHT_INV, "1/(distance^2)"),
	    new Tag(WEIGHT_SIM, "1-distance")
	};

	/** Method (cleanness or undersampling) to be used in the
algorithm. **/
	private int m_Method = METH_CLEAN;

	public void setMethod(SelectedTag newMethod) {

		if (newMethod.getTags() == TAGS_METHOD) {
		  this.m_Method = newMethod.getSelectedTag().getID();
		}

	}

	public SelectedTag getMethod() {
		return new SelectedTag(this.m_Method, TAGS_METHOD);
	}

	public static final int METH_CLEAN = 1;
	public static final int METH_UNDER = 2;
	public static final Tag [] TAGS_METHOD = {
	    new Tag(METH_CLEAN, "Cleanness"),
	    new Tag(METH_UNDER, "Undersampling"),
	};


	/** Definitions and structures to use in this filter. **/
	private Instances m_Input;
	private double m_MajorityClassValue;
	private Vector<Integer> m_InstancesToRemove = new
Vector<Integer>();

	/** Description of parameters **/
	public String knnTipText() {
		return "Number of nearest neighbors (k).";
	}

	public String distanceTipText() {
```

```
            return "Distance function.";
      }

      public String weightingTipText() {
            return "Distance weighting.";
      }

      public String methodTipText() {
            return "Method (cleanness or undersampling) to be used in
the algorithm.";
      }

      /** Parses a given list of options. **/
      public void setOptions(String[] options) throws Exception {

            String knnString = Utils.getOption('K', options);
          if (knnString.length() != 0) {
            setKnn(Integer.parseInt(knnString));
          } else {
            setKnn(1);
          }

            if (Utils.getFlag('V', options)) {
              setDistance(new SelectedTag(DIST_HVDM, TAGS_DISTANCE));
            } else if (Utils.getFlag('M', options)) {
              setDistance(new SelectedTag(DIST_MANHATTAN,
TAGS_DISTANCE));
          } else {
            setDistance(new SelectedTag(DIST_HEOM, TAGS_DISTANCE));
          }

            if (Utils.getFlag('I', options)) {
                  setWeighting(new SelectedTag(WEIGHT_INV,
TAGS_WEIGHTING));
            } else if (Utils.getFlag('S', options)) {
                  setWeighting(new SelectedTag(WEIGHT_SIM,
TAGS_WEIGHTING));
            } else {
                  setWeighting(new SelectedTag(WEIGHT_NONE,
TAGS_WEIGHTING));
            }

            if (Utils.getFlag('U', options)) {
                  setMethod(new SelectedTag(METH_UNDER,
TAGS_METHOD));
            } else {
                  setMethod(new SelectedTag(METH_CLEAN,
TAGS_METHOD));
            }

            Utils.checkForRemainingOptions(options);

      }


      /** Gets the current settings of KnnImputation. */
      public String [] getOptions() {

          String [] options = new String [7];
          int current = 0;
```

```
          options[current++] = "-K"; options[current++] = "" +
getKnn();

          if (m_Distance == DIST_HVDM) {
            options[current++] = "-V";
          }

          if (m_Distance == DIST_MANHATTAN) {
            options[current++] = "-M";
          }

          if (m_Weighting == WEIGHT_INV) {
            options[current++] = "-I";
          }

          if (m_Weighting == WEIGHT_SIM) {
            options[current++] = "-S";
          }

          if (m_Method == METH_UNDER) {
            options[current++] = "-U";
          }

          while (current < options.length) {
                  options[current++] = "";
            }

          return options;

      }

      /** Returns an enumeration describing the available options.
**/
      public Enumeration<Option> listOptions() {

            Vector<Option> newVector = new Vector<Option>(6);

          newVector.addElement(new Option(
               "\tNumber of nearest neighbors (k).\n"
                +"\t(Default = 3)",
                "K", 1,"-K <number of neighbors>"));

          newVector.addElement(new Option(
                        "\tHeterogeneous Euclidean-Value Distance
Metric.\n",
                        "V", 0, "-V"));

          newVector.addElement(new Option(
                        "\tHeterogeneous Manhattan-Overlap
Metric.\n",
                        "M", 0, "-M"));

          newVector.addElement(new Option(
                        "\tWeight neighbors by inverse of their
squared distance.\n",
                        "I", 0, "-I"));

          newVector.addElement(new Option(
                        "\tWeight neighbors by similarity.\n",
                        "S", 0, "-S"));
```

```
            newVector.addElement(new Option(
                        "\tUses the method to undersampling.\n",
                        "U", 0, "-U"));

            return newVector.elements();

      }


      /** Vector used to store the neighbors according their
proximity **/
      private Vector<Neighbor> neighbors = new Vector<Neighbor>();

      /** Class that defines a neighbor **/
      private class Neighbor implements Serializable {

            /**
             * Generated by Eclipse
             */
            private static final long serialVersionUID = -
3536862725297518804L;

            double dist;
            Instance inst;
            int index;

            Neighbor(double dist, Instance inst, int index) {
                  this.dist = dist;
                  this.inst = inst;
                  this.index = index;
            }

            public double getDist() { return dist; }
            public Instance getInst() { return inst; }
            public int getIndex() { return index; }

      }


      /** Initializes the input and output formats. **/
      public boolean setInputFormat(Instances instanceInfo) throws
Exception {

            super.setInputFormat(instanceInfo);
            setOutputFormat(instanceInfo);

            m_Input = instanceInfo;

            return true;

      }

    /** Input an instance for filtering. **/
      public boolean input(Instance instance) {

          if (m_Input == null) {
            throw new IllegalStateException("No input instance format
defined");
          }
```

```
            if (m_NewBatch) {
              resetQueue();
              m_NewBatch = false;
            }

            if (isFirstBatchDone()) {
                    push(instance);
                    return true;
              } else {
                    bufferInput(instance);
                    return false;
              }

      }

      /** Signify that this batch of input to the filter is finished.
**/
      public boolean batchFinished() throws Exception {

            if (m_Input == null) {
                  throw new IllegalStateException("No input instance
format defined");
            }

            if (!isFirstBatchDone()) {

                  //Initializes the vectors and determines the
initial values to calculate the HVDM function
                if (m_Distance == DIST_HVDM) {

                  m_StdDev = new double [m_Input.numAttributes()];

                  for (int i = 0; i < m_Input.numAttributes(); i++) {

                        if (m_Input.attribute(i).isNominal()) {

                              m_VetVDM.add(new
double[m_Input.attribute(i).numValues()]);
                              m_MatVDM.add(new
double[m_Input.attribute(i).numValues()][m_Input.numClasses()]);

                        } else {

                              //Calculates the standard deviation for
each attribute
                              AttributeStats as =
m_Input.attributeStats(i);
                                    m_StdDev[i] =
as.numericStats.stdDev;

                              m_VetVDM.add(new double[0]);
                              m_MatVDM.add(new double[0][0]);

                        }

                  }

                } else {
                  m_Min = new double [m_Input.numAttributes()];
                      m_Max = new double [m_Input.numAttributes()];
```

```
                        for (int i=0; i < m_Input.numAttributes(); i++)
{
                          m_Min[i] = Double.MAX_VALUE;
                          m_Max[i] = Double.MIN_VALUE;
                        }

                    }

                      evaluateData();

                      if (m_Method == METH_UNDER)
                            majorityClass();

                      for(int i = 0; i < m_Input.numInstances(); i++) {
                            evaluateInstance(m_Input.instance(i), i);
                      }

                      for (int i = m_InstancesToRemove.size()-1; i >= 0;
i--) {

      m_Input.delete(m_InstancesToRemove.elementAt(i));
                      }

                      //Push pending input instances
                      for(int i = 0; i < m_Input.numInstances(); i++) {
                            push(m_Input.instance(i));
                      }

                }

                //Free memory
                flushInput();

                m_NewBatch = true;
                m_FirstBatchDone = true;
                return (numPendingOutput() != 0);

      }

      /** Verify the majority class. **/
      private int majorityClass() {

                int[] classes = new int[m_Input.numClasses()];
                int counter = 0;

                m_MajorityClassValue = -1.0;

                for(int i = 0; i < m_Input.numInstances(); i++) {

                      classes[(int)m_Input.instance(i).classValue()]++;

                      if (classes[(int)m_Input.instance(i).classValue()]
> counter) {
                            m_MajorityClassValue =
m_Input.instance(i).classValue();
                            counter++;
                      }

                }

                return counter;
```

```
	}

	/** Evaluate an instance to mark instances to remove. **/
	private void evaluateInstance(Instance instance, int index)
throws Exception {

		if (m_Method==METH_UNDER &&
instance.classValue()!=m_MajorityClassValue) {
			return;
		}

		double classification = estimate(instance);

		if (classification != instance.classValue()) {
			m_InstancesToRemove.add(index);
		}

	}



	/** Estimate values. **/
	public double estimate(Instance instance) throws Exception {

		double dist;
		int i;
		int att = instance.classIndex();

		neighbors.clear();
		m_Bigger = Double.NaN;

		for (int index = 0; index < m_Input.numInstances();
index++) {

			//Current instance
			Instance actualInstance =
(Instance)m_Input.instance(index);

			if (actualInstance.isMissing(att))
				continue;


			dist = distance(instance, actualInstance, att);

			//If the calculated distance is greater than K-th
neighbor yet calculated, returns -1
			if (dist>0) {
				Neighbor v = new Neighbor(dist,
actualInstance, index);

				if (neighbors.size()==0)
					neighbors.add(v);
				else {
					if ( (neighbors.size()>=m_Knn) )
						i = search(0,m_Knn-1,dist);
					else
						i = search(0,neighbors.size()-
1,dist);
```

```
                              neighbors.add(i, v);

                        }

                        if( (neighbors.size()>=m_Knn) )
                              m_Bigger = neighbors.elementAt(m_Knn-
1).getDist();

                  }

            }

            return mode(att);

      }

      /** Function that seeks the position where a neighbor should be
inserted into the
       * vector, based on binary search. **/
      public int search(int begin, int end, double value) {

            int center=(int)(begin+end)/2;

            while (begin<=end) {

                  center=(int)(begin+end)/2;

                  if (neighbors.elementAt(center).getDist() < value )
{

                        if (center<neighbors.size()-1 &&
                              neighbors.elementAt(center+1).getDist()
>= value){
                              return center+1;
                        }

                        begin = center+1;

                  } else if (neighbors.elementAt(center).getDist() >
value ) {

                        if (center>0 &&
                              neighbors.elementAt(center-1).getDist()
<= value) {
                              return center;
                        }

                        end = center-1;
                  } else {
                        return center;
                  }

            }

            if (neighbors.elementAt(center).getDist() < value) {
                  return center+1;
            } else {
                  return center;
            }

      }
```

```
      /** Searches the training data to find the maximum and minimum
values
       * or the values used in VDM function. **/
      public void evaluateData() {

            //Enumerate instances
            Enumeration<?> enu = m_Input.enumerateInstances();
            while (enu.hasMoreElements()) {

                  //Current instances
                  Instance trainInstance = (Instance)
enu.nextElement();

                  //Searches attributes of the current instance
                  for (int i = 0; i < m_Input.numAttributes(); i++) {


                        //If the attribute is numeric, evaluates the
min and max
                        if ((m_Distance != DIST_HVDM)
                                    &&
(m_Input.attribute(i).isNumeric())
                                    && (!trainInstance.isMissing(i)))
{

                              if (trainInstance.value(i) < m_Min[i])
                                    m_Min[i] =
trainInstance.value(i);
                              if (trainInstance.value(i) > m_Max[i])
                                    m_Max[i] =
trainInstance.value(i);

                        }
                        //If the attribute is nominal and the
distance function is HVDM, counts the number of occurences
                        else if ((m_Distance == DIST_HVDM)
                                    &&
(m_Input.attribute(i).isNominal())
                                    && (!trainInstance.isMissing(i))
) {


      (m_VetVDM.elementAt(i))[(int)trainInstance.value(i)]++;

      (m_MatVDM.elementAt(i))[(int)trainInstance.value(i)][(int)train
Instance.classValue()]++;

                        }

                  }

            }

      }

      /** Calculates the distance between two instances. **/
      public double distance(Instance inst1, Instance inst2, int att)
{
```

```
            double dist = 0;

            //Searches attributes of the instance
            for(int i = 0; i < m_Input.numAttributes(); i++) {

                  //Do not calculate distance of the attribute att
                  if (i == att) {
                        continue;
                  }

                  //If one or both value are missing, sum the maximum
distance
                  if (inst1.isMissing(i) || inst2.isMissing(i)) {
                        dist += 1;
                  } else {

                        //Nominal attribute
                        if (m_Input.attribute(i).isNominal()) {

                              if (m_Distance == DIST_HVDM) {

                                    if ( (int)inst1.value(i) !=
(int)inst2.value(i) )
                                          dist += norm_vdm(inst1,
inst2, i);


                              } else {

                                    if ( (int)inst1.value(i) !=
(int)inst2.value(i) )
                                          dist += 1;

                              }

                        //Numeric attribute
                        } else {

                              if (m_Distance==DIST_HVDM)
                                    dist+=norm_diff(inst1, inst2, i);
                              else {

                                    if (!(Double.isNaN(m_Min[i])) &&
!(Utils.eq(m_Max[i], m_Min[i]))) {


    dist+=range_norm_diff(inst1, inst2, i);

                                    } else {
                                          dist+=1;
                                    }
                              }

                        }

                  }

                  if (!Double.isNaN(m_Bigger) && dist > m_Bigger)
                        return -1;

            }
```

```
            return dist;

      }


      /** Calculates the normalized difference between two numeric
values. **/
      public double norm_diff(Instance inst1, Instance inst2, int
att) {

            if (m_StdDev[att]==0) return 1;

            return Math.pow((inst2.value(att) - inst1.value(att)) /
                        (4*m_StdDev[att]),2);

      }


      /** Calculates the range normalized difference between two
numeric values. **/
      public double range_norm_diff(Instance inst1, Instance inst2,
int att) {

            double d;

            if (m_Max[att]==m_Min[att]) return 1;

            if (m_Distance == DIST_MANHATTAN) {

                  d = Math.abs((inst2.value(att) - inst1.value(att))
/
                                          (m_Max[att]-m_Min[att]));
                  return d;


            } else {

                  d = (inst2.value(att) - inst1.value(att)) /
                              (m_Max[att]-m_Min[att]);
                  return d*d;

            }

      }


      /** Calculates the distance between two nominal values using
the VDM function **/
      public double norm_vdm(Instance inst1, Instance inst2, int att)
{


            int i;
            int ncl =
m_Input.numDistinctValues(m_Input.classIndex());

            double d=0, nx1, nx2;
            double nx1C[] = new double[ncl];
            double nx2C[] = new double[ncl];

            nx1 = m_VetVDM.elementAt(att)[(int)inst1.value(att)];
```

```
                nx2 = m_VetVDM.elementAt(att)[(int)inst2.value(att)];

                nx1C = m_MatVDM.elementAt(att)[(int)inst1.value(att)];
                nx2C = m_MatVDM.elementAt(att)[(int)inst2.value(att)];

                for (i=0;i<ncl;i++) {

                        if (nx1>0 && nx2>0) {

                                d += Math.pow( ((nx1C[i]/nx1) -
        (nx2C[i]/nx2)) , 2.0);


                        } else {

                                return 1.0;

                        }

                }

                //(sqrt(d))^2==d
                return d;

        }


        /** Calculates mode of k-nearest neighbors
         *  considering the distance weighting. **/
        public double mode(int att) {

                int i, aux, k=getKnn();

                int n = m_Input.numAttributes();

                double dist=0,mode=0,maior=0;

                boolean out = false;
                boolean zero = false;

                int t =
m_Input.numDistinctValues(m_Input.attribute(att));
                double vet[] = new double[t];

                for (i=0;!out;i++) {

                        if (i>=neighbors.size()) {
                                out=true;
                                continue;
                        }

                        Neighbor v = neighbors.elementAt(i);

                        if ( i>=k && (v.getDist()-dist>0.0001) ) {
                                out = true;
                                continue;
                        }

                        dist = v.getDist();

                        Instance inst = v.getInst();
```

```
                    aux = (int)inst.value(inst.attribute(att));

                    //In this case, if exists neighbors with distance
0, calculates
                    //the mode only among them
                    if (m_Weighting == WEIGHT_INV) {

                          if (!zero && dist==0)
                                zero = true;


                          if (!zero) {
                                vet[aux] += (1/(dist*dist));
                          } else if (m_Weighting == WEIGHT_SIM) {
                                vet[aux]++;
                          } else {
                                out = true;
                          }


                    } else if (m_Weighting == WEIGHT_SIM) {

                          vet[aux] += (n-dist);

                    } else {

                          vet[aux]++;

                    }

                    if (vet[aux] > maior) {
                          maior = vet[aux];
                          mode = aux;
                    }

              }

              return mode;

        }


        public static void main(String [] argv) {

              try {
                    if (Utils.getFlag('b', argv)) {
                          Filter.batchFilterFile(new EditedNN(), argv);
                    } else {
                          Filter.filterFile(new EditedNN(), argv);
                    }
              } catch (Exception ex) {
                    System.out.println(ex.getMessage());
              }
        }


}
```

Appendix E

Java codes for Neigbhourhood Cleaning Rule (NCL) Class

```
/**
 * Class Neighbourhood Cleaning Rule (NCL)
 *
**/


package weka.filters.supervised.instance;

import java.io.Serializable;
import java.util.*;

import weka.filters.*;
import weka.core.*;
import weka.core.Capabilities.Capability;

public class NeighborhoodCleaning extends Filter implements
SupervisedFilter, OptionHandler {


      /**
       * Generated by Eclipse.
       */
      private static final long serialVersionUID =
8333537819558930169L;


      /** Returns the revision string. **/
      public String getRevision() {
            return RevisionUtils.extract("$Revision: 1.0 $");
      }

      /** Returns default capabilities of the classifier. **/
      public Capabilities getCapabilities() {

            Capabilities result = super.getCapabilities();

            // attributes
            result.enable(Capability.NOMINAL_ATTRIBUTES);
            result.enable(Capability.NUMERIC_ATTRIBUTES);
            result.enable(Capability.DATE_ATTRIBUTES);
            result.enable(Capability.MISSING_VALUES);

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

            // instances
            result.setMinimumNumberInstances(0);

            return result;

      }

      /** Description of the classifier in Weka's graphical mode. **/
      public String globalInfo() {
            return "";
      }
```

```
	/** Class constructors. **/
	public NeighborhoodCleaning(int k) {
		setKnn(k);
		m_Distance = DIST_HEOM;
		m_Weighting = WEIGHT_INV;
	}

	public NeighborhoodCleaning() {
		setKnn(3);
		m_Distance = DIST_HEOM;
		m_Weighting = WEIGHT_INV;
	}



	/**
	* Stores the maximum and minimum values and standard deviation
	* of each attribute (depending on the metric used).
	**/
	private double[] m_Min, m_Max, m_StdDev;

	/** Structures to store data to calculate the VDM metric. **/
	Vector < double[] > m_VetVDM = new Vector<double[]>();
	Vector < double[][] > m_MatVDM = new Vector<double[][]>();

	/** Auxiliary variable **/
	double m_Bigger;


	/** Number of nearest neighbor (k). **/
	private int m_Knn = 3;
	public void setKnn(int m_Knn) { this.m_Knn = m_Knn; }
	public int getKnn() { return this.m_Knn; }

	/**
	 * The degree of bias towards uniform (nominal) class
distribution. -1 means
	 * apply algorithm with no regard to class distribution. 0
means apply the
	 * algorithm, but return the selected majority class examples
plus all the
	 * minority ones. >0 represent the proportion of minority-
majority class
	 * example, n majority examples for one minority example
	 */
	private double m_BiasToUniformClass = -1.0;

	public void setBiasToUniformClass(double m_BiasToUniformClass)
{
		this.m_BiasToUniformClass = m_BiasToUniformClass;
	}

	public double getBiasToUniformClass() {
		return this.m_BiasToUniformClass;
	}

	/** Distance function to be used in the algorithm. **/
	private int m_Distance = DIST_HEOM;
```

```
        public void setDistance(SelectedTag newMethod) {

              if (newMethod.getTags() == TAGS_DISTANCE) {
                this.m_Distance = newMethod.getSelectedTag().getID();
            }

        }

        public SelectedTag getDistance() {
              return new SelectedTag(this.m_Distance, TAGS_DISTANCE);
        }

        public static final int DIST_HEOM = 1;
        public static final int DIST_HVDM = 2;
        public static final int DIST_MANHATTAN = 3;
        public static final Tag [] TAGS_DISTANCE = {
            new Tag(DIST_HEOM, "Heterogeneous Euclidean-Overlap
Metric"),
            new Tag(DIST_MANHATTAN, "Heterogeneous Manhattan-Overlap
Metric"),
            new Tag(DIST_HVDM, "Heterogeneous Value Distance Function")
        };


        /** Distance weighting. **/
        private int m_Weighting;

        public void setWeighting(SelectedTag newMethod) {

              if (newMethod.getTags() == TAGS_WEIGHTING) {
                this.m_Weighting = newMethod.getSelectedTag().getID();
            }

        }

        public SelectedTag getWeighting() {
              return new SelectedTag(this.m_Weighting, TAGS_WEIGHTING);
        }

        public static final int WEIGHT_NONE = 1;
        public static final int WEIGHT_INV = 2;
        public static final int WEIGHT_SIM = 3;
        public static final Tag [] TAGS_WEIGHTING = {
            new Tag(WEIGHT_NONE, "No weight"),
            new Tag(WEIGHT_INV, "1/(distance^2)"),
            new Tag(WEIGHT_SIM, "1-distance")
        };

        /** Definitions and structures to use in this filter. **/
        private Instances m_Input;
        private double m_MajorityClassValue, m_Proportion;
        private Vector<Integer> m_InstancesToRemove = new
Vector<Integer>();

        /** Description of parameters **/
        public String knnTipText() {
              return "Number of nearest neighbors (k).";
        }

        public String distanceTipText() {
              return "Distance function.";
```

```
	}

	public String weightingTipText() {
		return "Distance weighting.";
	}

	/** Parses a given list of options. **/
	public void setOptions(String[] options) throws Exception {

		String knnString = Utils.getOption('K', options);
	    if (knnString.length() != 0) {
		setKnn(Integer.parseInt(knnString));
	    } else {
		setKnn(1);
	    }

	    String biasString = Utils.getOption('B', options);
	    if (biasString.length() != 0) {
		setBiasToUniformClass(Double.parseDouble(biasString));
	    } else {
		setBiasToUniformClass(0);
	    }

		if (Utils.getFlag('V', options)) {
		  setDistance(new SelectedTag(DIST_HVDM, TAGS_DISTANCE));
		} else if (Utils.getFlag('M', options)) {
		  setDistance(new SelectedTag(DIST_MANHATTAN,
TAGS_DISTANCE));
	    } else {
		setDistance(new SelectedTag(DIST_HEOM, TAGS_DISTANCE));
	    }

		if (Utils.getFlag('I', options)) {
			setWeighting(new SelectedTag(WEIGHT_INV,
TAGS_WEIGHTING));
		} else if (Utils.getFlag('S', options)) {
			setWeighting(new SelectedTag(WEIGHT_SIM,
TAGS_WEIGHTING));
		} else {
			setWeighting(new SelectedTag(WEIGHT_NONE,
TAGS_WEIGHTING));
		}

		Utils.checkForRemainingOptions(options);

	}


	/** Gets the current settings of NCL. */
	public String [] getOptions() {

	    String [] options = new String [6];
	    int current = 0;
	    options[current++] = "-K"; options[current++] = "" +
getKnn();

	    options[current++] = "-B"; options[current++] = "" +
getBiasToUniformClass();

	    if (m_Distance == DIST_HVDM) {
		options[current++] = "-V";
```

```
        }

        if (m_Distance == DIST_MANHATTAN) {
          options[current++] = "-M";
        }

        if (m_Weighting == WEIGHT_INV) {
          options[current++] = "-I";
        }

        if (m_Weighting == WEIGHT_SIM) {
          options[current++] = "-S";
        }

        while (current < options.length) {
                options[current++] = "";
          }

        return options;

      }

      /** Returns an enumeration describing the available options.
**/
      public Enumeration<Option> listOptions() {

            Vector<Option> newVector = new Vector<Option>(6);

        newVector.addElement(new Option(
             "\tNumber of nearest neighbors (k).\n"
                +"\t(Default = 1)",
                "K", 1,"-K <number of neighbors>"));

        newVector.addElement(new Option(
                    "\tThe degree of bias towards uniform (nominal)
class distribution.\n"
                    +"\t(Default = 0)",
                    "B", 1,"-B <number>"));

        newVector.addElement(new Option(
                    "\tHeterogeneous Euclidean-Value Distance
Metric.\n",
                    "V", 0, "-V"));

        newVector.addElement(new Option(
                    "\tHeterogeneous Manhattan-Overlap
Metric.\n",
                    "M", 0, "-M"));

        newVector.addElement(new Option(
                    "\tWeight neighbors by inverse of their
squared distance.\n",
                    "I", 0, "-I"));

        newVector.addElement(new Option(
                    "\tWeight neighbors by similarity.\n",
                    "S", 0, "-S"));

        return newVector.elements();
      }
```

```
      /** Vector used to store the neighbors according their
proximity **/
      private Vector<Neighbor> neighbors = new Vector<Neighbor>();

      /** Class that defines a neighbor **/
      protected class Neighbor  implements Serializable  {

            /**
             * Generated by Eclipse
             */
            private static final long serialVersionUID = -
4185966236010333089L;


            double dist;
            Instance inst;
            int index;

            Neighbor(double dist, Instance inst, int index) {
                  this.dist = dist;
                  this.inst = inst;
                  this.index = index;
            }

            public double getDist() { return dist; }
            public Instance getInst() { return inst; }
            public int getIndex() { return index; }

      }



      /** Initializes the input and output formats. **/
      public boolean setInputFormat(Instances instanceInfo) throws
Exception {

            super.setInputFormat(instanceInfo);
            setOutputFormat(instanceInfo);

            m_Input = instanceInfo;

            return true;

      }

    /** Input an instance for filtering. **/
      public boolean input(Instance instance) {

          if (m_Input == null) {
            throw new IllegalStateException("No input instance format
defined");
          }

          if (m_NewBatch) {
            resetQueue();
            m_NewBatch = false;
          }

          if (isFirstBatchDone()) {
                  push(instance);
                  return true;
```

```
            } else {
                  bufferInput(instance);
                  return false;
            }

      }

      /** Signify that this batch of input to the filter is finished.
**/
      public boolean batchFinished() throws Exception {

            if (m_Input == null) {
                  throw new IllegalStateException("No input instance
format defined");
            }

            //Initializes the vectors and determines the initial
values to calculate the HVDM function
          if (m_Distance == DIST_HVDM) {

            m_StdDev = new double [m_Input.numAttributes()];

            for (int i = 0; i < m_Input.numAttributes(); i++) {

                  if (m_Input.attribute(i).isNominal()) {

                        m_VetVDM.add(new
double[m_Input.attribute(i).numValues()]);
                        m_MatVDM.add(new
double[m_Input.attribute(i).numValues()][m_Input.numClasses()]);

                  } else {

                        //Calculates the standard deviation for each
attribute
                         AttributeStats as =
m_Input.attributeStats(i);
                               m_StdDev[i] = as.numericStats.stdDev;

                        m_VetVDM.add(new double[0]);
                        m_MatVDM.add(new double[0][0]);

                  }
            }
          } else {
            m_Min = new double [m_Input.numAttributes()];
                m_Max = new double [m_Input.numAttributes()];

                for (int i=0; i < m_Input.numAttributes(); i++) {
                  m_Min[i] = Double.MAX_VALUE;
                  m_Max[i] = Double.MIN_VALUE;
                }

          }

            evaluateData();

            //Verify the majority class
            int[] classes = new int[m_Input.numClasses()];
```

```
            m_MajorityClassValue = 0.0;
            int counter = 0;

            for(int i = 0; i < m_Input.numInstances(); i++) {

                  classes[(int)m_Input.instance(i).classValue()]++;

                  if (classes[(int)m_Input.instance(i).classValue()]
> counter) {
                        m_MajorityClassValue =
m_Input.instance(i).classValue();
                        counter++;
                  }

            }

            //System.out.println(m_MajorityClassValue + ":" +
m_Bigger);

            if (m_BiasToUniformClass > 0.0) {

                  m_Proportion = (double)(m_Input.numInstances()-
counter) / counter;

                  int aux=0;

                  while (m_Proportion < 1/m_BiasToUniformClass
                              && aux != m_Input.numInstances()) {

                        //System.out.println(m_Input.numInstances() +
":" + counter);
                        //System.out.println(m_Proportion + ":" +
1/m_BiasToUniformClass);

                        aux = m_Input.numInstances();
                        m_InstancesToRemove.clear();

                        for(int i = 0; i < m_Input.numInstances();
i++) {
                              evaluateInstance(m_Input.instance(i),
i);
                        }

                        Collections.sort(m_InstancesToRemove);

                        for (int i = m_InstancesToRemove.size()-1; i
>= 0; i--) {

      //System.out.println(m_Input.instance(m_InstancesToRemove.eleme
ntAt(i)));

      m_Input.delete(m_InstancesToRemove.elementAt(i));
                        }

                        counter = 0;

                        for(int i = 0; i < m_Input.numInstances();
i++) {

                              if (m_Input.instance(i).classValue() ==
m_MajorityClassValue) {
```

```
                                    counter++;

                              }


                        }

                        m_Proportion =
(double)(m_Input.numInstances()-counter) / counter;

                  }

            } else {

                  for(int i = 0; i < m_Input.numInstances(); i++) {
                        evaluateInstance(m_Input.instance(i), i);
                  }

                  Collections.sort(m_InstancesToRemove);

                  for (int i = m_InstancesToRemove.size()-1; i >= 0;
i--) {

      //System.out.println(m_Input.instance(m_InstancesToRemove.eleme
ntAt(i)));

      m_Input.delete(m_InstancesToRemove.elementAt(i));
                  }

            }

            //Convert pending input instances
            for(int i = 0; i < m_Input.numInstances(); i++) {
                  push(m_Input.instance(i));
            }

            //Free memory
            flushInput();

            m_NewBatch = true;
            return (numPendingOutput() != 0);

      }

      /** Evaluate a instance to mark instances to remove. **/
      private void evaluateInstance(Instance instance, int index)
throws Exception {

            double classification = estimate(instance);

            if (instance.classValue() == m_MajorityClassValue) {

                  if (classification != instance.classValue()
                              &&
!m_InstancesToRemove.contains(index)) {
                        m_InstancesToRemove.add(index);
                  }


            } else {
```

```
                  if (classification != instance.classValue()) {

                        for(int i=0; i<m_Knn; i++) {

                              if
(neighbors.elementAt(i).getInst().classValue() ==
m_MajorityClassValue
                                    &&
!m_InstancesToRemove.contains(neighbors.elementAt(i).getIndex())) {

      m_InstancesToRemove.add(neighbors.elementAt(i).getIndex());
                              }

                        }

                  }

            }

      }



      /** Estimate values. **/
      public double estimate(Instance instance) throws Exception {

            double dist;
            int i;
            int att = instance.classIndex();

            neighbors.clear();
            m_Bigger = Double.NaN;

            for (int index = 0; index < m_Input.numInstances();
index++) {

                  //Current instance
                  Instance actualInstance =
(Instance)m_Input.instance(index);

                  if (actualInstance.isMissing(att))
                        continue;


                  dist = distance(instance, actualInstance, att);

                  //If the calculated distance is greater than K-th
neighbor yet calculated, returns -1
                  if (dist>0) {
                        Neighbor v = new Neighbor(dist,
actualInstance, index);

                        if (neighbors.size()==0)
                              neighbors.add(v);
                        else {
                              if ( (neighbors.size()>=m_Knn) )
                                    i = search(0,m_Knn-1,dist);
                              else
                                    i = search(0,neighbors.size()-
1,dist);
```

```
                        neighbors.add(i, v);

                    }

                    if( (neighbors.size()>=m_Knn) )
                        m_Bigger = neighbors.elementAt(m_Knn-
1).getDist();

                }

            }

            return mode(att);

        }

        /** Function that seeks the position where a neighbor should be
inserted into the
         * vector, based on binary search. **/
        public int search(int begin, int end, double value) {

            int center=(int)(begin+end)/2;

            while (begin<=end) {

                center=(int)(begin+end)/2;

                if (neighbors.elementAt(center).getDist() < value )
{

                    if (center<neighbors.size()-1 &&
                        neighbors.elementAt(center+1).getDist()
>= value){
                        return center+1;
                    }

                    begin = center+1;

                } else if (neighbors.elementAt(center).getDist() >
value ) {

                    if (center>0 &&
                        neighbors.elementAt(center-1).getDist()
<= value) {
                        return center;
                    }

                    end = center-1;
                } else {
                    return center;
                }

            }

            if (neighbors.elementAt(center).getDist() < value) {
                return center+1;
            } else {
                return center;
            }
```

```
	}


	/** Searches the training data to find the maximum and minimum
values
	 * or the values used in VDM function. **/
	public void evaluateData() {

		//Enumerate instances
		Enumeration<?> enu = m_Input.enumerateInstances();
		while (enu.hasMoreElements()) {

			//Current instances
			Instance trainInstance = (Instance)
enu.nextElement();

			//Searches attributes of the current instance
			for (int i = 0; i < m_Input.numAttributes(); i++) {


				//If the attribute is numeric, evaluates the
min and max
				if ((m_Distance != DIST_HVDM)
						&&
(m_Input.attribute(i).isNumeric())
						&& (!trainInstance.isMissing(i)))
{

					if (trainInstance.value(i) < m_Min[i])
						m_Min[i] =
trainInstance.value(i);
					if (trainInstance.value(i) > m_Max[i])
						m_Max[i] =
trainInstance.value(i);

				}
				//If the attribute is nominal and the
distance function is HVDM, counts the number of occurences
				else if ((m_Distance == DIST_HVDM)
						&&
(m_Input.attribute(i).isNominal())
						&& (!trainInstance.isMissing(i))
) {


	(m_VetVDM.elementAt(i))[(int)trainInstance.value(i)]++;

	(m_MatVDM.elementAt(i))[(int)trainInstance.value(i)][(int)train
Instance.classValue()]++;

				}

			}

		}

	}

	/** Calculates the distance between two instances. **/
	public double distance(Instance inst1, Instance inst2, int att)
{
```

```
            double dist = 0;

            //Searches attributes of the instance
            for(int i = 0; i < m_Input.numAttributes(); i++) {

                  //Do not calculate distance of the attribute att
                  if (i == att) {
                        continue;
                  }

                  //If one or both value are missing, sum the maximum
distance
                  if (inst1.isMissing(i) || inst2.isMissing(i)) {
                        dist += 1;
                  } else {

                        //Nominal attribute
                        if (m_Input.attribute(i).isNominal()) {

                              if (m_Distance == DIST_HVDM) {

                                    if ( (int)inst1.value(i) !=
(int)inst2.value(i) )
                                          dist += norm_vdm(inst1,
inst2, i);


                              } else {

                                    if ( (int)inst1.value(i) !=
(int)inst2.value(i) )
                                          dist += 1;

                              }

                        //Numeric attribute
                        } else {

                              if (m_Distance==DIST_HVDM)
                                    dist+=norm_diff(inst1, inst2, i);
                              else {

                                    if (!(Double.isNaN(m_Min[i])) &&
!(Utils.eq(m_Max[i], m_Min[i]))) {


      dist+=range_norm_diff(inst1, inst2, i);

                                    } else {
                                          dist+=1;
                                    }
                              }

                        }

                  }

                  if (!Double.isNaN(m_Bigger) && dist > m_Bigger)
                        return -1;
```

```
            }

            return dist;

      }


      /** Calculates the normalized difference between two numeric
values. **/
      public double norm_diff(Instance inst1, Instance inst2, int
att) {

            if (m_StdDev[att]==0) return 1;

            return Math.pow((inst2.value(att) - inst1.value(att)) /
                        (4*m_StdDev[att]),2);

      }


      /** Calculates the range normalized difference between two
numeric values. **/
      public double range_norm_diff(Instance inst1, Instance inst2,
int att) {

            double d;

            if (m_Max[att]==m_Min[att]) return 1;

            if (m_Distance == DIST_MANHATTAN) {

                  d = Math.abs((inst2.value(att) - inst1.value(att))
/
                                                (m_Max[att]-m_Min[att]));
                  return d;


            } else {

                  d = (inst2.value(att) - inst1.value(att)) /
                              (m_Max[att]-m_Min[att]);
                  return d*d;

            }

      }


      /** Calculates the distance between two nominal values using
the VDM function **/
      public double norm_vdm(Instance inst1, Instance inst2, int att)
{

            int i;
            int ncl =
m_Input.numDistinctValues(m_Input.classIndex());

            double d=0, nx1, nx2;
            double nx1C[] = new double[ncl];
            double nx2C[] = new double[ncl];
```

```
                nx1 = m_VetVDM.elementAt(att)[(int)inst1.value(att)];
                nx2 = m_VetVDM.elementAt(att)[(int)inst2.value(att)];

                nx1C = m_MatVDM.elementAt(att)[(int)inst1.value(att)];
                nx2C = m_MatVDM.elementAt(att)[(int)inst2.value(att)];

                for (i=0;i<ncl;i++) {

                        if (nx1>0 && nx2>0) {

                                d += Math.pow( ((nx1C[i]/nx1) -
        (nx2C[i]/nx2)) , 2.0);


                        } else {

                                return 1.0;

                        }

                }

                //(sqrt(d))^2==d
                return d;

        }


        /** Calculates mode of k-nearest neighbors
         *  considering the distance weighting. **/
        public double mode(int att) {

                int i, aux, k=getKnn();

                int n = m_Input.numAttributes();

                double dist=0,mode=0,maior=0;

                boolean out = false;
                boolean zero = false;

                int t =
m_Input.numDistinctValues(m_Input.attribute(att));
                double vet[] = new double[t];

                for (i=0;!out;i++) {

                        if (i>=neighbors.size()) {
                                out=true;
                                continue;
                        }

                        Neighbor v = neighbors.elementAt(i);

                        if ( i>=k && (v.getDist()-dist>0.0001) ) {
                                out = true;
                                continue;
                        }

                        dist = v.getDist();
```

```
                    Instance inst = v.getInst();

                    aux = (int)inst.value(inst.attribute(att));

                    //In this case, if exists neighbors with distance
0, calculates
                    //the mode only among them
                    if (m_Weighting == WEIGHT_INV) {

                          if (!zero && dist==0)
                                zero = true;


                          if (!zero) {
                                vet[aux] += (1/(dist*dist));
                          } else if (m_Weighting == WEIGHT_SIM) {
                                vet[aux]++;
                          } else {
                                out = true;
                          }


                    } else if (m_Weighting == WEIGHT_SIM) {

                          vet[aux] += (n-dist);

                    } else {

                          vet[aux]++;

                    }

                    if (vet[aux] > maior) {
                          maior = vet[aux];
                          mode = aux;
                    }

              }

              return mode;

        }


        public static void main(String [] argv) {

              try {
                    if (Utils.getFlag('b', argv)) {
                          Filter.batchFilterFile(new
NeighborhoodCleaning(), argv);
                    } else {
                          Filter.filterFile(new NeighborhoodCleaning(),
argv);
                    }
              } catch (Exception ex) {
                    System.out.println(ex.getMessage());
              }
        }

}
```

Appendix F

Java codes for Condense Nearest Neighbour (**CNN) Class**

```
/**
 * Class Condense Nearest Neighbour (CNN)
 *
**/

package weka.filters.supervised.instance;

import weka.classifiers.Evaluation;
import weka.classifiers.lazy.IBk;
import weka.core.Capabilities;
import weka.core.DistanceFunction;
import weka.core.EuclideanDistance;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.Option;
import weka.core.OptionHandler;
import weka.core.RevisionUtils;
import weka.core.Utils;
import weka.core.Capabilities.Capability;
import weka.filters.Filter;
import weka.filters.SupervisedFilter;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Enumeration;
import java.util.Random;
import java.util.Vector;

public class CNN extends Filter implements SupervisedFilter,
OptionHandler {

      /** The subsample size, percent of original set, default 100%.
*/
      /*
       * protected double m_SampleSizePercent = 100;
       */
      /** The random number generator seed. */
      protected int m_RandomSeed = 1;

      /**
       * The degree of bias towards uniform (nominal) class
distribution. -1 means
       * apply algorithm with no regard to class distribution 0 means
apply the
       * algorithm, but return the selected majority class examples
plus all the
       * minority ones >0 represent the proportion of minority-
majority class
       * example, n majority examples for one minority example
```

```
         */
        protected double m_BiasToUniformClass = 0;

        /**
         * Whether to invert the selection (only if instances are drawn
WITHOUT
         * replacement).
         *
         * @see #m_NoReplacement
         */
        protected boolean m_InvertSelection = false;

        /**
         * Which variant of CNN algorithm is used. Basic means basic
algorithm as
         * presented in Hart67. Other options are tomek, corresponds to
method 2 in
         * Tomek 76.
         */
        protected double variant = -1;

        protected int max, min=0;

        /**
         * Returns a string describing this filter.
         *
         * @return a description of the filter suitable for displaying
in the
         *         explorer/experimenter gui
         */
        public String globalInfo() {
              return "Produces a supervised subsample of a dataset
based on Cnn algotirthm "
                          + "The original dataset must "
                          + "fit entirely in memory. The number of
instances in the generated "
                          + "dataset may be specified. The dataset must
have a nominal class "
                          + "attribute.";
        }

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

              result.addElement(new Option(
                          "\tSpecify the random number seed (default
1)", "S", 1,
                          "-S <num>"));

              result.addElement(new Option(
                          "\tThe size of the output dataset, as a
percentage of\n"
                                      + "\tthe input dataset (default
100)", "Z", 1,
                          "-Z <num>"));
```

```
            result
                        .addElement(new Option(
                                        "\tBias factor towards uniform
class distribution.\n"
                                                + "\t-1 = Cnn will
act regardless class distribution --"
                                                + " n = indicate the
class distribution bias.\n"
                                                + "\t(default 0)",
"B", 1, "-B <num>"));

            result.addElement(new Option("\tDisables replacement of
instances\n"
                        + "\t(default: with replacement)", "no-
replacement", 0,
                        "-no-replacement"));

            result
                        .addElement(new Option(
                                        "\tInverts the selection - only
available with '-no-replacement'.",
                                        "V", 0, "-V"));

            result.addElement(new Option("\tAlgorithm
modification.\n"
                        + "\tbasic = Cnn basic algorithm--"
                        + " tomek= tomek link modification.\n" +
"\t(default basic)",
                        "M", 1, "-M <basic/tomek>"));

            return result.elements();
      }

      /**
       * Parses a given list of options.
       * <p/>
       *
       * <!-- options-start --> Valid options are:
       * <p/>
       *
       * <pre>
       *  -S &lt;num&gt;
       *  Specify the random number seed (default 1)
       * </pre>
       *
       * <pre>
       *  -Z &lt;num&gt;
       *  The size of the output dataset, as a percentage of
       *  the input dataset (default 100)
       * </pre>
       *
       * <pre>
       *  -B &lt;num&gt;
       *  Bias factor towards uniform class distribution.
       *  0 = distribution in input data -- 1 = uniform distribution.
       *  (default 0)
       * </pre>
       *
       * <pre>
       *  -no-replacement
       *  Disables replacement of instances
```

```
         *  (default: with replacement)
         * </pre>
         *
         * <pre>
         *  -V
         *  Inverts the selection - only available with '-no-
replacement'.
         * </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 tmpStr;

              tmpStr = Utils.getOption('S', options);
              if (tmpStr.length() != 0)
                    setRandomSeed(Integer.parseInt(tmpStr));
              else
                    setRandomSeed(1);

              tmpStr = Utils.getOption('B', options);

              if (tmpStr.length() != 0)
                    setBiasToUniformClass(Double.parseDouble(tmpStr));
              else
                    setBiasToUniformClass(0);

              tmpStr = Utils.getOption('V', options);
              if (tmpStr.length() != 0)
                    setVariant(Double.parseDouble(tmpStr));
              else
                    setVariant(-1);

              /*
               * tmpStr = Utils.getOption('Z', options); if
(tmpStr.length() != 0)
               * setSampleSizePercent(Double.parseDouble(tmpStr)); else
               * setSampleSizePercent(100);
               */

              /*
               * setNoReplacement(Utils.getFlag("no-replacement",
options));
               *
               * if (getNoReplacement())
setInvertSelection(Utils.getFlag('V',
               * options));
               */

              if (getInputFormat() != null) {
                    setInputFormat(getInputFormat());
              }
        }

        /**
         * 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("-B");
             result.add("" + getBiasToUniformClass());

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

             result.add("-V");
             result.add("" + getVariant());

             /*
              * result.add("-Z"); result.add("" +
getSampleSizePercent());
              */

             /*
              * if (getNoReplacement()) { result.add("-no-
replacement"); if
              * (getInvertSelection()) result.add("-V"); }
              */

             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 biasToUniformClassTipText() {
             return "Whether to use bias towards a uniform class. A
value of 0 leaves the class "
                    + "distribution as-is, a value of 1 ensures
the class distribution is "
                    + "uniform in the output data.";
       }

       /**
        * Gets the bias towards a uniform class. A value of 0 leaves
the class
        * distribution as-is, a value of 1 ensures the class
distributions are
        * uniform in the output data.
        *
        * @return the current bias
        */
       public double getBiasToUniformClass() {
             return m_BiasToUniformClass;
       }

       /**
```

```
	 * Sets the bias towards a uniform class. A value of 0 leaves
the class
	 * distribution as-is, a value of 1 ensures the class
distributions are
	 * uniform in the output data.
	 *
	 * @param newBiasToUniformClass
	 *            the new bias value, between 0 and 1.
	 */
	public void setBiasToUniformClass(double newBiasToUniformClass)
{
		m_BiasToUniformClass = newBiasToUniformClass;
	}

	/**
	 * Gets variant value.
	 *
	 * @return variant.
	 */
	public double getVariant() {

		return variant;
	}

	public void setVariant(double var) {

		variant = var;
	}

	/**
	 * 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 "Sets the random number seed for subsampling.";
	}

	/**
	 * Gets the random number seed.
	 *
	 * @return the random number seed.
	 */
	public int getRandomSeed() {
		return m_RandomSeed;
	}

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

	/**
	 * Returns the tip text for this property.
```

```
         *
         * @return tip text for this property suitable for displaying
in the
         *         explorer/experimenter gui
         */
        public String sampleSizePercentTipText() {
              return "The subsample size as a percentage of the
original set.";
        }

        /**
         * Gets the subsample size as a percentage of the original set.
         *
         * @return the subsample size
         */
        /*
         * public double getSampleSizePercent() { return
m_SampleSizePercent; }
         */

        /**
         * Sets the size of the subsample, as a percentage of the
original set.
         *
         * @param newSampleSizePercent
         *            the subsample set size, between 0 and 100.
         */
        /*
         * public void setSampleSizePercent(double
newSampleSizePercent) {
         * m_SampleSizePercent = newSampleSizePercent; }
         */

        /**
         * Returns the tip text for this property.
         *
         * @return tip text for this property suitable for displaying
in the
         *         explorer/experimenter gui
         */
        public String invertSelectionTipText() {
              return "Inverts the selection (only if instances are
drawn WITHOUT replacement).";
        }

        /**
         * Gets whether selection is inverted (only if instances are
drawn WIHTOUT
         * replacement).
         *
         * @return true if the replacement is disabled
         * @see #m_NoReplacement
         */
        public boolean getInvertSelection() {
              return m_InvertSelection;
        }

        /**
         * Sets whether the selection is inverted (only if instances
are drawn
         * WIHTOUT replacement).
```

```
         *
         * @param value
         *            if true then selection is inverted
         */
        public void setInvertSelection(boolean value) {
              m_InvertSelection = value;
        }

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

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

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

              return result;
        }

        /**
         * 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);
              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 (isFirstBatchDone()) {
			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 Exception
	 * @throws IllegalStateException
	 *             if no input structure has been defined
	 */
	public boolean batchFinished() throws Exception {

		if (getInputFormat() == null) {
			throw new IllegalStateException("No input instance
format defined");
		}

		if (!isFirstBatchDone()) {
			// Do the subsample, and clear the input instances.
			createSubsample();
		}
		flushInput();

		m_NewBatch = true;
		m_FirstBatchDone = true;
		return (numPendingOutput() != 0);
	}

	@SuppressWarnings("unchecked")
	protected void createSubsample() throws Exception {

		// Sort according to class attribute, instances with
missing values at the end.
		getInputFormat().sort(getInputFormat().classIndex());

		// Create an index of where each class value starts
		int[] classIndices = new
int[getInputFormat().numClasses() + 1];
		int currentClass = 0;
		classIndices[currentClass] = 0;
```

```
            for (int i = 0; i < getInputFormat().numInstances(); i++)
{
                  Instance current = getInputFormat().instance(i);

                  if (current.classIsMissing()) {
                        for (int j = currentClass + 1; j <
classIndices.length; j++) {
                              classIndices[j] = i;
                        }
                        break;
                  } else if (current.classValue() != currentClass) {
                        for (int j = currentClass + 1; j <=
current.classValue(); j++) {
                              classIndices[j] = i;
                        }
                        currentClass = (int) current.classValue();
                  }
            }//end for

            if (currentClass <= getInputFormat().numClasses()) {
                  for (int j = currentClass + 1; j <
classIndices.length; j++) {
                        classIndices[j] =
getInputFormat().numInstances();
                  }
            }

            Vector<Integer>[] indices = new
Vector[classIndices.length - 1];
            Vector<Integer>[] indicesNew = new
Vector[classIndices.length - 1];

            // generate list of all indices to draw from, indices[0]
list the indexes of first class,
            //indices[1]lists indexes of the second class. IndicesNew
is empty and has the  same capacity of indeces
            for (int i = 0; i < classIndices.length - 1; i++) {
                  indices[i] = new Vector<Integer>(classIndices[i +
1]
                              - classIndices[i]);
                  indicesNew[i] = new
Vector<Integer>(indices[i].capacity());
                  for (int n = classIndices[i]; n < classIndices[i +
1]; n++) {
                        indices[i].add(n);
                  }
            }//end for


                  cnn(indices, indicesNew);

      }


      //work for just two classes
      protected void cnn(Vector<Integer>[] indices, Vector<Integer>[]
indicesNew)
                  throws Exception {

            double classMinority = 0;
```

```
            int min = 0, max = 0;
            indices[0].trimToSize();
            indices[1].trimToSize();

            // indicate the index of maiority and minority class
            if (indices[0].size() > indices[1].size()) {

                  min = 1;
                  max = 0;
            } else {

                  min = 0;
                  max = 1;
            }

            //store the class value of minority class
            classMinority =
getInputFormat().instance(indices[min].get(1))
                        .classValue();

            Instances original = new Instances(getInputFormat());
        Instances temp = new Instances(getInputFormat());
            temp.delete();

            Instances subsample = new Instances(getInputFormat());
            subsample.delete();

            ArrayList<Instance> majoritySample = new
ArrayList<Instance>();

            Instance in;
            int count;
            int random;
            boolean goon = true;

            while (goon) {
                  random = new
Random().nextInt(original.numInstances());
                  in = original.instance(random);
                  original.delete(random);

                  subsample.add(in);

                  if (in.classValue() != classMinority)
                        majoritySample.add(in);
                  count = 0;

                  while (original.numInstances() > 0) {
                        random = new
Random().nextInt(original.numInstances());
                        in = original.instance(random);
                        IBk ibk = new IBk(1);
                        ibk.buildClassifier(subsample);

                        Evaluation eval = new Evaluation(subsample);
                        double ev = eval.evaluateModelOnce(ibk, in);
                        if (ev == in.classValue()) {
                              temp.add(in);
                        } else {
                              subsample.add(in);
                              count++;
```

```
                              if (in.classValue() != classMinority)
                                    majoritySample.add(in);

                        }
                        original.delete(random);
                  } // end while
                  if (count == 0)
                        goon = false;
                  else {
                        original = new Instances(temp);
                        temp.delete();
                  }
            }// end external while


            // apply CNN with no regard to class distribution
            if (variant < 0) {
                  for (int k = 0; k < subsample.numInstances(); k++)
{
                        Instance out = new
Instance(subsample.instance(k));
                        push((Instance) out.copy());
                  }

      //apply CNN all minority from the original dataset plus all
majority after CNN
            } else if (variant == 0) {
                  for (int k = 0; k < subsample.numInstances(); k++)
{
                        Instance out = new
Instance(subsample.instance(k));
                        out.setDataset(getInputFormat());
                        if (out.classValue() != classMinority)
                              push((Instance) out.copy());
                  }
                  for (int k = 0; k < indices[min].size(); k++) {
                        Instance out = new
Instance(getInputFormat().instance(
                                    indices[min].get(k)));
                        push((Instance) out.copy());
                  }

            //apply CNN all minority from the original dataset plus
majority but following a proportion
            } else if (variant > 0) {
                  int size = (int) (indices[min].size() *
m_BiasToUniformClass);
                  if (majoritySample.size() >= size) {
                        for (int k = majoritySample.size() - 1; k >
majoritySample
                                    .size()
                                    - size - 1; k--) {
                              push((Instance)
majoritySample.get(k).copy());

                        }

                        for (int k = 0; k < indices[min].size(); k++)
{
```

```
                                    Instance out = new
Instance(getInputFormat().instance(
                                                indices[min].get(k)));
                                    push((Instance) out.copy());
                              }

                        } else {
                              for (int k = 0; k < subsample.numInstances();
k++) {
                                    Instance out = new
Instance(subsample.instance(k));
                                    out.setDataset(getInputFormat());
                                    if (out.classValue() != classMinority)
                                          push((Instance) out.copy());
                              }
                              for (int k = 0; k < indices[min].size(); k++)
{
                                    Instance out = new
Instance(getInputFormat().instance(
                                                indices[min].get(k)));
                                    push((Instance) out.copy());
                              }

                        }
                  }
            }


            public String getRevision() {
                  // TODO Auto-generated method stub
                  return null;
            }

            public static void main(String[] args) {

                  CNN cnn = new CNN();
                  //cnn.setBiasToUniformClass(1);
                  //cnn.setVariant("tomek");
                  try {
                        cnn.setOptions(weka.core.Utils.splitOptions("-M
tomek -B 0"));
                  } catch (Exception e3) {
                        // TODO Auto-generated catch block
                        e3.printStackTrace();
                  }
                  Instances data = null;
                  Instances newData = null;

                  try {
                        data = new Instances(new BufferedReader(new
FileReader(new File(
                                    args[0])));
                  } catch (FileNotFoundException e2) {
                        // TODO Auto-generated catch block
                        e2.printStackTrace();
                  } catch (IOException e2) {
                        // TODO Auto-generated catch block
                        e2.printStackTrace();
                  }
                  data.setClassIndex(data.numAttributes() - 1);
```

```
	//	System.out.println("data n instances " +
data.numInstances());
			try {
				cnn.setInputFormat(data);
			} catch (Exception e1) {
				// TODO Auto-generated catch block
				e1.printStackTrace();
			}

			try {
				newData = Filter.useFilter(data, cnn);
			} catch (Exception e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
			BufferedWriter br = null;
			try {
				br = new BufferedWriter(new
FileWriter("CnnOutput.arff"));
			} catch (IOException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
			//System.out.println(data.numInstances());
	//	System.out.println("dataset size after sampling:
"+newData.numInstances());
			try {
				br.write(newData.toString());
				br.close();
			} catch (IOException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
	//	System.out.println(newData.attributeStats(100));
		}

}
```

Appendix G

Java codes for Random Under Sampling (RUS) **Class**

```
/*
 *    Class Random Under-sampling
*
 */

package weka.filters.supervised.instance;

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.UnassignedClassException;
import weka.core.UnsupportedClassTypeException;
import weka.core.Utils;
import weka.core.Capabilities.Capability;
import weka.filters.Filter;
import weka.filters.SupervisedFilter;

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

/**
 <!-- globalinfo-start -->
 * Produces a random subsample of a dataset. The original dataset
must fit entirely in memory. This filter allows you to specify the
maximum "spread" between the rarest and most common class. For
example, you may specify that there be at most a 2:1 difference in
class frequencies. When used in batch mode, subsequent batches are
NOT resampled.
 * <p/>
 <!-- globalinfo-end -->
 *
 <!-- options-start -->
 * Valid options are: <p/>
 *
 * <pre> -S &lt;num&gt;
 *  Specify the random number seed (default 1)</pre>
 *
 * <pre> -M &lt;num&gt;
 *  The maximum class distribution spread.
 *  0 = no maximum spread, 1 = uniform distribution, 10 = allow at
most
 *  a 10:1 ratio between the classes (default 0)</pre>
 *
 * <pre> -W
 *  Adjust weights so that total weight per class is maintained.
 *  Individual instance weighting is not preserved. (default no
 *  weights adjustment</pre>
 *
 * <pre> -X &lt;num&gt;
 *  The maximum count for any class value (default 0 = unlimited).
 * </pre>
```

```
 *
 <!-- options-end -->
 **/
public class RUS
  extends Filter
  implements SupervisedFilter, OptionHandler {

  /** for serialization */
  static final long serialVersionUID = -3947033795243930016L;

  /** The random number generator seed */
  private int m_RandomSeed = 1;

  /** The maximum count of any class */
  private int m_MaxCount;

  /** True if the first batch has been done */
  private double m_DistributionSpread = 0;

  /**
   * True if instance weights will be adjusted to maintain
   * total weight per class.
   */
  private boolean m_AdjustWeights = false;

  /**
   * Returns a string describing this filter
   *
   * @return a description of the filter suitable for
   * displaying in the explorer/experimenter gui
   */
  public String globalInfo() {

    return "Produces a random subsample of a dataset. The original
dataset must "
      + "fit entirely in memory. This filter allows you to specify
the maximum "
      + "\"spread\" between the rarest and most common class. For
example, you may "
      + "specify that there be at most a 2:1 difference in class
frequencies. "
      + "When used in batch mode, subsequent batches are NOT
resampled.";

  }

  /**
   * Returns the tip text for this property
   *
   * @return tip text for this property suitable for
   * displaying in the explorer/experimenter gui
   */
  public String adjustWeightsTipText() {
    return "Wether instance weights will be adjusted to maintain
total weight per "
      + "class.";
  }

  /**
   * Returns true if instance  weights will be adjusted to maintain
   * total weight per class.
```

```
     *
     * @return true if instance weights will be adjusted to maintain
     * total weight per class.
     */
    public boolean getAdjustWeights() {

      return m_AdjustWeights;
    }

    /**
     * Sets whether the instance weights will be adjusted to maintain
     * total weight per class.
     *
     * @param newAdjustWeights whether to adjust weights
     */
    public void setAdjustWeights(boolean newAdjustWeights) {

      m_AdjustWeights = newAdjustWeights;
    }

    /**
     * Returns an enumeration describing the available options.
     *
     * @return an enumeration of all the available options.
     */
    public Enumeration listOptions() {

      Vector newVector = new Vector(4);

      newVector.addElement(new Option(
                "\tSpecify the random number seed (default 1)",
                "S", 1, "-S <num>"));
      newVector.addElement(new Option(
                "\tThe maximum class distribution spread.\n"
                +"\t0 = no maximum spread, 1 = uniform distribution, 10
= allow at most\n"
              +"\ta 10:1 ratio between the classes (default 0)",
                "M", 1, "-M <num>"));
      newVector.addElement(new Option(
                "\tAdjust weights so that total weight per class is
maintained.\n"
                +"\tIndividual instance weighting is not preserved.
(default no\n"
                +"\tweights adjustment",
                "W", 0, "-W"));
      newVector.addElement(new Option(
                "\tThe maximum count for any class value (default 0 =
unlimited).\n",
                "X", 0, "-X <num>"));

      return newVector.elements();
    }


    /**
     * Parses a given list of options. <p/>
     *
     <!-- options-start -->
     * Valid options are: <p/>
     *
     * <pre> -S &lt;num&gt;
```

```
   *  Specify the random number seed (default 1)</pre>
   *
   * <pre> -M &lt;num&gt;
   *  The maximum class distribution spread.
   *  0 = no maximum spread, 1 = uniform distribution, 10 = allow at
most
   *  a 10:1 ratio between the classes (default 0)</pre>
   *
   * <pre> -W
   *  Adjust weights so that total weight per class is maintained.
   *  Individual instance weighting is not preserved. (default no
   *  weights adjustment</pre>
   *
   * <pre> -X &lt;num&gt;
   *  The maximum count for any class value (default 0 = unlimited).
   * </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 seedString = Utils.getOption('S', options);
    if (seedString.length() != 0) {
      setRandomSeed(Integer.parseInt(seedString));
    } else {
      setRandomSeed(1);
    }

    String maxString = Utils.getOption('M', options);
    if (maxString.length() != 0) {
      setDistributionSpread(Double.valueOf(maxString).doubleValue());
    } else {
      setDistributionSpread(0);
    }

    String maxCount = Utils.getOption('X', options);
    if (maxCount.length() != 0) {
      setMaxCount(Double.valueOf(maxCount).doubleValue());
    } else {
      setMaxCount(0);
    }

    setAdjustWeights(Utils.getFlag('W', options));

    if (getInputFormat() != null) {
      setInputFormat(getInputFormat());
    }
  }

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

    String [] options = new String [7];
    int current = 0;
```

```
    options[current++] = "-M";
    options[current++] = "" + getDistributionSpread();

    options[current++] = "-X";
    options[current++] = "" + getMaxCount();

    options[current++] = "-S";
    options[current++] = "" + getRandomSeed();

    if (getAdjustWeights()) {
      options[current++] = "-W";
    }

    while (current < options.length) {
      options[current++] = "";
    }
    return options;
  }

  /**
   * Returns the tip text for this property
   *
   * @return tip text for this property suitable for
   * displaying in the explorer/experimenter gui
   */
  public String distributionSpreadTipText() {
    return "The maximum class distribution spread. "
      + "(0 = no maximum spread, 1 = uniform distribution, 10 = allow
at most a "
      + "10:1 ratio between the classes).";
  }

  /**
   * Sets the value for the distribution spread
   *
   * @param spread the new distribution spread
   */
  public void setDistributionSpread(double spread) {

    m_DistributionSpread = spread;
  }

  /**
   * Gets the value for the distribution spread
   *
   * @return the distribution spread
   */
  public double getDistributionSpread() {

    return m_DistributionSpread;
  }

  /**
   * Returns the tip text for this property
   *
   * @return tip text for this property suitable for
   * displaying in the explorer/experimenter gui
   */
  public String maxCountTipText() {
    return "The maximum count for any class value (0 = unlimited).";
```

```
}

/**
 * Sets the value for the max count
 *
 * @param maxcount the new max count
 */
public void setMaxCount(double maxcount) {

  m_MaxCount = (int)maxcount;
}

/**
 * Gets the value for the max count
 *
 * @return the max count
 */
public double getMaxCount() {

  return m_MaxCount;
}

/**
 * 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 "Sets the random number seed for subsampling.";
}

/**
 * Gets the random number seed.
 *
 * @return the random number seed.
 */
public int getRandomSeed() {

  return m_RandomSeed;
}

/**
 * Sets the random number seed.
 *
 * @param newSeed the new random number seed.
 */
public void setRandomSeed(int newSeed) {

  m_RandomSeed = newSeed;
}

/**
 * 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);

    return result;
  }

  /**
   * 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 UnassignedClassException if no class attribute has been
set.
   * @throws UnsupportedClassTypeException if the class attribute
   * is not nominal.
   */
  public boolean setInputFormat(Instances instanceInfo)
      throws Exception {

    super.setInputFormat(instanceInfo);
    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 (isFirstBatchDone()) {
      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
   */
  public boolean batchFinished() {

    if (getInputFormat() == null) {
      throw new IllegalStateException("No input instance format
defined");
    }

    if (!isFirstBatchDone()) {
      // Do the subsample, and clear the input instances.
      createSubsample();
    }

    flushInput();
    m_NewBatch = true;
    m_FirstBatchDone = true;
    return (numPendingOutput() != 0);
  }


  /**
   * Creates a subsample of the current set of input instances. The
output
   * instances are pushed onto the output queue for collection.
   */
  private void createSubsample() {

    int classI = getInputFormat().classIndex();
    // Sort according to class attribute.
    getInputFormat().sort(classI);
    // Determine where each class starts in the sorted dataset
    int [] classIndices = getClassIndices();

    // Get the existing class distribution
    int [] counts = new int [getInputFormat().numClasses()];
    double [] weights = new double [getInputFormat().numClasses()];
    int min = -1;
    for (int i = 0; i < getInputFormat().numInstances(); i++) {
      Instance current = getInputFormat().instance(i);
      if (current.classIsMissing() == false) {
        counts[(int)current.classValue()]++;
        weights[(int)current.classValue()]+= current.weight();
      }
    }

    // Convert from total weight to average weight
    for (int i = 0; i < counts.length; i++) {
      if (counts[i] > 0) {
        weights[i] = weights[i] / counts[i];
      }
      /*
      System.err.println("Class:" + i + " " +
getInputFormat().classAttribute().value(i)
                         + " Count:" + counts[i]
```

```
                          + " Total:" + weights[i] * counts[i]
                          + " Avg:" + weights[i]);
      */
    }

    // find the class with the minimum number of instances
    int minIndex = -1;
    for (int i = 0; i < counts.length; i++) {
      if ( (min < 0) && (counts[i] > 0) ) {
        min = counts[i];
        minIndex = i;
      } else if ((counts[i] < min) && (counts[i] > 0)) {
        min = counts[i];
        minIndex = i;
      }
    }

    if (min < 0) {
      System.err.println("SpreadSubsample: *warning* none of the
classes have any values in them.");
      return;
    }

    // determine the new distribution
    int [] new_counts = new int [getInputFormat().numClasses()];
    for (int i = 0; i < counts.length; i++) {
      new_counts[i] = (int)Math.abs(Math.min(counts[i],
                                             min *
m_DistributionSpread));
      if (i == minIndex) {
        if (m_DistributionSpread > 0 && m_DistributionSpread < 1.0) {
          // don't undersample the minority class!
          new_counts[i] = counts[i];
        }
      }
      if (m_DistributionSpread == 0) {
        new_counts[i] = counts[i];
      }

      if (m_MaxCount > 0) {
        new_counts[i] = Math.min(new_counts[i], m_MaxCount);
      }
    }

    // Sample without replacement
    Random random = new Random(m_RandomSeed);
    Hashtable t = new Hashtable();
    for (int j = 0; j < new_counts.length; j++) {
      double newWeight = 1.0;
      if (m_AdjustWeights && (new_counts[j] > 0)) {
        newWeight = weights[j] * counts[j] / new_counts[j];
        /*
        System.err.println("Class:" + j + " " +
getInputFormat().classAttribute().value(j)
                           + " Count:" + counts[j]
                           + " Total:" + weights[j] * counts[j]
                           + " Avg:" + weights[j]
                           + " NewCount:" + new_counts[j]
                           + " NewAvg:" + newWeight);
        */
      }
```

```
      for (int k = 0; k < new_counts[j]; k++) {
        boolean ok = false;
        do {
        int index = classIndices[j] + (Math.abs(random.nextInt())
                                          % (classIndices[j + 1] -
classIndices[j])) ;
        // Have we used this instance before?
          if (t.get("" + index) == null) {
            // if not, add it to the hashtable and use it
            t.put("" + index, "");
            ok = true;
          if(index >= 0) {
              Instance newInst =
(Instance)getInputFormat().instance(index).copy();
              if (m_AdjustWeights) {
                newInst.setWeight(newWeight);
              }
              push(newInst);
            }
          }
        } while (!ok);
      }
    }
  }

  /**
   * Creates an index containing the position where each class starts
in
   * the getInputFormat(). m_InputFormat must be sorted on the class
attribute.
   *
   * @return the positions
   */
  private int[] getClassIndices() {

    // Create an index of where each class value starts
    int [] classIndices = new int [getInputFormat().numClasses() +
1];
    int currentClass = 0;
    classIndices[currentClass] = 0;
    for (int i = 0; i < getInputFormat().numInstances(); i++) {
      Instance current = getInputFormat().instance(i);
      if (current.classIsMissing()) {
        for (int j = currentClass + 1; j < classIndices.length; j++)
{
          classIndices[j] = i;
        }
        break;
      } else if (current.classValue() != currentClass) {
        for (int j = currentClass + 1; j <= current.classValue();
j++) {
          classIndices[j] = i;
        }
        currentClass = (int) current.classValue();
      }
    }
    if (currentClass <= getInputFormat().numClasses()) {
      for (int j = currentClass + 1; j < classIndices.length; j++) {
        classIndices[j] = getInputFormat().numInstances();
      }
    }
```

```
    return classIndices;
  }

  /**
   * Returns the revision string.
   *
   * @return            the revision
   */
  public String getRevision() {
    return RevisionUtils.extract("$Revision: 5542 $");
  }

  /**
   * Main method for testing this class.
   *
   * @param argv should contain arguments to the filter:
   * use -h for help
   */
  public static void main(String [] argv) {
    runFilter(new SpreadSubsample(), argv);
  }
}
```