



Run Run Shaw Library

香港城市大學
City University of Hong Kong

Copyright Warning

Use of this thesis/dissertation/project is for the purpose of private study or scholarly research only. ***Users must comply with the Copyright Ordinance.***

Anyone who consults this thesis/dissertation/project is understood to recognise that its copyright rests with its author and that no part of it may be reproduced without the author's prior written consent.

CITY UNIVERSITY OF HONG KONG
香港城市大學

Comprehensive Analysis of Data Sampling Approaches on
the Performance of Software Defect Prediction Models
數據抽樣方法對軟件缺陷預測模型性能的綜合分析

Submitted to
Department of Computer Science
電腦科學系

in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
哲學博士學位

by

Kwabena Ebo Bennin

August 2018
二零一八年八月

Abstract

Software defect prediction models can conveniently identify faulty modules and these models aid in the prioritization of scarce testing resources. The performance of defect prediction models are known to be dependent on the datasets used for training the models. Unfortunately, defect datasets are imbalanced with the non-defective modules dominating the defective modules. These imbalanced datasets thus make accurate and reliable predictions difficult. A prevalent approach to alleviate the class imbalance of defect datasets before model training is the application of data sampling approaches. The application of data sampling approaches to imbalanced defect data during model training has been shown to affect performance of defect prediction models in previous studies. However, very little is known about the best distribution for attaining high performance especially in a more practical scenario. There are still inconclusive results pertaining to the suitable ratio of defect and clean instances(resampling rate), the statistical and practical impacts of resampling approaches on prediction performance and the more stable resampling approach across several performance measures. Furthermore, the effects of most sampling methods have been evaluated in terms of accuracy/geometric mean/F1-measure in previous studies; however, these measures do not consider the effort needed to fix faults.

In this thesis, we investigate the impact that sampling methods have on prediction performance. An in-depth analysis of sampling methods and the various components (i.e. different types, configuration parameters) on performance of defect prediction models is conducted. A case study on both static and process metric open-source projects reveal that (1) the effectiveness of sampling methods are dependent on the resampling rate (2) sampling methods significantly improve performance but it is accompanied with significantly high false alarms. (3) sampling methods specifically under-sampling do not improve prediction performance when evaluated with an effort-aware measure. Our findings suggest that for better prediction performance, researchers should carefully select the resampling rate based on the sampling approach. Stimulated by our findings that most of the data sampling approaches result in over-generalization (high rates of false alarms) and generate near-duplicated data instances (less diverse data), a diversity-based oversampling method named MAHAKIL is further proposed which aims to produce high recall(probability of detection) and low pf (false alarms). Our experiments indicate that MAHAKIL improves the prediction performance for all the models and achieves better and more significant pf values than the other oversampling approaches, based on Brunner's statistical significance test and Cliff's effect sizes. Therefore, MAHAKIL is

strongly recommended as an efficient alternative for defect prediction models built on highly imbalanced datasets.

Keywords - Software Defect Prediction Modelling; Imbalanced data; Data Sampling Techniques; Empirical software engineering.

CITY UNIVERSITY OF HONG KONG
Qualifying Panel and Examination Panel

Surname: BENNIN
First Name: Kwabena Ebo
Degree: PhD
College/Department: Department of Computer Science

The Qualifying Panel of the above student is composed of:

Supervisor(s)

Dr. KEUNG Wai Jacky Department of Computer Science
City University of Hong Kong

Qualifying Panel Member(s)

Dr. YU Yuen Tak Department of Computer Science
City University of Hong Kong

Prof. ZHANG Qingfu Department of Computer Science
City University of Hong Kong

This thesis has been examined and approved by the following examiners:

Dr. KWOK Lam For Department of Computer Science
City University of Hong Kong

Prof. WANG Jianping Department of Computer Science
City University of Hong Kong

Dr. KEUNG Wai Jacky Department of Computer Science
City University of Hong Kong

Dr. LAI Richard Y Department of Computer Science and Information
Technology
La Trobe University

Acknowledgements

My greatest expression of gratitude goes to my primary supervisor Dr. Jacky Wai Keung for his patience, guidance, financial support and encouragement which has been very key to my PhD completion. His advice and support really made it possible for me to comfortably adjust and fully concentrate on my research studies in Hong Kong. His practical research skills has been a blessing to me as I have been able to learn from his thinking and writing skills. I am also grateful to him for trusting and believing in my research skills and potential and patiently helping me improve and develop this research skill. I am also very grateful to him for giving me the opportunity to travel for research internships and for introducing me to experienced Software engineering researchers across the globe. These are priceless opportunities which will be with me for the rest of my life and have been key to the completion of my PhD research studies.

This thesis would not have been successfully completed without the resolute support of my mentor, Dr. Akito Monden of Okayama University who introduced me to this research area of software engineering. Being confused with regard to my research area, Dr. Monden introduced me to defect prediction and patiently explained to me how to effectively conduct research in this popular but significant domain of software engineering. I have been humbled to work under your supervision and guidance. Although very busy, you have always had the time and patience to review my articles and provide constructive feedback which has enabled me publish in top conferences and Journals. Your kindness and technical research skills significantly gave me confidence in conducting my research. I am very grateful for your encouragement and financial support towards my research internship in your laboratory and which also enabled me attend several high-profile international conferences.

I am also grateful to my qualifying panel members who have over the past four years annually reviewed my research progress reports and provided significant and crucial feedback that has helped improve my work.

My Ph.D studies would not have been possible without any financial support. I am thus very grateful to the School of Graduate Studies and the University Grants Council for granting me very generous scholarships specifically the Chow Yei Ching Entrance Scholarship and the Hong Kong Fellowship Scheme which has supported my student life in Hong Kong. The research travel grants have been very beneficial as it allowed me to travel for several conferences without any financial burden.

This thesis would definitely not have been possible without the support of my family. I couldn't have the peace of mind and comfort to complete my PhD

without the unflinching support and love of my dear wife Hannah Tetteh who graciously combined her student life with family and cared for me. I am very grateful to you for sacrificing your time to care for our dear son Aiden so that I could concentrate on my research work. My gratitude goes to my mother, Nancy Addo who spent some months with us during the PhD period in helping to take care of my son. I am also grateful to my brother Seth who took good care of my mother and other family matters whilst I was away from home. My sincere gratitude goes to my friend Jonathan who took great care of some family affairs whilst I concentrated on my studies. Having a peace of mind to study is invaluable and my good friend Ernest Pobee has been of great support in this remarkable journey. Thank you my brother for all the great time we have spent together during my short stay in Hong Kong.

Finally, to my colleagues and friends, I say a big thank you for your support and love.

Contents

Abstract	i
Acknowledgements	iv
1 Introduction	1
1.1 Introduction	1
1.1.1 Problem Statement	4
1.1.2 Thesis Contributions	5
1.1.3 Thesis Organisation	6
2 Defect Prediction and Class Imbalance Learning	7
2.1 Introduction	7
2.1.1 Chapter Organization	8
2.2 Class Imbalance and Imbalanced Defect Datasets	8
2.2.1 Data Re-sampling Approaches for Defect Prediction Models	9
SMOTE	10
ADASYN	10
Borderline-SMOTE	11
Safe-level SMOTE	11
Random Over Sampling	11
Random Under Sampling	11
2.3 Evaluation of Prediction models	12
2.3.1 Performance Measures	12
False positive rate (FPR)	12
Recall (pd)	12
Precision	12
F-measure	13
Area Under the ROC Curve (AUC)	13
Cost-Effective Evaluation ($\text{Norm}(P_{opt})$)	14
2.3.2 Performance assessment of prediction models trained on	
Imbalanced Datasets	14
2.4 Chapter Summary	16

3	A Motivating Study: Empirical Evaluation of Cross-Release Effort-Aware Defect Prediction Models	17
3.1	Introduction	17
3.1.1	Chapter Organisation	19
3.2	Research Questions	19
3.2.1	Which is the best fault prediction model when testing effort is considered?	19
3.2.2	Which is the best fault prediction model when assuming cross-release prediction?	20
3.3	Experiment Description	20
3.3.1	Target Project	20
3.3.2	Measurement Metrics	20
3.3.3	Experimental data set	21
3.3.4	Pre-processing of Data	21
3.3.5	Estimation of hyper parameters	23
3.3.6	Model Evaluation criteria	23
3.3.7	Experiment execution environment	23
3.4	Results and Discussion	23
3.4.1	Evaluation of fault density prediction models that take into account testing effort	23
3.4.2	Evaluation of fault density prediction model across versions	25
3.4.3	Discussions	26
3.5	Threats to Validity	29
3.6	Related Work	30
3.6.1	Evaluation of models that take into account the testing effort	30
3.6.2	Evaluation of models that span the version	31
3.7	Chapter Summary	31
3.8	Concluding Remarks	32
4	Related Works	35
4.1	Introduction	35
4.2	Software Defect Prediction	35
4.3	Approaches for Tackling Class Imbalanced Defect Datasets Learning	36
4.4	Comparative studies of data resampling approaches in the Literature	37

4.5	Evaluation measures of Prediction models trained on Imbalanced data	39
4.6	Parameter Configuration of sampling methods	40
4.7	Chapter Summary	41
5	Assessing the Impacts of Balanced Training and Testing Datasets on Effort-Aware Fault Prediction Models	43
5.1	Introduction	43
5.1.1	Chapter Organization	44
5.2	Background and Related Work	45
5.2.1	Class Imbalance learning	45
5.2.2	Effort-Aware Classifiers	46
5.3	Research Questions	47
5.4	Experimental Setting	48
5.4.1	Datasets	48
5.4.2	Sampling Techniques	48
5.4.3	Experimental Method	49
5.5	Empirical Results and Discussion	51
5.6	Threats to Validity	55
5.7	Chapter Summary	56
5.8	Concluding Remarks	57
6	Assessing the Significant Impact of Data Sampling Approaches on Software Defect Prioritization and Classification	59
6.1	Introduction	59
6.1.1	Chapter Organization	61
6.2	Related Work	61
6.3	Background	63
6.3.1	Data Sampling Approaches	63
	Random Over-sampling (ROS)	63
	Random Under-sampling (RUS)	64
	Synthetic Minority Over-sampling TEchnique (SMOTE)	64
	Adaptive Synthetic Sampling (ADASYN)	64
	Borderline-SMOTE	64
6.3.2	Effect Size	65
6.4	Case Study Design	66
6.4.1	Research Questions	66
6.4.2	Datasets	66
6.4.3	Evaluation Criteria	67

6.4.4	Experimental Setup	68
6.4.5	Performance Comparison	69
6.5	Case Study Results	70
6.6	Discussion	75
6.7	Chapter Summary	76
6.7.1	Concluding Remarks	77

7 MAHAKIL: Diversity based Oversampling Approach to Alleviate the Class Imbalance Issue in Software Defect Prediction 79

7.1	Introduction	79
7.1.1	Chapter Organization	81
7.2	Motivation	81
7.3	Chromosomal Theory of Inheritance and Diversity	83
7.4	MAHAKIL - New Oversampling technique for Software Defect Prediction	84
7.4.1	Overview	84
7.4.2	Phase 1: Diversity Measurement	86
7.4.3	Phase 2: Data Partitioning and Pairing	87
7.4.4	Phase 3: Synthetic Sample Generation	88
7.5	Experimental Methodology	89
7.5.1	Benchmark Datasets	89
7.5.2	Baselines	90
7.5.3	Experimental Design	92
7.5.4	Evaluation Performance Metrics	93
7.5.5	Performance Comparison	95
7.6	Empirical Results	95
7.6.1	Overall Results	95
7.6.2	Statistical Comparison of Techniques	97
7.7	Discussion	99
7.7.1	Stability of MAHAKIL	99
7.7.2	Choice of Prediction models	100
7.7.3	Why MAHAKIL Performs best	101
7.7.4	Threats to Validity	102
	Construct Validity	102
	Internal Validity	103
	Statistical Conclusion Validity	104
	External Validity	104
7.8	Chapter Summary	105

7.8.1	Concluding Remarks	105
8	Assessing the Impact of the Distribution Parameter of Data Sampling Approaches on Software Defect Prediction Models	109
8.1	Introduction	109
8.1.1	Chapter Organization	110
8.2	Related Works and Research Questions	110
8.3	Case Study Design	112
8.3.1	Studied Sampling Approaches	112
8.3.2	Datasets	113
8.3.3	Evaluation Criteria	115
8.3.4	Experimental Setup	116
8.3.5	Statistical Performance Comparison	117
8.4	Case Study Results and Discussions	118
8.4.1	RQ1: How does tweaking the <i>Pfp</i> parameter impact performance of sampling approaches on defect prediction models?	118
8.4.2	RQ2: To what degree (percentage of fault-prone modules <i>Pfp</i>) should one resample or balance the original data set?	121
8.4.3	RQ3: Which are the high-performing resampling approaches for defect prediction?	122
8.5	Threats to Validity	124
8.6	Chapter Summary	124
8.6.1	Concluding Remarks	125
9	Conclusion	127
9.1	Introduction	127
9.2	Research Findings and Implications	127
9.3	Research Contributions	131
9.4	Future Research	132
9.4.1	More Analysis of sampling approaches	132
9.4.2	Improving the MAHAKIL oversampling approach	133
9.4.3	Considering the impact of other parameters of sampling approaches	133
	Bibliography	135
A	Appendix	149
A.1	Extra Results for MAHAKIL (Chapter 7)	149

B	Appendix	155
B.1	Appendix B: Process Metrics Data Used This Study (Chapter 8)	155
C	Publications	157

List of Figures

1.1	Impact of Sampling on a classification model	3
2.1	Construction process of defect prediction models	8
2.2	Kernel Density plots of faults and clean modules in the ivy-1.4 dataset.	9
2.3	How the SMOTE algorithm works ($k=5$).	10
2.4	Common ROC plot [114].	13
2.5	An example of LOC-based cumulative lift chart [58]	15
3.1	Deñsar's significance diagram in RQ1	27
3.2	Deñsar's significance diagram in RQ2	27
3.3	Prediction performance of each model on the 25 OSS projects sorted according to P_{fp}	28
3.4	Cross-release prediction performance of models on the 25 data sets sorted according to P_{fp}	29
5.1	Average Norm(P_{opt}) performances across all ten datasets for the sampling techniques on different Effort-Aware Models at different P_{fp} values	52
5.2	Average performance of Effort-Aware Models across different P_{fp} values	53
5.3	Effort-Aware Models trained on 7 different datasets (train-20, train-30, train-40, train-50, train-60, train-70 and train-80) tested by seven different testing datasets, increasing from 20% faulty (x-axis=20) to 80% faulty (x-axis=80)	54
6.1	Quartile charts of performance values for all sampling techniques on different Predictive Models at different (P_{fp}) values across all ten Static metrics datasets.	72
6.2	Performance plots for the effect sizes computed for each sampling methods compared to the no sampling method on different prediction models across all ten cross-project datasets per each P_{fp} values.	73

7.1	Effect of sub-clusters of unequal or small sizes on the class imbalance problem; (a) the estimated decision boundary (dashed line) does not approximate the ideal or true decision boundary (solid line); (b) generating synthetic minority data without considering the position and distance of the nearest neighbors from the minority example (e.g. data point A) creates near duplicated or erroneous data and widens the estimated decision boundary; and (c) the ideal diverse data after resampling ensuring that the estimated and true decision boundaries are almost equal	82
7.2	Illustration of MAHAKIL. The red and green dots represent the minority samples with the latter representing the synthetic data generated by MAHAKIL.	86
7.3	Framework of experiments	94
7.4	Performance plot of the sampling techniques per each prediction model for the pd performance metric on 20 imbalanced datasets across different percentage of fault-prone modules (Pfp) values.	96
7.5	Performance plot of the sampling techniques per each prediction model for the pf performance metric on 20 imbalanced datasets across different percentage of fault-prone modules (Pfp) values.	97
7.6	Brunner's statistical test win-tie-loss and effect-size comparison of MAHAKIL vs. ROS, ADASYN, Borderline-SMOTE, SMOTE and NONE across all 20 datasets per each defect prediction model, performance measure (pd , pf) and Pfp value (30%, 40%, 50%).	98
7.7	Average diversity value at each Pfp for all 20 projects	102
8.1	Pseudocode for win-tie-loss computation between predictors k and l with performance measure values $Perf_k$ and $Perf_l$. Note that pf values are negated before comparison since the lower the pf value, the better.	117
8.2	Performance plots (median values) for all performance measures of the sampling techniques on different Prediction Models at different Pfp values across all ten cross-release static metrics datasets	118
8.3	Performance plots (median values) for all performance measures of the sampling techniques on different Prediction Models at different Pfp values across all ten cross-release process metrics datasets	120

8.4	Total counts of wins of each <i>Pfp</i> rate for each Sampling method across all ten static metrics datasets and all prediction models per each performance metric	121
8.5	Total counts of wins of each <i>Pfp</i> rate for each Sampling method across all ten process metrics datasets and all prediction models per each performance metric	122

List of Tables

2.1	Confusion Matrix for the Two-class Problem	12
3.1	Prediction methods and hyper parameters in our experiments . .	21
3.2	The number of modules and their respective faults of two ver- sions' projects used in experiments	22
3.3	Measurement Metrics	22
3.4	Prediction performance of each model when testing effort is con- sidered	25
3.5	Prediction performance considering testing effort for cross-release analysis	26
5.1	Prediction methods and hyperparameters in our experiments . .	49
5.2	Average Norm(P_{opt}) of Effort-Aware Models among different Sam- pling Techniques across all P_{fp} values	52
6.1	Summary of selected 20 Imbalanced datasets extracted from PROMISE data repository	65
6.2	Description of Static Code Metrics [49]	67
6.3	Confusion Matrix for two-class classification problem	68
6.4	Predictive Models and their hyperparameter configurations . . .	70
6.5	Performance in terms of losses aggregated across four P_{fp} values on the 10 datasets. Lower losses values indicates higher perfor- mance. Records highlighted in light gray indicate that applica- tion of sampling methods results in better performance whilst those in dark gray indicate otherwise.	73
7.1	Description of the Static Code Metrics [49]	90
7.2	Summary Statistics of Selected 20 Imbalanced Datasets Extracted from the PROMISE Data Repository	91
7.3	Classification Models and their Hyperparameter Configurations	92
7.4	Confusion Matrix for the Two-class Problem	94

7.5	Performance in terms of wins, losses, and wins-losses aggregated from the two performance measures on 20 datasets per each <i>Pfp</i> value. Higher wins(%), wins-losses(%) and lower losses(%) indicate higher performance of the predictor. The predictors are ordered by wins-losses.	100
8.1	Summary of selected 40 Imbalanced datasets	114
8.2	Predictive Models and their hyperparameter configurations . . .	116
8.3	Kendall's Tau rank correlation analysis between <i>Pfp</i> and each prediction performance measure per each prediction model and metric-type dataset. Significant correlation results (p-value<0.05) are highlighted in gray.	119
8.4	Performance in terms of wins, losses and wins-losses aggregated from all five performance measures across all datasets. Higher wins and wins-losses indicate better performance, whereas lower losses also indicate better performance. The ranks are ordered by wins-losses	123
A.1	AUC Performance values of SMOTE, ADASYN, ROS, Borderline-SMOTE and MAHAKIL on Real World Data sets using C4.5 at different <i>Pfp</i> rates.	149
A.2	AUC Performance values of SMOTE, ADASYN, ROS, Borderline-SMOTE and MAHAKIL on Real World Data sets using KNN at different <i>Pfp</i> rates.	150
A.3	AUC Performance values of SMOTE, ADASYN, ROS, Borderline-SMOTE and MAHAKIL on Real World Data sets using NNET at different <i>Pfp</i> rates.	150
A.4	AUC Performance values of SMOTE, ADASYN, ROS, Borderline-SMOTE and MAHAKIL on Real World Data sets using RF at different <i>Pfp</i> rates.	151
A.5	AUC Performance values of SMOTE, ADASYN, ROS, Borderline-SMOTE and MAHAKIL on Real World Data sets using SVM at different <i>Pfp</i> rates.	151
A.6	G-mean values of SMOTE, ADASYN, ROS, Borderline-SMOTE and MAHAKIL on Real World Data sets using C4.5 at different <i>Pfp</i> rates.	152
A.7	G-mean values of SMOTE, ADASYN, ROS, Borderline-SMOTE and MAHAKIL on Real World Data sets using KNN at different <i>Pfp</i> rates.	152

A.8	G-mean values of SMOTE, ADASYN, ROS, Borderline-SMOTE and MAHAKIL on Real World Data sets using NNET at different <i>Pfp</i> rates.	153
A.9	G-mean values of SMOTE, ADASYN, ROS, Borderline-SMOTE and MAHAKIL on Real World Data sets using RF at different <i>Pfp</i> rates.	153
A.10	G-mean values of SMOTE, ADASYN, ROS, Borderline-SMOTE and MAHAKIL on Real World Data sets using SVM at different <i>Pfp</i> rates.	154
B.1	Summary of versions and collection period for the 10 OSS projects used.	156

Chapter 1

Introduction

1.1 Introduction

Software defect prediction models can conveniently identify software modules or classes that are likely to be defective. It is very crucial to ensure that defective modules are being accurately detected during defect prediction due to the huge economic impact faulty software could have. Media reports in the past decade detailed the tremendous loss of revenue for large software projects due to software bugs [104]. Advancement in data collection automation has led to the amassment of software project data which could be used by project managers during testing and maintenance phases of a software project to ensure correct allocation and prioritization of testing resources. Prioritization of testing resources is a key challenge of any software engineering project. Thus, several maintenance and quality assurance measures (critical inspection of source codes, user acceptance tests and others) have been employed and implemented by organizations. The goal to detect faults in software applications has instigated proposals for several fault prediction techniques in recent research. This helps in focusing on the specific modules, which really needs attention with regards to code inspection and efficiently prioritize and allocate the limited testing resources to those modules identified as likely to be defective [75, 88, 135].

Defect prediction datasets are usually skewed or imbalanced with the non fault-prone modules dominating the fault-prone modules [135]. This is not so surprising as several high quality assurance activities are employed during the testing phase of a software project hence drastically reducing the total number of faults in the software [30]. A defect prediction model trained on such a skewed data set will thus be more biased towards the non fault-prone modules or majority class and inadvertently disregard the minority or fault-prone modules [139]. This is referred to as class imbalance learning in machine learning and data mining domain and using such a trained model to classify the target project will most often predict a fault-prone module as non fault-prone. Researchers

and software quality teams are however more interested in the minority faulty classes [116].

In the past decade, different approaches proposed to tackle the issue of class imbalanced data set by researchers include: data resampling techniques, assigning specific cost to the classes aimed at reducing the classification cost [29, 105, 123] and use of ensemble learning techniques [73, 143]. However, adoption of data resampling techniques as a measure to alleviate the issue of imbalanced datasets by adding new instances to the minority class or removing majority instances known as over and under-sampling respectively is very common due to the ease of implementation and relatively positive effect on classification performance. Previous studies [38, 50, 117] conclude that oversampling approaches perform better than undersampling approaches due to the fact that no information is lost and as such, most of the recent resampling approaches proposed in the literature are all based on oversampling [10, 22, 44, 48]. Figure 1.1 shows the impact of oversampling on a classification model. The shift in decision boundary built before and after applying oversampling is clear, implying that oversampling has a significant effect on the classification performance. Specifically, before oversampling the data, the decision boundary of the classifier is largely induced by the dominating non-defective modules. This model will over predict the non-defective modules and will have a low probability of predicting the defective modules. However, after oversampling the data, the classification model built will now have an equal probability of predicting both the defective and non-defective modules. Consequently, the performance of the model is largely improved. Despite the obvious fact that class imbalance is one of the critical factors that affects the performance of software defect prediction models [3, 43, 124], most studies [106, 113, 135] focus on applying or proposing data resampling approaches without considering the amount of resampling or percentage of fault-prone modules (Pfp) actually required. The objective functions of most machine learning algorithms assume that datasets are balanced prior to training [22, 73, 108]. The question we therefore ask ourselves is "Is balanced the best distribution?" Agrawal and Menzies [1] noted that the performance of prediction models can be significantly improved when the parameters of SMOTE sampling method are configured. Their study suggests that configuring the parameters of sampling methods can improve the performance of prediction models. Different sampling methods however, have different configurable parameters and it is impractical to consider and configure all these parameters during model construction. Nevertheless, the Pfp which controls

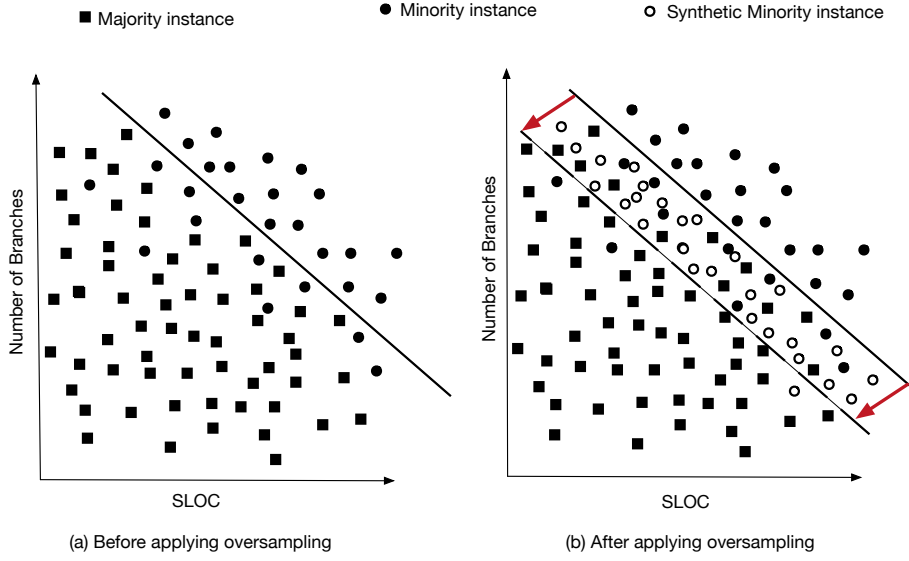


FIGURE 1.1: Impact of Sampling on a classification model

the amount of data added or deleted from the minority class samples respectively is a constant parameter present in all sampling methods. This study focuses on the effect of Data Sampling approaches and the Pfp parameter on the performance of prediction models considering other sampling methods including the SMOTE method. Additionally, empirical research conducted till date used different evaluation measures and did not statistically evaluate the results leading to the inconclusive stability as to the best resampling approach and Pfp .

Before we examine and discuss in detail the main body of this thesis, we initially conducted a comprehensive study to investigate the best performing prediction model. Subsequently, we observe that previous studies have been unable to conclude on the best fault prediction model due to three major drawbacks: (1) comparison of few fault prediction models considering small number of data sets, (2) use of evaluation measures that ignore testing efforts and (3) use of n-fold cross validation instead of the more practical cross-release validation. To address these concerns, we conduct cross-release evaluation of 11 fault density prediction models using data sets collected from 2 releases of 25 open source software projects with an effort-aware performance measure known as $\text{Norm}(P_{opt})$. Our result shows that, whilst M5 and K^* had the best performances; they were greatly influenced by the percentage of faulty modules present (class imbalance of dataset) and the size of the data set. Using $\text{Norm}(P_{opt})$ produced an overall average performance of more than 50% across all the selected models clearly indicating the importance of considering testing efforts in building fault-prone

prediction models. Our findings suggest that performance of prediction models are highly influenced by the training datasets specifically the size and amount of defective and non-defective modules methods and empirical investigations on data sampling approaches using robust statistical methods are needed to ascertain the impact and practical benefits of these data sampling approaches.

1.1.1 Problem Statement

We formulate and state below the thesis statement based on the empirical findings from our initial study (see Chapter 3).

Thesis Statement: Imbalanced Defect datasets influence the prediction performance of defect prediction models. Empirical investigations on the impact of data sampling methods and their configuration parameters on defect prediction performance are required to attain more practical implications from defect prediction models.

In the domain of defect prediction, several studies have raised concerns about the impact of the dataset used for model training. Gray et al. [40] reported the damaging effects datasets extracted from open-source repositories do have on defect prediction performance. They suggested that datasets should be properly pre-processed to appropriately alleviate issues such as noise and outliers. A predominant feature of defect datasets is the imbalanced nature of the class distributions [34, 40, 135]. Menzies et al. [91] suggested that for better prediction performance, the information content of the training data should be improved. This improvement can be achieved by using data sampling methods prior to model training, which has not yet been extensively investigated. The statistical significance of these data sampling methods on defect prediction performance is not known and the required amount of sampling has never been explored. Additionally, the most robust and best sampling method to use for defect prediction studies is unknown.

Thus, this thesis aims to investigate and assess the impact of data sampling methods on defect prediction performances. A systematic analysis to examine the extent data resampling approaches improve prediction performance is conducted. The relevance of data sampling methods considering two overlooked experimental components (i.e., choice of evaluation performance measure and the sampling distribution parameters) on the performance of defect prediction

models are investigated. Furthermore, the statistical and practical significance of sampling methods are empirically assessed and a robust and efficient sampling method is proposed.

1.1.2 Thesis Contributions

In this thesis, we investigate the impact that sampling methods have on defect prediction performance. An in-depth analysis of several sampling methods and the various components (i.e different types, configuration parameters) on performance of defect prediction models is conducted. We provide several contributions to the field of defect prediction. The key contributions are highlighted as follows:

1. We empirically validate several resampling methods across more datasets and statistically test the significance using robust statistical measures. Use of robust statistical significance tests enables us to better understand the techniques that are beneficial for defect prediction models and identify the most stable and effective data resampling method for SDP.
2. We demonstrate that the choice of evaluation performance measure has significant and practical effects on sampling methods and prediction performance. Extensive empirical experiments with several types of performance measures reveal that software quality teams can use sampling methods for defect classification purposes but not for defect prioritization activities.
3. We empirically demonstrate that the choice of resampling rates (Pfp) significantly impacts prediction performance. We evaluate the practical effects of different resampling rates (Pfp) on defect prediction performance and find a stable Pfp rate for data resampling methods.
4. Practical guidelines for achieving higher prediction performance for each resampling method. We contribute by providing recommendations and guidelines on which data resampling method to use or avoid for software quality teams and researchers.
5. We demonstrate that generation of diverse synthetic datasets is more important than generating duplicates of existing data instances and thus develop a novel diversity based sampling approach based on Chromosomal theory of Inheritance for diversity.

1.1.3 Thesis Organisation

The thesis is structured as follows. Chapter 2 discusses the background of defect prediction and the dominance and effect of imbalanced datasets on prediction performance. Definitions of commonly used data resampling approaches are also presented in this chapter. In Chapter 3, a motivating empirical study assessing eleven prediction models evaluated with an effort-aware measure with the aim of finding the best performing model is conducted. Unable to find a stable prediction model due to imbalanced nature of the training datasets, we are driven to investigate and conduct more experiments to assess the impact and benefits of data sampling approaches on performances of defect prediction models. The main research work of this thesis are therefore presented in Chapters 5 to 8. Chapter 4 summarises several related works of data sampling approaches and extract four main problems with existing studies. In Chapter 5, an empirical study assessing the impact of a balanced dataset on prediction performance is presented. A few number (4) of data sampling approaches are evaluated with effort-aware models. In Chapter 6, we evaluated the validity and impact of several data sampling approaches by adopting robust statistical tests. The practical significance of data sampling approaches is examined by adopting several conventional performance measures. The performance of 800 combinations of data sampling approaches and prediction models are evaluated across two types of dataset metrics. Analyses of the results led us to develop and propose a new data sampling approach named MAHAKIL in Chapter 7. In chapter 8, the impact of the parameter configuration of the data sampling approaches is investigated. A conclusion stability experiment is conducted by comparing the performance of our proposed method to other selected and studied data sampling approaches. The research work of this thesis is finalised in Chapter 8 where we sought to find a stable Pfp (% of fault-prone modules) or amount of resampling of all data sampling approaches for defect prediction studies. We conclude the thesis in Chapter 9 with a discussion of findings from all the experiments conducted from the previous Chapters and highlight some promising future directions of data sampling approaches.

Chapter 2

Defect Prediction and Class Imbalance Learning

2.1 Introduction

With the increasing pressures of expediting software projects that is always increasing in size and complexity to meet rapidly changing business needs, quality assurance activities such as fault prediction models have thus become extremely important. The main purpose of a fault prediction model is the effective allocation or prioritization of quality assurance effort (test effort and code inspection effort) [75, 88]. Construction of these prediction models are mostly dependent on historical or previous software project data referred to as a dataset. Figure 2.1 visualizes the construction process of a defect prediction model.

However, a prevalent problem in data mining is the skewness of a dataset [34, 106]. Fault prediction datasets are not excluded from this phenomenon. Most datasets have the majority of the instances being either clean or not faulty [106, 124] and conventional learning methods are primarily designed for balanced datasets [22]. Common classifiers such as Neural Networks (NN), Support Vector Machines (SVM) and decision trees work best towards optimizing their objective functions that leads to the maximum overall accuracy - ratio of correctly identified faults to the number of total predicted faults. The use of imbalanced datasets for training a classifier will most likely generate a classifier that tend to over-predict the presence of the majority class but a lower probability of predicting the minority or faulty modules. Should the model predict the minority class, it usually comes with a high rate of error than predicting the majority class [52, 106]. This therefore affects the prediction performance of the classifiers and in machine learning, is referred to as learning from imbalanced datasets. Inaccurate performances are derived from prediction models trained on default imbalanced datasets. Several researches have stated the negative impact imbalanced datasets have on prediction performance. Insights derived

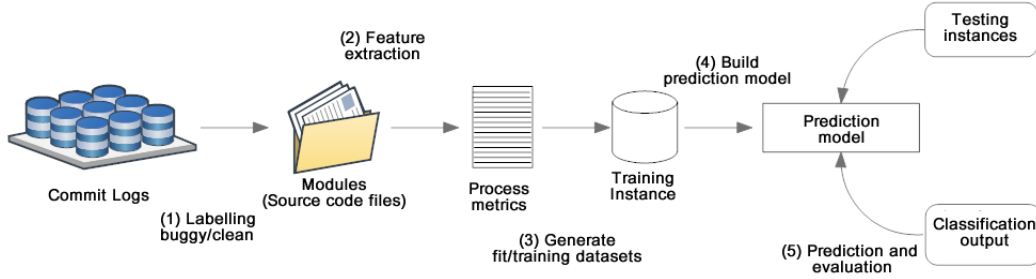


FIGURE 2.1: Construction process of defect prediction models

from these prediction performances are not useful and practical for software quality teams and researchers due to the class imbalance of the defect datasets. Alleviating this challenging issue with empirical evidence is thus necessary for a more informed decision making process.

2.1.1 Chapter Organization

This chapter is organized as follows. We provide detailed information on the nature of defect datasets in defect prediction, class imbalance and the different types of data resampling approaches available in literature. We further discuss the different types of performance evaluation measures for defect prediction models and the more appropriate performance measures used for evaluating prediction models trained on imbalanced datasets.

2.2 Class Imbalance and Imbalanced Defect Datasets

Class Imbalance, one of the top 10 challenges in data mining [35] has received significant attention in the past decade because most real world datasets are imbalanced. An imbalanced dataset is one in which some classes have relatively very few samples compared to other classes in the same dataset. In a two class dataset, the dominating class sample is referred to as the majority class and the few class samples, minority class. It is very prevalent in several application domains such as software defect prediction [113], human missense phenotype prediction [137] and credit card fraud detection [33] where clean, neutral and non-fraudulent instances dominate the buggy, deleterious and fraudulent instances respectively in the dataset. The minority instances are however of the greatest interest to the investigator. To visually illustrate the distribution of defective and non-defective modules in a typical defect dataset, we present the Kernel density plot of a commonly used dataset in defect prediction studies.

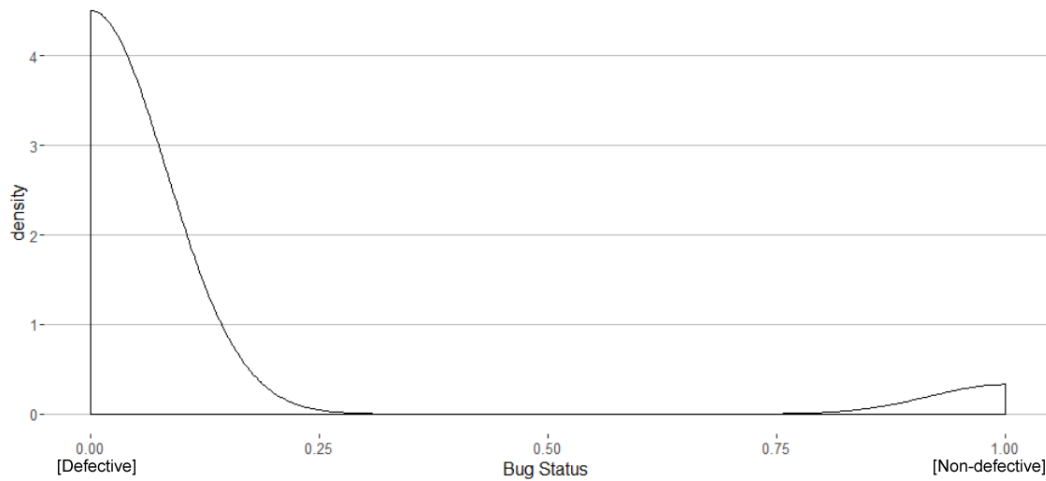
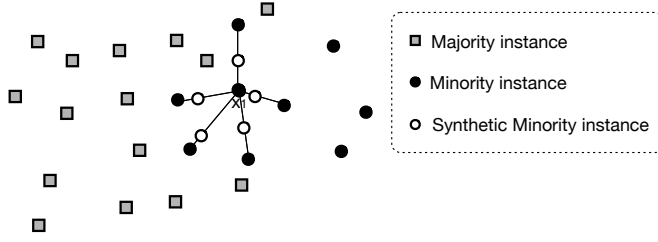


FIGURE 2.2: Kernel Density plots of faults and clean modules in the ivy-1.4 dataset.

From figure 2.2, we clearly observe that the distribution of the data is not normal and the non-defective modules dominate the defective modules. As argued by Phung et al. [108], the class imbalance problem is either caused by an intrinsic property in some application domains where for example we expect very few cases of defective instances compared to non-defective instances in a defect prediction dataset or caused by factors such as cost, time, privacy and effort which hinders one from acquiring a very representative data. Conventional supervised learning algorithms such as decision trees and neural networks are not able to classify the minority class suitably because the objective functions of these algorithms were designed for rightly balanced datasets [108] and thus assume an equal misclassification cost for both classes [73, 125]. Executing such algorithms or classifiers on a highly imbalanced dataset will result in a high misclassification error for the minority class because the minority class is somehow overlooked [110, 123]. In the literature, several methods and approaches have been proposed and formulated to solve this issue. In this thesis, we focus on one of the most popular and widely used approaches used to alleviate the class imbalance issue in defect prediction studies.

2.2.1 Data Re-sampling Approaches for Defect Prediction Models

Data re-sampling in this context refers to a pre-processing procedure to correct the imbalance target classes in a dataset by either increasing or decreasing the target cases before a model could be built for the dataset. Several data resampling approaches do exist in literature. These resampling approaches are

FIGURE 2.3: How the SMOTE algorithm works ($k=5$).

categorized into two main groups – Oversampling and Undersampling. Oversampling refers to the process of increasing the number of minority cases in a dataset. Undersampling on the hand is the converse of over sampling whereby we reduce the majority cases in a dataset by deletion. We define below the different types of resampling approaches found in literature.

SMOTE

Chawla et al. [22] proposed the commonly known Synthetic Minority Oversampling Technique (SMOTE) that creates synthetic data instances in the minority class region so as to shift the bias of the classifier towards the minority class. SMOTE automatically deletes some majority instances and increases the minority instances. The SMOTE algorithm generates synthetic instances by:

1. Selecting an instance from the minority class samples, for example, $x1$ in Figure 2.
2. Find the defined k minority class nearest neighbors instances of $x1$. ($k=5$)
3. New synthetic instances are created along the line segment that join each neighbor instance and the selected instance. The synthetic instances are randomly positioned between the the two minority instances.

These steps are repeated until the degree of oversampling required is attained.

ADASYN

He et al. [48] proposed the adaptive synthetic (ADASYN) sampling approach which applies a weighted distribution method for assigning weight to different minority classes distinguishing the harder-to-classify minority samples from the easier-to-learn minority samples and adaptively shifts the classification boundary toward the difficulty minority samples. In contrast to the SMOTE algorithm which generates equal synthetic samples for each minority class, the learning algorithm is induced to focus on the hard to learn examples within the minority class samples therefore samples generated are not equal for all samples.

Borderline-SMOTE

Focusing only on the minority samples closer to the borderline, Han et al. [44] proposed the borderline-SMOTE which generates synthetic samples along the line between some minority and majority samples based on the selected nearest neighbors. It aims at strengthening the borderline minority samples. The minority class instances are divided into three regions namely noise, borderline and safe based on the number of majority instances found among the k nearest neighbors. If all the nearest neighbors of a minority instance P are majority instances, P is denoted as noise. P is considered as a borderline instance when majority of its nearest neighbors are majority class samples. A safe P is when its neighbors are mostly minority instances. The SMOTE technique is then applied to only the borderline instances to generate more synthetic data.

Safe-level SMOTE

This is another modification of the SMOTE algorithm proposed by Bunkhumpornpat et al. [20]. A "safe-level" value is computed for each minority instance before applying the SMOTE algorithm. Synthetic samples generated are positioned closer to the largest "safe-level" ensuring that all synthetic instances are located only in the safe regions. A safe-level value is defined as the number of majority instances among the k nearest neighbors of a minority instance.

Random Over Sampling

Random Over Sampling (ROS) increases the number of minority instances by randomly replicating already existing minority class instances. As a simple and easy approach to implement, it has performed favourably well in previous empirical studies [32, 59]. However, no "new" data is generated since the oversampled instances are duplicates of old instances.

Random Under Sampling

Random Under Sampling (RUS) reduces the majority instances of a dataset by randomly deleting them. The minority instances are left intact whilst the majority instances are reduced. RUS is mostly convenient for datasets with very large number of majority samples as it reduces the training time. A major drawback of RUS is the loss of information which might be very influential to a classifier's performance.

TABLE 2.1: Confusion Matrix for the Two-class Problem

	Predicted Positive	Predicted Negative
Actual Positive	TP	FN
Actual Negative	FP	TN

2.3 Evaluation of Prediction models

2.3.1 Performance Measures

To observe the success and effectiveness of defect prediction models, several evaluation performance measures are used. For each prediction model, evaluation of the models are computed from a confusion matrix (table 7.4). Definitions of the acronyms in the table are defined as follows:

True Positives (TP): defective modules predicted as defective

False Positives (FP): non-defective modules predicted as defective

True Negatives (TN): non-defective modules predicted as not defective (clean)

False Negatives (FN): defective modules predicted as not defective (clean)

False positive rate (FPR)

Also referred to as probability of false alarm (pf), it measures the total non-defective modules predicted as defective among all non-defective modules.

$$pf = \frac{FP}{TN + FP} \quad (2.1)$$

Recall (pd)

Also referred to as true positive rate (TPR) or probability of detection (pd) is the ratio of correctly predicted defective modules to the total number of defective modules

$$Recall(pd) = \frac{TP}{TP + FN} \quad (2.2)$$

Precision

Precision is defined as the ratio of correctly identified defects to the number of total predicted defects.

$$Precision = \frac{TP}{TP + FP} \quad (2.3)$$

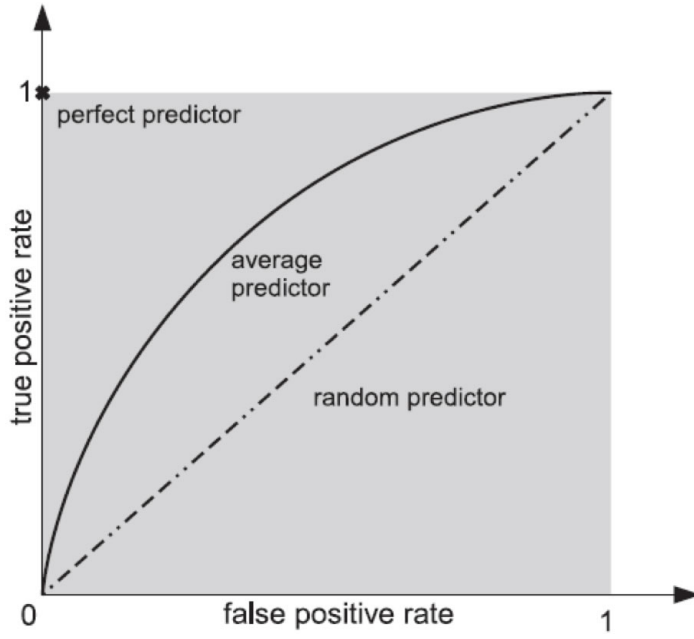


FIGURE 2.4: Common ROC plot [114].

F-measure

This refers to the harmonic mean of precision and recall. This is mostly used to solve the trade-off issues on both recall and precision.

$$F - measure = \frac{2 * precision * recall}{precision + recall} \quad (2.4)$$

Area Under the ROC Curve (AUC)

AUC uses the receiver operating characteristic (ROC) curve by measuring the area under the ROC curve. Recall (pd) and pf are used in plotting the ROC curve (Figure 2.4). A curve that is very close to the top left corner is regarded as better and more accurate. For a perfect prediction model, recall (pd) should be 1 and a false positive rate (pf) of 0. A random model should have the curve which is closer to the straight dotted line in the figure with an AUC value of 0.5. A random curve implies that for any threshold value, the probability for defective or non-defective prediction is equal. This evaluation measure has been recommended as the primary accuracy indicator for comparative studies in software defect prediction because of its ability to distinguish defective and non-defective modules clearly and class and cost distributions of each prediction model since performance of the model is dependent on the characteristics of the project or data. Therefore, the AUC-based evaluation has the potential to significantly improve convergence across studies.

Cost-Effective Evaluation (Norm(P_{opt}))

An effort-aware evaluation measure proposed by Mende and Koschke [87] and upgraded by Kamei et al. [58], this performance measure considers the effort needed to fix the fault detected which is ignored by most of the evaluation measures described above. It thus distinguishes between a predicted fault in a small module and a predicted fault in a large module. This is a more realistic and practical evaluation measure in agreement with Arisholm et al.'s [4] suggestion that evaluation and comparison of fault-proneness prediction models should not only consider their prediction accuracy but also take into consideration the potential cost-effectiveness of applying such models. Special consideration could be given to such models in the form of detailed inspection (effort) in a descending order of level of defectiveness [40]. P_{opt} value which is defined as difference in the LOC-based cumulative lift charts of the optimal model and the prediction model represented as Δ_{opt} (Figure 2.5) is computed first. From figure 2.5, the number of defects is represented on the y-axis whilst the cumulative size of modules or number of LOC is represented on the x-axis. The modules are ordered by defects and values of P_{opt} are dependent on the number of defects for each data set hence the output will vary. Normalizing the P_{opt} values for the prediction model is thus necessary since this will put them in the same scale and make comparison easier and reliable for each technique. A P_{opt} larger value implies a small difference between the optimal and predicted model and hence the best performing model.

The normalised P_{opt} is defined in equation (1)

$$Norm(P_{opt}) = \frac{P_{opt} - \min(P_{opt})}{\max(P_{opt}) - \min(P_{opt})} \quad (1)$$

From equation (1), $\max(P_{opt})$ and $\min(P_{opt})$ are computed based on the lift chart when all modules are ordered by fault densities. Evidently, $\max(P_{opt})$ will always be 1 since $\Delta_{opt} = 0$ for the best curve. To find $\min(P_{opt})$, modules are ordered increasingly on the x-axis using their actual fault densities similar to finding the worst case of prediction.

2.3.2 Performance assessment of prediction models trained on Imbalanced Datasets

Assessing the performance of prediction models on an imbalanced dataset using the overall accuracy is inappropriate due to the unequal error cost associated

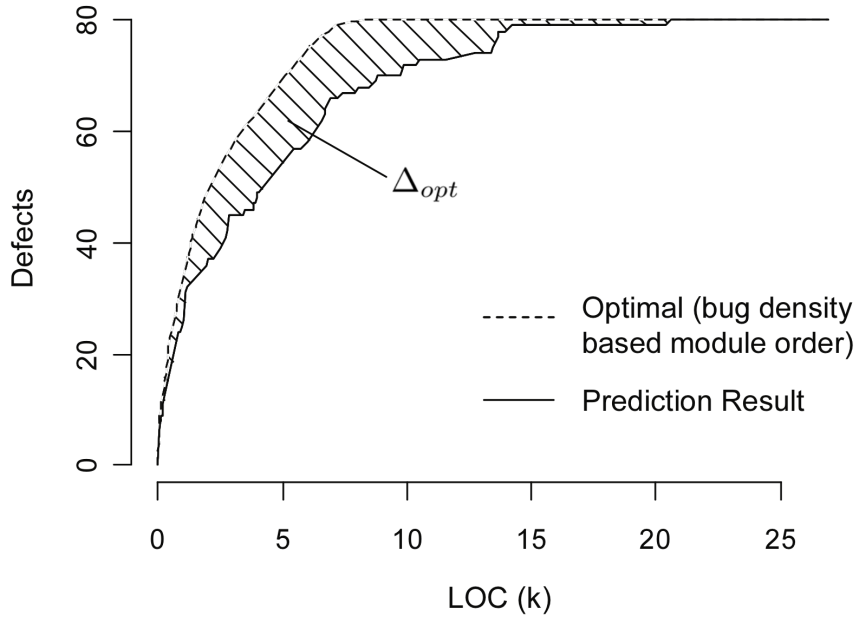


FIGURE 2.5: An example of LOC-based cumulative lift chart [58]

with the dataset [21, 88, 101]. In defect prediction studies, the goal is to precisely detect the defect-prone modules, that is the minority class samples in the datasets. As such, metrics such as Precision, Recall (pd) and probability of false alarms (pf) employed by previous studies [49, 54, 88] for evaluating the performance of classification algorithms on minority classes are more preferable. Recall (pd) and pf was recommended by Menzies et al. [92] as a more stable performance metric for highly imbalanced datasets. Measures such as AUC, *balance* and *g-mean* are considered for this thesis. The Area Under the ROC Curve (AUC) performance measure computed from the Receiver Operating Characteristics (ROC) curve handles the trade-offs between the true and false positive error rates [14, 74], thus does not sacrifice one class over the other. ROC is a graphical evaluation method which aids in defect classification and prioritization. The best predictor is achieved when $pd=1$ and $pf=0$ on the ROC curve implying that all defects were actually detected with no errors. The AUC is a single value between 0 and 1. Another practical measure for software quality teams is *balance* [88]. This measure tries to find a balance between pd and pf . The Euclidean distance between the actual point (pf, pd) and preferred point ($pf=0, pd=1$) is the *balance*. Similarly, the Geometric mean (*g-mean*) performance measure proposed by Kubat and Matwin [71] computes the balance between the two classes within the dataset. The measures (AUC, *balance* and *g-mean*) are used to evaluate the overall performance of the prediction models

in this thesis. Higher values denote better prediction performance. From Table 7.4, the mathematical definitions of *balance* and *g-mean* evaluation metrics are defined in the equations below.

$$balance(bal) = 1 - \frac{\sqrt{(0 - pf)^2 + (1 - pd)^2}}{\sqrt{2}} \quad (2.5)$$

$$g - mean = \sqrt{\frac{TP}{TP + FN} * \frac{TN}{TN + FP}} \quad (2.6)$$

2.4 Chapter Summary

In this chapter, definitions and explanations of the defect prediction process and the effect of imbalanced datasets are presented. Conventional evaluation methods and popular data sampling approaches are also explained. A motivational analysis of several common defect prediction models using a rarely used but more practical evaluation measure is conducted in the next chapter. The study aims to find the best performing defect prediction model and highlights the factors which lead to conclusion instability issue of finding the best performing prediction models. Consequently, the motivational study lays the foundation for the rest of our analysis in this thesis.

Chapter 3

A Motivating Study: Empirical Evaluation of Cross-Release Effort-Aware Defect Prediction Models

3.1 Introduction

In software testing and maintenance process, prediction of fault-prone modules (having high probability of containing a fault) is helpful in determining the allocation and priorities of review/test effort [34]. Fault prediction models predict the presence or absence of a fault in a module (single source code file) using metrics such as number of source code lines, cyclomatic complexity number and others, as explanatory variables [93], [99].

In addition, prediction of the number of faults [60], [76] and the fault density [65] have also been studied. However, in conventional studies, there are problems with the experimental settings for prediction model evaluation. Therefore, the stable or conclusive ranking of fault prediction models has not been firmly established yet. Specifically, these are the following three major problems:

- Small number of predictive models and small sample size of data sets are used for performance comparison and evaluation. As a result, the best prediction model varies among different research papers as observed by Myrtveit et al. [97].
- Cost-effectiveness (hereby referred to as testing effort in this paper) of applying fault-prone prediction models during evaluation and comparison of their prediction performance is ignored in most research studies as pointed out by Arisholm et al. [4]. Several previous studies predict the presence or absence of faults in a module, and evaluate the prediction

performance by precision/recall/F-measure/ROC-Curve and others [51], [75], [4]. However, most of these performance measures ignores effort needed to fix faults, that is, they do not distinguish between a predicted fault in a small module and a predicted fault in a large module. From the viewpoint of finding more bugs with less testing effort (cost of additional quality assurance such as verification cost which includes code inspections, coverage testing and others) on selected faulty modules, it would be better to predict high fault-density modules, which contain many faults in a few lines of code and to evaluate the prediction performance using effort-aware measures [65].

- N-fold cross-validation has been commonly used in previous studies for evaluating prediction models. It derives both fit and test data set from a single data set. However, this is a quite unrealistic situation in actual software development environment. In the software development environment, it is assumed that a model is constructed using the data of the developed software in the past (old version), to predict the fault of a developing software (the current version) and therefore evaluation of predictive model that span such version is required [112].

Based on these current issues, we address the following two Research Questions (RQ) using a data set from 25 open-source projects.

RQ1. Which is the best fault prediction model when testing effort is considered?

For RQ1, we experimentally demonstrate the superiority or inferiority of prediction performance of 11 kinds of fault density prediction models. In this experiment, we use a rating scale that takes into account the testing effort required after prediction. The experimental results shows that K* outperformed the other 10 prediction models although it was not statistically significant in comparison to the other prediction models.

RQ2. Which is the best fault prediction model when assuming cross-release prediction?

For RQ2, considering the use of a real past project, a model is constructed using data of the developed software (old version) to predict the fault density of each module in the next version. The proposed approach is a more reliable approach in the software development environment as noted by Kamei et al. [58]. We assess how the choice of using a previous release of a software for training a predictive model affects the accuracy of the predictive models based

on the evaluation criteria which considers the testing effort. The M5 tree model performed better in comparison to the other prediction models.

3.1.1 Chapter Organisation

This chapter is organized as follows. Background of research questions and the eleven predictive models are discussed in section 3.2. Section 3.3 presents the design of the experiment and Section 3.4 presents and discusses the results. Section 3.5 presents the threats to validity and Section 3.6 gives an overview of related works. Finally, we conclude in Section 3.7.

3.2 Research Questions

We address the research questions for the purpose of 11 predictive models in a real project environment.

3.2.1 Which is the best fault prediction model when testing effort is considered?

Kamei et al. [58] proposed a fault prediction model which is evaluated considering the testing effort. Kamei et al's experiment considers the number of source code lines as testing efforts and used fault density (number of fault/source code number of lines) as the target variable. With the use of prediction performance, developers can observe that for high fault density, it is possible to prioritize and allocate testing effort for the modules with large fault number per unit lines of code thus, creating an efficient test plan.

However, comparison of prediction performances of fault density prediction models using large number of data sets have not been well implemented in several previous studies. Therefore, to compare the types of fault density prediction models considering the number of source code lines in RQ1, 11 techniques were chosen to cover six categories of prediction models. We conduct an empirical experiment to reveal the superiority of the various prediction models by analysing their performances when testing effort is considered. Overview and name of each defect prediction model used and their hyper parameter values are shown in Table 5.1.

3.2.2 Which is the best fault prediction model when assuming cross-release prediction?

For a developer to use the fault density prediction model in real projects, the model should be built using data of the past developed software (the last version) as pointed out by Radjenovic et al. [112]. However, in conventional studies, cross-validation methods have been widely evaluated on the same version. Therefore, in RQ2 we build a fault density prediction model by using the previous version of the project or data, to evaluate the prediction performance when applied to the next version. Furthermore, upon examining and verifying the differences and correlation between the results of RQ1 and RQ2, developers could select the best fault density prediction model to use across versions.

3.3 Experiment Description

In this section, the experiment setting is described. Target projects and measurement metrics used are discussed.

Data is collected from 25 Open Source Software (OSS) projects and fault density (number of faults divided by number of lines of code) is predicted using 11 different approaches and finally compared considering the effort-aware measure $\text{Norm}(P_{opt})$. We describe the preprocessing activities performed on the data sets, the hyper parameters configuration, modelling tool used for the execution of this experiment and evaluation criteria used to verify the RQs.

3.3.1 Target Project

The records of each metric value was collected from 25 OSS project data sets used for the experiment. The number of project modules and faults are shown in Table 3.2. Buggy modules were labelled in the commit logs as "error" and thus were used to compute the error density. Bugs which were fixed in the next release were also labelled as "Bugfixes". For all the data set, version B is a newer version of A.

3.3.2 Measurement Metrics

One module is regarded as one source code file. For this experiment, we collected seven typical process metrics as proposed by Moser et al [95] for a module unit. We used the commit logs from collection of version control tools (such as CVS, Git, etc.) as the target repository. Names of each metric and overview

TABLE 3.1: Prediction methods and hyper parameters in our experiments

Category	Model	Overview	Hyper parameters
Statistical approach	LM	Linear regression	No
	LARS	Least Angle Regression [31]	No
	RVM	Relevance Vector Machine [12]	No
Neighborhood Law	K-NN	k neighborhood method [75]	$k=\{1, \dots, 15\}$
	K*	k neighborhood method using distance based on entropy [24]	No
Neural Net	NNET	3-layer neural net [60]	size = $\{4, \dots, 28\}$, decay = $\{0.1, 0.2\}$
Support Vector Machine	SVM	support vector machine [144]	$C = \{2^{-6}, \dots, 2^{10}\}$
The tree model	M5	classification tree [36]	No
	RPART	CART [61]	cp= $\{0.05, 0.1, \dots, 0.7\}$
Collective Learning	GBM	classification using boosting trees [19]	n.tree= $\{10, 50, 100, 200\}$
	RF	Random Forest [41]	mtry= $\{10, 50, 100, 200, 250, 500, 1000\}$

is shown in Table 3.3. In addition, we considered the specific words in the commit log measured in module units ("Fix" and "Refactoring") as BugFixes and Refactorings.

3.3.3 Experimental data set

For RQ1, the fit data is randomly selected using 2/3 of the modules from the data set Version A and the remaining 1/3 as the test data as shown in Table 3.2

For RQ2, the model was built using the data set version A as the fit data and predicted using data set version B as test data as shown in Table 3.2.

3.3.4 Pre-processing of Data

Because refactoring in several projects were not performed, the value of refactorings for these projects were zero (0). Hence, metric values of refactorings which were zero were excluded in advance from the model building to prevent

TABLE 3.2: The number of modules and their respective faults of two versions' projects used in experiments

Project Name	Short Name	Version A		Version B	
		Module	Fault	Module	Fault
Clam Antivirus	CLAM	1597	93	8723	56
eCos	ECOS	630	110	3480	67
Exim Internet Mailer	EXIM	184	28	61	31
freedesktop	FREE	1982	477	2855	603
Ganymede	GANY	184	29	95	30
Helma	HELM	312	38	100	17
Hibernate Core	HIBE	406	87	5032	1191
HylaFAX	HYLA	137	70	109	35
iptables	IPTA	151	96	276	91
Koffice	KOFF	930	363	990	232
NetBSD	NBSD	6781	548	10960	299
OpenBSD	OBSD	1706	275	5964	158
OpenCms	OCMS	1727	193	2821	93
openNMS	ONMS	1203	123	1416	112
OTRS	OTRS	420	111	763	169
PostgreSQL	POST	863	435	1211	325
ProFTPD	PROF	195	64	220	61
Samba	SAMB	1623	543	2559	439
Scilab	SCIL	2636	239	1248	244
Spring Security	SPRI	926	126	639	132
squid	SQUI	250	59	185	92
Wine	WINE	1627	548	1382	288
Xdoclet	XDOC	105	30	135	5
XFree86	XFRE	713	350	761	259
XORP	XORP	1696	158	1755	217

TABLE 3.3: Measurement Metrics

Name	Overview
Codechurn	Number of changed rows (number of additional lines + number of deleted rows)
LOCAdded	Additional number of rows (Lines of codes)
LOCDeleted	Deleted rows (Lines of codes)
Revisions	Revision number
Age	Last number of days since it was fixed
BugFixes	Number of corrected faults
Refactorings	Number of refactoring

any adverse effect on the model accuracy. In addition, multicollinearity tends to decrease the reliability of the model and as such, metrics causing multicollinearity were also excluded from the model building. The Codechurn and LOCAdded metrics were excluded since they had high correlation values above 0.9.

3.3.5 Estimation of hyper parameters

During prediction model building, it is necessary to set the parameters. Among the parameters that affect the entire model are the hyper parameters which has a significant impact on classification accuracy. Therefore, the optimal hyper parameter values were estimated using a 10-fold cross-validation method in this experiment. Candidate values of hyper parameters and the estimation of each method is shown in Table 5.1. However, the statistical approach is labeled "No" because they have no hyper parameters.

3.3.6 Model Evaluation criteria

For the experiment evaluation, we evaluated the performance of the prediction models based on testing efforts as proposed and used by Kamei et al [58]. They used the modification of P_{opt} evaluation metrics [86] called normalized P_{opt} due to its ability to evaluate the performance considering the testing effort. To find the best performing prediction model, we employed Nemenyi statistical test and Deñsar's significance diagram to find the significant differences between the different models. The statistical test considers the rank orders of the $\text{Norm}(P_{opt})$ values. The null hypothesis that the average rank (AR) values for each predictive model are equal and cannot be rejected and a significant level of 5% was used for the test.

3.3.7 Experiment execution environment

Construction and evaluation of the fault density prediction model was done using R [129] which is an open source statistical processing environment. Also, determination of hyper parameters and prediction model involving different data set split of 10-fold cross-validation was implemented using the caret packages [72] that is available in R.

3.4 Results and Discussion

3.4.1 Evaluation of fault density prediction models that take into account testing effort

Table 3.4 shows the experimental results for the cross-validation analysis using the same data set or release. From the table, each row represent the $\text{Norm}(P_{opt})$ values for each data set for a specific prediction model. The AR at the bottom

represents rank averages of $\text{Norm}(P_{opt})$ values on all data sets for a specific technique. The equation (2) shows the AR_i formula for the entire data set k . Here, r_j^i represents the rank order of $\text{Norm}(P_{opt})$ values (descending) when you apply the i th classifier to the j th data set.

$$AR_i = \frac{1}{k} \sum_{j=1}^k r_j^i \quad (3.1)$$

From Table 3.4, the bolded and underlined value in each row represents the highest $\text{Norm}(P_{opt})$ value, which indicates higher accuracy hence, the best performing predictive model for that specific data set. AVG is the average value of $\text{Norm}(P_{opt})$ values for each prediction model on all data sets. In the AR region, the most accurate value is bolded and underlined, which indicates a low (high accuracy) value. For almost all the data sets, $\text{Norm}(P_{opt})$ values for the prediction models are more than 50% and even some models had values greater than 70%. It was observed that different predictive models performed quite well for different data sets, that is no single predictive model had the best performance for all the data sets. However, two models K* and RF performed admirably well (9 and 7 out of 25 with respect to their ranks) whilst RVM and RPART couldn't perform for any of the data set. K* also had the best AVG and AR values of 72.79 and 3.16 respectively.

Furthermore, for the comparison of the prediction models, we followed the experimental framework proposed by Lessmann et al. [75]. Nemenyi test results' was expressed using Deġsar's significance diagram (Figure 2) to statistically compare the relative merits between different models. In the diagram, the classification techniques are listed in ascending order of ranked performance on the y-axis and the average rank (AR) of prediction performance across all 25 data sets are displayed on the x-axis. From Figure 2, the deeply shaded bullet beginning each solid horizontal line represents the AR value for each effort-aware model and the vertical dashed line identifies the end of the best model's tail that is the lowest AR value (lower average of ranked performance) which is K*. It also represents the maximum number of AR values that cannot be rejected by the null hypothesis. The diagram shows that we cannot reject the null hypothesis since none of the AR values (deeply shaded bullet) started after the vertical dashed lines thus there is no significant difference among the models. However, RPART performs worse than the other effort-aware models with an AR value of 8.84. The results implies that prediction performance of all models improved since we couldn't reject all the models. Observing the average (AVG) $\text{Norm}(P_{opt})$ for each predictive model on each data set, the lowest

TABLE 3.4: Prediction performance of each model when testing effort is considered

Data sets	Predictive Models										
	LM	LARS	RVM	k-NN	K*	NNET	SVM	M5	RPART	GBM	RF
CLAM	<u>94.2</u>	93.7	50.1	62.7	93.4	38.9	88.2	92.3	73.8	93.1	93.4
ECOS	58.3	62.6	51.6	49.6	<u>73.3</u>	54.4	46.5	64.4	60.4	65.5	72.3
EXIM	44.3	51.8	24.4	58.7	<u>72.8</u>	23.8	68.1	54.0	24.4	55.6	70.1
FREE	71.4	75.7	61.5	75.0	<u>80.2</u>	68.0	63.4	76.6	59.0	72.1	<u>80.2</u>
GANY	43.2	46.4	49.4	51.4	53.9	<u>68.2</u>	49.2	46.4	45.6	54.9	54.8
HELM	69.9	70.4	32.6	67.6	64.6	30.9	64.5	64.2	36.2	61.8	<u>73.3</u>
HIBE	67.0	<u>76.7</u>	59.6	50.6	74.0	76.1	49.4	56.8	52.6	72.0	71.2
HYLA	46.9	88.3	78.6	74.3	83.1	55.0	70.7	82.7	87.6	76.9	<u>92.5</u>
IPTA	57.8	57.9	65.0	66.3	65.8	<u>72.7</u>	68.8	69.8	58.0	61.9	68.0
KOFF	52.8	60.6	59.3	62.9	61.6	64.6	51.1	67.6	58.2	<u>75.8</u>	62.2
NBSD	67.6	73.7	44.5	64.1	<u>78.4</u>	43.1	64.9	74.6	35.3	71.2	<u>78.4</u>
OBSD	75.3	81.9	29.4	61.2	81.5	51.4	65.4	<u>82.1</u>	25.8	69.3	78.9
OCMS	<u>55.8</u>	53.7	42.2	51.1	51.8	47.8	48.5	54.6	41.9	51.2	45.3
ONMS	56.8	61.0	60.5	46.7	<u>74.0</u>	65.8	48.2	73.5	51.7	72.5	69.0
OTRS	52.0	60.6	<u>70.6</u>	59.2	65.6	65.8	66.7	64.2	70.0	58.0	61.1
POST	47.0	75.4	60.8	70.9	73.3	74.6	64.1	<u>76.1</u>	42.1	67.1	70.9
PROF	71.0	67.4	55.5	60.4	<u>71.1</u>	43.3	67.3	68.8	56.9	68.4	63.0
SAMB	46.6	53.7	42.3	65.9	<u>75.2</u>	68.0	56.7	75.0	41.6	42.2	58.9
SCIL	73.1	70.4	58.0	59.5	76.8	67.9	58.1	76.1	66.5	74.3	<u>78.7</u>
SPRI	73.4	73.0	61.9	66.5	72.7	70.2	59.0	76.1	63.3	74.7	<u>76.7</u>
SQUI	66.5	62.1	37.2	49.6	60.5	38.5	63.9	<u>68.0</u>	37.5	63.5	64.1
WINE	62.9	<u>76.6</u>	50.3	68.2	66.2	53.0	56.1	76.2	48.9	62.8	66.7
XDOC	72.6	79.7	55.7	58.2	<u>84.8</u>	61.4	64.5	82.4	46.8	82.5	82.9
XFRE	51.0	77.6	61.6	75.8	77.1	75.0	67.8	68.5	71.9	75.1	<u>81.2</u>
XORP	84.3	80.8	30.9	72.8	<u>88.1</u>	47.9	67.7	82.4	79.7	83.0	86.0
AVG	62.47	69.27	51.74	61.97	<u>72.79</u>	57.05	61.55	70.94	53.43	68.22	71.99
AR	6.52	4.64	8.80	6.88	<u>3.16</u>	7.00	7.56	3.84	8.84	5.20	3.32

value was more than 50% which agrees with Mende et al's [86] conclusion that inclusion of effort-awareness in prediction models improves their performance from a practical point of view.

3.4.2 Evaluation of fault density prediction model across versions

For fault density prediction model constructed using data set version A to predict fault-prone accuracy of data set version B, the $\text{Norm}(P_{opt})$ values for the cross-release are represented in Table 3.5. The values in each row represents the $\text{Norm}(P_{opt})$ for each predictive model on each data set. For almost all the data sets, $\text{Norm}(P_{opt})$ values for the prediction models are more than 50% and values greater than 60% were observed on some few models with the best performing model for each data set bolded and underlined. Similar to the experimental results of the cross-validation study on a single software version, no single predictive model outperformed the others. However, M5 was the best performing model across all data sets with average rank (AR) value of 3.44 whilst RVM,

TABLE 3.5: Prediction performance considering testing effort for cross-release analysis

data sets	Predictive Models										
	LM	LARS	RVM	K-NN	K*	NNET	SVM	M5	RPART	GBM	RF
CLAM	79.2	94.7	83.3	69.3	88.8	67.1	85.1	90.1	87.9	96.5	94.6
ECOS	58.0	59.8	51.7	55.8	55.6	45.7	52.9	58.6	49.9	53.4	53.1
EXIM	55.1	52.8	56.0	42.1	24.3	83.1	55.2	34.7	56.0	32.5	36.4
FREE	73.2	79.2	66.4	81.1	81.2	81.6	64.2	81.4	69.2	78.2	80.0
GANY	56.7	53.6	50.8	53.5	49.4	54.9	55.5	54.6	51.0	47.5	48.4
HELM	41.7	40.6	48.1	43.7	53.8	58.2	42.4	40.5	48.3	50.2	44.9
HIBE	42.3	49.4	45.4	55.2	54.6	58.5	45.1	58.9	41.7	56.4	53.4
HYLA	73.8	77.6	73.6	68.2	65.8	57.9	70.3	73.5	78.0	76.5	76.7
IPTA	64.3	58.4	52	56.5	72.4	59.6	58.4	60.6	49.6	65.3	64.2
KOFF	50.8	62.5	51.4	59.8	55.1	56.4	53.2	63.9	54.5	56.4	53.6
NBSD	61.4	66.7	46.8	56.8	61.8	42.9	55.1	65.0	37.4	65.8	61.2
OBSD	70.6	74.9	39.6	64.6	69.9	52.8	46.3	74.7	27.1	58.7	61.2
OCMS	76.4	75.7	64.8	63.3	65.7	50.7	68.8	66.9	70.6	68.2	66.5
ONMS	54.7	57.3	48.9	60.1	53.3	55.2	41.9	60.2	47.4	61.3	60.0
OTRS	80.9	88.8	88.4	65.3	86.6	79.7	61.9	85.2	88.5	88.1	78.0
POST	59.6	56.5	59.4	60.3	64.1	62.0	55.6	65.7	48.0	60.6	62.9
PROF	68.5	64.1	58.1	51.8	53.6	58.7	54.5	65.6	60.1	57.5	67.9
SAMB	62.9	68.7	37.2	62.2	68.3	49.4	56.4	69.5	37.3	56.8	69.8
SCIL	58.2	62.7	40.5	55.3	62.5	61.8	52.4	63.3	51.8	61.5	62.4
SPRI	66.3	69.3	49	59.3	60.4	64.1	57.7	69.3	53.9	65.9	68.0
SQUI	64.2	75.4	37.0	54.4	59.5	44.9	56.0	70.7	36.1	70.3	48.6
WINE	64.7	68.7	50.8	65.6	61.2	58.7	60.3	71.8	57.0	68.1	66.7
XDOC	69.4	60.1	34.4	55.8	74.6	37.3	66.1	55.6	43.0	66.0	69.2
XFRE	65.0	63.6	64.1	64.4	73.7	66.3	53.5	74.3	62.7	68.5	66.6
XORP	67.2	80.2	22.8	60.2	81.3	51.2	67.1	85.8	82.1	81.9	82.5
AVG	63.40	66.45	52.82	59.38	63.90	58.35	57.44	66.42	55.56	64.48	63.87
AR	5.44	4.04	8.60	6.88	5.28	6.68	7.92	3.44	7.76	4.76	5.04

SVM and RPART could not perform for any of the data set hence having the lowest average(AVG) Norm(P_{opt}) values or highest AR values.

Comparing the significant differences between the models using Deñsar's significance diagram from Figure 3, there were no significant differences statistically since the vertical dashed line fails to reject all the models.

Comparing the results to that of RQ1 in the case of fault-prone module prediction on single software versions, the overall Norm(P_{opt}) values were low. Furthermore, examining the relationship between the prediction models of RQ1 and RQ2, the correlation coefficient of the average ranks of RQ1 and RQ2 showed a strong positive correlation of 0.87. This implies that for fault-prone module prediction using fault density, utilization of cross-validation methods for the model building in previous versions may help in the selection of the best fault-prone models for cross versions prediction.

3.4.3 Discussions

In light of the analysis conducted for the two RQs, it was observed that not one effort-aware model proved very dominant and was statistically significant

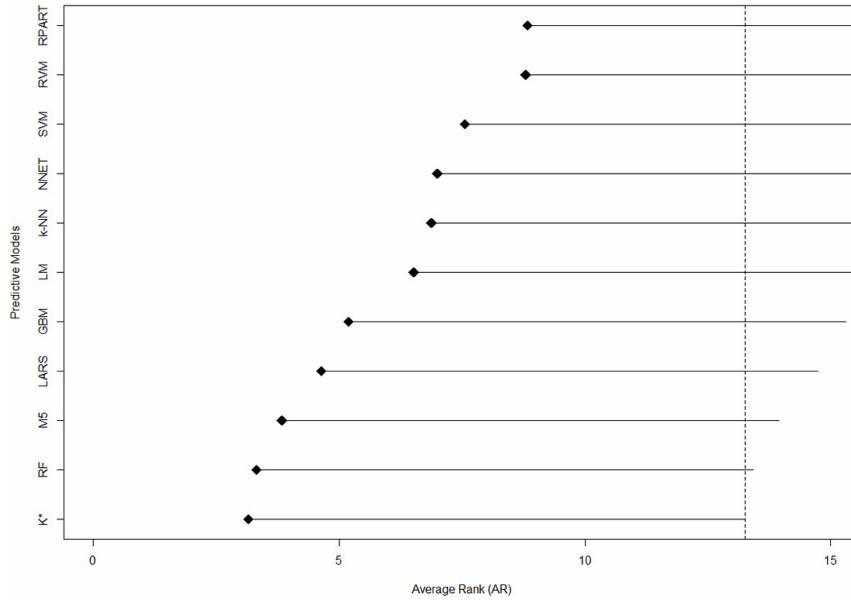


FIGURE 3.1: Deñsar's significance diagram in RQ1

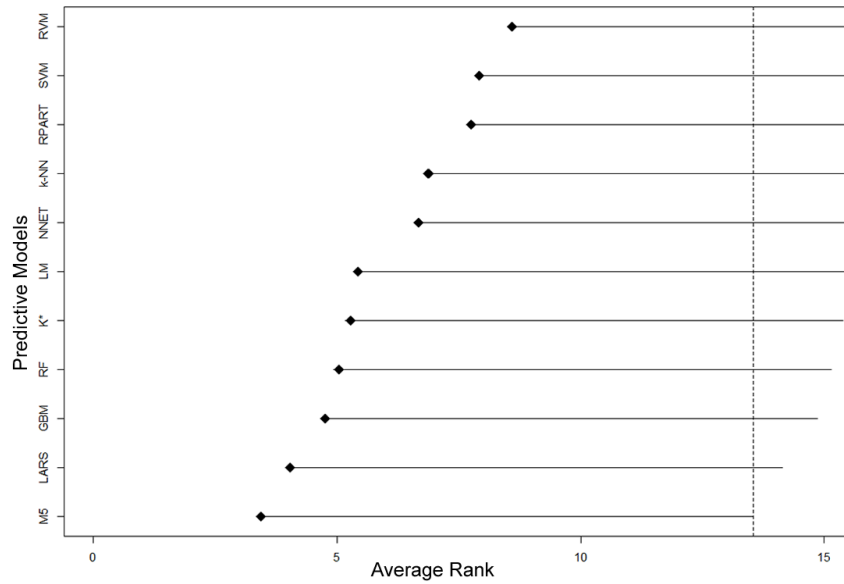


FIGURE 3.2: Deñsar's significance diagram in RQ2

from the others. Also the best prediction model varied according to the data sets in both RQs. We sought to find the external factors causing the oscillation in the results and thus considered the P_{fp} (% of fault-prone modules) in the fit data set for each OSS project. It was evident that the data set size was one key reason for the variation in the performances of the models as was observed from the results presented in Tables IV and V. We next considered the P_{fp} because of the unpredictability of real world data sets. Also, the advent of several sampling techniques provides the means to modify or control the P_{fp} in defect data sets.

For RQ1, a stratification method was employed in the splitting of each data

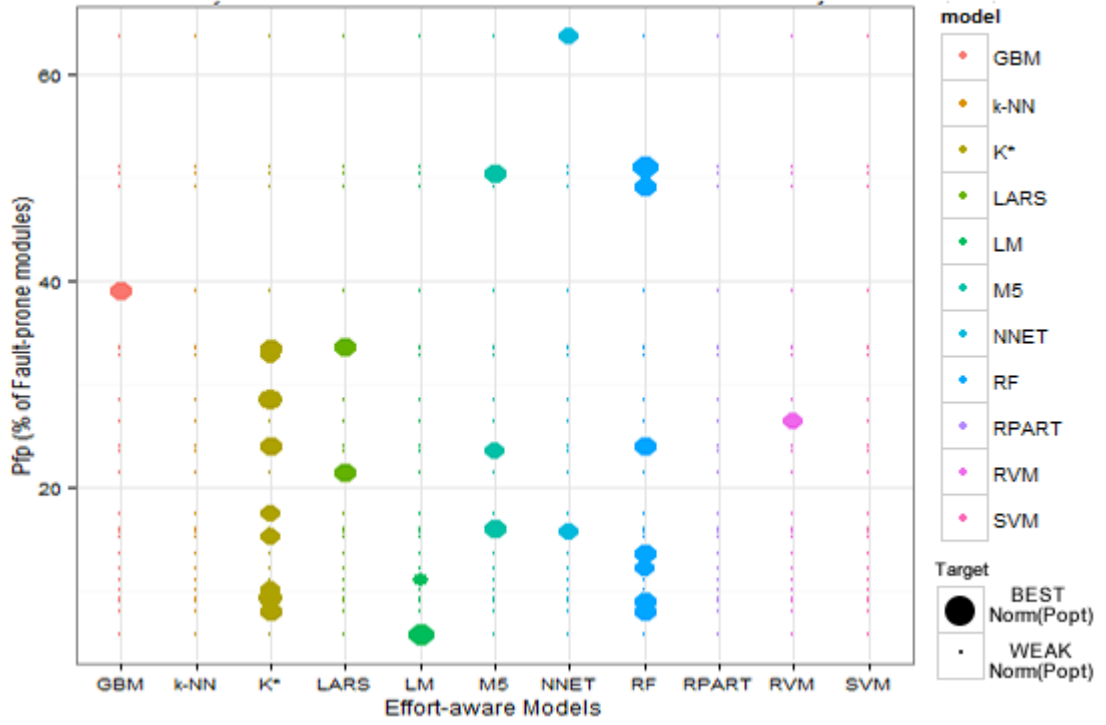


FIGURE 3.3: Prediction performance of each model on the 25 OSS projects sorted according to P_{fp}

set to get the fit and test data sets ensuring that the P_{fp} was same and maintained in both fit and test data sets. We present a bubble chart to show the relationship between the P_{fp} and performance of each effort-aware model. Figure 4 displays a bubble chart where the deeply shaded bubbles represent the highest $\text{Norm}(P_{opt})$ value of a specific prediction model for each data set horizontally which have been sorted according to their P_{fp} on the y-axis.

It could be observed that K^* and RF won in most cases when P_{fp} was below 40%. When P_{fp} was above 40%, RF and M5 performed best and above 60%, NNET performed best. Contrasting results were achieved in RQ2 as presented in Figure 5, it is evident that variation in the P_{fp} of the test data greatly affects performance of effort-aware models. The LARS and M5 models were dominant when P_{fp} was below 40%.

The prediction models having best performance are different in the case of cross validation and that of cross release analysis due to the variation of data set size and P_{fp} modules in each data set. Comparing our results to the results achieved by D'Ambros et al. [28], we do attain similar results when the effort-aware performance measure was used. Factors such as the data set size and class distribution had significant effect on the results, hence not one prediction model could outperform the others. We thus concur with their conclusion that external circumstances do affect prediction performance and thus, limits the

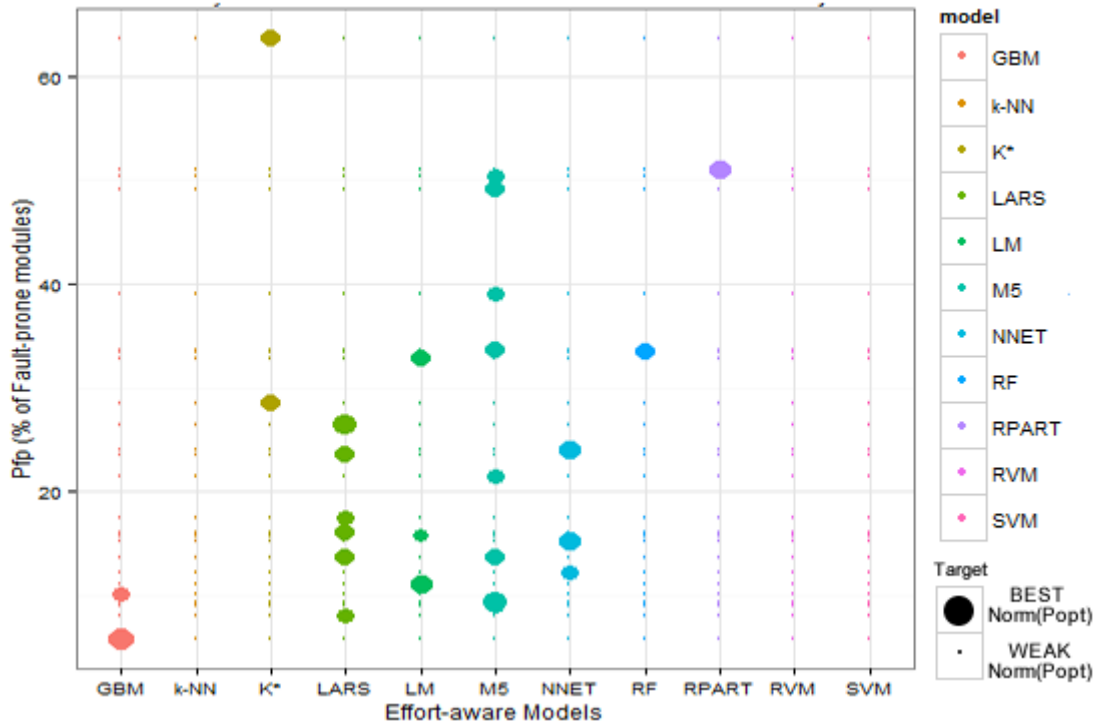


FIGURE 3.4: Cross-release prediction performance of models on the 25 data sets sorted according to P_{fp}

generalization of our results to other contexts. We however, recommend that in using effort-aware models, K* model should be considered if the P_{fp} is the same in both train and test data set which is less likely in a real world data set hence M5 could be considered since it performed well for test data sets of varying P_{fp} values in both RQ's.

3.5 Threats to Validity

In the conduct of our experiments, several potential threats which could undermine our results are discussed in this section. The data sets used are a possible source of bias in relation to the representativeness and measurement accuracy to aid in the generalization of our conclusion results. Since we considered purely OSS data sets, it may be difficult to generalize the results to closed source projects or the industrial settings. Also, relying solely on faults recorded in the commit logs from version control tools such as CVS and GIT implies that faults not recorded in such tools would be omitted from our study. Considering the keywords used to find the faults in our study, likely faults which were not labeled as "bugfix" or "fix" would also be excluded. Further analysis is required to accurately ascertain and collect all faults.

The sampling and splitting techniques could also affect our results. We randomly split the data sets into fit (2/3) and test (1/3) for RQ1. This procedure has been well applied in several empirical experiments and thus an established approached to predictive data modelling [127]. Also, the large size of each data set complements the splitting procedure. The selection of the hyper parameters could also affect our results since the values were estimated using a 10-fold cross-validation method.

Finally, the predictive models used for this experiment may not be exhaustive of all models available. With the availability of several models, other models which could have performed better were not considered. However, the selection was made with the aim of finding a meaningful balance between the several categories of techniques (machine leaning, statistical, data mining and others). We thus believe that, the models applied are representative of the most important techniques found in literature.

3.6 Related Work

3.6.1 Evaluation of models that take into account the testing effort

Arisholm et al. [4] stipulated the need of evaluating predictive models by assessing their potential cost-effectiveness and not only by considering their accuracy if the economic viability and benefits of these fault-proneness prediction models are to be realised. A study by Mende et al. [86] showed an example of module size used to evaluate a fault-prone module prediction model considering the cost-effectiveness of applying the model. They proposed bug prediction models that are effort-aware (effort for review or code testing) and considered number of lines of code of the module to be proportional to the testing effort which were evaluated with the prediction accuracy of each model. Further, Kamei et al. [58] performed a fault-prone module prediction by extending Mende et al.'s method and observed very interesting findings. For example, the process metrics when the testing effort was considered showed that it is possible to construct a good predictive model than product metrics. D'Ambros et al. [28] extensively experimented and evaluated several prediction models using traditional and effort-aware performance measures considering different software metrics. They conclude that some approaches performed better than others statistically but external validity limits the generalization of results to different contexts.

Therefore, among the various evaluation measures (such as precision/recall/F-measure/ROC-Curve), the normalised P_{opt} proposed by Kamei et al. [58] is a more realistic performance evaluation of predictive models due to its ability to consider testing effort. Monden et al. [94] also applied normalised P_{opt} with the assumption that effort is not distributed uniformly across all modules since some modules will be more expensive to test or review than others. The normalised P_{opt} was further discussed in Section III.

In this chapter, considering the testing effort (module scale) to compare the performance of the prediction model, 25 distinct projects were used in the experiment as a way of increased number of projects.

3.6.2 Evaluation of models that span the version

Traditionally, cross-validation as an evaluation method of fault-prone module prediction model (N-fold cross validation) has been widely used [75], [86], [95]. Menzies et al. [89] argued that for a model to have useful future validity, the learned theory should be tested on data which was never used in building the model since failure to do that could result in over-estimating the learned model. More recently, evaluation of prediction model's performance has been carried out using multiple versions [58], [42], [98]. For example, Kamei et al. [58] constructed a model from modules of some specific old versions of three open source software types.

In this chapter, for a more realistic evaluation performance, we built a prediction model using the historical version of the data and evaluated its predictive performance when applied to the next version.

3.7 Chapter Summary

In this chapter, we empirically evaluate the best performing fault-prone prediction model when testing effort is being considered using two proposed research questions (RQs). Based on 25 open source projects, we extensively experimented and found the best models by evaluating them using the normalized P_{opt} because it considers testing efforts which helps in finding the cost of additional quality assurance activities. For RQ1, we compared 11 different kinds of fault prediction models on the data sets of same version. Among all the models used, K^* was the best performing model. As a result of comparison with the other techniques to K^* , lower differences was observed statistically to the remaining techniques but not significant. In RQ2, 11 types of fault density

prediction models were compared using previous software versions as fit data set and current (new) version as test data set. As expected, the $\text{Norm}(P_{opt})$ values dropped in comparison to the cross validation method for the same software version (RQ1) and the M5 tree model had the highest value between versions but no significant differences were also observed statistically in relation to the other models. Two different models performed well for both research questions and this could be explained by the different data sets used for testing in both instances. Also, the data set size and percentage of faulty modules present in the data set greatly affected the performance of the prediction models. This is in concordance to what was observed by [75] that the distribution of target variable and size of the data set (fit and test) has an effect on the performance measure. Further analysis of the average $\text{Norm}(P_{opt})$ values for both RQs shows that prediction performance of all models considered had an overall average performance value of more than 50%.

Also, it was observed that there was a strong correlation between average rank of the prediction accuracies for both the intra-release and cross-release versions when cross-validation method was employed. Therefore, it is possible to make use of cross-validation method in previous versions which may be useful for model building techniques for fault density prediction across versions.

From these results, an appropriate scenario of applying fault prediction models in a real development environment has been shown in this paper.

3.8 Concluding Remarks

The objective of this chapter is to find the best performing fault prediction model. Previous studies have been unable to conclude on the best fault prediction model due to three major drawbacks: (1) comparison of few fault prediction models considering small number of data sets, (2) use of evaluation measures that ignore testing efforts and (3) use of n-fold cross validation instead of the more practical cross-release validation. To address these concerns, we conduct cross-release evaluation of 11 fault density prediction models using data sets collected from 2 releases of 25 open source software projects with an effort-aware performance measure known as $\text{Norm}(P_{opt})$. Our result shows that, whilst M5 and K* has the best performances; they are greatly influenced by the percentage of faulty modules present (class imbalance of dataset) and size of data set. Using $\text{Norm}(P_{opt})$ produced an overall average performance of more than 50% across all the selected models clearly indicating the importance of considering testing efforts in building fault-prone prediction models. We discuss in the

next chapter some related works that assessed the impact of class imbalance on prediction performance and the missing gaps in these studies. Subsequent chapters will discuss further empirical evidence on the impact of class imbalance and the most appropriate environment for deriving higher performances when data sampling approaches are applied during model training.

Chapter 4

Related Works

4.1 Introduction

Class imbalance and its effect on the performance of software defect prediction models has received little attention until recently. Approaches proposed to handle the class imbalance issue could be categorised into three. They are: (a) Resampling the original datasets (data resampling approaches) by rebalancing the distribution of the class [22, 71], (b) Assigning specific cost to the classes to reduce classification cost [79, 123, 146] and (c) applying ensemble learning techniques [73, 143]. Data resampling approaches are however more prevalent in literature because it is not dependent on the prediction model used [59].

Prior work has examined the effect of resampling methods on performance of prediction models by considering few resampling methods, single performance measure with no statistical evaluations and ignoring the impact of the distribution parameter of the sampling methods. In this chapter, summary discussions on the application and impact of data resampling approaches in software defect prediction studies is presented. The related works are organized into comparative studies of several resampling methods, impact of the amount of resampling on prediction performances, evaluation criteria and its effect on the prediction performance.

4.2 Software Defect Prediction

Defect predictors have been used to improve the quality of software and reduce the overhead cost of producing software. Several machine learning, data mining and statistical models have been proposed and evaluated for software defect prediction. Using basic discriminant analysis, Munson and Khoshgof-taar [96] observed that there is a strong useful relationship between software faults and software complexity metrics during development and suggested that

it was possible to use predictive models to help determine program faults. Afterwards, several conventional methods of constructing fault-prone module prediction models have been proposed [7, 16, 60, 102, 121]. For example, logistic regression analysis has been used in many studies to determine the given value of a set of explanatory variables (set of metrics values measured from the module) as demonstrated by Basili et al [7].

Modeling methods such as linear discriminant analysis [102], classification tree [61], neural network [60], support vector machine [144], Bayesian classifier [103], k-neighbor classifier [134], have also been used. Recent studies [75],[94] have recommended the Random Forest model because it is fast to train, less complex and more robust in relation to the parameter settings.

The main goal of existing research has been to increase the prediction performance. Several methods have been used to achieve this, such as considering the modeling methods, using robust feature selection and preprocessing techniques and proposing new defect metrics. Modification of the class distribution using data re-sampling approaches is another method of improving prediction performance. This approach is widely gaining recognition because it is independent of the prediction model and its effects are clearly observable. In the following sections, we consider studies rectifying the imbalanced nature of the defect datasets to improve the prediction performance.

4.3 Approaches for Tackling Class Imbalanced Defect Datasets Learning

The application of sampling methods has been shown to be more successful and dominant in recent studies. The data sampling techniques focus on balancing the majority and minority class distribution samples by either increasing the minority class, which is known as over-sampling, or by reducing the majority class, which is known as under-sampling, or by effectively combining both methods. The simplest and most common form of under-sampling is Random Under-Sampling (RUS) [50] in which the majority data samples are randomly removed through deletion. Similarly, Random Over-Sampling (ROS) randomly duplicates the minority samples so as to increase the minority samples. However, ROS adds no further or new information to the classifier because the datasets consist of duplicates and therefore lead to over-fitting [38]. An improved method proposed by Chawla et al. [22] known as Synthetic Minority Over-sampling TEchnique (SMOTE), increases the minority class samples

by generating new synthetic samples that contribute vital information to the dataset. This method creates new instances along a line segment that joins each instance and some defined k minority class nearest neighbors instances. Several synthetic instances are thus created but depending on the degree of oversampling required, the number of synthetic samples are randomly selected.

Several other modifications of the SMOTE algorithm have been proposed [6, 23, 44, 48]. Han et al. [44] observed that data instances on the borderline of the classification boundary were the most likely to be misclassified and proposed the Borderline-SMOTE method, which generates synthetic data along the line separating the minority and majority class samples in a bid to strengthen the minority samples found on the decision border. Another form of the SMOTE is the Adaptive synthetic sampling approach (ADASYN) proposed by He et al. [48]. They consider the classification boundary and apply a weighted distribution method that assigns weights depending on the learning attributes of the minority class samples thus differentiating the harder-to-classify minority class samples from the easier-to-learn minority class samples.

Other sampling methods use clustering algorithms to partition the dataset before under or oversampling the dataset. The majority weighted minority oversampling technique (Mwmote) proposed by Barua et al. [6] partitions the datasets using clustering before using the Euclidean distance similarity measure to find very close class samples and synthetically generate samples based on the weights assigned to the minority class samples.

Several studies prefer over-sampling to under-sampling [38, 50, 117]. Because under-sampling removes the class samples, information crucial for building an effective predictive model is lost.

Observation 1: With the abundance of several resampling approaches proposed in literature, little is known about the most robust and high performing data sampling approach for software defect prediction studies

4.4 Comparative studies of data resampling approaches in the Literature

The problem of class imbalance makes it challenging to conduct experiments with datasets in the field of software defect prediction. Few studies have considered the effects of class imbalance and how to alleviate this issue in software defect prediction. Kamei et al. [59], were the first to investigate the effects

of resampling methods on performance of defect prediction models. They empirically evaluated four resampling methods (random over sampling, synthetic minority over sampling, random under sampling and one-sided selection) assessed with four defect prediction models-Linear Discriminant Analysis (LDA), Logistic Regression Analysis(LRA), Neural Network (NNET) and Classification Tree (CT) on two sets of an industry legacy software. Their results showed that prediction performance of LDA and LRA were improved when the four resampling methods were applied to the data prior to model training. However, the performance of two other models - NNET and CT were not improved. The authors also observed an inconclusive result regarding the best resampling approach and the rate of resampling required to use for defect prediction. A major limitation of this work was the use of only a single dataset that is not publicly available.

Pelayo and Dick [106] applied the SMOTE method to four NASA datasets extracted from the PROMISE repository and assessed the performance with the C4.5 decision tree model. Using the geometric mean performance measure, they observed an improvement of at least 23% on all four datasets. Similarly, Riquelme et al. [113] also evaluated the performance of SMOTE and Random oversampling using the Naive Bayes and C4.5 models on five datasets extracted from PROMISE repository. Their results showed that the AUC performance measure is improved when the SMOTE method is applied to the datasets before training. Contrary to previous studies, a recent study by Shatnawi [118] showed that undersampling and SMOTE do not improve the AUC performance of prediction models. Menzies et al [91] observed that resampling methods increase the information content of a training data which consequently improves the performance of prediction models. Although these studies indicate that resampling of defect datasets is important for better performance of prediction models, they fail to indicate the amount of resampling required. Secondly, they all validated their experiments on ten or less datasets and two or less prediction models. Several prediction models and datasets exist. Lastly, they mostly adopted the SMOTE method whilst there exist several other resampling methods in literature.

Other comparative studies incorporated ensemble and boosting techniques. Sun et al. [124] proposed an ensemble method based on coding schemes that transforms the binary-class into a multi-class dataset before training a model. Comparing their method to other conventional resampling methods on 14 NASA datasets and four prediction models, the performance of C4.5, Ripper and Random Forest were improved with their ensemble method. Wang and Yao [135]

conducted an in-depth investigation of how class imbalance learning methods could improve software defect prediction performance. By comparing class imbalance learning methods of three main types (undersampling, threshold-moving and boosting-based ensembles) with two popular base classifiers (Naive Bayes and Random Forest) on ten PROMISE datasets, their results showed that AdaBoost.NC, an ensemble method had the best performance in terms of the G-mean and AUC measures.

Kitchenham et al. [64] in their recent study noted that most empirical findings in software engineering studies were not conclusive and practical because these studies fail to perform robust statistical tests and effect size computations. Robust statistical tests can aid in appropriately concluding on the efficiency and performance of several methods applied on software prediction and estimation models. Kitchenham et al. [64] argue that these robust statistical test are very necessary given the fact that most real world data for software engineering experiments are either skewed, heavy-tailed or have unstable variances and could heavily impact the results produced from these datasets. The authors recommended non-parametric methods such as Brunner's test for statistical significance and Cliff's delta for effect size computation.

Observation 2: Previous comparative studies consider very few sampling methods for their experiments and failed to perform statistical tests and compute effect sizes to support their results and conclusions (e.g. [59, 106, 113]) given that the results obtained could be heavily affected by the unstable variances and skewed class distribution of the datasets used as reported by Kitchenham et al. [64].

4.5 Evaluation measures of Prediction models trained on Imbalanced data

Extensive studies by Shepperd and Kadoda[119] and Jiang et al.[51] concluded that experimental conditions such as different performance measures and datasets used does affect the performances of prediction models. Performance of prediction models have been predominantly evaluated using Accuracy [28, 100]. Menzies et al. [88] and Chawla et al. [21] however, argue that evaluating predictive models trained on class imbalanced dataset with accuracy is unsuitable as a result of the differing cost of error for the classes in the dataset or misclassification cost. Misclassification cost is very crucial and critical during software testing

therefore using accuracy will be devastating to the investigator or business. For example, a usual software fault dataset could contain 95% bug free modules and 5% buggy modules. A model trained on this dataset could produce a 95% accuracy for the bug free modules. However, this predictive accuracy would be relevant if it reflects a higher rate of the minority modules (faulty) and allows a small error rate for majority bug free modules. Performance measures such as Receiver Operating Characteristics (ROC) curve, Area Under the ROC Curve (AUC), F-measure, Recall, Precision and G-mean have thus been considered for evaluating models trained on imbalanced datasets in several studies [28, 59, 106, 108, 113, 135], because these measures in one way or another manages the misclassification error rates. However, for a fault prediction model to be beneficial in an industrial setting, a software quality ranking system provides such benefits. Gray et al. [40] reports that using a software quality ranking system, an ordered list of the most likely error-prone modules could be produced. In effect, special consideration could be given to such modules in the form of detailed inspection (effort) in a descending order of level of defectiveness. Arisholm et al. [4] also suggests that evaluation and comparison of fault-proneness prediction models should not only consider their prediction accuracy but also take into consideration the potential cost-effectiveness of applying such models.

Observation 3: Evaluatuion of the performance of sampling techniques with more practical measures is unknown. Effort-aware measures that consider the effort needed to fix faults is necessary.

4.6 Parameter Configuration of sampling methods

Several studies have assessed the impact of sampling methods on defect prediction performance. There are however contradictory findings about the performance of these sampling methods especially concerning SMOTE.

Given that the distribution of the training data could impact prediction performance, most comparative studies [59, 106, 135] focus on applying or proposing data resampling approaches without considering the amount of resampling or percentage of fault-prone modules (P_{fp}) actually required. P_{fp} which controls the amount of data added or deleted from the minority class samples respectively is a constant parameter in all sampling methods.

A recent study by Agrawal and Menzies [1] confirmed that the performance of sampling methods specifically SMOTE depends on the default distribution of the dataset being used for training the prediction models and the default parameters of the sampling method. They concluded that tuning the parameters of SMOTE depending on the distribution of the dataset provides significant AUC improvement over the use of default parameters of SMOTE.

The amount of resampling (*Pfp*) applied is an important factor in determining the performance of a defect prediction model on a resampled dataset [32, 138]. To maximize overall classification accuracy, most conventional prediction models assume the two classes in the datasets to be as balanced as possible or equal [139, 145]. However, a balanced dataset may not result in the best prediction performance. Further analysis on the best resampling rate for data resampling approaches is very necessary.

Observation 4: Little is known about the most appropriate data sampling rate which results in best or highest performance for software defect prediction studies.

4.7 Chapter Summary

In this chapter, a brief survey of previous comparative studies related to the application of data sampling approaches and defect prediction is conducted. We observe that most studies apply data sampling approaches without any robust statistical analysis of the results. Also, most comparative studies consider very few data sampling approaches with few performance measures. These studies overlook the impact of the resampling rate on prediction performance. Lastly, it is still unknown as to the best and most efficient data sampling approach for defect prediction studies. We describe the empirical studies conducted to tackle the missing gaps found in our literature in the following chapters of this thesis. The next chapter describes the impact of conventional data sampling approaches on the performance of prediction models evaluated with an effort-aware measure.

Chapter 5

Assessing the Impacts of Balanced Training and Testing Datasets on Effort-Aware Fault Prediction Models

5.1 Introduction

To prioritize software quality assurance efforts, fault prediction models have been proposed to distinguish faulty modules from clean modules. The performances of such models are often biased due to the skewness or class imbalance of the datasets considered. Several methods have been proposed in machine learning for dealing with the class imbalanced issue such as random over and under sampling [22, 32] creating synthetic data [44, 48], application of cleaning techniques for data sampling [131] and cluster-based sampling [52]. With a significant amount of literature in machine learning for imbalanced datasets, very few studies have tackled it in the area of fault prediction. The first of such studies by Kamei et al. [59] showed that sampling techniques improved the prediction performance of linear and logistics models whilst other two models (neural network and classification tree) did not have a better performance upon application of the sampling techniques.

Interestingly, sampling techniques applied to datasets during fault prediction are mostly evaluated in terms of Accuracy, AUC, F1-measure, Geometric Mean Accuracy [28, 59, 106, 108, 113, 135] just to name a few; however, these measures ignore the effort needed to fix faults, i.e., they do not distinguish between a predicted fault in a small module and a predicted fault in a large module. Nickerson et al [101] concludes that to evaluate the performance of classifiers on imbalanced datasets, accuracy or its inverse error rate should never be used.

Chawla et al. [21] also alludes to the conclusion that simple predictive accuracy might not be appropriate for an imbalanced dataset.

The goal of this chapter is to improve the prediction performance of fault-prone module prediction models, applying over and under sampling approaches to rebalance number of fault-prone modules and non-fault-prone modules in the training dataset and to find the appropriate distribution or proportion of faulty and non-faulty modules that results in the best performance. The experiment focuses on the use of $\text{Norm}(P_{opt})$, which is an effort-aware measure proposed by Kamei et al. [58] to evaluate the effect of over/under sampling on prediction models to find out if the over/under sampling is still effective in a more realistic setting.

Ten datasets are extracted from sampled open-source softwares (OSS) datasets with various degrees of sizes and distribution of fault-prone modules and 10 effort-aware prediction models are constructed considering different P_{fp} (percentage of fault-prone modules) values for 4 sampling techniques. Ideally, a balanced dataset should have $P_{fp} = 50\%$ but this might not be the best choice for each model and as such eight (8) P_{fp} values of (default, 20, 30, 40, 50, 60, 70 and 80)% are applied. In total, we conducted 800 or $10 \times 10 \times 8$ experiments for each of the four sampling techniques and compared the performance of these models to identify the most influential sampling approaches and required P_{fp} value for effort-aware fault prediction model. Our results shows that over-sampling with a P_{fp} of 20% to 30% improved the performances for all effort-aware models but the Relevance Vector Machine (RVM) model. We thus recommend over-sampling with P_{fp} between (20-30)% but caution should be exercised when applying it for use on RVM models since over-sampling by replication does not present new information to aid in re-estimating the RVM's parameters.

5.1.1 Chapter Organization

The organization of this chapter is as follows. Section 5.2 reviews the most closely related work in connection with imbalanced dataset and sampling techniques and discusses the background of effort-aware and evaluation of models on imbalanced datasets. Section 5.3 presents a discussion on our research objectives. In Section 5.4, we describe the experimental methodology and performance evaluation. The results are presented in Section 5.5 and a discussion on threats to validity and future work of our study is in Section 5.6. Finally, a summary of the overall work is presented in section 5.7.

5.2 Background and Related Work

This section introduces the class imbalance learning problem and current research in this area of software engineering. We further discuss effort-aware models and evaluation measures for class imbalance and effort-aware models.

5.2.1 Class Imbalance learning

The class imbalance issue has been tackled by the research communities in two ways. Reducing the classification cost by assigning distinct costs to fit dataset [29, 105] and resampling the original dataset by over-sampling the minority class [22] or under-sampling the majority class [52, 71]. Over-sampling refers to the process of increasing the number of minority cases in a dataset. Several methods such as Random Over Sampling (ROS), Synthetic Minority Over Sampling Techniques (SMOTE), ADASYN [48] and MWMOTE [6] have been proposed. Oversampling is known to cause overfitting [124] hence care is taken when applying such techniques. Under sampling on the hand is the converse of over sampling whereby we reduce the majority cases in a dataset by deletion. Random Under Sampling (RUS) and One-Sided Selection (OSS) are some techniques available for under sampling. Chawla et al. [21] argues that removal of repeated data points could substantially destabilize the different class distributions. This may therefore boost or reduce classifier performance due to the class imbalance problem.

In the area of software fault prediction, learning from imbalanced fault datasets has received little attention. Kamei et al. [59], the first to apply sampling techniques to fault-proneness models experimentally evaluated the effects of four sampling techniques (ROS, SMOTE, RUS, OSS) applied to four fault-proneness models (Linear Discriminant Analysis-LDA, Logistic Regression-LR, Neural Networks-NN and Classification Tree-CT). Using two module sets of an industry legacy software and F1-Values as accuracy measure, all four sampling techniques improved the LR and LDA whilst the contrary was true for the NN and CT. They concluded that, the most appropriate sampling technique varied among the models.

Pelayo and Dick [106] applied SMOTE algorithm to determine if classification performance could be improved for defect-prone modules using the C4.5 decision tree model and the cost of misclassification. Using the average geometric mean classification accuracy as the performance indicator, they observed an improvement of at least 23% on four datasets. Riquelme et al. [113] analyses two balancing techniques with two predictive models on five datasets from

the PROMISE repository aimed at improving the performance measure. They conclude that the balancing techniques may not necessarily improve the percentage of correctly classified instances, they however improve the AUC measure of classifying better buggy instances which are of more interest to the researcher.

Wang and Yao [135] investigated how the class imbalance learning methods improves software fault prediction. They applied three class imbalance learning methods of three main types (undersampling, threshold-moving and boosting-based ensembles) and compared with two popular base classifiers (Naive Bayes and Random Forest) on ten datasets from the PROMISE repository. Measuring classification performance using balance, G-mean and AUC, Adaboost.NC- an ensemble predictive model, produced the best overall performance.

All of these studies considered datasets and fault prediction models of five or less. The study in this chapter extensively applies four sampling techniques to ten open source datasets using the Normalised P_{opt} , an effort-aware evaluation measure to assess the performance and effects of sampling approaches on ten fault prediction models. It also aims at finding the appropriate class distribution for training datasets during application of sampling techniques on effort-aware defect prediction models.

5.2.2 Effort-Aware Classifiers

Fault prediction, which is a method of predicting the number of defects in a software is valuable in organizing a project's scarce test resources. Given limited resources for software testing, fault predictors could focus test engineers on the modules most likely to be faulty [3, 133]. For predictive models to be practical in an industrial setting, effort aware models have been proposed [3]. The intuition behind the effort-aware models is that time spent on reviewing a potentially buggy file is dependent on the file size and complexity, therefore in discovering bugs, less effort is required for small modules than for large modules [4]. The effort required to inspect a file or module during fault prediction has thus been considered in several recent studies. In a study by Mende and Koschke [87], authors proposed an effort-aware evaluation measure and considered lines of code as a proxy for effort. Kamei et al. [58] revisited common bug prediction findings using effort-aware performance measure, which was an upgrade on Mende and Koschke's performance measure and discovered that process metrics outperformed product metrics when testing effort was considered. Focusing on predicting fault-prone parts of large legacy system, Arisholm and Briand [3] assessed the cost-effectiveness of using an effort-aware logistic model considering the verification effort to predict the fault-proneness of the legacy system.

Similarly, Menzies et al. [90] considered effort during the fault prediction construction and found that learners which considered effort out-performed other simple learners.

5.3 Research Questions

We presume effort-aware prediction models will be improved if the suitable sampling technique together with the appropriate class distribution is chosen for an imbalanced fault dataset. We thus embark on an experimentation to compare different effort-aware model performances when sampling techniques are applied to the training dataset. The objectives of this paper are to empirically demonstrate the effect (positive and negative) of sampling techniques on the performance of software effort-aware fault prediction models and the effectiveness of having the same class proportions in both the training and testing dataset. To achieve the objectives as mentioned above, three research questions are stated:

RQ1. Is sampling approach effective on effort-aware prediction models? Several sampling approaches have been proposed with each model claiming to be more effective than another. In the domain of fault prediction, we sought to find which sampling approach has greater influence on effort-aware models.

RQ2. Could the right distribution of faulty and clean classes be identified during sampling for fault prediction? The search for the right distribution is a fascinating and prevalent problem which emerges during class imbalance learning. Intuitively, it may seem reasonable that making the dataset as balanced as possible would produce the best performance but that may not always be the case.

RQ3. Should the resampled training dataset have same class distribution as the testing dataset and which effort-aware models are more affected by imbalanced distributions? Menzies et al. [92] reports that resampling could produce a training set that contains classes of buggy/non-buggy = 1, although the test set still retains the original distribution. It may seem reasonable that training a predictive model on a dataset whose class distribution has been tuned on the testing dataset's distribution or similar to that of the test dataset should perform better. We explore this domain to investigate the role a "balanced" testing dataset has on a predictive model.

5.4 Experimental Setting

We begin this section by describing the datasets used in the study. We conducted a large study with 10 datasets which are publicly available. Sampling techniques, model construction and evaluation were executed using the open source statistical processing tool R [129]. The library packages Caret [72] and Unbalanced [109] were also used.

5.4.1 Datasets

We experiment on 10 different OSS project datasets available online to ensure the reproducibility of our experiments. These datasets were sampled from the datasets used for the motivating study in Chapter 3. For more details on the datasets, please refer to Chapter 3. With varying degree in size and class distributions, these datasets provide an extensive domain for the investigation of the effects of sampling techniques on the performance of the effort-aware models to be constructed.

5.4.2 Sampling Techniques

For this study, Sampling in this context refers to a preprocessing procedure to correct the imbalanced target classes in a dataset by either increasing or decreasing the data samples before a model could be built for the dataset. We apply four sampling techniques RUS, ROS, SMOTE, PSMOTE at different *percentage of fault prone modules* (P_{fp}) values. P_{fp} refers the total number of minority class divided by the total number of data points in a dataset. The Sampling techniques considered in this study are discussed below.

RUS: Randomly remove clean or majority modules from the data until we reach the desired P_{fp} value whilst the minority modules are kept intact.

ROS: Randomly increase the minority or faulty modules until the desired P_{fp} value is attained. The increase is done by duplicating already existing minority modules.

SMOTE: This approach synthetically over-samples the minority class by considering each class instance and introduces a synthetic example along the line of segments which joins any of the k nearest neighbors [22]. The k nearest neighbor is randomly selected depending on the P_{fp} value required.

Purely SMOTE (PSMOTE): The original SMOTE algorithm oversamples the minority and undersamples the majority in a bid of shifting the initial bias of the learner from the majority class (negative) towards the minority class

TABLE 5.1: Prediction methods and hyperparameters in our experiments

Category	Model	Description	Hyper parameters Used
Statistical approach	LM	Linear regression	No
	LARS	Least Angle Regression [31]	No
	RVM	Relevance Vector Machine [12]	No
Neighborhood Law	K-NN	k neighborhood method [75]	$k=\{1, \dots, 15\}$
	K*	k neighborhood method using distance based on entropy [24]	No
Neural Net	NNET	3-layer neural net [60]	size = $\{4, \dots, 28\}$, decay = $\{0.1, 0.2\}$
The tree model	M5	classification tree [36]	No
	RPART	CART [61]	cp= $\{0.05, 0.1, \dots, 0.7\}$
Collective Learning	GBM	classification using boosting tress [19]	n.tree= $\{ 10, 50, 100, 200 \}$
	RF	Random Forest [41]	mtry= $\{ 10, 50, 100, 200, 250, 500, 1000\}$

(positive) [22]. However, due to the effects of undersampling - removing important data points, PSMOTE applies the algorithm only to oversample the minority whilst keeping the majority classes intact. This is due to the fact that we do not want to lose any important module. When a P_{fp} value is applied to oversample the minority so as to become the majority class, the once majority class is still kept intact whilst the previous minority class is oversampled until it becomes the majority class.

5.4.3 Experimental Method

To answer the research questions raised above, two sets of empirical experiments are conducted. We compare and examine the effects of four different sampling techniques at different P_{fp} values on ten effort-aware prediction models (Table 5.1) using the procedure in Algorithm 1 for experiment 1. We furthermore investigate the influence a balanced testing dataset has on the performance on an effort-aware model after resampling the training datasets in experiment 2. The fault density for each module is predicted and not the presence or absence of defects. This produces an ordered list of modules and as noted by D'Ambros et al. [27], it helps in resource allocation.

Algorithm 1 Outline of Experiment 1

Given a training set comprised of clean(c) and buggy(b) data, $W_{train}(\text{Version A}) = (W_c, W_b)$, $|W_b| < |W_c|$

and $Z_{test}(\text{Version B})$ be set of testing data set

Procedure Begin

1. For training dataset $W_{train}\{W_1, \dots, W_n\}$ and $Z_{test}\{Z_1, \dots, Z_n\}$ **repeat**
2. Initialize $\text{Norm}(\text{Popt}) = \emptyset$; $\text{samp}=[\text{ROS}, \text{RUS}, \text{SMOTE}, \text{PSMOTE}]$;
3. Initialize $pfp=[\text{default}, 20, 30, 40, 50, 60, 70, 80]$; $C=[\text{GBM}, K^*, \text{KNN}, \text{LARS}, \text{LM}, \text{M5}, \text{NNET}, \text{RF}, \text{RPART}, \text{RVM}]$
4. **for** $k = 1, \dots, \text{length}(\text{samp})$ **do**
5. **for** $i = 1, \dots, \text{length}(pfp)$ **do**
6. Generate N balanced training sets from $W_{train} : W_{pfp[1]}, \dots, W_{pfp[n]}$,
 where $W_{pfp[i]} = \text{resample}(W, \text{samp}[k], pfp[i])$;
7. Train classifier C_i on modified data $W_{pfp[i]}$
8. Classify data points in Z_{test} using the classifier C_i
9. Compute: $\text{Norm}(P_{opt})$ for each classifier C_i
10. **end for end for**
11. **until** W_{train} is empty (i.e., the training data is exhausted)

End

We refer to a resampled training dataset as a dataset which has been pre-processed using a sampling technique to an appropriate P_{fp} value. A 10-fold cross-validation method is employed during the model construction on the resampled training datasets. The resampled training dataset is split into ten partitions with nine partitions being used for training and the single one partition used for validation. This is done several times to ensure that the optimal parameters are found and selected after validating them on each fold of data when the default set parameters was used initially for building models. Since all the datasets have P_{fp} values less than 20% by default, we use P_{fp} values of 20% and above during the resampling process to generate seven independent training datasets (train-20, train-30, train-40, train-50, train-60, train-70, train-80). Default parameter configurations are used for the SMOTE algorithm where k-nearest neighbor is five ($k=5$).

Experiment 2 follows the procedure of experiment 1 but makes modification at step 8 in Algorithm 1 where the Z_{test} is modified to reflect the different P_{fp}

values in relation to **RQ3**'s assertion of the effects a balanced testing dataset could have on the performance of predictive models. We consider five datasets and five effort-aware models based on their high $\text{Norm}(P_{opt})$ performance from experiment 1 and create seven independent testing datasets (test-20, test-30, test-40, test-50, test-60, test-70, test-80) using the same number of testing data points but varying the class distribution (P_{fp} from 20%-80%) to test the seven trained models constructed for each effort-aware model after applying ROS to the training datasets (train-20, train-30, train-40, train-50, train-60, train-70, train-80) in Algorithm 1.

5.5 Empirical Results and Discussion

RQ1 presents the comparison of performances of sampling techniques and its effects on the models. **RQ2** closely investigates the best proportion of class distribution to be used during sampling and **RQ3** examines the effect of a balanced testing dataset on an effort-aware fault prediction model.

RQ1. *Is sampling approach effective on effort-aware fault prediction models?*

We compare the performance of each sampling technique on the effort-aware models to the performance of the effort-aware models when no sampling technique is applied to the dataset which is denoted as NOS. Figure 2 shows the average $\text{Norm}(P_{opt})$ evaluation measure across all ten datasets for each sampling technique on the effort-aware fault prediction models. Effort-aware prediction results on individual datasets are not displayed due to space restrictions. From the plots, we observe the performance of the sampling techniques and using no sampling (NOS) as a threshold, it is evident only ROS performed relatively better. The other three sampling techniques negatively affected the performances of the models. All the models considered in this study were clearly induced or affected by the sampling approaches. ROS performs better in all the models except RVM compared to SMOTE and this could be explained by the fact that ROS merely duplicates the instances instead of introducing new information to the classifier hence the RVM model, a probabilistic model whose functional form is equivalent to the Support Vector Machine (SVM) [12] could only have a decrease in performance because it requires new information to re-estimate its parameters. Study by Wei et al. [136] also confirmed the negative effect of ROS on the performance of SVM and RVM. The over-sampling techniques also had a positive effects on the NNET and RPART models since most of the $\text{Norm}(P_{opt})$ values were above the threshold line. Interestingly, the performance of most

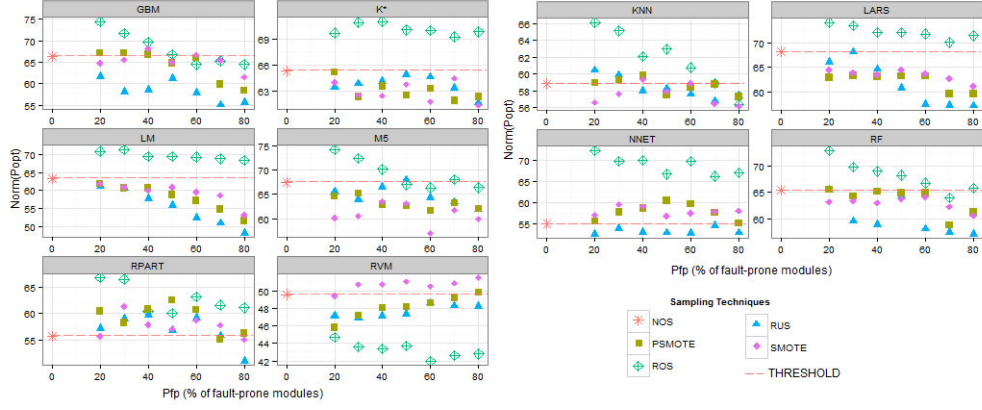


FIGURE 5.1: Average $\text{Norm}(P_{opt})$ performances across all ten datasets for the sampling techniques on different Effort-Aware Models at different P_{fp} values

models when no sampling (NOS) was applied to the technique was better in comparison to the other three techniques (RUS, SMOTE, PSMOTE). Table IV presents the summarised results of the average $\text{Norm}(P_{opt})$ for each model and sampling technique across all the P_{fp} values considered for this study. The first column displays the no sampling approach (NOS) on the models, the last row shows the overall average of each sampling technique across all models and the best $\text{Norm}(P_{opt})$ for each sampling technique is bolded and underlined. Clearly, the over-sampling approaches outperforms the under-sampling approaches with ROS performing best with an average of 65.407 across all models. The SMOTE and PSMOTE achieved very similar $\text{Norm}(P_{opt})$ values. Similarly, NOS also outperforms the under-sampling techniques which implies that under-sampling approach should not be applied alone if the $\text{Norm}(P_{opt})$ evaluation measure is employed but could be considered when combined with an over-sampling approach.

TABLE 5.2: Average $\text{Norm}(P_{opt})$ of Effort-Aware Models among different Sampling Techniques across all P_{fp} values

Models	NOS	RUS	ROS	SMOTE	PSMOTE
GBM	66.34	58.27	<u>68.11</u>	65.93	64.3
K*	65.31	63.72	<u>70.09</u>	63.21	63.06
KNN	58.84	58.32	<u>61.75</u>	57.81	58.56
LARS	68.19	61.54	<u>72.14</u>	63.74	62.06
LM	63.37	55.25	<u>69.72</u>	60.34	57.93
M5	67.44	64.77	<u>69.22</u>	61.02	63.15
NNET	54.97	53.19	<u>68.84</u>	57.97	57.81
RF	65.44	60.07	<u>68.13</u>	63.28	63.52
RPART	55.64	56.93	<u>62.83</u>	58.06	59.13
RVM	49.55	47.64	43.24	<u>50.52</u>	48.08
AVERAGE	61.509	57.97	<u>65.407</u>	60.188	59.76

RQ2. *Could the appropriate distribution of classes be identified during sampling for fault prediction?*

In light of the analysis presented in RQ1, it will be of great essence to examine the effort-aware predictive models and find the appropriate class distribution required during sampling. Examining the results from the plot in Figure 3, it could be observed that performance of sampling techniques on the effort-aware models varied across different P_{fp} values for different sampling techniques. For

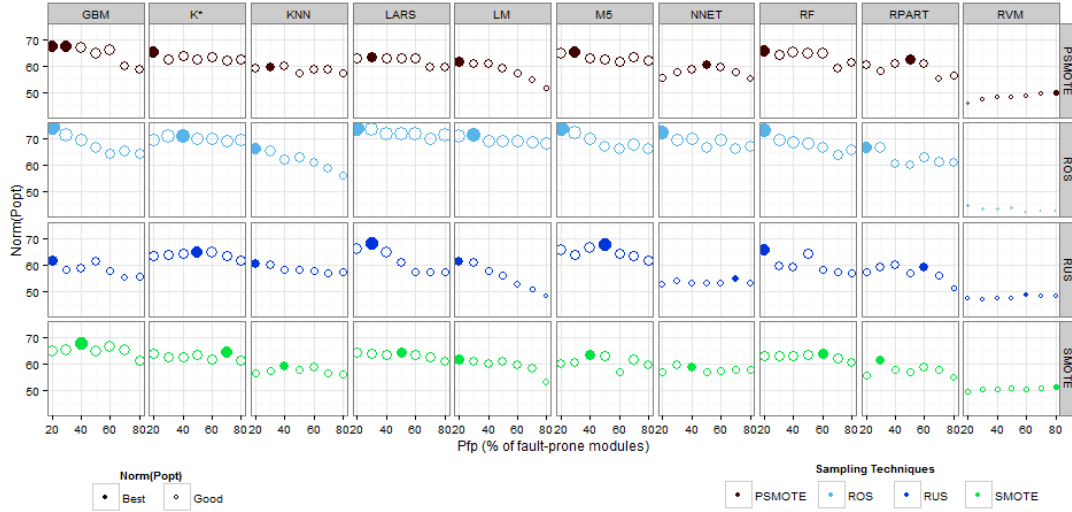


FIGURE 5.2: Average performance of Effort-Aware Models across different P_{fp} values

most of the models, the best performance is achieved during the initial P_{fp} values. For all the models, except LM and K*, ROS performs best when P_{fp} is 20%. Similarly, RUS and PSMOTE attains better performance for most of the models when P_{fp} is either 20% or 30%. From Figure 3, it is evident that at P_{fp} values of 20% and 30%, most models performed best in comparison to the other P_{fp} values. However, for SMOTE, the best performance seems to alternate at different P_{fp} values with four of the models achieving best performances at 40%. This is not unusual since this techniques generate synthetic instances which are new as compared to the ROS which duplicates already existing instances hence the synthetic technique will have a better performance when more are being generated. A stable P_{fp} value for SMOTE could not be attained but majority of the models achieved best performance at P_{fp} values less than 50%. Generally, most models exhibit better performances when the sampling approaches are applied at P_{fp} level less than 50% and thus conclude that most models performs best when sampling is applied at a P_{fp} level of 20-30%.

RQ3. *Should the resampled training dataset have the same class distribution as the testing dataset and which effort-aware models are more affected by imbalanced distributions?*

To find out whether the performance of the effort-aware models trained on the resampled training datasets were influenced by the proportion of the class distribution in the testing dataset, we explored further by conducting experiment 2. For each of the five datasets considered, Figure 4 shows the $\text{Norm}(P_{opt})$ performance of the seven training set models tested by seven different testing datasets for each model and dataset. $\text{Norm}(P_{opt})$ values are displayed on the y-axis and the P_{fp} values of the testing dataset are represented on the x-axis. The plots in Figure 4 reveals an interesting trend in the $\text{Norm}(P_{opt})$ performance achieved by all models.

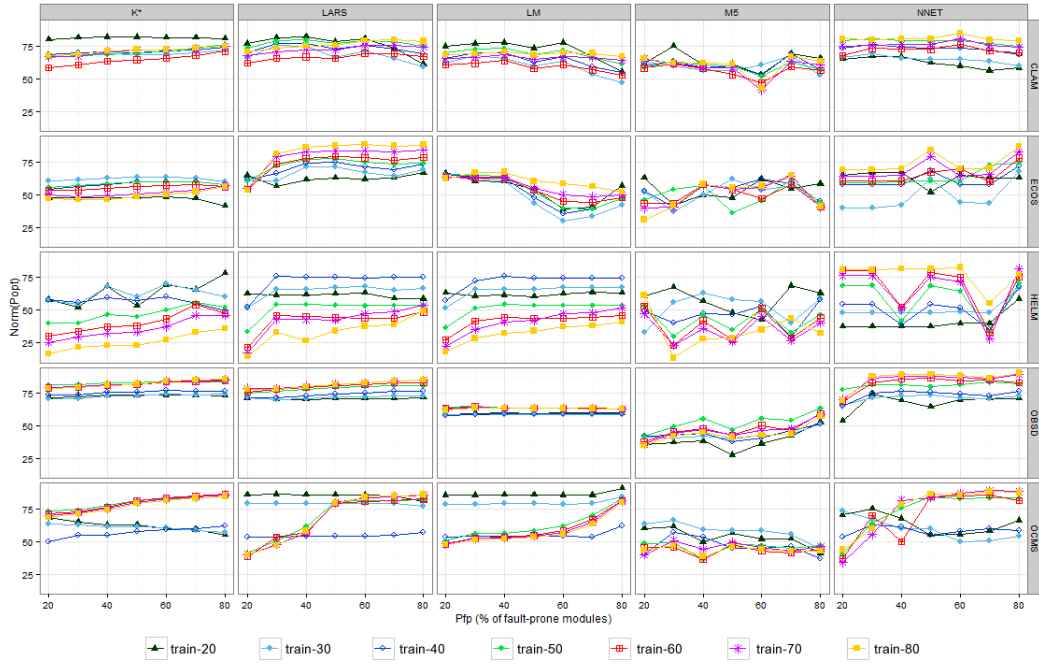


FIGURE 5.3: Effort-Aware Models trained on 7 different datasets (train-20, train-30, train-40, train-50, train-60, train-70 and train-80) tested by seven different testing datasets, increasing from 20% faulty (x-axis=20) to 80% faulty (x-axis=80)

Observing the K^* model, the $\text{Norm}(P_{opt})$ curves for all the datasets except HELM are not significantly slanted, they are almost flat when the percentage of the fault prone modules (P_{fp}) increases from 20% to 80%. On the HELM dataset however, there seems to be a positive trend with four of the models increasing as the P_{fp} values increases in the testing datasets. The model trained with dataset train-20 achieves the best performance when P_{fp} is 80 for the test dataset. LARS and LM models display similar results across all datasets. For both models, we observe that a combination of test-80 with train-80 achieves the best performance on three of the datasets showing a steady rise in performance as the P_{fp} in the test dataset is increasing. Similarly, they both display a sharp increase in performance as P_{fp} increases from 20% to 30% and maintains a stable and no significant change in performance afterwards with the train-20

model being the only exception which exhibits a flat or stable performance for the HELM dataset. Also for the OCMS dataset, models trained with train-50 and above exhibit some significant increase in performance in the initial stages as P_{fp} increases from 20% to 50% and stalls afterward for the LARS model whilst it the converse for the LM model which maintains the same performance until P_{fp} reaches 50% where performance displays a significant increase afterwards. With train-20 and train-80 performing well on the three models considered so far, the train-20 model however performs poorly whilst the train-80 still performs better for the NNET model. It is also interesting to note that train-20, train-30 and train-80 fails to react at test-40 when all the models seem to fall or decrease dramatically for the HELM dataset. Train-20 and train-30 exhibits resistance to change but finally increases at test-80. On all datasets, models trained on P_{fp} values 50% and above performs better than those below 50%. The M5 model overall seems to be the most affected model by class distributions in the test dataset. There could not be established a stable trend in performance as the $\text{Norm}(P_{opt})$ curves oscillate when the P_{fp} values in the test data kept on increasing.

In summary, we could conclude the following: (1) initial increase in the P_{fp} of the testing dataset results in most of the models having a significant increase/decrease in performance. (2) equal P_{fp} value in both training and testing dataset does not achieve the maximum $\text{Norm}(P_{opt})$ in most cases (3) models trained on a moderately rebalanced training datasets such as train-20 and train-30 are the most resistant to performance fluctuations notwithstanding the proportions of two class distributions in the testing datasets.

5.6 Threats to Validity

As an empirical study, there are several potential limitations. We discuss below the internal and external threats to validity of the study.

This study obtains the number of faults in source code files through the analysis of files and commits in software repositories. This process is commonly used in fault prediction research [28, 58, 95], but has the limitation that faults not recorded in CVS log comments cannot be collected. Further research is required to improve the accuracy of faults collection from repositories. Considering only process metrics, we cannot generalize our results to all areas of fault prediction. However, process metrics has been shown in literature as a better option for software fault prediction [28, 58, 87, 95]. Consideration of other metrics such as the static code metrics or product metrics is left for future studies. Our study

was also limited to ten purely OSS datasets and ten effort-aware models which could also be a source of bias to the results. However, the depth and size of our datasets were large enough for a comprehensive experiment. We acknowledge the availability of diverse effort-aware predictive models which could have been considered but we argue that the models selected covers different domains of techniques ranging from statistical, data mining and machine learning techniques hence these models are representative of all the several models that exist in literature.

5.7 Chapter Summary

We have assessed and examined the role of balanced and imbalanced training and testing datasets on the performance of software defect prediction models considering ten datasets and ten effort-aware models evaluated using an effort-aware measure, $\text{Norm}(P_{opt})$. Of interest to us were how the percentage of target variable distribution in a fault dataset affects the performance of a predictive model with the background knowledge that rate of faulty modules in real world fault dataset is unstable and highly imbalanced. Assessing the performance of four sampling techniques on ten effort-aware predictive models, our results indicate that moderate application of sampling technique was required to achieve better results for most effort-aware models considered since the best percentage or rate of fault-prone modules in a dataset (P_{fp}) was found to be between the ranges of 20-30%. In comparison of the best sampling technique, the over-sampling approaches outperformed the under-sampling approach with Random Over-Sampling edging out the others.

Our results indicate no sampling is better than applying under-sampling technique to a faulty dataset when the minority class of the original dataset is less than 20% of the total dataset. Under-sampling could be considered when it is combined with other over-sampling techniques. Most studies have reported the positive effects of applying sampling techniques to fault prediction models but our study contrasted that view when we evaluated using the $\text{Norm}(P_{opt})$. Random over-sampling (ROS) proved to be the only positive approach. Lastly, the study showed that performance of trained models is highly dependent on the class distribution in a testing dataset and thus extra caution should be considered when evaluating and comparing effort-aware models across different testing datasets with varied proportions of buggy and clean instances. We have demonstrated in a practical setting, the effects of sampling approach on fault

prediction models and intend to apply more advanced sampling techniques on very large datasets for future studies.

5.8 Concluding Remarks

The objective of this chapter is to investigate the effect of commonly-used data resampling approaches on effort-aware fault prediction models. The performances of fault prediction models are often biased due to the skewness or class imbalance of the datasets considered. To improve the prediction performance of these models, sampling techniques have been employed to rebalance the distribution of fault-prone and non-fault-prone modules. The effect of these techniques have been evaluated in terms of accuracy/geometric mean/F1-measure in previous studies; however, these measures do not consider the effort needed to fix faults. To empirically investigate the effect of sampling techniques on the performance of software fault prediction models in a more realistic setting, this study employs $\text{Norm}(P_{opt})$, an effort-aware measure that considers the testing effort. We performed two sets of experiments aimed at (1) assessing the effects of sampling techniques on effort-aware models and finding the appropriate class distribution for training datasets (2) investigating the role of balanced training and testing datasets on performance of predictive models. Of the four sampling techniques applied, the over-sampling techniques outperformed the under-sampling techniques with Random Over-sampling performing best with respect to the $\text{Norm}(P_{opt})$ evaluation measure. Also, performance of all the prediction models improved when sampling techniques were applied between the rates of (20-30)% on the training datasets implying that a strictly balanced dataset (50% faulty modules and 50% clean modules) does not result in the best performance for effort-aware models. Our results also indicate that performances of effort-aware models are significantly dependent on the proportions of the two types of the classes in the testing dataset. Models trained on moderately balanced datasets are more likely to withstand fluctuations in performance as the class distribution in the testing data varies.

Very few data sampling approaches were considered in this chapter and no statistical tests were applied for more practical results and implications. We investigate in the next chapter the statistical differences and practical effects (with regards to prediction performances) when data sampling methods are applied to training data (resampled data) in building prediction models compared to the application of no sampling method (default data).

Chapter 6

Assessing the Significant Impact of Data Sampling Approaches on Software Defect Prioritization and Classification

6.1 Introduction

Software defect prediction models predict buggy modules in a software system and aids in the efficient prioritization and allocation of scarce testing resources on the predicted buggy modules [40, 75, 88]. The performances of defect prediction models are known to be dependent on data used for training them [4, 28, 75]. The algorithms of most conventional prediction models are made for balanced datasets so as to maximize the overall classification accuracy [139, 145].

However, defect datasets are mostly imbalanced with the non-defective modules dominating the defective modules [34, 40, 135]. Models trained on imbalanced datasets will inadvertently predict most minority (defective) class instances of a new project as majority (non-defective) instances [110]. A high misclassification error is usually associated with the classifier on the rare occasions that the classifier predicts the minority class samples compared to predicting the majority class instances. Software quality teams are however, interested in a model with high prediction accuracy on the minority class [82, 92].

A popular approach for dealing with the imbalanced learning is the application of data sampling methods [5, 22, 37] to the dataset prior to training. The two primary methods, under-sampling and over-sampling alter the distribution of the data by reducing the majority class and increasing the minority class respectively. The first to conduct a study using sampling methods in Software Defect Prediction (SDP) were Kamei et al. [59] evaluated on an

industrial dataset. They suggested that sampling methods can be effectively used for defect prediction. Subsequent studies [106, 113, 135] examined the effects of sampling methods on prediction models using open source projects from PROMISE repository. These previous studies all conclude that sampling methods does improve prediction performance but do not report whether the improvements are really practical or important for software defect prediction studies.

Most of these studies failed to perform statistical tests and compute effect sizes to support their results and conclusions (e.g. [59, 106, 113]) given that the results obtained could be heavily affected by the unstable variances and skewed class distribution of the datasets used as reported by Kitchenham et al. [64]. With the abundance of several sampling methods proposed in literature, knowing the actual degree and significant effect a sampling method could have on the performance of prediction models has real world benefits for Software Quality Assurance (SQA) teams. SQA teams will adopt the right sampling method for the efficient allocation of scarce resources for quality assurance activities such as code review efforts for testing all classified defective modules or a subset (top-n) defective modules. Substantial evidence is thus required to validate the ability of sampling methods in improving performance of defect prediction models notwithstanding the prediction model or evaluation measure. Robust statistical significance tests together with an effect size metric that quantifies the strength or weakness of the relationship between investigated variables (sampling methods and prediction performance) is thus very necessary.

The main objective of this chapter is to investigate whether there are statistical differences and practical effects (with regards to prediction performances) when data sampling methods are applied to training data (resampled data) in building prediction models compared to the application of no sampling method (default data). We consider the practical effects with respect to the classification and prioritization of defective modules. As such, we structure our experiments to address the following two research questions:

1. **RQ1: Do sampling methods improve defect classification?** Yes, we observe a clear distinction between prediction performance (pd , pf and G-mean) for models trained on resampled data and models trained on default data based on the large and significant effect sizes obtained. Our results show that existing sampling methods could help in classifying all defective modules.
2. **RQ2: Do sampling methods improve defect prioritization?** No,

Brunner’s statistical test [17] results obtained on the AUC performance values show that, prediction models trained on resampled datasets could not outperform models trained on the default datasets. The effect sizes computed from Cliff’s method with Hochberg’s method [17] were insignificant indicating that existing sampling methods cannot improve the prediction of defect-proneness itself.

The experimental results show that all sampling methods significantly improved the G-mean and pd performance values whilst they increased pf for all prediction models with large effect sizes. Sampling methods had no statistical and practical effect on the AUC values. None of the sampling methods studied could concurrently increase pd and decrease pf . We encourage the use of sampling methods only for defective modules classification and not prioritization. To help improve defect prioritization, sampling methods which could concurrently achieve low pf and high pd is needed. Overall, the simple RUS method outperformed the complex SMOTE and its variants. We thus recommend the use of RUS when large data samples of a particular project exist.

6.1.1 Chapter Organization

The organization of this chapter is as follows. Section 6.2 reviews some previous studies on the application of sampling methods for defect prediction. Section 6.3 presents a brief background description of the sampling methods and the effect size measure used for the experiments and in Section 6.4, we describe the experimental methodology, data processing steps and performance evaluation measures. The empirical results and analysis is presented in Section 6.5. Further analysis and discussions together with the threats to validity are presented in Section 6.6 and conclusion and discussion of future works are presented in section 6.7.

6.2 Related Work

The problem of class imbalance makes it challenging to conduct experiments with datasets in the field of software defect prediction. Few studies have considered the effects of class imbalance and how to alleviate this issue in software defect prediction.

Kamei et al. [59] were the first to suggest the use of sampling methods as a preprocessing technique for SDP. They experimentally evaluated the effects of

four sampling methods (ROS, SMOTE, RUS, ONESS) on four prediction models using two sets of industrial legacy software evaluated using the F1-value. Sampling methods improved performance for two models (Linear Discriminant Analysis and Logistic Regression) whilst the opposite was true for Neural Networks and Classification Trees. The best performing sampling method was different among the models and datasets. The limitation of this study is that they used a dataset which is not public and not readily available. Menzies et al. [91] empirically examined several training data for defect prediction models. They studied the effects of over-, under- and micro-sampling on prediction performance and concluded that these methods improve the information content within the training data and thus improve prediction performance. By conducting an empirical study considering five PROMISE projects, two prediction models and two sampling methods, Riquelme et al. [113] demonstrated that sampling methods especially SMOTE does improve the AUC performance when a fully balanced dataset was generated. Similarly, Pelayo and Dick [106] analysed the effects of SMOTE on C4.5 prediction model using four benchmarked datasets. They observed an improvement of at least 23% in the average geometric mean (G-mean).

A more comprehensive study by Wang and Yao [135] investigated the effects of different types of class imbalance learning methods comprising of sampling method (RUS), threshold moving and boosting-based ensemble algorithms together with two base classifiers (Naive Bayes and Random Forests). Using three evaluation measures and 10 PROMISE datasets, all methods improved performance. However, AdaBoost.NC, the ensemble technique had the best overall performance.

Our recent study [9](Chapter 5) investigated the effects of sampling approaches on effort-aware prediction models. We conducted an empirical comparisons among 10 effort-aware models trained on 10 manually extracted open source software projects after resampling with four data sampling approaches and evaluated these models using the effort-aware measure $\text{Norm}(P_{opt})$. Results showed that ROS outperformed the SMOTE approach and no sampling outperformed RUS when the dataset is highly imbalanced that is, has percentage of fault-prone modules (P_{fp}) less than 20%.

Most of these previous studies consider very few sampling methods for their comparative studies (e.g. [106, 113]) and fail to perform statistical tests and effect size computations to aid conclude the efficiency of these methods on the prediction models. Benefits of using statistical test especially robust ones for evaluating software engineering empirical results were reported in the recent

work of Kitchenham et al. [64]. They argue that these robust statistical test are very necessary given the fact that most real world data for software engineering experiments are either skewed, heavy-tailed or have unstable variances and could heavily impact the results produced from these datasets. The authors recommended non-parametric methods such as Brunner’s test for statistical significance and Cliff’s delta for effect size computation.

This study builds on and extends previous studies [59, 106, 113] by investigating additional oversampling approaches and using more prediction models. However, this study differentiates itself from [106, 113] in that, we use different datasets and conduct cross-release prediction. We further statistically evaluate the performance of the models trained on these resampled datasets to examine if the improvements were significant or not using robust statistical test recommended by Kitchenham et al. [64].

6.3 Background

In this section, we present a brief description of the sampling methods and the effect size metric adopted and used in our experiments. These methods are selected because they are the most commonly used in several defect prediction studies.

6.3.1 Data Sampling Approaches

Data sampling approaches are mostly considered as preprocessing procedures which are applied to imbalanced datasets by either increasing or decreasing data instances (samples) so as to alleviate the class imbalance before a prediction model is trained on the datasets. They comprise of over-sampling whereby minority data instances are increased and under sampling whereby majority data instances are decreased.

Random Over-sampling (ROS)

The basic and simplest form of over-sampling is ROS. This method adds more minority instances to the datasets by randomly duplicating existing minority instances. It provides no new information to the trained classifier and is known to cause over-fitting. It is mostly preferred because it is easy to implement and requires no complex algorithm.

Random Under-sampling (RUS)

RUS decreases the number of majority instances by randomly deleting already existing majority instances. As a simple and easy approach, the approach is repeated until the desired sampling rate is achieved. Due to the randomness of deleting, majority instances which provide vital information to the classifier in clearly identifying the decision boundary could be deleted.

Synthetic Minority Over-sampling TEchnique (SMOTE)

SMOTE has been extensively applied in several defect prediction studies [113, 117]. Proposed by Chawla et al. [22], this technique over-samples the minority class in a dataset by generating new “synthetic” samples. This technique adopts the k-nearest neighbor approach to over-sample the minority class instances. To generate these “synthetic” samples, each minority class sample is considered and the new samples are introduced along the line segments that joins any of the k minority class nearest neighbors.

Adaptive Synthetic Sampling (ADASYN)

ADASYN was proposed by He et al. [48] as an alternative over-sampling approach that focuses on the minority classes that are difficult to classify. Weights are assigned to the minority classes and dynamically adjusted so as to reduce the bias in the imbalanced dataset by considering the characteristics of the data distribution. The ADASYN algorithm incorporates a density distribution in automatically deciding the number of synthetic samples needed for each minority class instance resulting in unequal data samples being generated for each considered minority data instance.

Borderline-SMOTE

As a modified version of the SMOTE technique, this method proposed by Han et al. [44] mainly focuses on the harder-to-classify minority instances found on the border-line of a classifier. By adopting the K-nearest neighbor technique, minority data instances with more than a specified k-nearest neighbors of majority instances than minority instances are labeled and considered for over-sampling. The SMOTE technique is subsequently applied to the labeled instances. A major advantage of this method is the strengthening of the borderline between the majority and minority class samples.

TABLE 6.1: Summary of selected 20 Imbalanced datasets extracted from PROMISE data repository

Project Name	Short Name	Version A				Version B			
		Release	Module	# Defects	Pfp(%)	Release	Module	# Defects	Pfp(%)
Ant1	ANT1	1.3	125	20	16	1.4	178	40	22
Ant2	ANT2	1.5	293	32	10.9	1.6	351	92	26.2
Camel1	CAM1	1.0	339	13	3.8	1.2	608	216	35.5
Camel2	CAM2	1.4	872	145	17	1.6	965	188	19.5
Forrest	FORR	0.7	29	5	17.2	0.8	32	2	6.2
Ivy	IVY	1.4	241	16	7	2.0	352	40	11.4
Jedit	JEDIT	4.2	367	48	13.1	4.3	492	11	2.2
Synapse	SYNA	1.0	157	16	10.2	1.1	222	60	27
Xalan	XALA	2.4	723	110	15.2	2.5	803	387	48.2
Xerces	XERC	1.2	440	71	16.1	1.3	453	69	15.2

6.3.2 Effect Size

Any indicator that measures the magnitude of a treatment effect is referred to as an effect size. Usually, the p-values obtained from statistical significance tests are affected by the sample size [130] hence, the need for effect sizes which provide a more practical significance and objective measure of the experimental effect, regardless of the statistical significance of the test statistics used [64, 83]. Effect sizes provides an appropriate evaluation measure for determining the magnitude and importance of the statistical results obtained [122]. The benefits of effect sizes has been extensively reviewed and reported by Tomczak and Tomczak [130]. Cliff's delta (δ) [25, 141] has been reported to outperform other effect size metrics such as Cohen's d and Hedges g statistics [70]. Cliff's method with Hochberg's method proposed by Brunner et al. [17] for controlling multiple tests was adopted for computing the effect sizes in our study. This method was recently recommended by Kitchenham et al. [63] as a more robust and appropriate choice for non-normal data since it is reliable when ties do exist within the data. The method, related to Cliff's delta uses the midranks ($\bar{R}$) of the treatments to calculate the test metric (p-hat($\hat{P}$)) which represents the effect size and can be used to compute the delta value ($\hat{\delta}$). A p-value is also produced which helps indicate if the effect size is statistically significant or not. The mathematical definitions of the effect size metric adopted for this study are presented as follows:

$$\bar{R}_j = \frac{i}{n_j} \sum_{i=1}^{n_j} R_{ij} \quad (6.1)$$

$$\hat{P} = \frac{1}{N}(\bar{R}_1 - \bar{R}_2) + \frac{1}{2} \quad (6.2)$$

$$\hat{\delta} = 1 - 2\hat{P} \quad (6.3)$$

6.4 Case Study Design

This section describes the research questions, datasets, prediction models, performance measures and comparison criteria used for the study. In emulation of real-life setting, we used two sets of project datasets for training and testing. Detailed but simple experiments were conducted at different resampling rates (Pfp) to aid in the examination and analyses of the effects of sampling methods on the chosen prediction models.

6.4.1 Research Questions

Our empirical experiments are designed to address the following two research questions:

RQ1: Do sampling methods improve defect classification?

RQ2: Do sampling methods improve defect prioritization?

As we pointed out earlier in Section 6.2, previous studies report that sampling methods do improve performance of defect prediction models but the practical effects and implications of these sampling methods are mostly not reported. Jiang et al. [51] reports that the type of evaluation measure used for performance assessment of prediction models could have a practical value for software engineers. They report that measures such as the G-mean, F-measure, pd , pf and precision offer comparison of classification performance but not sufficient for model selection. RQ1 adopts some of these measures to examine the practical effects of sampling methods on classification performance on prediction models. On the other hand, for more safety critical systems with very limited resources available for verification and validation activities, graphical model evaluation approaches such as the AUC will aid in fault resolution and prioritization activities. RQ2 examines the effect of sampling methods on defect prediction models for software verification and prioritization activities.

6.4.2 Datasets

Twenty (20) versions of ten (10) open source software project datasets were extracted from the PROMISE data repository. Table 6.1 shows a summary of description of these datasets. These projects donated by Jurezcko and Madeyski [56], and Jurezcko and Spinellis [57] were written in JAVA where a module corresponds to a class. Twenty (20) static code metrics measured using BugInfo and ckjm¹ tools were extracted from each module. A non-defective module is labeled

¹<http://www.spinellis.gr/sw/ckjm>

TABLE 6.2: Description of Static Code Metrics [49]

Abbreviation	Description
WMC	Weighted methods per class
DIT	Depth of Inheritance Tree
NOC	Number of Children
CBO	Coupling between object classes
RFC	Response for a Class
LCOM	Lack of cohesion in methods
LCOM3	Lack of cohesion in methods, different from LCOM
NPM	Number of Public Methods
DAM	Data Access Metric
MOA	Measure of Aggregation
MFA	Measure of Functional Abstraction
CAM	Cohesion Among Methods of Class
IC	Inheritance Coupling
CBM	Coupling Between Methods
AMC	Average Method Complexity
Ca	Afferent couplings
Ce	Efferent couplings
CC	McCabe's cyclomatic complexity
Max(CC)	Maximum value of CC methods of the investigated class
Avg(CC)	Arithmetic mean of the CC value in the investigated class
LOC	Lines of Code
Bug	Number of detected bugs in the class

as zero and defective modules are labeled with the number of bugs present in the module. Table 6.2 presents the static code metrics. Previous or old release datasets (version A) were used as the train datasets whilst the latest or newer releases (version B) were used as the testing datasets. The datasets selected consist of very low and high imbalanced ratio or percentage of fault-prone modules (Pfp) with the training datasets having a Pfp ranging between 3.8-17.2%. These datasets have also been widely used in several empirical studies in the defect prediction domain [10, 49, 56, 57]

6.4.3 Evaluation Criteria

Evaluation of the predictive models are done considering the confusion matrix (Table 6.3). From the table, TP is the number of true positives, FP is the number of false positives, FN is the number of false negatives and TN is the number of true negatives. We use Recall (pd), AUC, Geometric-mean and pf .

TABLE 6.3: Confusion Matrix for two-class classification problem

	Predicted Positive	Predicted Negative
Actual Positive	TP	FN
Actual Negative	FP	TN

$$Recall(pd) = \frac{TP}{TP + FN} \quad (6.4)$$

$$pf = \frac{FP}{FP + TN} \quad (6.5)$$

$$g - mean = \sqrt{\frac{TP}{TP + FN} * \frac{TN}{TN + FP}} \quad (6.6)$$

These measures are known to be stable for evaluating performance of prediction models on imbalanced datasets and ranges between the values of 0-100%. Recall (pd) measures how much of the defective modules were actually detected. A higher pd denotes better performance. Probability of false alarm (pf) measures the rate of predicted modules that were non-defective or wrongly predicted as defective. A low or zero value for pf implies a better prediction model. For a highly imbalanced dataset, the Geometric-mean (G-mean) proposed by Kubat and Matwin [71] is preferred as it considers both the pf and pd values and does not value one measure over the other. Measures such as precision and F-measure were excluded since precision has been refuted by Menzies et al. [92] because they are unstable for highly imbalanced datasets. We also use the AUC measure because it does not sacrifice one class over the other and optimally handles the tradeoffs between the true and false positive error rates. The mathematical definitions of pd , pf and G-mean measures are presented in equations 4-6.

6.4.4 Experimental Setup

For the experiments, implementation of the ROS, RUS, SMOTE resampled datasets, all prediction models construction and evaluations were executed using the open source statistical processing tool, R [129]. The library packages WRS [140] and Caret [72] were used for the statistical tests and prediction model construction respectively. The ADASYN and Borderline-SMOTE generated datasets were implemented following the instructions given by the authors using MATLAB tool. We use all features or metrics for each training dataset as suggested by Menzies et al. [88]. For this study, five prediction models (Table 8.2) widely used in defect prediction studies [75, 81] were chosen. The parameter configurations for each of the prediction models are also presented in Table

8.2. We considered the Random Forests [75], C4.5 decision tree [111], K-NN algorithm [26], NNET [60] and SVM [144]. It should be noted that we predicted the presence of bugs in a module and not the number of bugs in a module. The experiments conducted are described below.

We examined the performance of the defect prediction models when the 5 different sampling methods as described in Section 6.3.1 are applied to each project data at different Pfp rates (default, 20, 30, 40, 50) to create 5 independent training data. The default value indicates that no sampling method is applied and will be represented as NOS in our experiments. We stop resampling at 50% because our pilot experiments shows that application of sampling methods above 50% could cause serious over-fitting and RUS could totally deplete the training data. Considering 10 test and 10 training datasets, we conducted 10 cross-project experiments per each Pfp making 50 experiments for each prediction model in all for each sampling method. After obtaining the training data, a 10-fold cross validation method is applied during the model construction. This method ensures that for a particular model training, the dataset is split into 10 folds where 90% of the dataset is used for training and the remaining 10% is used for testing. It is iterated until all folds in the dataset have been used once as a test dataset. After model training, the model is tested on the separate test data sets (version B) and evaluated using the performance measures discussed in Section 8.3.3. Algorithm 1 presents the procedure followed for the experiments. From the algorithm, steps 1 to 3 initializes the parameters used for the experiment. These include the input variables such as the source and target data, sampling methods, prediction models and Pfp rates. The process continues from step 4 through to step 17 where the classifiers are constructed on the resampled or modified training data (P_{pfp}) and then tested on the test data. For both SMOTE and ADASYN, we set $k=5$ as used by the original authors of these approaches [22, 48].

6.4.5 Performance Comparison

We compare the results of the performance measures of each prediction model on the original unmodified dataset (simple predictors) and the resampled data (advanced predictors). A superior sampling method should produce better results on the prediction model, notwithstanding the performance measure used. Sampling methods generating high values for pd , G-mean, AUC and low value for pf is thus preferred. We adopted the *win-tie-loss* statistics, one of the most prevalent statistical evaluation method used in several Software effort estimation studies [66, 68]. This procedure performs the statistical test by name Brunner's

TABLE 6.4: Predictive Models and their hyperparameter configurations

Model	Overview	Hyperparameters
C4.5	J48 Decision Tree	$c = \{0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7\}$
NNET	3-layer Neural Network	$size = \{4, \dots, 28\}, decay = \{0.1, 0.2\}$
SVM	Support Vector Machine	$c = \{2^{-6}, \dots, 2^{10}\}$
RF	Random Forest	$mtry = \{10, 50, 100, 200, 250, 500, 1000\}$
KNN	K-Nearest Neighbor	$c = \{2^*(0, \dots, 7) + 1\}$

test [17] to find the significant differences in the performances of the models. This test is known to be more robust in comparison to other non-parametric tests [64]. Three counters by name *wins*, *ties* and *losses* are initially set for each predictor. For two pairs of selected predictors (combination of prediction model and sampling method), the statistical test is performed across all datasets for each performance measure and the *win*, *tie* or *loss* value is reported based on the significance value ($p < 0.05$). If the performance distributions of 2 predictors are not significantly different, the *ties* counters of both predictors are increased by 1; otherwise the *wins* counter of the significant predictor is increased by 1 and the *losses* counter of the other predictor also increased by 1. Lastly, we compute the effect size if a predictor is significant and outperformed other predictors using Cliff's method with Hochberg's method proposed by Brunner et al. [17] (Section 6.3.2). The rationale behind Cliff's method is "What is the probability that a random observation from the set of simple predictors is greater than an observation from the set of advanced predictors?". The p-value obtained from this method helps indicate if the effect size computed is statistically significant or not. We interpret the effect sizes using the proposed magnitude labels of Kraemer and Kupfer [69] - small ($\hat{\delta} \leq 0.112$), medium ($0.112 < \hat{\delta} < 0.428$) and large ($\hat{\delta} \geq 0.428$).

6.5 Case Study Results

We present the prediction results of sampling methods applied to the dataset at different *Pfp* rates. Due to space limitation, the results are visualized using quartile charts. The charts are generated for each performance measure, *Pfp* rate and prediction model by sorting the performance values for all data sets to isolate the median, lower and upper quartiles. Figure 6.1 shows the quartile

Algorithm 2 Outline of Experiment**Procedure Begin**

1. For source projects $P_{source}(\text{Old version}) = \{P_1, \dots, P_n\}$ and target projects $Q_{test}(\text{New version}) = \{Q_1, \dots, Q_j\}$, **repeat**
2. Initialize $pf, pd, auc, g\text{-mean} = \emptyset$; $sampling = [\text{SMOTE}, \text{ADASYN}, \text{BORDERLINE-SMOTE}, \text{ROS}, \text{RUS}]$;
3. Initialize $pfp = [\text{default}, 20, 30, 40, 50]$; $learner = [\text{C4.5}, \text{SVM}, \text{KNN}, \text{NNET}, \text{RF}]$;
4. **for** $j = 1, \dots, \text{length}(P_{source})$ **do**
5. Select each train data P_j and its respective test data Q_j
6. **for** $k = 1, \dots, \text{length}(sampling)$ **do**
7. **for** $i = 1, \dots, \text{length}(pfp)$ **do**
8. Generate modified dataset ($P_{pfp[i]}$) from P_{source} where $P_{pfp[i]} = \text{resample}(P_j, sampling[k], pfp[i])$;
9. **for** $l = 1, \dots, \text{length}(learner)$ **do**
10. Train classifier $learner_l$ on modified data $P_{pfp[i]}$ using 10-fold cross-validation
11. Classify data points in Q_j using the classifier $learner_l$
12. Compute: $pf, pd, AUC, G\text{-mean}$ for each classifier $learner_l$
13. **end for**
14. **end for**
15. **end for**
16. **end for**
17. **until** P_{source} is empty (i.e., the training data is exhausted)

End

charts of performance values for each sampling method and prediction model across all datasets. From the performance results in Figure 6.1, the median is represented by a solid dot and the 25th and 75th percentiles are represented by vertical lines. In this figure, we compare the performance of the prediction models on the default dataset (NOS) with the resampled datasets. To statistically evaluate the effects of applying sampling methods on the prediction performance, we extract the total count of losses from the win-tie-loss statistic

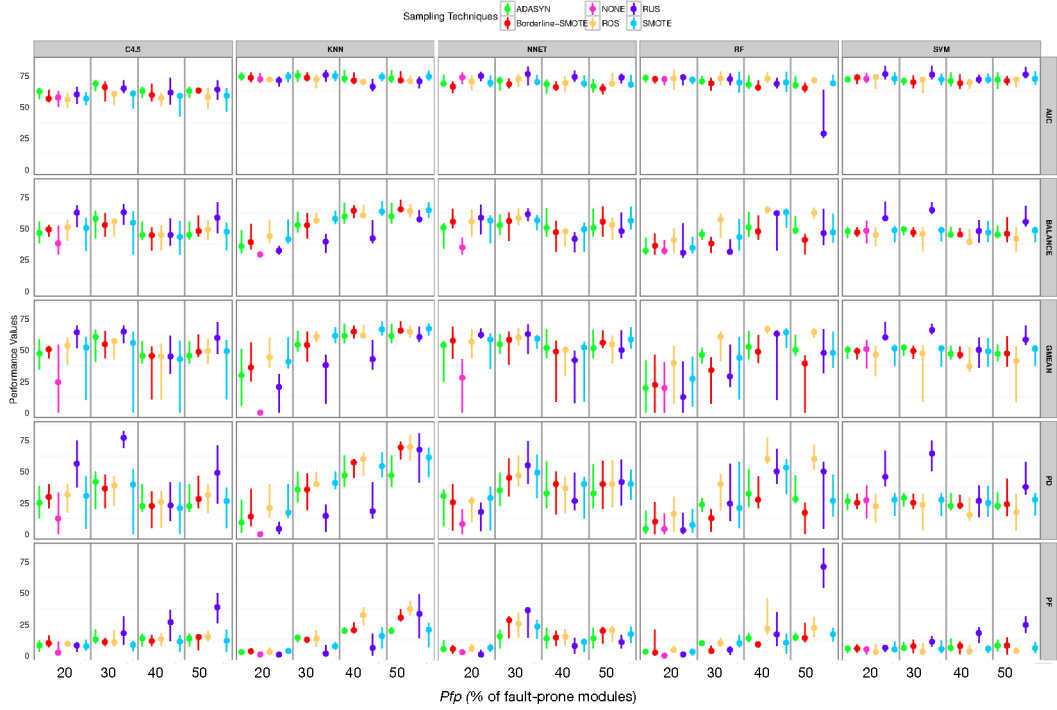


FIGURE 6.1: Quartile charts of performance values for all sampling techniques on different Predictive Models at different (Pfp) values across all ten Static metrics datasets.

procedure explained in Section 6.4.5. Using this statistical procedure, we compared the prediction performance value between prediction models trained on NOS datasets (simple predictor) and prediction models trained on each sampling method applied to all datasets (advanced predictors) across 4 (20,30,40,50 %) Pfp rates. The total losses generated across all datasets for each prediction model and performance measure is presented in Table 6.5. For a record in the table, the minimum loss is 0, which either indicates a predictor was never outperformed by its respective predictor being compared, and the maximum losses is 4 indicating the predictor lost across all 4 Pfp rates. Records with the lowest or zero losses indicate better performance. For more practical results, we present the effect sizes computed for each pairwise comparison between the simple predictor and each other advanced predictors in Figure 6.2. Similar to the *win-tie-loss* statistic procedure, we compute the effect size per each performance measure across all 4 Pfp values. If the effect size was significant based on the p-value, we report the magnitude of significance based on magnitude labels of Kraemer and Kupfer [69]. Based on the results presented in the Figures and table, we answer the following two research questions.

RQ1: Do sampling methods improve defect classification?

In this section, we measure classification performance using G-mean, pd

TABLE 6.5: Performance in terms of losses aggregated across four Pfp values on the 10 datasets. Lower losses values indicates higher performance. Records highlighted in light gray indicate that application of sampling methods results in better performance whilst those in dark gray indicate otherwise.

PERF	MODEL	TOTAL LOSSES									
		NOS	ADASYN	NOS	BSMOTE	NOS	ROS	NOS	RUS	NOS	SMOTE
AUC	C4.5	0	0	0	0	0	0	2	0	0	0
	KNN	0	0	0	0	0	1	0	1	1	0
	NNET	0	1	0	3	0	0	0	0	0	0
	SVM	0	0	0	0	0	0	0	1	0	0
	RF	0	0	0	0	0	0	0	0	0	0
GMEAN	C4.5	0	0	0	0	0	0	3	0	0	0
	KNN	4	0	4	0	4	0	4	0	4	0
	NNET	4	0	4	0	4	0	3	0	4	0
	SVM	3	0	1	0	3	0	2	0	2	0
	RF	0	0	0	0	0	1	3	0	0	0
PD	C4.5	1	0	1	0	0	0	3	0	0	0
	KNN	4	0	4	0	4	0	4	0	4	0
	NNET	4	0	4	0	4	0	3	0	4	0
	SVM	3	0	2	0	4	0	3	0	3	0
	RF	0	0	0	0	0	1	3	0	0	0
PF	C4.5	0	3	0	1	0	4	0	3	0	2
	KNN	0	4	0	4	0	4	0	1	0	4
	NNET	0	2	0	3	0	4	0	3	0	3
	SVM	0	4	0	3	0	4	0	3	0	4
	RF	0	0	0	0	0	0	0	3	0	1
Total Losses		23	14	20	14	23	19	33	15	22	14

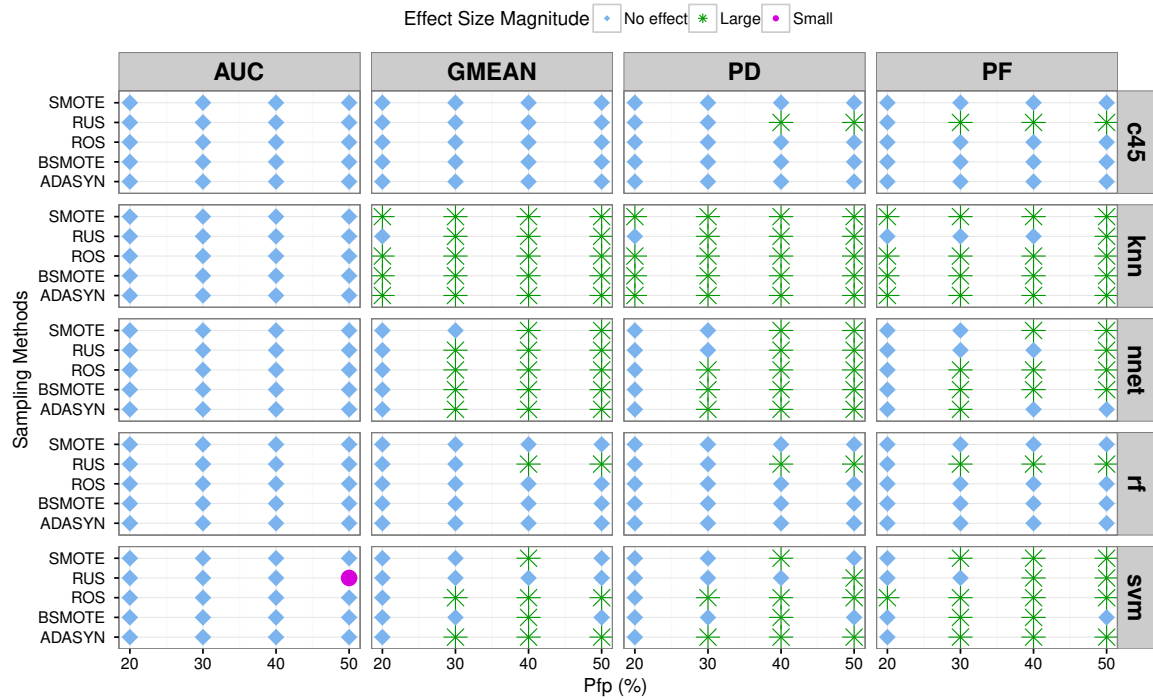


FIGURE 6.2: Performance plots for the effect sizes computed for each sampling methods compared to the no sampling method on different prediction models across all ten cross-project datasets per each Pfp values.

and pf evaluation measures. From Figure 6.1, all sampling methods have an impact on the performance values of the models. It is evident that sampling methods performed better than no sampling across all prediction models except SVM with regards to the G-mean and pd performance values. With no sampling (NOS) as the baseline, all sampling methods had positive impacts on the prediction models except the ROS where it had a negative impact on the SVM model regarding the G-mean and pd values. However, the pf values were consistently high with the application of sampling methods. For statistical significance, we observe from Table 6.5 that for all performance measures except pf , application of different sampling methods did improve performance. On rare occasions did NOS outperform the other sampling methods. The losses are mostly reduced to a 0 or 1 after applying sampling methods. The large differences in the total losses after applying sampling methods across all performance measures indicates the positive influence of sampling methods. Several large effect sizes were observed for the G-mean and pd evaluation measures across most prediction models (Figure 6.2). The sampling methods had large and positive impacts on the KNN, NNET and SVM models as the Pfp rate increased. We can conclude based on the statistical and practical significance that applying sampling methods on prediction models does improve the classification performance of prediction models.

Summary: Application of sampling methods improved the G-mean and pd performance measures of prediction models and were statistically significant with large effect sizes in comparison to prediction models trained on datasets with no sampling method applied. Existing sampling methods can properly set the threshold between buggy and clean groups

RQ2: Do sampling methods improve defect prioritization?

From Figure 6.1, for most of the prediction models, the AUC performance values of NOS prediction models seem to be no different from prediction models trained on datasets with sampling methods applied. The pf values were consistently low for the NOS models. The statistical tests results presented in Table 6.5 show very few losses even after applying sampling methods to the prediction models for the AUC measure. NOS significantly outperformed all sampling methods across all models regarding the pf performance measure. RUS and ROS produced the worst and significant increase in pf .

The effect size results (Figure 6.2) for the pf measure indicates that application of sampling methods on prediction models does increase the false alarms

(pf) significantly since the effect sizes computed were large. From the figure, it was observed that all results obtained excluding one (RUS on SVM at Pfp of 50) were not practically significant when evaluated with the AUC measure. Application of sampling methods cannot improve the prediction of defect-proneness due to the high pf values making it difficult for engineers to rank and choose the top- n ranked modules for testing purposes.

Summary: Application of sampling methods significantly increased the pf performance values of prediction models with large effect sizes and had no significant effect on the AUC performance value in comparison to prediction models trained on datasets with no sampling method applied. Sampling methods do not significantly improve the detection of defective modules.

6.6 Discussion

Practical Implications of Results: Our empirical results show the positive effects sampling methods have on prediction performance. With models trained on resampled datasets statistically outperforming the models trained on default dataset with regards to the G-mean and pd , it implies that most or all defective modules are correctly classified. However, with the high pf values, these modules will be contaminated with non-defective modules. Sampling methods are useful for Engineers/SQA teams that wish to classify faulty/clean modules so as to test all faulty modules.

With insignificant AUC results and high pf values produced by prediction models trained on resampled data, existing sampling methods are not useful to prioritize modules for quality assurance activities (such as testing and code review). Engineers/SQA teams that wish to pick top $n\%$ faulty modules for testing will struggle to select those faulty modules due to the contamination of non-defective modules in the obtained results.

Comparisons with past findings: We revisited some previous studies and compared our findings to theirs. Riquelme et al. [113] reported that SMOTE does improve the AUC performance of prediction models. Our findings from Figure 2 does show some performance changes with respect to the AUC values but after conducting statistical tests and effect size computations, our results however reveals that although some sampling methods had significant AUC values, the effect sizes computed were impractical or insignificant across all

models. We can thus conclude that evaluation of prediction models using the AUC measure is not affected by the sampling methods.

Compared with previous work, our results agree that sampling methods does improve prediction performance values of G-mean and pd values. Simple methods such as the ROS and RUS had very positive influence on all prediction models and could be used for fast computations. The results obtained with statistical significance from this experiments clearly shows the positive impacts sampling methods do have on prediction models and does reinforce that of previous studies [59, 106]: the use of resampling approaches does improve the classification performance compared to not using resampling approaches. With large effect sizes obtained for the G-mean and pd values, we recommend future software prediction studies to consider using sampling methods for classification purposes.

Threats to Validity As an empirical study, there are several potential limitations. We discuss below the internal and external threats to validity of the study. Conditions under which the experiments were performed relates to the internal threats to validity. The configuration parameters chosen for each prediction model had an impact on our results. Secondly, the amount of resampling or Pfp is a potential threat. With no agreed amount of resampling to be used for sampling methods in literature, we used low to high Pfp rates so as to eliminate any bias towards some datasets which are not severely imbalanced. Considering only open source software projects with static code metrics poses an external threat. These metrics extracted with automated softwares are easy to collect but the results cannot be generalized to all other projects that have different metrics apart from static code or projects in the commercial domain. Also, the number of projects considered could have an effect on our results. We however considered a wide range and sufficient sizes of projects and therefore produced robust results, which we expect to be similar for other unconsidered projects. We intend to consider commercial projects in future studies. A limited number of prediction models were also considered. However, these are widely used models in several defect prediction studies [106, 113]. Considering different types of prediction models could have different implications.

6.7 Chapter Summary

The magnitude (degree and power) of the effect of sampling methods on the classification and prioritization performances of defect prediction models has not been established. The focus of this chapter is to investigate the statistical

and practical significance of using resampled data for constructing defect prediction models. We examine the practical effects of six data sampling methods on performances of five defect prediction models. The prediction performances of the models trained on default datasets (no sampling method) are compared with that of the models trained on resampled datasets (application of sampling methods). To decide whether the performance changes are significant or not, robust statistical tests are performed and effect sizes computed. Twenty releases of ten open source projects extracted from the PROMISE repository are considered and evaluated using the AUC, pd , pf and G-mean performance measures. Results: There are statistical significant differences and practical effects on the classification performance (pd , pf and G-mean) between models trained on resampled datasets and those trained on the default datasets. However, sampling methods have no statistical and practical effects on defect prioritization performance (AUC) with small or no effect values obtained from the models trained on the resampled datasets. Existing sampling methods can properly set the threshold between buggy and clean samples, while they cannot improve the prediction of defect-proneness itself. Sampling methods are highly recommended for defect classification purposes when all faulty modules are to be considered for testing.

6.7.1 Concluding Remarks

The purpose of this chapter was to investigate the statistical and practical significance of using resampled data for constructing defect prediction models. Employing five different sampling techniques, we analyzed the performance of these sampling methods on prediction models evaluated with four different measures. There is an interesting relation between the choice of evaluation measure and the sampling method used. Evaluation measure that do not rely on the confusion matrix like the AUC was not influenced by the sampling method whilst the converse was true for those computed from a confusion matrix. Sampling methods had a significant and positive influence on prediction models when evaluated with the G-mean and pd performance measures. Effect sizes computed for the pairwise comparison between the simple predictors (models with no sampling method) and advanced predictors (models with sampling methods) were very large implying the importance of sampling methods in improving the classification performance of prediction models. Consequently, these sampling methods also significantly increases the pf which is very undesirable and impractical for defect prediction studies. The use of sampling methods for prioritization purposes is thus not recommended.

The sampling approaches studied could not achieve high pd and low pf simultaneously. An increase in pd leads to an increase in false alarms. The performance of the sampling methods were heavily influenced by the rate of sampling (Pfp). In the next chapter, we examine the drawbacks of these sampling approaches and propose a novel oversampling approach that achieves high pd and low pf simultaneously.

Chapter 7

MAHAKIL: Diversity based Oversampling Approach to Alleviate the Class Imbalance Issue in Software Defect Prediction

7.1 Introduction

Software defect prediction models can conveniently identify software modules or classes that are more likely to be defective [88, 135]. Accurate prediction helps prevent defects by suggesting that developers focus more on these modules or classes to efficiently prioritize and allocate the limited testing resources to them. The accuracy of the prediction models are often hindered by the imbalanced nature of the datasets [9, 40], in which there are naturally fewer defective classes than the non-defective classes [34, 124, 135]. The main problem is that common prediction algorithms assume that the classes in any dataset are equally balanced [139, 145]. Thus, models trained on imbalanced software defect datasets are generally biased towards the non-defective class samples and ignore the defective class samples [110]. Consequently, the class imbalance issue is well-recognized as one of the major causes of the poor performance of software defect prediction models [4, 43]. Accordingly, a considerable body of research [59, 106, 124] has sought to alleviate this problem in the past decade.

The prevalent approach to solving the problem of class imbalance is to use data sampling techniques. To date, most of the data sampling approaches are synthetic based [6, 22, 44, 48]. The most popular among them being the Synthetic Minority Over-sampling Technique (SMOTE) [22], whereby new synthetic or artificial data samples are intelligently introduced into the minority class samples. These synthetic methods tend to introduce some bias towards

the minority class, thus improving the performance of prediction models on the minority class.

Notwithstanding the positive effect that these synthetic methods have on classifiers, Barua et al. [6] argued that synthetic methods such as SMOTE and the Adaptive Synthetic Sampling Approach (ADASYN) [48] erroneously enlarge the minority class region. This leads to many majority class samples being misclassified by a classifier because synthetic samples are accidentally introduced into the region of the majority class. These approaches also generate data samples that are non-diverse and sometimes very similar to existing samples because they consider only nearest neighbor samples.

Similarly, Wong et al. [143] noted that SMOTE causes over-generalization and leads to a similar problem introduced by cross-company defect prediction which is mainly the increase in the false alarm rate (pf) [133] whereby non-defective modules are wrongly classified as defective. Our recent study [11] also confirmed the findings by Wong et al. [143]. A high pf implies that more effort is unnecessarily allocated to inspect those wrongly classified modules. In agreement with Barua et al. [6] and Wong et al. [143], Hang et al. [45] concluded with respect to anomaly detection that SMOTE increases pf despite the significant increase in classification accuracy. The problem of a high pf in highly imbalanced defect datasets was discussed in detail by Menzies et al. in [92]. Menzies et al. suggested high recall (pd) and low pf as the more stable performance measures for prediction models trained on highly imbalanced defect datasets. As such, data re-sampling approaches that result in high pf are undesirable.

In this Chapter, we propose a novel resampling technique that achieves both a high pd and a low pf simultaneously. We believe that this can be achieved by generating as much diverse synthetic data as possible restricted within the region of the minority class. When the oversampling approach generates synthetic samples that are widely distributed but appropriately reside within the decision boundary or region of the minority class, the pf can be conveniently reduced without sacrificing the overall performance.

Our proposed oversampling technique takes inspiration from the field of biology by considering how diverse populations of humans or living organisms differ although they are of the same species. This phenomenon is explained by the Chromosomal Theory of Inheritance [126], which justifies how offspring inherit traits from their parents by obtaining chromosomes from each parent in equal quantity. The proposed technique named MAHAKIL (oversampling based on the theory of inheritance and the Mahalanobis distance), increases the diversity

within the minority class by uniquely creating new synthetic minority instances based on a "typical" case (having a small diversity measure distance value) and an "atypical" case (having a large diversity measure distance value) so that the resultant instance becomes not too typical and not too atypical. The proposed technique offers several benefits over the other complex and conventional techniques. The over-generalization problem is alleviated by this technique because the newly generated samples are evenly distributed within the data and automatically induced to reside within the boundary and region of the minority class samples. Generating data instances from two dissimilar instances ensures that over fitting is avoided because this technique does not generate duplicated samples, thus in effect increasing the diversity within the resampled dataset.

We conducted empirical experiments to systematically show the performance of MAHAKIL in comparison to four common over-sampling approaches: Random Over-Sampling (ROS), SMOTE, Borderline-SMOTE and ADASYN using 20 imbalanced defect datasets from the PROMISE repository¹ and considering five classification algorithms evaluated on the resampled datasets. Based on robust Brunner statistical test and *win-tie-loss* statistics, our results show that MAHAKIL significantly improves the recall(*pd*) performance and reduces the *pf* for all five models over SMOTE, Borderline-SMOTE, ADASYN and ROS.

7.1.1 Chapter Organization

The organization of this paper is as follows. Section 7.2 discusses some concerns over the typical sampling techniques and the motivations behind the MAHAKIL approach. Section 7.3 summarizes the chromosomal theory and its application to synthetic data generation. In Section 7.4, a detailed description of the MAHAKIL approach and its main components are presented. In Section 7.5, we describe the experimental methodology and performance evaluation measures used to validate our approach. The experimental results are presented in Section 7.6 and Section 7.7 discusses the impacts of our findings and the limitations of our study. Finally, a summary of the study is presented in Section 7.8.

7.2 Motivation

Synthetic oversampling methods such as SMOTE [22], Borderline-SMOTE [44] and ADASYN [48] share similar procedures for generating synthetic data. These methods use the k nearest neighbor (k -NN) technique to aid in selecting similar

¹<http://openscience.us/repo/>

neighbors for oversampling. These techniques select the data instances that are the nearest neighbors or very close to the data instance under consideration with respect to their Euclidean distance values. However, selecting only data instances that are very close (neighbors) poses the potential challenge of generating less diverse data points. For example, when $k=1$, two very similar parents are selected, thereby generating data instances that are mere duplicates of the parents. This is almost the same as the ROS approach, which simply duplicates existing data instances and thus provides no new information to the classifier during training. The minority class boundary can also be accidentally widened by the conventional k -NN methods when data instances on the boundary are considered for oversampling. From Figure 7.1 (b), most of the new data instances generated tend to fall outside the previous minority decision boundary and fall in the region of the majority class when data point C is considered.

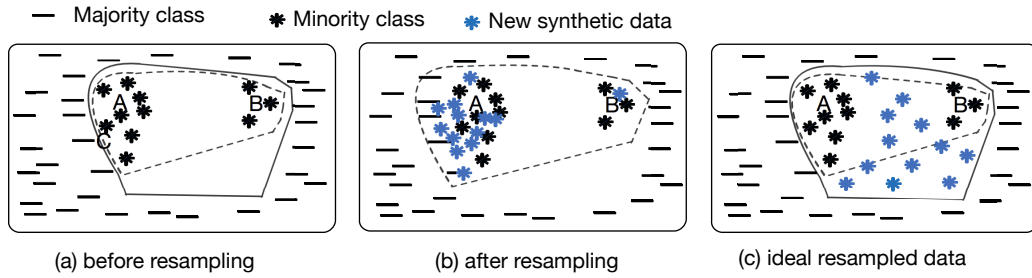


FIGURE 7.1: Effect of sub-clusters of unequal or small sizes on the class imbalance problem; (a) the estimated decision boundary (dashed line) does not approximate the ideal or true decision boundary (solid line); (b) generating synthetic minority data without considering the position and distance of the nearest neighbors from the minority example (e.g. data point A) creates near duplicated or erroneous data and widens the estimated decision boundary; and (c) the ideal diverse data after resampling ensuring that the estimated and true decision boundaries are almost equal

Lastly, if the minority class samples comprise several sub-clusters of unequal size (intra-class imbalanced), generating synthetic data using the conventional k -NN methods such as SMOTE will only increase the size of the sub-clusters because each generated data instance will fall into a specified cluster. Therefore, these samples worsen the intra-class imbalanced problem. Figure 7.1 (a) displays an example of small and fairly dense subsets of samples within the minority class samples. If data points A and B are considered for the oversampling, all new generated data points will fall within their respective clusters if the synthetic methods that use k -NN methods are applied. The estimated decision boundary (dashed line) can be largely induced towards the dominating sub-clusters (i.e., cluster of A) and will not approximate as expected with the ideal or true decision boundary (solid line), as observed from Figure 7.1 (a). Moreover, an “empty space” between the major clusters will not be filled with

data because all new data samples generated fall within a cluster. The ideal resampled data are presented in Figure 7.1 (c), where the newly generated synthetic data are diversely spread and induce the estimated decision boundary to almost equal the true decision boundary.

These examples demonstrate that sampling methods based on the k -NN method are still deficient and are sometimes inappropriate for certain disparate scenarios of defect prediction. They can generate identical or duplicate data points and worsen the intra-class imbalanced issue. As noted by Barua et al. [6], these problems occur because these approaches select all k nearest neighbors with no regard to the position and distance of these neighbors from the data point under consideration. They thus generate erroneous samples and widen the region of the minority class. The determination of k in advance also poses a challenge for these methods.

7.3 Chromosomal Theory of Inheritance and Diversity

According to elementary biology and genetics, genes, the basic unit of heredity are located on the chromosomes and are passed on to children in equal amount with half from both parents during fertilization of the gametic (egg and sperm) chromosomes. This theory known as the Chromosomal Theory of Inheritance was proposed in 1902 by Walter Sutton and Theodor Boveri [126]. The theory implies that new offspring receive 50% of chromosomes from each parent, thereby making the new offspring similar to both parents and at the same time, unique from both parents. To maintain diversity within species, the sex(gender) of the species is of great importance because it helps in selecting two opposite members that can reproduce. The natural process of reproduction involving chromosomes can be defined as follows. Two different sets of chromosomes population $M = (m_1, m_2, m_3, \dots, m_q)$ and $F = (f_1, f_2, f_3, \dots, f_q)$ are created with the same size q . Each chromosome $c = (c_{j1}, c_{j2}, c_{j3}, \dots, c_{jp})$ represents a p -dimensional vector comprised of genes, where c_{jk} is the k th gene value ($k = 1, 2, \dots, p$) of the j th chromosome ($j = 1, 2, \dots, q$) in each of the populations. A new offspring is produced when two chromosomes, one from each set are combined with each parent randomly contributing half of the genes inherited by the offspring.

By elucidating the heredity law in animals and plants, researchers in the

fields of biology and agriculture have extensively applied this theory and produced significant benefits. For example, improved methods for introducing new breeds of various plants and animals have been developed by agricultural specialists [2]. This theory has also provided valuable information about the DNA within a cell, hence facilitating the development of specific kinds of breeds in the domain of agriculture on a more rational basis whereby breeders, for example discard fish that are less productive and focus more on the productive animals [39, 80]. Farmers segregate animals and plants into groups based on their environmental and genetic features to develop (generate) specific traits in the plants and animals. Thus, the chromosomal theory of inheritance forms the basis of our understanding of the patterns of inheritance of many different traits in humans and other organisms.

The underlying principles behind selective animal and plant breeding are applied to an imbalanced defect dataset in order to generate (introduce) new data instances of faulty modules. During selective breeding, methods such as out-crossing, inbreeding and line-breeding [53] have been used to segregate the populations of animals and plants. Similarly, we can rationally oversample the faulty classes by considering a similarity measure to aid in segregating our data samples. Based on the chromosomal theory of inheritance and by adopting the segregation approach, we aim to develop an oversampling technique that can generate new samples by examining the minority defective samples and merging distinct samples to produce synthetic data instances that are maximally different from the parents and contribute to the diversity (variation) within the minority class distribution. The proposed oversampling algorithm based on inheritance theory is different from genetic algorithms because the software modules are considered to be individuals and not solutions to a specific problem. Furthermore, similar to other oversampling techniques, we do not want to lose the few minority data instances in the datasets after generating new data instances, which is in contrast to genetic algorithms whereby parents or older generations of parents die after generating new children data instances.

7.4 MAHAKIL - New Oversampling technique for Software Defect Prediction

7.4.1 Overview

We exploit the challenge of the above mentioned synthetic methods in formulating our novel MAHAKIL oversampling method. We take inspiration from

Walter Sutton's [126] chromosomal theory of inheritance by considering the features (metrics) of the defective modules as the chromosomes. The theory fits our objective of endeavoring to generate diverse data instances. We seek to generate new minority instances that take features from two different data instances but are also uniquely different and not a direct duplicate of the data instances involved. The basic intuition is that two different instances, which are not necessarily close in distance, can produce a new instance that is similar to both instances. MAHAKIL aims to generate a synthetic instance that has both peculiar and common attributes, and it involves three simple and fundamental phases.

In the first phase, the segregation measure values for the minority samples are computed after separating the minority samples from the majority samples. This involves computing of the Mahalanobis distance (D^2) for the minority samples and arranging the samples in a descending order according to their D^2 values. In the second phase, the ordered samples are partitioned into two bins by using the midpoint of the distance matrix (D^2) or central instance to split the samples and sequentially assign a unique label to each member in both bins. In the last phase, the synthetic sample is generated by finding the average of two selected data samples (samples with same unique labels) from each partition. Algorithm 3 presents the complete process, in which phase 1 is described in steps 1 to 6, the second phase is from steps 7 to 9 and concludes with steps 10 to 14 for the final stage. Each component or phase is described in the following sections.

Figure 7.2 shows MAHAKIL in greater detail. The figure shows the complete generation of minority samples based on the number of samples required, which are represented as levels, and focuses on the minority class samples extracted from the whole data. In the first step, based on the Mahalanobis distance, the first and main parents (ancestors) are found and partitioned into two groups (node S1 and node S2) which are used to generate the set of new synthetic samples C01. To avoid new synthetic samples penetrating into the region of the majority class, the first parent nodes are used as the boundary such that all subsequent children generated automatically reside within the range of the parents. Each new child node with the help of its immediate parent, grand parent or ancestor is used to generate two or more extra nodes if more data instances are required. In step 2 (Level 0), the new synthetic node or children are included in the next process of data generation by pairing the children with the first node of parents (S1) and later with the second node (S2) to generate two new nodes of children. Consequently, in the next levels, the location of the children nodes

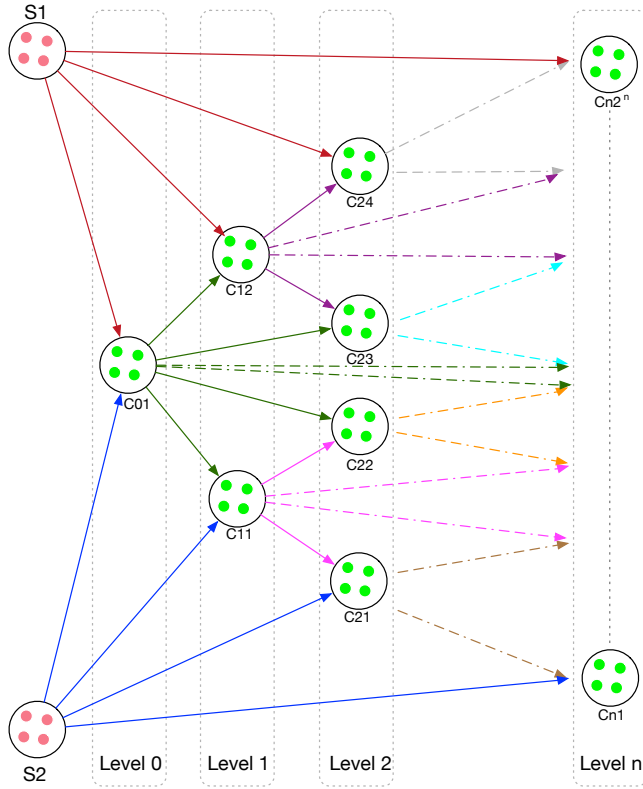


FIGURE 7.2: Illustration of MAHAKIL. The red and green dots represent the minority samples with the latter representing the synthetic data generated by MAHAKIL.

is considered. New samples are generated by considering the immediate parent node of each children node. If the samples generated by pairing the children and immediate parent nodes are less than the required number of samples, the predecessor nodes of the parent nodes are paired with the current/children node. If the generated samples are still less than the required number of samples, the process of pairing the child nodes with older generations is repeated until the samples are sufficient (greater than or equal to the required number of samples). From level 1 onwards, the pairing is done using the sequential labels inherited from the immediate parents of the nodes and thus, the Mahalanobis distances are not computed for any two samples selected for data generation. Only steps 10-14 of Algorithm 1 are invoked for all levels starting from level 0.

7.4.2 Phase 1: Diversity Measurement

To define a diversity measure for a set of minority defective samples, we initially consider the diversity between the smallest sample set possible, which will be a set of two defective instances. To declare two p -dimensional feature vectors in the defective class as being the most diverse, the diversity value between them

Algorithm 3 Overview of MAHAKIL(N, P)**Input:** Dataset of minority and majority class samples N ; Pfp value P **Output:** Balanced dataset at a set Pfp value**Procedure Begin**

1. Split dataset N into arrays of minority class N_{min} and majority class N_{maj}
2. Compute the number of additional minority classes to be generated (T) to attain Pfp P and $k = \text{arraylength}(N_{min})$
3. X_{new} : array for generated sample, initialized to 0
4. X_{newchk} : keeps count of number of synthetic samples generated,
5. Compute Mahalanobis distance (D^2) for N_{min} , $D^2 = (x - \mu)' \Sigma^{-1}(x - \mu)$ where μ =mean and Σ =covariance matrix
6. Save D^2 matrix and it's respective data samples in a decreasing order in $N_{mindist}$
7. Find the midpoint or middle instance i.e. $N_{mid} = k/2$
8. Create two partitions from $N_{mindist}$ making N_{mid} the last record for partition 1 and N_{mid+1} the starting record for partition 2 i.e. $N_{bin1} = \{y_1, y_2, \dots, y_{mid}\}$ and $N_{bin2} = \{y_{mid+1}, y_{mid+2}, \dots, y_k\}$, where $y_i \in N_{mindist}$
9. For each $y_i \in N_{bin1}$ and $y_i \in N_{bin2}$, sequentially assign unique labels (l_i) from $i = 1, \dots, mid$
10. **for** $i = 1, \dots, mid$
11. • select instances y_a, y_b from N_{bin1} and N_{bin2} where label (l_i) of y_a, y_b are same, i.e. $y_a(l_i) = y_b(l_i)$
- Generate a minority class synthetic sample x by $\text{average}(y_a, y_b)$
- Add x to X_{new} and increment X_{newchk} :
12. **end for**
13. If $X_{newchk} < T$, pair instances in X_{new} with instances in each parent bin N_{bin} and repeat steps 10 to 12 to create two different sets of $X_{new[j]}$ ($j = 1, 2$). If still $X_{newchk} < T$, pair each set of current X_{new} with it's immediate parent set and later with predecessor set (i.e., set that contributed to generating that current set) of immediate parent and repeat steps 10 to 12 for subsequent rounds.
14. If $X_{newchk} \geq T$, assign all samples in X_{new} with the label of the minority class (defective) and add to dataset N .

End

should be very large. We measure the diversity of two data instances using the Mahalanobis distance measure proposed by P. C. Mahalanobis [85]. This measure is adopted because it eliminates several limitations of the Euclidean metric. Defect datasets are known to contain noise or duplicates [40] due to the collection process of the metrics. In this case, the use of the Euclidean distance measure would fail to recognize the highly correlated or duplicated data samples that provide no new information during the classifier training. As a unit-less measure, the Mahalanobis distance measure provides a relative measure of an instance distance or residual from a common selected point and helps identify and detect the similarity between an unknown sample set and a known one, which also helps in detecting outliers. This measure is preferred because of its multivariate effect size, which considers the correlation of the dataset and is therefore, not scale dependent. The covariance among the data samples is also considered when calculating the distances thus, overcoming the inherent scale and correlation problems associated with the Euclidean distance measure.

Considering two data instances or vectors of the defective class $x = (x_1, x_2, x_3, \dots, x_n)^T$ and $y = (y_1, y_2, y_3, \dots, y_n)^T$, the Mahalanobis distance between them is defined as:

$$d(x, y) = \sqrt{(x - y)^T S^{-1} (x - y)}, \text{ where } S^{-1} \text{ is the covariance matrix}$$

We use this measure to help rank and sort the data instances according to their distance in a decreasing order. By sorting the data, we are able to distinguish data samples that are far or close from the central data instance.

7.4.3 Phase 2: Data Partitioning and Pairing

To avoid the problems associated with the k -NN based techniques, MAHAKIL uses a partitional cluster approach. Based on the concept of the equal chromosomes from two parents established in the chromosomal theory of inheritance, we partition or segregate the data samples into two bins based on having a small Mahalanobis distance (typical case) and having a large Mahalanobis distance (atypical case). Using two bins to partition the sample also results in the best and generalized output of synthetic data after several runs of simulation experiments. We believe that it is practical to use two parents because this approach still works even when there are very few defective instances. The two partitions are obtained through sorting using the middle instance found during the ranking to partition them. All data samples with Mahalanobis distance values higher than or equal to the middle data sample are grouped into one partition while the second partition comprises the other data samples. Individual samples within the two partitions are sequentially labeled and then paired. Two

instances from each bin are assigned the same label to uniquely identify two independent instances from both bins that have been systematically and orderly paired as a "couple" (Steps 7- 9 of Algorithm 1). The pairing is done in a systematic order such that the first instances of both partitions are paired followed by the second to last instances in sequential order. These pairs are identified by their assigned labels. This is done to ensure that there are no overlapping samples and that the consequent samples to be generated reside within the minority decision boundary and fill the space between the two clusters formed by the partitioning process.

7.4.4 Phase 3: Synthetic Sample Generation

In the final phase, the synthetic samples are generated by aggregating and computing the mean or average between two paired instances from each partition, which, is then added to the original data N_{min} (steps 10 to 12 of algorithm 1). Unlike the SMOTE algorithm and k -NN based techniques, which use the k -nearest neighbor to determine and generate a new synthetic instance, this approach considers two instances that may not be close, as in being neighbors, and orderly merges two unfamiliar instances by considering them as "male" and "female" instances. In other words, based on the two partitions created in the previous phase, the resultant instances that are generated will be distinctly unique but related to the parents from both partitions. Maintaining an exact order between the pair of bins ensures that the generated samples also follow the trend of their parent instances. Thus, all of the synthetic data instances are positioned within both bins so that no data instance falls out of the decision boundary of the minority class. This approach is beneficial because the synthetic samples generated do not overlap in the majority region. Another advantage of MAHAKIL over the k -NN based techniques is that, whereas the synthetic samples generated by the k -NN based techniques tend to cluster or organize themselves in a particular group within the minority class samples, thus providing limited information to the trained classifier, the synthetic samples generated by MAHAKIL are evenly distributed to exhaust all possible minority samples within the two clusters thus, providing more information to the classifier. Analytically, MAHAKIL generates synthetic data samples of the minority class within the convex hull of the minority samples. By strictly working within the boundary of minority class, MAHAKIL ensures no data samples are created outside the decision boundary of the minority class. Additionally, the synthetic samples are convex combinations of well segregated parents that are dissimilar

regarding the Mahalanobis distance measure ensuring duplicate instances are not generated.

Based on the Pfp value, the algorithm computes the amount of synthetic data instances to be generated and prunes other data that are generated but not required. All data instances in the nodes or levels above the last level are included in the final dataset. The pruning starts from the last level of data generated by dividing the amount of data samples left to make our required final set complete by the number of nodes on that level. The value computed is then taken from each node on that level, which means that each node on the last level will contribute equal amount to the final set.

7.5 Experimental Methodology

The research question we ask in this study is "Can MAHAKIL perform better than other oversampling approaches?" The primary goal of this study is to improve performance of defect prediction models trained on imbalanced datasets using our proposed oversampling approach. Comparative experiments are conducted in order to answer the research question. The performance of MAHAKIL is compared with that of SMOTE [22], Borderline-SMOTE [44], Random Over-Sampling (ROS), ADASYN [48] and the no sampling approach (NONE).

Below, we describe the datasets, selected oversampling approaches, predictive models, evaluation metrics and statistical tests used for the study. For a comprehensive study, the datasets chosen vary in terms of the imbalance ratio and sample size. Predictive models spanning different domains of machine learning and statistics are also applied and a further comparison is made using statistical tests to observe the statistical significance and dominance of our sampling approach on predictive models over the other four data sampling approaches considered.

7.5.1 Benchmark Datasets

We conduct an experiment on 20 different benchmarked datasets extracted from PROMISE Repository² which is freely available online. Jurezcko and Madeyski [56], and Jurezcko and Spinellis [57] kindly donated the projects, which are stored in the PROMISE Repository. The PROMISE Repository stores a collection of datasets that are commonly used by the software engineering research community to construct predictive software models [115]. Twenty static code

²<http://openscience.us/repo/>

TABLE 7.1: Description of the Static Code Metrics [49]

Abbreviation	Description
WMC	Weighted methods per class
DIT	Depth of Inheritance Tree
NOC	Number of Children
CBO	Coupling between object classes
RFC	Response for a Class
LCOM	Lack of cohesion in methods
LCOM3	Lack of cohesion in methods, different from LCOM
NPM	Number of Public Methods
DAM	Data Access Metric
MOA	Measure of Aggregation
MFA	Measure of Functional Abstraction
CAM	Cohesion Among Methods of Class
IC	Inheritance Coupling
CBM	Coupling Between Methods
AMC	Average Method Complexity
Ca	Afferent couplings
Ce	Efferent couplings
CC	McCabe's cyclomatic complexity
Max(CC)	Maximum value of CC methods of the investigated class
Avg(CC)	Arithmetic mean of the CC value in the investigated class
LOC	Lines of Code
Bug	Number of detected bugs in the class

metrics (Table 7.1) are measured from classes or modules of these projects which are written in JAVA. Non-defective modules are labeled as zero and defective modules are labeled with the total number of defects present in the module. A summary of the data sets, including the total sample size, number of features and percentage of fault-prone modules (Pfp) or defects for each dataset, is presented in Table 7.2. With varying degrees of sample size, features and imbalance ratios, these datasets provide an extensive domain for the evaluation of the sampling techniques.

7.5.2 Baselines

MAHAKIL is compared to four other common oversampling techniques and the no sampling approach (NONE). A brief explanation of how these techniques work is provided below.

TABLE 7.2: Summary Statistics of Selected 20 Imbalanced Datasets Extracted from the PROMISE Data Repository

#	Release	# modules	# defects	defects (%)
1	ant-1.3	125	20	16.0
2	ant-1.4	178	40	22.5
3	ant-1.5	293	32	10.9
4	ant-1.6	351	92	26.2
5	arc	234	27	11.5
6	camel-1.4	875	145	16.6
7	camel-1.6	965	188	19.5
8	ivy-1.4	241	16	6.6
9	ivy-2.0	352	40	11.4
10	jedit-4.0	306	75	24.5
11	jedit-4.1	312	79	25.3
12	jedit-4.2	367	48	13.1
13	log4j-1.0	135	34	25.2
14	pbeans2	51	10	19.6
15	redaktor	176	27	15.3
16	synapse-1.0	157	16	10.2
17	systemdata	65	9	13.8
18	tomcat	858	77	9.0
19	xerces-1.2	440	71	16.1
20	xerces-1.3	453	69	15.2

SMOTE: Proposed by Chawla et al. [22], this technique over-samples the minority class in a dataset by creating "synthetic" samples. It over-samples the minority class in a bid to make the dataset as balanced as possible based on the configuration parameter values. To generate these synthetic samples, each minority class sample is considered and the new samples are introduced along the line segments that join any of the k minority class nearest neighbors.

Borderline-SMOTE: This is a modification of the SMOTE technique but with the main focus on harder-to-classify minority class data instances, which are referred to as borderline data instances [44] . The algorithm first finds minority class instances that have more majority class instances as nearest neighbors than minority class instances and applies the SMOTE technique to such instances. This approach has the advantage of strengthening the borderline between the majority and minority class data instances.

ADASYN: ADASYN was proposed by He et al. [48] and it assigns weights to the minority classes and dynamically adjusts the weights in a bid to reduce the bias in the imbalanced dataset by considering the characteristics of the data distribution. The ADASYN algorithm incorporates a density distribution in automatically deciding the number of synthetic samples needed for each minority

TABLE 7.3: Classification Models and their Hyperparameter Configurations

Model	Overview	Hyperparameters
C4.5	J48 Decision Tree	$c = \{0.05, 0.10, 0.20, 0.30, 0.40, 0.50, 0.60, 0.70\}$
NNET	3-layer Neural Network	$\text{size} = \{4, \dots, 28\}, \text{decay} = \{0.10, 0.20\}$
KNN	K- Nearest Neighbor	$c = \{2^*(0, \dots, 7) + 1\}$
RF	Random Forest	$\text{mtry} = \{10, 50, 100, 200, 250, 500, 1000\}$
SVM	Support Vector Machine	$c = \{2^{-6}, \dots, 2^{10}\}$

class sample. The learning algorithm is induced to focus on the hard to learn (classify) examples within the minority class samples. Therefore, the samples generated are not equal for all samples.

ROS: ROS increases the number of minority instances by randomly replicating the already existing minority class instances. A simple and easy approach to implement, ROS has performed favorably in previous empirical studies [9, 59].

7.5.3 Experimental Design

For the experiments, we use five classification algorithms (Table 7.3) which have been used in similar class imbalance studies [6, 78, 81] to evaluate the effect of each sampling approach on the datasets. Conventional methods such as Neural Networks [60], SVM [144] and the C4.5 decision tree [111] are used. We also consider Random Forests (RF) which is known to improve classification accuracy because they grow an ensemble of classification trees and allow them vote on the classification decision [81]. Lessman et al. [75] showed that RF was significantly better than 21 other prediction models. KNN [26] is also used because it is easy to implement and it classifies an unknown instance by considering its nearest neighbors. Hyper-parameter configurations for the classification models estimated using a 10-fold cross-validation method are presented in Table 7.3.

The training data are randomly selected using 2/3 of the sample size and the remaining 1/3 as the testing data for each dataset. The division is done using stratification such that the proportion of minority class distribution or P_{fp} is kept intact within both the training and testing datasets. Figure 7.3 shows the steps followed for the empirical evaluation of the data re-sampling methods for each dataset. To reduce the impact of bias caused by the random

splitting method, the whole experiment is conducted 5 times. The results are averaged across the independent runs and presented. The sampling techniques are applied only to the training datasets. Because a significant number of the projects have fewer minority samples than the number of features, we use a multicollinearity technique to eliminate features with very high correlations ($>90\%$) and zero-variance predictors. This helps to successfully invert the singular matrices during the computation of the Mahalanobis distance for the MAHAKIL approach.

We also consider a 10-fold cross-validation in choosing the parameters of the validation set during the model construction on the preprocessed or re-sampled training datasets i.e., we randomly partition the whole dataset into 10 folds of data and use all but one for training and validating on the remaining fold. This procedure ensures that the optimal parameters are found and selected after validating them on each fold of data. The experiments are conducted on each dataset after re-sampling with each over-sampling approach at different levels of Pfp . With all datasets having a Pfp of less than 30%, we conduct three different experiments for those datasets by applying each over-sampling approach until the Pfp reaches 30%, 40% and finally 50%. We stop over-sampling at 50% based on the assumption of a balanced dataset in most algorithms and also based on the conclusion of Ahmad Abu, et al. [117] that over-sampling to at most 50% achieves better results.

The experiments were all conducted using R [129], an open source statistical tool. The library packages Caret [72] and DMwR [132] were used for the model construction and SMOTE sampling approach respectively. The ADASYN technique was implemented following the instructions given by the authors [48] using the MATLAB tool. For SMOTE, Borderline-SMOTE and ADASYN which considers a k -neighbor value, k was set to the default value of five.

7.5.4 Evaluation Performance Metrics

A typical defect dataset is categorized into two groups: defective and non-defective. Conventionally, minority or defective classes are labeled as positive and majority classes labeled as negative in a two-class problem. From the confusion matrix (Table 7.4), True Positives (TP) are the number of positive samples correctly classified as positive, False Positives (FP) are the number of negative samples wrongly classified as positive, True Negatives (TN) are the number of negative samples correctly classified as negative and False Negatives (FN) are the number of positive samples wrongly classified as negative. Indicators such as Precision, Recall or probability of detection(pd) and F-measure have also

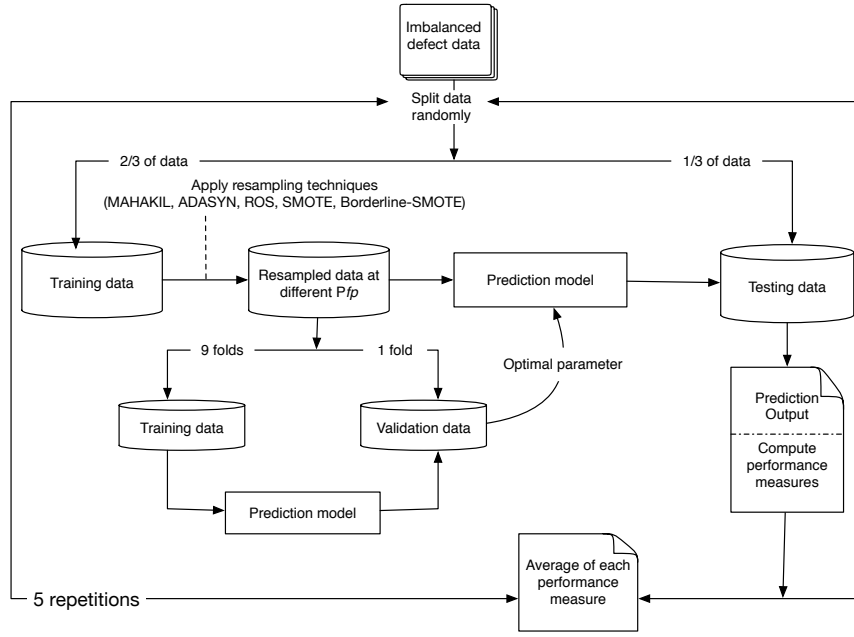


FIGURE 7.3: Framework of experiments

TABLE 7.4: Confusion Matrix for the Two-class Problem

	Predicted Positive	Predicted Negative
Actual Positive	TP	FN
Actual Negative	FP	TN

$$Recall(pd) = \frac{TP}{TP + FN} \quad (7.1)$$

$$pf = \frac{FP}{TN + FP} \quad (7.2)$$

been used to assess the performance of models on the minority class [18, 54]. Precision and the F-measure indicator which computes the harmonic mean between recall and precision are excluded from our experiment because precision is known to be an unstable indicator for models trained on highly imbalanced datasets [88]. We adopt the recall(pd) and pf measures for our experiment. Recall (pd) is a measure of completeness that attains a high value of 100% if the number of false negatives is zero. The probability of false alarms (pf) which measures how many of the data instances that activated or prompted the detector did not actually have the defect concept [107] is also computed. A lower pf implies that less time and effort will be required in determining the true defects. A high pd and low pf indicate a better prediction model [88]. The evaluation metrics used for this study are defined in equations (1) to (2) computed from Table 7.4.

7.5.5 Performance Comparison

To statistically evaluate and compare MAHAKIL to the other four techniques, we adopt the *win-tie-loss statistics* which has been used in several empirical studies [67, 68]. Three counters-*wins*, *ties* and *losses* are used to assess the overall performance of a predictor (model*sampling technique) in a pairwise manner for all possible pairs of predictors. By using Brunner’s statistical test ($p < 0.05$) [17], two predictors are compared and if the performance distributions of the two predictors are not significantly different, the *ties* counters of both estimators are increased by 1. Otherwise, the *wins* counter of the predictor with a higher performance is increased and the *losses* counter of the other predictor with low performance is also increased. All of the counters are summed per the *Pfp* value to find the best performing predictors. Furthermore, to ascertain the practical significance of the results, we compute the effect size using Cliff’s method with Hochberg’s method proposed by Brunner et al. [17] and recommended by Kitchenham et al. [63] as a more robust statistical test for non-normal data. The effect sizes ($\hat{\delta}$) are interpreted using the magnitude labels (small ($\hat{\delta} \leq 0.112$), medium ($0.112 < \hat{\delta} < 0.428$) and large ($\hat{\delta} \geq 0.428$)) proposed by Kraemer and Kupfer [69].

7.6 Empirical Results

7.6.1 Overall Results

In this section, we present the results of each sampling approach on the prediction models and compare the performance of MAHAKIL with ROS, SMOTE, Borderline-SMOTE, ADASYN and No Sampling (NONE). We present the results for each sampling technique on each dataset per each percentage of fault-prone modules (*Pfp*) value. For each dataset, we present the performance values for all of the *Pfp* values and the prediction models in Figures 7.4 and 7.5. In the figures, the dot plot compares the performance of each sampling method (represented with different shapes) across the various *Pfp* values (represented with different colors) per prediction model. It can be clearly seen that the application of the sampling methods at different *Pfp* rates produces a better performance than NONE regarding the *pd* performance values across all prediction models. However, NONE has the lowest and better *pf* values compared to the application of the sampling methods across most of the prediction models and datasets.

From Figures 7.4 and 7.5, we observe the following.

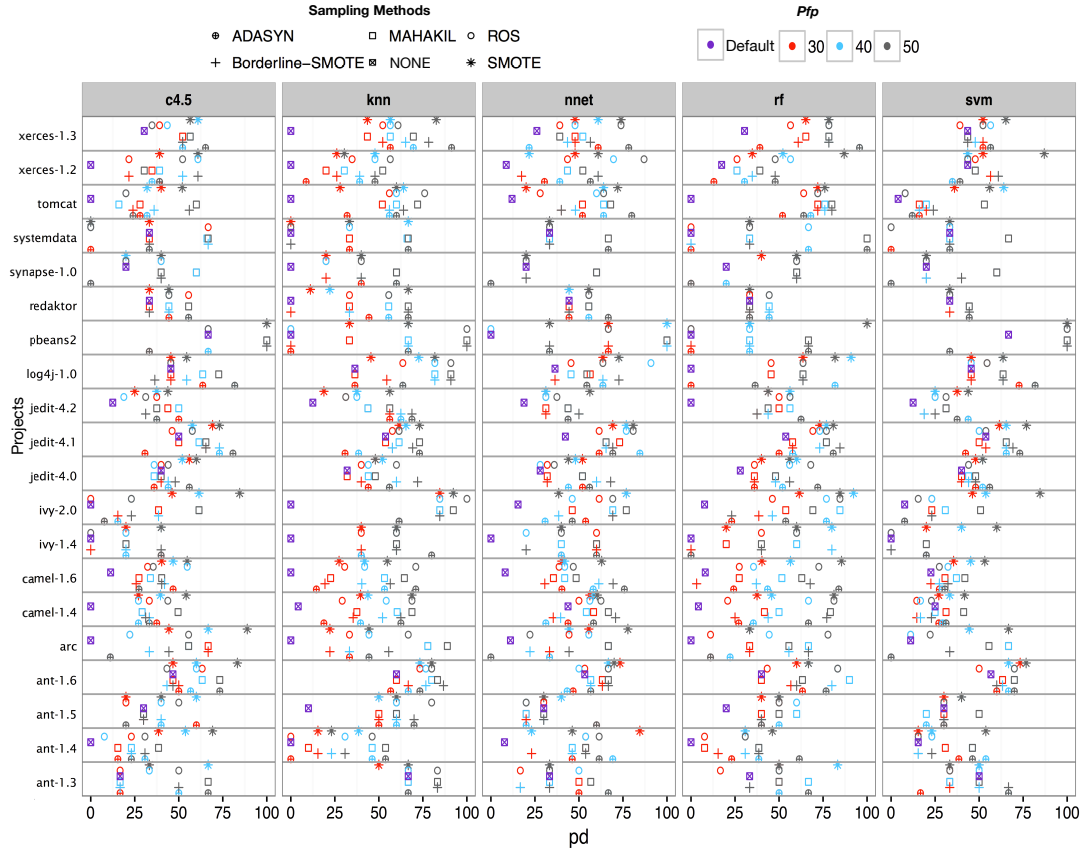


FIGURE 7.4: Performance plot of the sampling techniques per each prediction model for the pd performance metric on 20 imbalanced datasets across different percentage of fault-prone modules (Pfp) values.

1. Across all prediction models, SMOTE and MAHAKIL consistently performed better regarding the pd values than Borderline-SMOTE, ADASYN and ROS. Their distributions were quite similar with SMOTE slightly edging MAHAKIL.
2. SMOTE resulted in the worst pf values whiles NONE and MAHAKIL resulted in the best pf values across all prediction models.
3. Across all prediction models and datasets, the best pd performance was achieved when Pfp was increased to 50% for all sampling methods.

Across all models and datasets, MAHAKIL performed relatively better with high pd and low pf values. This can be explained by the uniqueness and closeness or the representativeness of the new synthetic data injected into the dataset. The models trained on the SMOTE resampled datasets resulted in the highest mean pd but with the highest mean pf . This implies that the application of SMOTE results in the models detecting more defective models, although this approach is not cost effective. The results of the statistical tests conducted to validate the results are presented in the next section.

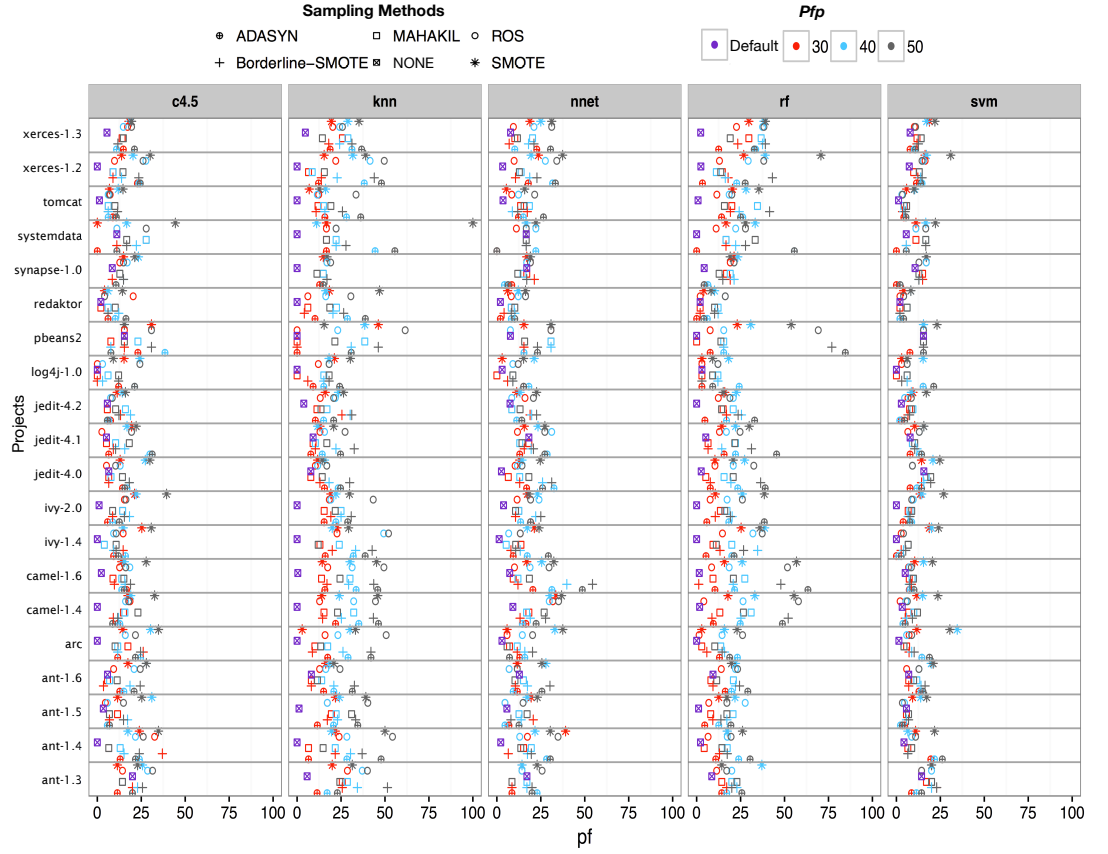


FIGURE 7.5: Performance plot of the sampling techniques per each prediction model for the pf performance metric on 20 imbalanced datasets across different percentage of fault-prone modules (Pfp) values.

7.6.2 Statistical Comparison of Techniques

To investigate in detail the performance of MAHAKIL and demonstrate the effectiveness of MAHAKIL in achieving better (significant) performance measures, we present the win-tie-loss results of MAHAKIL against those of SMOTE, Borderline-SMOTE, ADASYN, ROS and NONE per each Pfp for each performance measure across all 20 datasets in Figure 7.6. Additionally, the practical effect of the statistical results is also computed and presented in 7.6. We compute the wins, ties and losses for each predictor on all datasets per each round of experiment (i.e., Pfp value). For each Pfp rate, the win-tie-loss statistic is presented together with the magnitude of the effect size. We observe from the figure that the models trained on MAHAKIL resampled data significantly outperformed most of the sampling techniques, with the Pfp increasing from 30 to 50 for both performance measures. MAHAKIL also proved to be statistically and significantly better in terms of the pf value having atleast a win across all of the prediction models. Most of the wins were practically significant as confirmed by the magnitudes of the effect size. However, NONE consistently

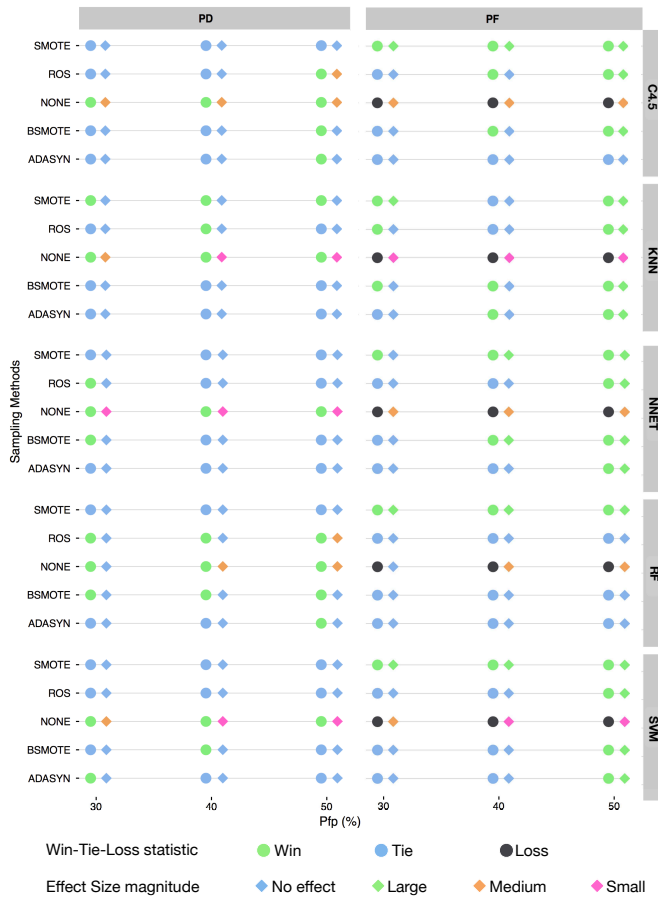


FIGURE 7.6: Brunner's statistical test win-tie-loss and effect-size comparison of MAHAKIL vs. ROS, ADASYN, Borderline-SMOTE, SMOTE and NONE across all 20 datasets per each defect prediction model, performance measure (pd , pf) and Pfp value (30%, 40%, 50%).

outperformed MAHAKIL achieving magnitudes of small or medium effect sizes indicating that for the lowest pf , the no sampling method produces better results than using the sampling methods.

MAHAKIL consistently outperformed the NONE method with significant and practical effect sizes regarding the pd results. There were several ties between MAHAKIL and each of the other four sampling techniques regarding the pd results across all models. From the figure, we can conclude that MAHAKIL achieves better or comparable prediction results than the other four sampling techniques. The results also show that for best results, MAHAKIL should be applied at a Pfp of 50% because all of the models trained on datasets resampled at this Pfp value produced significant and practical results.

For more details about the overall performance of MAHAKIL and to determine which predictors (combination of prediction models and sampling methods) produce the best performance, the total win-tie-loss results per Pfp value

are reported in Table 8.4. For each performance measure, we compute the total wins, ties and losses for each predictor on all datasets per each round of experiment (i.e. Pfp value). Each single predictor is compared with 29 other predictors (5 models * 6 Sampling techniques - 1). We rank the results by their total wins-losses. Out of the 58 (29*2) overall win-tie-loss comparisons aggregated from two performance measures per Pfp value, the application of MAHAKIL always resulted in a positive win-loss value. The models trained on MAHAKIL resampled datasets consistently had positive wins-losses values and were in the top half of the predictors. The models trained on data resampled with ROS and SMOTE performed worse with the highest numbers of losses and negative wins-losses values. Overall, the results shows that MAHAKIL clearly outperforms all four sampling techniques and the no sampling method. All five prediction models combined with MAHAKIL performed better than any other combinations, with an impressive number of wins and minimum losses when Pfp was 50%. Overall, the combination of RF and KNN with MAHAKIL produced the best performance with the highest number of wins and few losses. Based on the results, RF with MAHAKIL is recommended for highly imbalanced defect datasets when a Pfp of 40 or less is applied. For a balanced dataset (Pfp of 50), the KNN prediction model combined with MAHAKIL is recommended. In agreement with Figure 7.6, we observe that for the best results, MAHAKIL should be applied at a Pfp of 50% because the models trained at this Pfp value occupied the top five positions.

7.7 Discussion

7.7.1 Stability of MAHAKIL

The datasets used for the experiments vary in terms of the imbalance ratio. We consider datasets with an imbalance ratio of less than 15% as severely/highly imbalanced and the remaining ones as lowly imbalanced. Severely imbalanced datasets imply that more artificial data instances will be generated compared to the low imbalanced data when we apply the sampling methods at a specified Pfp rate. From the pd plot in Figure 7.4, we observe that as the Pfp increases, the pd values increase across all of the prediction models. For example, when we consider the systemdata project, which is severely imbalanced, there are only nine default defective instances, with the test data consequently comprising only three instances. NONE achieves a minimum value of 0% and a maximum value of 33% across all prediction models. The sampling methods except MAHAKIL

TABLE 7.5: Performance in terms of wins, losses, and wins-losses aggregated from the two performance measures on 20 datasets per each Pfp value. Higher wins(%), wins-losses(%) and lower losses(%) indicate higher performance of the predictor. The predictors are ordered by wins-losses.

Pfp 30					Pfp 40					Pfp 50				
Model	Sampling	Wins	Losses	Wins-Losses	Model	Sampling	Wins	Losses	Wins-Losses	Model	Sampling	Wins	Losses	Wins-Losses
RF	MAHAKIL	20	5	15	RF	MAHAKIL	27	11	16	KNN	MAHAKIL	34	11	23
RF	NONE	26	12	14	C45	MAHAKIL	24	11	13	RF	MAHAKIL	32	11	21
NNET	MAHAKIL	24	11	13	KNN	MAHAKIL	26	14	12	NNET	MAHAKIL	29	9	20
RF	ADASYN	19	9	10	NNET	MAHAKIL	23	11	12	C45	MAHAKIL	29	13	16
C45	MAHAKIL	16	7	9	RF	ADASYN	24	14	10	SVM	MAHAKIL	23	12	11
RF	BORDERLINE	19	11	8	RF	BORDERLINE	26	18	8	RF	SMOTE	21	15	6
NNET	ADASYN	19	12	7	SVM	ROS	24	16	8	NNET	ADASYN	19	14	5
RF	ROS	19	12	7	SVM	MAHAKIL	20	14	6	RF	BORDERLINE	27	22	5
SVM	MAHAKIL	11	4	7	NNET	ROS	16	13	3	C45	NONE	28	27	1
SVM	ADASYN	18	13	5	RF	SMOTE	19	16	3	NNET	BORDERLINE	18	17	1
KNN	MAHAKIL	12	8	4	C45	BORDERLINE	19	17	2	RF	ADASYN	24	23	1
SVM	ROS	12	8	4	RF	NONE	28	26	2	RF	NONE	28	27	1
C45	NONE	27	25	2	RF	ROS	26	24	2	KNN	NONE	28	28	0
NNET	ROS	13	12	1	C45	NONE	28	27	1	SVM	NONE	27	27	0
C45	BORDERLINE	10	10	0	KNN	NONE	28	28	0	NNET	NONE	26	27	-1
KNN	NONE	28	28	0	SVM	NONE	27	27	0	RF	ROS	24	26	-2
RF	SMOTE	12	12	0	C45	SMOTE	13	16	-3	NNET	ROS	16	19	-3
SVM	NONE	27	27	0	NNET	NONE	25	28	-3	SVM	SMOTE	17	20	-3
C45	ADASYN	9	11	-2	KNN	SMOTE	14	18	-4	SVM	ROS	16	20	-4
NNET	SMOTE	17	19	-2	NNET	ADASYN	9	13	-4	C45	SMOTE	14	19	-5
C45	ROS	8	12	-4	NNET	BORDERLINE	12	16	-4	KNN	ADASYN	18	23	-5
NNET	NONE	19	24	-5	SVM	ADASYN	12	17	-5	KNN	BORDERLINE	17	24	-7
SVM	BORDERLINE	7	13	-6	C45	ADASYN	15	21	-6	C45	ADASYN	18	26	-8
NNET	BORDERLINE	6	14	-8	NNET	SMOTE	11	19	-8	C45	BORDERLINE	18	26	-8
KNN	ADASYN	6	15	-9	SVM	BORDERLINE	10	18	-8	SVM	ADASYN	11	20	-9
C45	SMOTE	8	18	-10	C45	ROS	16	25	-9	SVM	BORDERLINE	13	22	-9
KNN	BORDERLINE	5	15	-10	SVM	SMOTE	16	25	-9	KNN	ROS	14	24	-10
SVM	SMOTE	12	22	-10	KNN	ADASYN	15	25	-10	KNN	SMOTE	11	22	-11
KNN	ROS	5	21	-16	KNN	ROS	10	22	-12	NNET	SMOTE	13	24	-11
KNN	SMOTE	4	28	-24	KNN	BORDERLINE	10	23	-13	C45	ROS	15	30	-15

obtained a pd value of 0% across all the prediction models. MAHAKIL achieved the highest pd value as the Pfp increased to 50%, which could be due to the quantity of diverse and important data instances introduced into the training data by MAHAKIL as the Pfp increased. When we consider a dataset that is lowly imbalanced, such as pbeans2, we observe similar results regarding the MAHAKIL method while the other methods except SMOTE obtain a 0% pd value across the prediction models. We argue that our approach is more stable and even the smallest amount of synthetic data introduced into the training data improves the pd performance of the prediction models. Similarly, the pf plot in Figure 7.5 shows a similar pattern to the pd plot, with the values of pf increasing as Pfp increases. However, MAHAKIL obtains the lowest values across all sampling methods.

7.7.2 Choice of Prediction models

The KNN and RF models achieved the best results when trained on MAHAKIL resampled datasets. We attribute this to how these models work and the positioning of the synthetic generated samples within the minority class by MAHAKIL. MAHAKIL generates minority samples that are distributed within the

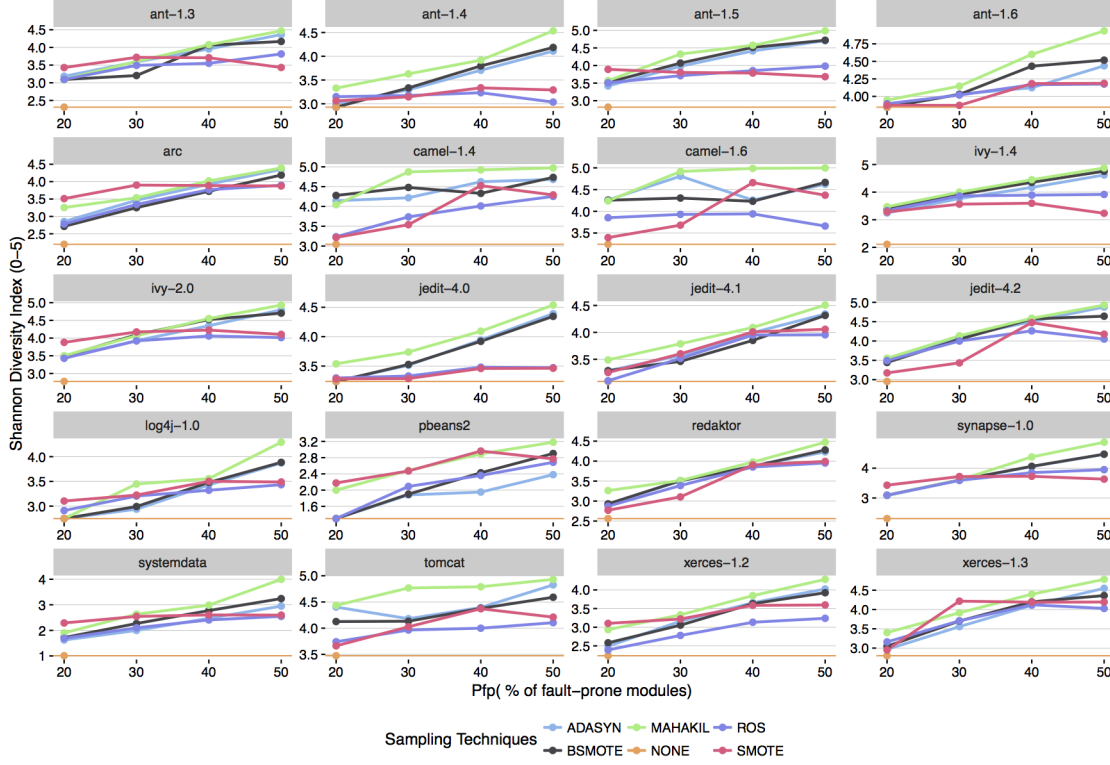
sample space. Consequently, the test set's similar instances are found in the generated minority space more easily by these prediction models, especially the KNN model which selects samples closer to the test instance. As an ensemble method, RF is more robust against overfitting and the parameter settings [15]. Based on the internal parameters, RF is able to compute the importance of each predictor and choose the most informative predictor. It is also able to measure the internal structure and pattern of the training data [77], which makes it easy to identify the patterns of the MAHAKIL resampled datasets because the minority region is easy to identify. Lessman et al. [75] confirmed that the RF classifier was a more effective classifier for software defect prediction studies. Impressively, at a Pfp rate of 50%, all of the prediction models trained on MAHAKIL resampled datasets outperformed the other sampling methods.

7.7.3 Why MAHAKIL Performs best

The findings of this study provide substantial empirical support demonstrating that unique data instances can be generated with predetermined characteristics. For all models, MAHAKIL performed relatively better with high pd and low pf values. This can be explained by the uniqueness and closeness or the representativeness of the new synthetic data injected into the dataset.

The performance of MAHAKIL can be attributed to the succinct and almost uniformly spread of the newly generated samples within the boundary of the minority class. Based on the strategy used to generate new data instances, all "empty spaces" within the minority data are filled if more data instances are generated which, produces diverse data instances. To demonstrate the diversity, we compute the diversity (variety of different data instances) for each metric in the minority class of the resampled dataset. The diversity for each metric was computed using the Shannon-Wiener diversity index (H)[55], which is a widely accepted metric for computing and quantifying diversity in ecological and environmental studies. This method assigns the same weight to common and rare species. H ranges between 0 and 5, with a higher H value implying higher diversity and a lower likelihood of selecting two instances of the same species, and hence less duplicates. The line plots in Figure 7.7 present the average diversity values of the metrics for each dataset and sampling method.

We plot the diversity for the data when no sampling is applied (NONE) at the Pfp -20 rate. For all sampling methods, Figure 7.7 shows that the diversity within the minority class increases as the Pfp increases or more data instances are produced. However, the diversity value of the SMOTE resampled datasets decreases as the Pfp reaches 40%. Across all datasets, MAHAKIL achieves

FIGURE 7.7: Average diversity value at each Pfp for all 20 projects

the highest diversity values for all the metrics per each Pfp especially at a Pfp of 50%. Based on these results, the success of MAHAKIL confirms that prediction models not only require ample training data but also very relevant and exhaustive data that contribute more information to the model. Data quality is a very crucial issue in the domain of defect prediction and using sampling approaches that generate less informative data tends to only worsen the situation.

7.7.4 Threats to Validity

Threats to Validity refers to the extent to which a research study actually studies what the researcher purports to investigate [120]. To address the observed potential limitations in our findings, the following discusses the threats to the construct, internal, statistical conclusion and external validity of the study as recommended by Wohlin et al. [142].

Construct Validity

Our proposed approach assumes that defective classes can be appropriately separated into two clusters, which may not always be the case for defect datasets.

Some defect datasets consist of several multi-clusters and even majority instances mix with minority instance clusters. Restricting our technique to identifying mini clusters and working in local patches will improve the method and reduce the *pf*. We leave this for future studies. The metrics considered for our study are a potential threat to our experimental results. By using a single set or type of metrics, we cannot generalize our results to other types of defect metrics. Nonetheless, static code metrics are known to perform very well and have been used in several empirical studies on defect prediction because they are easy to collect and can be collected from any software once the source codes are available [90]. We also acknowledge that the sampling and splitting approach could affect the classification results. We randomly split the datasets into training (2/3) and testing (1/3) sets. Most empirical studies use this procedure for predictive data modelling [8, 75, 127] making it an accepted approach. It would be interesting to examine the effects of our proposed oversampling method on cross-version datasets. However, we leave this for future studies. Because there are no universally agreed upon performance measures for assessing the prediction performance on imbalanced data, we used two different measures: Recall (*pd*) and *pf*. Recall (*pd*) and *pf* enabled us to know the performance of the predictors on the minority class. The assessment results for the performance of the prediction models using the AUC and G-mean measures are provided in the supplementary file. Other measures not used for our study are left for future studies.

Internal Validity

Our approach is dependent on the use of the Mahalanobis distance metric. The advantages of the Mahalanobis distance notwithstanding its use comes with a high price. For very large data records, the computation of the covariance matrices can be difficult to determine accurately, especially if there are a large number of features. More memory and time resources are required because these resources grow quadratically rather than linearly with the number of features. These problems may be insignificant when only a few features are needed. This challenge also affects methods based on the Euclidean distance measure and a more advanced and time-efficient technique is required to handle the computation of covariance matrices for large dimensional features. Moreover, covariance matrices cannot be inverted for datasets with fewer minority samples than the number of features (singular matrix). We adopted multicollinearity techniques to eliminate metrics with very high correlations. A generalized inverse method

can be incorporated to compute an inverse-like matrix that will ignore the non-informative metrics and the impact of such a method on our algorithm is left for future studies. The hyper-parameters configuration set could introduce some bias in our results. However, we used 10-fold cross validation at each run and repeated the overall experiments five times, which enabled us to report the average results across all run times. An important known validity in empirical experiments is the quality of the data, which is difficult to obtain and verify. We do acknowledge that noise or outliers inherent in datasets extracted from most open-source projects could have significant effects on prediction performance as reported by Gray et al. [40]. As such, applications of data cleansing techniques such as noise detection and elimination measures [62, 128] are left for future studies.

Statistical Conclusion Validity

In this study, we used Brunner’s statistical test for the win-tie-loss analysis and Cliff’s method with Hochberg’s method for the effect size computations. These statistics are recommended by Kitchenham et al. [64] as a more robust and reliable test than other statistical tests for evaluating the performance of prediction models. The impact of this threat on our study is thus limited.

External Validity

Our study used a limited number of datasets. The projects used were all from one source, the PROMISE repository and have been extensively used in several defect prediction studies. However, our results may not be generalizable to all software projects, especially proprietary projects, because we used only open source projects. Nevertheless, we provided a detailed description of the experimental setup, which will be useful for replicating our experiment using commercial software systems. We also used only five classification techniques spanning the machine learning, data mining and statistical based families. The effects of our proposed technique on other classification models that were not examined in this study are not known and extra evaluation of other models are required to generalize our results. Lastly, we compared our proposed technique to four other sampling techniques. We intend to compare our technique with other simple and more complex techniques in a future study.

7.8 Chapter Summary

With most real world defect datasets highly imbalanced, resampling approaches have been adopted to alleviate the class imbalance learning issues. The majority of resampling approaches are synthetic based, where synthetic minority data instances are generated to balance the distribution between the minority and majority class samples. Although these synthetic methods have been shown to increase the prediction performance, they sometimes generate erroneous or duplicated data instances, which leads to high false positives. They also tend to generate less diverse data points within the minority class. Exploiting these challenges, we proposed a novel over-sampling approach that systematically generates new minority samples based on their Mahalanobis distance. Motivated by the chromosomal theory of inheritance, MAHAKIL uses features from two parent instances to produce a new synthetic instance that has the characteristics of both parent instances and, thus, ensures that the synthetic data fall within the decision boundary of any classification algorithm. Our approach generates new samples by finding the average between two instances merged together based on the Mahalanobis distance disparity.

We empirically evaluated our approach by comparing it to four other over-sampling approaches (ROS, SMOTE, Borderline-SMOTE and ADASYN) using five classification models (C4.5, NNET, KNN, RF, SVM) on 20 imbalanced datasets. In total, over 200 different combinations of experiments were conducted and MAHAKIL consistently performed better than the other four approaches with regards to the performance measures. With respect to the individual prediction models, MAHAKIL was significantly better than all four approaches regarding the pf measure, with the RF and KNN models outperforming all of the models for all of the performance measures considered.

The performance of MAHAKIL was superior to the other methods because the data generated by the MAHAKIL algorithm inherited features from the parent data instances, which helped in correctly classifying most of the minority classes used during testing. We demonstrated in a more practical scenario that MAHAKIL works irrespective of the sample size of the data.

7.8.1 Concluding Remarks

Highly imbalanced data typically make accurate predictions difficult. Unfortunately, software defect datasets tend to have fewer defective modules than

non-defective modules. Synthetic oversampling approaches address this concern by creating new minority defective modules to balance the class distribution before a model is trained. Notwithstanding the successes achieved by these approaches, they mostly result in over-generalization (high rates of false alarms) and generate near-duplicated data instances (less diverse data). In this chapter, we introduce MAHAKIL, a novel and efficient synthetic oversampling approach for software defect datasets that is based on the chromosomal theory of inheritance. Exploiting this theory, MAHAKIL interprets two distinct subclasses as parents and generates a new instance that inherits different traits from each parent and contributes to the diversity within the data distribution. We extensively compare MAHAKIL with SMOTE, Borderline-SMOTE, ADASYN, Random Oversampling and the No sampling approach using 20 releases of defect datasets from the PROMISE repository and five prediction models. Our experiments indicate that MAHAKIL improves the prediction performance for all the models and achieves better and more significant pf values than the other oversampling approaches, based on Brunner's statistical significance test and Cliff's effect sizes. Therefore, MAHAKIL is strongly recommended as an efficient alternative for defect prediction models built on highly imbalanced datasets.

The performance of the sampling methods were heavily influenced by the rate of sampling (Pfp). In the next chapter, we examine and conduct in-depth analysis on the impact of the Pfp on several sampling approaches.

Chapter 8

Assessing the Impact of the Distribution Parameter of Data Sampling Approaches on Software Defect Prediction Models

8.1 Introduction

Prioritization of testing resources is a key challenge of any software engineering project. Allocating the same testing efforts to all modules of a software is not ideal since defects are not uniformly distributed within the software. Software quality teams have adopted defect prediction models to predict buggy modules in software and this helps in prioritization of scarce testing resources [75, 88]. The effectiveness of these prediction models are however dependent on the data used in training them [4, 28, 75].

Defect datasets are typically imbalanced with the non-defective samples dominating the defective samples [34, 40, 135]. In SE research, the prevalent and easy approach for solving the class imbalance issue is the application of data sampling methods [5, 22]. These sampling methods have configurable parameters that control the characteristics of the datasets produced for training classifiers. For example, the number of neighbors (K) considered for data generation in SMOTE can be configured prior to being applied on the training data. Different sampling methods have different configurable parameters and it is impractical to configure all these parameters of the various sampling methods during model construction. Nevertheless, a constant parameter for all sampling method is the amount of resampling or percentage of fault-prone modules (Pfp) which controls the amount of data added to or removed from the minority class samples or the majority class samples respectively.

Our literature analysis reveal that prior defect prediction studies that apply sampling methods as preprocessing techniques do not consider the impact of these parameters and its effect on prediction performance. Agrawal and Menzies [1] note that when the parameters of SMOTE are tuned based on the data distribution, significant improvements could be observed. Most sampling methods apply a fixed balanced resampling rate since an optimal Pfp rate for these existing sampling methods is a challenge. Our previous study [9] in Chapter 5 shows that the application of a fixed Pfp for sampling models may degrade performance of prediction models due to the difference in imbalance ratio and size (number of samples) of training datasets.

In this chapter, we investigate and assess the significant effect of tuning the Pfp rate of sampling methods on performances of defect prediction models. Different Pfp rates are selected and configured for each sampling method before being applied to the training datasets prior to classifier training. We conduct an extensive experiment on seven sampling methods and five prediction models evaluated with five performance measures across 20 releases of 10 open-source software projects. We further perform a mini ranking experiment to obtain the best performing predictors. Our analysis indicate that sampling methods have a strong correlation with Pfp . The stable ranking experiments show that a stable predictor is difficult to achieve across different evaluation measures. No predictor was stable across the Pfp rates. In conclusion, we observe that performance of all sampling methods studied can be improved when the parameters are tuned and encourage the tuning of this parameter.

8.1.1 Chapter Organization

The organization of this chapter is as follows. Section 8.2 reviews some previous studies on the application of sampling methods for defect prediction. Section 8.3 describes the experimental methodology, sampling methods and performance evaluation measures. The empirical results and analysis is presented in Section 8.4. The threats to validity are presented in Section 8.5 and conclude in section 8.6.

8.2 Related Works and Research Questions

Several studies have assessed the impact of sampling methods on defect prediction performance. There are however contradictory findings about the performance of these sampling methods especially concerning SMOTE.

These studies report the positive impact sampling methods have on prediction models. Kamei et al. [59] experimentally evaluated the effects of four sampling methods on four prediction models using two sets of industrial legacy software evaluated using the F1-value. They observed that sampling methods improved the performance of two out of four models. Similarly, Menzies et al. [91], Riquelme et al. [113], Pelayo and Dick [106] and Wang and Yao [135] concluded that sampling methods improve prediction performance. These studies applied the sampling methods using the default parameter values. In our recent study [9], we investigated the effects of four sampling approaches on ten effort-aware prediction models where we tuned the resampling parameters. Results showed that ROS outperformed the SMOTE approach and no sampling outperformed RUS when the dataset is highly imbalanced. In this chapter, we conduct an experiment that differentiates itself from previous studies in that, we assess the performance of more than five sampling methods when the Pfp parameter is tweaked. Additionally, we conduct cross-release predictions with five performance measures. We further statistically evaluate the performance of the models trained on these resampled datasets to examine if the improvements were significant or not using robust statistical test recommended by Kitchenham et al. [64]. We thus formulate three research questions.

RQ1: How does tweaking the Pfp parameter impact performance of sampling approaches on defect prediction models?

The recent study by Agrawal and Menzies [1] where they tuned several parameters of SMOTE showed that performance can be further improved when these parameters are tuned according to the distribution of the training data. However, there exist several sampling methods in literature and each sampling method has different configurable parameters. Tuning each parameter for the numerous sampling methods available is not practical. We focus on the distribution parameter of the sampling methods. Additionally, we study other sampling methods including SMOTE.

RQ2: To what degree (percentage of fault-prone modules Pfp) should one resample or balance the original data set?

The amount of resampling (Pfp) applied is an important factor in determining the performance of a defect prediction model on a resampled dataset [32, 138]. To maximize overall classification accuracy, most conventional prediction models assume the two classes in the datasets to be as balanced as possible or

equal [139, 145]. Intuitively, it may seem ideal if we stop resampling at 50%. For RQ2, we investigate the effect of applying the resampling methods at different Pfp rates to the training datasets with the aim of finding the most appropriate resampling rate which results in best or highest performance.

RQ3: Which are the high-performing resampling approaches for defect prediction?

With the abundance of several resampling approaches proposed in literature, software quality teams face a challenge of choosing the appropriate resampling method to use. Empirically validating these approaches and finding the more stable and highest performing resampling approach has the real world benefit of helping these quality teams adopt the best resampling method to use for defect prioritization or classification. We follow up on RQ1 by conducting an overall assessment to determine and recommend the best performing resampling methods based on a robust statistical test. Thus, we apply a comparison procedure among the resampling methods for each performance measure across all twenty datasets.

8.3 Case Study Design

To assess the effect of tweaking the distribution parameter of sampling methods on prediction performance, we briefly discuss the sampling methods, datasets, performance measures and comparison criteria used for the study. In emulation of real-life setting, two sets of project datasets are used for the experiments in order to examine the impact of applying sampling methods on the performance of the cross-release project prediction models. Detailed but simple experiments were conducted at different resampling rates (Pfp) to aid in the examination and analyses of the effects of sampling methods on the chosen prediction models.

8.3.1 Studied Sampling Approaches

Data sampling approaches were proposed with the aim of increasing the sample size of the minority class or decreasing the sample size of the majority class referred to as over-sampling and undersampling respectively. Seven sampling methods were considered for these experiments.

RUS: Random Undersampling is the simplest form of undersampling. Majority instances of the dataset are randomly selected and deleted whereas the minority instances are left intact. Information crucial to the classifier during

training could be lost since instances are deleted. It is however very fast to execute for large datasets.

ROS: As a very simple and effective oversampling method, Random Oversampling increases the minority sample by randomly selecting and duplicating minority instances. This method provides no "new" information to the classifier.

SMOTE: Proposed by Chawla et al. [22], this technique over-samples the minority class in a dataset by generating new "synthetic" samples. This technique adopts the k-nearest neighbor approach to over-sample the minority class instances. SMOTE is one of the most popularly used sampling method in SE studies.

ADASYN: An adaptive oversampling method proposed by He et al. [48], ADASYN focuses on the minority classes that are difficult to classify. Weights are assigned to the minority classes and dynamically adjusted so as to reduce the bias in the imbalanced data.

Borderline-SMOTE: A modification of SMOTE proposed by Han et al. [44], this method mainly focuses on the harder-to-classify minority instances found on the border-line of a classifier. SMOTE is then applied to the instances labelled as "borderline".

Safe-Level SMOTE: Originally proposed by Bunkhumpornpat et al. [20], this method aims to avoid the generation of synthetic minority samples in the region of the majority instances. Samples are labelled as "Safe-Level" before the conventional SMOTE method is applied. Safe-level is defined as the number of majority instances among the k nearest neighbor of a minority instance.

MAHAKIL: The oversampling approach based on the chromosomal theory of inheritance proposed in Chapter 7. MAHAKIL aims to generate diverse minority instances by obtaining different traits from two dissimilar instances. MAHAKIL considers two unique and dissimilar instances (i.e., based on Mahalanobis distance values) for the generation of new instances. MAHAKIL alleviates overgeneralization of prediction models since synthetic data instances created are not clustered into a specific region of the minority class of the dataset.

8.3.2 Datasets

40 releases of 20 open source projects were used for the experiments. The datasets cover two types of metrics and have an imbalanced ratio between the ranges of 3.8 - 17.46%. This is to ensure the generalization of the experiments

TABLE 8.1: Summary of selected 40 Imbalanced datasets

Project Name	Metric	Version A				Version B			
		Release	Module	# Defects	Pfp(%)	Release	Module	# Defects	Pfp(%)
Ant1	Static	1.3	125	20	16	1.4	178	40	22
Ant2	Static	1.5	293	32	10.9	1.6	351	92	26.2
Camel1	Static	1.0	339	13	3.8	1.2	608	216	35.5
Camel2	Static	1.4	872	145	17	1.6	965	188	19.5
Forrest	Static	0.7	29	5	17.2	0.8	32	2	6.2
Ivy	Static	1.4	241	16	7	2.0	352	40	11.4
Jedit	Static	4.2	367	48	13.1	4.3	492	11	2.2
Synapse	Static	1.0	157	16	10.2	1.1	222	60	27
Xalan	Static	2.4	723	110	15.2	2.5	803	387	48.2
Xerces	Static	1.2	440	71	16.1	1.3	453	69	15.2
Clam									
An-tivirus	Process	0.96	1597	93	5.82	0.97	8723	56	0.64
eCos	Process	2.0	630	110	17.46	3.0	3480	67	1.93
Helma	Process	1.3.1	312	38	12.18	1.4.0	100	17	17
NetBSD	Process	4.0	6781	548	8.08	5.0	10960	299	2.73
OpenBSD	Process	4.5	1706	275	16.12	4.6	5964	158	2.65
OpenCMS	Process	7.0.0	1727	193	11.18	8.0.0	2821	93	3.3
openNMS	Process	1.7.0	1203	123	10.22	1.8.0	1416	112	7.91
Scilab	Process	5.2.0	2636	239	9.07	5.3.0	1248	244	19.55
Spring									
Secu-rity	Process	2.0.0	926	126	13.61	3.0.0	639	132	20.66
XORP	Process	1.0	1696	158	9.32	1.1	1755	217	12.36

considering the facts that we used datasets consisting of very low and high imbalanced ratio. 10 of these projects were extracted from the PROMISE repository [115] which were collected by Jureczko and Madeyski [56] and Jureczko and Spinellis [57]. These data have been well used in several empirical studies of defect prediction [49, 84]. Each project has 20 static code features (Table 7.1) and a labeled attribute (bug) that indicates the number of bugs for each row or instance. An instance is non-defective if the bug value is 0; and defective otherwise.

The second half of the projects were manually extracted. For replication, publicly available projects were chosen. Additional factors considered for extracting the projects include longevity (more than a year of development) and the community of developers (more than five). Through a careful analysis of the commit logs and files from version control tools (CVS, SVN), we obtained the number of bugs in source code files for each project. For this study, we refer to a module as one source code file. By using keywords such as "error" and "bugs", we identify modules which are defective from their commit logs. Similarly, "Bugfixes" or "fix" keywords found in the commit logs implies that the module had been revised in the subsequent release. If files are modified for bug fixes within "X" months (timespan between version B and A) after release, we identify the files that have post-defect. For files with multiple bugs, we target all kinds of revisions, so there is no distinction between bug fixes and other

$$Recall(pd) = \frac{TP}{TP + FN} \quad (8.1)$$

$$pf = \frac{FP}{FP + TN} \quad (8.2)$$

$$g - mean = \sqrt{\frac{TP}{TP + FN} * \frac{TN}{TN + FP}} \quad (8.3)$$

$$balance(bal) = 1 - \frac{\sqrt{(0 - pf)^2 + (1 - pd)^2}}{\sqrt{2}} \quad (8.4)$$

revisions. Commits that were incomplete or did not contain the customised keywords were thus not included in this study. The releases chosen and the time period in which the metrics were extracted can be found in the Appendix B where the "Faults recorded (Date)" column in Table B.1 under Appendix B is the time period where post release bugs were recorded for each release. Refactoring for most of the projects were not performed when we extracted. As such, the value of refactorings for these projects were zero (0). During preprocessing, we excluded all metric values with zero refactorings. These projects have been used in previous studies in Chapter 3 and Chapter 5. Detailed information on how the bugs were matched and the keywords used can be found in the appendix. We extracted eight typical process metrics proposed by Moser et al. [95] and commonly used in defect prediction studies [46] from these repositories and label each module as defective or non-defective. Table 8.1 presents the projects, number of modules and buggy labels. A module is defect-prone if the number of bugs is greater than zero. For all the data set, version B is a newer version of A.

8.3.3 Evaluation Criteria

Five commonly used performance metrics are adopted for the study. The AUC, g-mean, balance, pd and pf . The first three metrics measure the performance of the overall classifier while the last two are used for assessing the performance of the minority(defective) class. Higher AUC, g-mean, balance and pd denotes better performance. Probability of false alarm (pf) measures the rate of predicted modules that were non-defective or wrongly predicted as defective. A low or zero value for pf implies a better prediction model. Detailed explanations of these measures can be found in [51]. The mathematical definitions of pd , pf , balance and g-mean measures are presented in equations 1-4.

TABLE 8.2: Predictive Models and their hyperparameter configurations

Model	Overview	Hyperparameters
C4.5	J48 Decision Tree	$c = \{0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7\}$
NNET	3-layer Neural Network	$\text{size} = \{4, \dots, 28\}, \text{decay} = \{0.1, 0.2\}$
SVM	Support Vector Machine	$c = \{2^{-6}, \dots, 2^{10}\}$
RF	Random Forest	$\text{mtry} = \{10, 50, 100, 200, 250, 500, 1000\}$
KNN	K-Nearest Neighbor	$c = \{2^*(0, \dots, 7) + 1\}$

8.3.4 Experimental Setup

The experiments reports the performance of the defect prediction models when 7 different sampling methods are applied to each project data at different *Pfp* rates (default, 20, 30, 40, 50). This results in five independent training dataset (resampled data) per each sampling method and 350 ($5 \text{ } Pfp * 10 \text{ datasets} * 7 \text{ sampling methods}$) cross-release predictions in total. We represent the default distribution value as No Sampling (NONE) in our experiments. In agreements with the assumptions of classifiers that the dataset should be balanced, we stop resampling at 50%. Also, resampling above 50% could totally deplete the dataset for RUS. Five commonly used prediction models in defect prediction [75, 81] are selected for the experiment. They are presented in Table 8.2 together with the parameter configurations used. The experiments were executed with the open source statistical processing tool (R) [129]. The Caret [72] and WRS [140] library packages were used for the prediction model construction and statistical analyses respectively. According the recommendations of Menzies et al. [88], all the features (metrics) of the datasets are used during the model training. We strictly apply the parameters used by the original authors of all synthetic oversampling methods except the distribution values.

During the model construction, the 10-fold cross-validation method is applied to each resampled data. This method ensures that for a particular model training, the dataset is split into 10 folds where 90% of the dataset is used for training and the remaining 10% is used for testing. It is iterated until all folds in the dataset have been used once as a test dataset. The constructed model is tested on the separate test dataset (version B) and evaluated using the performance measures discussed in Section 8.3.3.

8.3.5 Statistical Performance Comparison

The *win-tie-loss* evaluation procedure mostly used in several Software effort estimation studies [66, 68] is adopted for the performance comparison. This procedure employs a statistical test for a pairwise comparison between predictors. Brunner's statistical test [17] is applied for significant differences since this test was recently recommended by Kitchenham et al. [64] as a more robust non-parametric test. Three counters *wins*, *ties* and *losses* are initially set for each predictor. For any two pairs of selected predictors, the statistical test is performed across all datasets for each performance measure and the *win*, *tie* or *loss* value is reported based on the significance value ($p < 0.05$). If the performance distributions of 2 predictors are not significantly different, the *ties* counters of both predictors are increased by 1; otherwise the *wins* counter of the significant predictor is increased by 1 and the *losses* counter of the other predictor also increased by 1. Figure 8.1 presents the *win-tie-loss* procedure used.

```

wink = 0, tiek = 0, lossk = 0
winl = 0, tiel = 0, lossl = 0
if Brunner-test(Perfk, Perfl, 0.95) says they are the same then
    tiek = tiek + 1
    tiel = tiel + 1
else
    if better(Perfk, Perfl) then
        wink = wink + 1
        lossl = lossl + 1
    else
        winl = winl + 1
        lossk = lossk + 1
    end if
end if

```

FIGURE 8.1: Pseudocode for win-tie-loss computation between predictors k and l with performance measure values $Perf_k$ and $Perf_l$. Note that pf values are negated before comparison since the lower the pf value, the better.

8.4 Case Study Results and Discussions

8.4.1 RQ1: How does tweaking the Pfp parameter impact performance of sampling approaches on defect prediction models?

To answer RQ1, we first compute the correlation between the amount of resampling (Pfp) and prediction performance for each sampling method. Kendall's Tau correlation test between Pfp and prediction performance is computed. The results are presented in Table 8.3. For the AUC performance measure, very low and insignificant correlation values less than 0.15% was observed overall across all prediction models indicating that all the sampling methods were not affected by the Pfp rate. On a rare occasion did the Pfp have a negative and significant correlation with RUS for the RF prediction model. Positive and significant correlations between the Pfp rate and the remaining performance measures are observed across almost all sampling methods and prediction models. However, a positive and significant correlation for pf measure indicates that sampling methods do increase the probability of false alarms. This is undesirable since the lower the false alarms, the better the performance. The C4.5 and SVM prediction models are most resistant to Pfp rates and are not affected by Pfp changes.

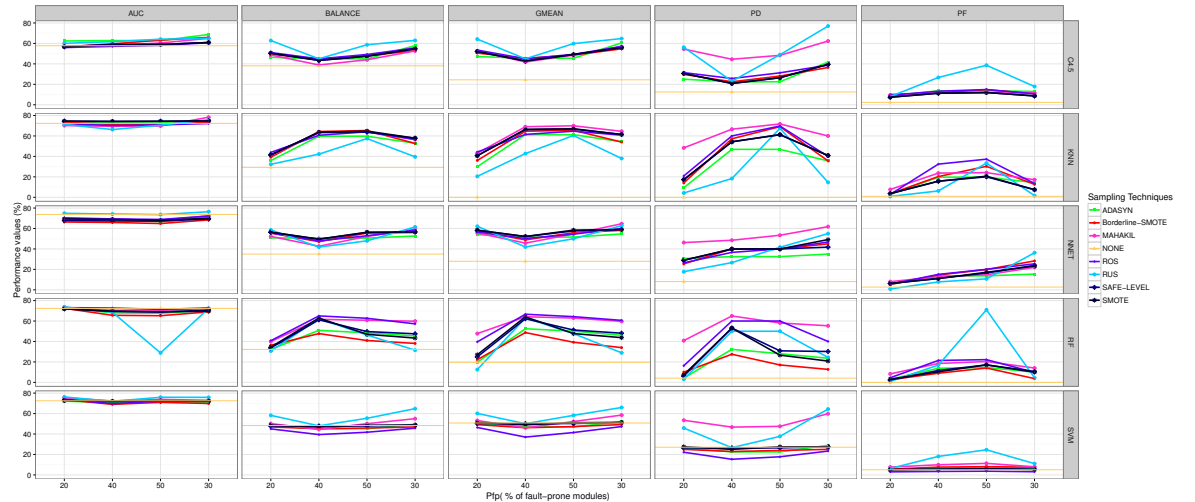


FIGURE 8.2: Performance plots (median values) for all performance measures of the sampling techniques on different Prediction Models at different Pfp values across all ten cross-release static metrics datasets

We further evaluate the effects of changing the Pfp on the prediction performance of seven sampling methods. The performance of each prediction model when sampling methods are applied are compared to the prediction model with

TABLE 8.3: Kendall's Tau rank correlation analysis between Pfp and each prediction performance measure per each prediction model and metric-type dataset. Significant correlation results (p-value<0.05) are highlighted in gray.

Perf	Model	Static Metric Datasets					
		ADASYN	BSMOTE	ROS	RUS	SAFE	SMOTE
AUC	c4.5	-0.06	0.08	0.01	0.13	-0.01	-0.01
	knn	-0.03	-0.07	-0.05	-0.2	0	0
	nnet	-0.11	-0.16	-0.06	0.08	-0.09	-0.04
	svm	-0.04	-0.05	-0.15	0.15	0.01	0.02
	rf	-0.13	-0.21	0.04	-0.35	-0.02	0.02
balance	c4.5	-0.02	-0.06	0	0.22	0.04	0.04
	knn	0.38	0.45	0.36	0.51	0.35	0.37
	nnet	0.13	0.21	0.3	0.39	0.17	0.21
	svm	-0.03	0	0.03	0.4	-0.02	0.01
	rf	0.27	0.06	0.41	-0.04	0.13	0.18
g-mean	c4.5	-0.04	-0.06	-0.01	0.21	0.02	0.02
	knn	0.38	0.41	0.3	0.51	0.32	0.32
	nnet	0.11	0.19	0.27	0.37	0.16	0.2
	svm	-0.03	-0.02	0.02	0.37	-0.03	0.05
	rf	0.25	0.04	0.39	0.04	0.13	0.16
pd	c4.5	-0.01	-0.06	0.05	0.41	0.05	0.05
	knn	0.4	0.56	0.49	0.55	0.45	0.45
	nnet	0.15	0.28	0.36	0.47	0.27	0.29
	svm	-0.03	-0.01	0.02	0.43	0	0
	rf	0.3	0.11	0.57	0.34	0.24	0.25
pf	c4.5	0.14	-0.04	0.19	0.44	0.08	0.08
	knn	0.49	0.73	0.64	0.48	0.38	0.38
	nnet	0.15	0.49	0.42	0.51	0.34	0.33
	svm	0.05	0.07	0.01	0.51	0.05	0.01
	rf	0.4	0.17	0.59	0.65	0.3	0.27
Process Metric Datasets							
AUC	c4.5	-0.19	-0.05	-0.03	-0.06	0.12	0.12
	knn	0.03	-0.1	0.01	-0.12	-0.03	0.02
	nnet	-0.02	-0.03	0.07	-0.09	0.04	0.02
	svm	0.03	-0.06	0.01	-0.11	-0.06	-0.03
	rf	0.13	0.15	0.17	-0.03	0.16	0.15
balance	c4.5	0.04	-0.05	-0.01	0.02	0.14	0.13
	knn	0.13	0.17	0.13	0.13	0.26	0.22
	nnet	0.2	0.33	0.45	0.11	0.37	0.28
	svm	0.12	0.15	0.01	0.03	0.07	0.04
	rf	0.11	0.39	0.25	0.11	0.14	0.11
g-mean	c4.5	0.03	-0.06	-0.02	0.03	0.13	0.13
	knn	0.13	0.15	0.13	0.12	0.24	0.2
	nnet	0.24	0.34	0.47	0.12	0.39	0.31
	svm	0.12	0.14	0.02	0.02	0.07	0.04
	rf	0.1	0.39	0.24	0.1	0.15	0.11
pd	c4.5	0.17	0.1	0.07	0.19	0.14	0.13
	knn	0.24	0.44	0.46	0.47	0.4	0.29
	nnet	0.2	0.44	0.47	0.37	0.46	0.32
	svm	0.17	0.19	0.02	0.32	0.08	0.03
	rf	0.19	0.5	0.39	0.28	0.29	0.22
pf	c4.5	0.13	0.13	0.15	0.18	0.18	0.15
	knn	0.31	0.52	0.53	0.4	0.45	0.33
	nnet	0.16	0.49	0.45	0.31	0.41	0.33
	svm	0.12	0.13	-0.01	0.29	0.09	0.05
	rf	0.15	0.49	0.42	0.31	0.3	0.27

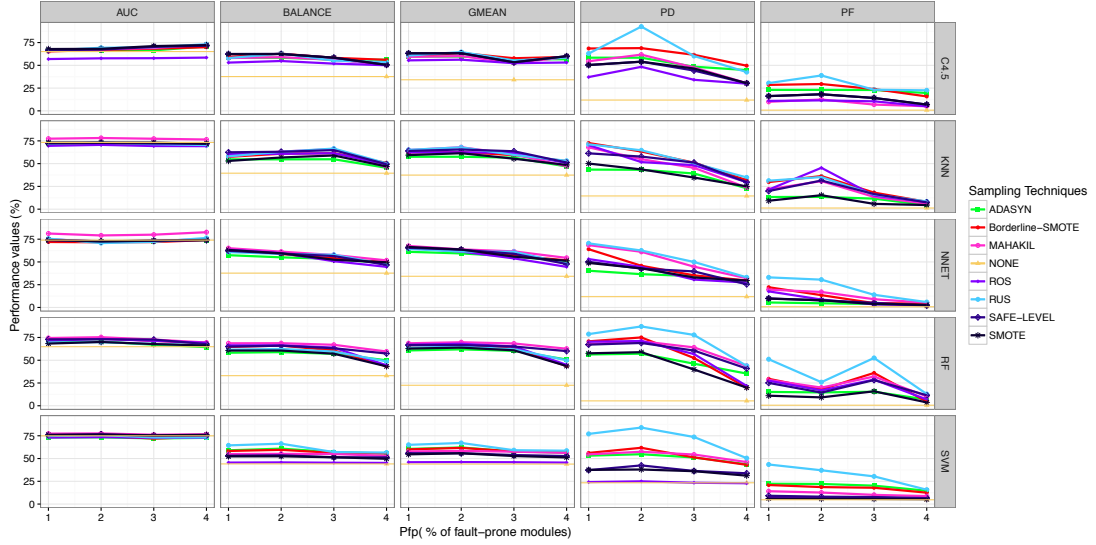


FIGURE 8.3: Performance plots (median values) for all performance measures of the sampling techniques on different Prediction Models at different Pfp values across all ten cross-release process metrics datasets

no sampling (default Pfp) across the Pfp values. Figures 8.2 and 8.3 presents plots of median performance values for each sampling method and prediction model across all static-metric and process-metric datasets respectively. By using the NONE as the benchmark, we observe change in performances as Pfp increases. AUC performance values are not significantly improved as Pfp increases. The AUC values for all sampling methods were almost the same as NONE. All other performance values improved with the exception of pf as Pfp increases for all the sampling methods. For both dataset metric type, it is obvious sampling methods perform better compared to no sampling (NONE) across all prediction models with regards to the *balance*, *g-mean* and *pd* performance values. However, *pf* performance values are worse when sampling methods are applied. For the AUC performance measure, performance values seem to be no different when sampling methods are used. The *g-mean* and *balance* performance values for most sampling methods increases as Pfp increases. The same applies for the *pd* performance measure where the RUS method increases at an increasing rate as Pfp increases across all prediction models in comparison to the other sampling methods. RUS performs better in terms of *pd* because the non-defective samples consistently reduces as Pfp is increasing. However, this is not desirable because in as much as we want to attain high *pd* especially on the defective modules, we also do not want to lose crucial information. It is however not a surprise that as *pd* increase, *pf* also increases implying RUS should be used with extra caution as a sampling method in software defect prediction research.

8.4.2 RQ2: To what degree (percentage of fault-prone modules Pfp) should one resample or balance the original data set?

To find the most appropriate Pfp rate to use per each sampling method, dot plots are presented to show the dominant Pfp rate that resulted in the highest performance across all datasets. More specifically, for each prediction model and sampling method, we count the number of wins (i.e., highest AUC, g -mean, $balance$, pd and lowest pf values) attained by each Pfp rate across all datasets. The results across all prediction models per each performance metric for the datasets are displayed in Figures 8.5 and 8.4. The best Pfp rates differed per the performance measure. For the AUC performance measure, the highest values are attained at a Pfp rate of 20% with ROS and SMOTE methods being the only exception where highest performance values were obtained consistently at a Pfp rate of 50%. There were no clear and consistent patterns for the $balance$ and g -mean performance measures with the best Pfp alternating between 40%-50% depending on the prediction model and sampling method. Across g -mean and $balance$ performance measures, the best Pfp for SMOTE and the variants (Borderline and Safe-level) was 50%. ADASYN however achieved more better performance for these measures when Pfp was 40%. These Pfp rates are however not stable since they were very close to the other Pfp rates. The gaps between the best Pfp rates and the next closest Pfp rates were too narrow and thus insignificant. For pd and pf performance measures, we observe more stable Pfp rates and consistent patterns for all sampling methods. A Pfp rate of 20% resulted in the lowest pf value across all sampling methods. In contrast, a high Pfp rate results in high pd values. A single Pfp rate could not be attained by ADASYN with the best ranging between 40% to 50%.

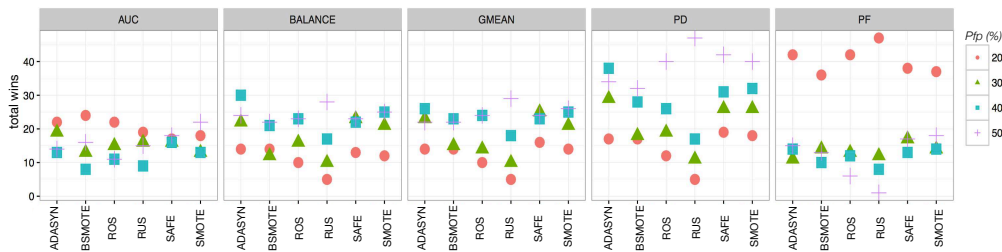


FIGURE 8.4: Total counts of wins of each Pfp rate for each Sampling method across all ten static metrics datasets and all prediction models per each performance metric

From the analysis and from the analysis of MAHAKIL in chapter 7, a stable Pfp rate of 20% was observed for all sampling methods when evaluated with

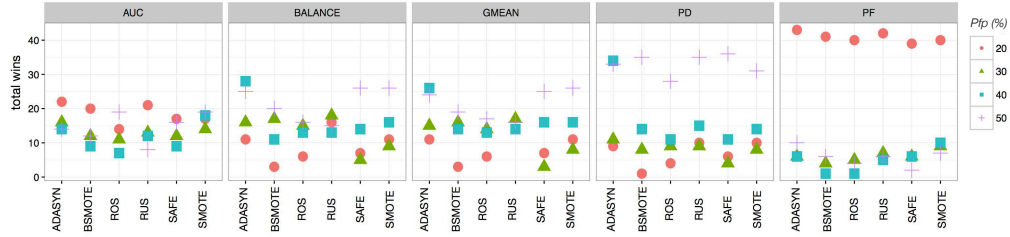


FIGURE 8.5: Total counts of wins of each Pfp rate for each Sampling method across all ten process metrics datasets and all prediction models per each performance metric

the pf performance measure across all prediction models. Similarly, with regards to pd , a stable Pfp rate of 50% was observed for most sampling methods. However, a stable Pfp rate could not be achieved for the sampling methods per the other three performance measures (AUC, g-mean, balance). It is therefore better to conduct several experiments with different Pfp and select the Pfp rate producing the highest performance value per each sampling method for these three performance measures.

8.4.3 RQ3: Which are the high-performing resampling approaches for defect prediction?

RQ3 aims to assess the performance of each sampling method on prediction models and determine the most effective one for defect prediction studies. To find the best sampling method, statistical comparisons are made per each performance measure using the Pfp rate that produced the highest performance for each sampling method and prediction model across the individual datasets using the win-tie-loss statistic procedure explained in Figure 8.1.

For each Pfp rate, comparisons are made among the five performance values of each sampling method and prediction model for every single dataset. We present the win-tie-loss results where each predictor is compared with 39 (5 models * 8 sampling methods - 1) other predictors for each Pfp rate in Table 8.4. We focus on the wins and losses of each predictor and thus exclude the tie values from the results. From the table, the best ranked predictors differed at each Pfp rate. A summary of the results from the tables are presented as follows:

1. Across all Pfp rates, combination of no sampling method (NONE) with all the five prediction models were consistently ranked in the bottom half

TABLE 8.4: Performance in terms of wins, losses and wins-losses aggregated from all five performance measures across all datasets. Higher wins and wins-losses indicate better performance, whereas lower losses also indicate better performance. The ranks are ordered by wins-losses

<i>Pfp</i> 20					<i>Pfp</i> 30					<i>Pfp</i> 40					<i>Pfp</i> 50				
Model	Sampling	Wins	Losses	Wins-Losses	Model	Sampling	Wins	Losses	Wins-Losses	Model	Sampling	Wins	Losses	Wins-Losses	Model	Sampling	Wins	Losses	Wins-Losses
RF	MAHAKIL	51	8	43	KNN	SAFE-LEVEL	64	6	58	KNN	SAFE-LEVEL	96	14	82	KNN	MAHAKIL	117	23	94
RF	ADASYN	46	9	37	KNN	SMOTE	64	6	58	KNN	SMOTE	96	14	82	RF	RUS	99	22	77
RF	BORDERLINE	46	9	37	SVM	MAHAKIL	66	15	51	KNN	MAHAKIL	83	23	60	KNN	SMOTE	92	17	75
SVM	MAHAKIL	49	12	37	RF	RUS	59	15	44	RF	RUS	75	15	60	KNN	SAFE-LEVEL	91	17	74
RF	SAFE-LEVEL	46	10	36	KNN	MAHAKIL	69	31	38	KNN	BORDERLINE	73	26	47	SVM	MAHAKIL	75	20	55
RF	NONE	46	10	36	RF	ADASYN	36	7	29	SVM	MAHAKIL	68	23	45	KNN	BORDERLINE	83	31	52
NNET	MAHAKIL	38	6	32	KNN	ADASYN	40	17	23	KNN	ADASYN	65	21	44	SVM	ROS	70	23	47
RF	RUS	41	10	31	NNET	RUS	30	7	23	SVM	ROS	62	21	41	KNN	ADASYN	62	23	39
KNN	MAHAKIL	42	12	30	RF	SAFE-LEVEL	37	14	23	RF	MAHAKIL	49	12	37	NNET	SMOTE	64	25	39
RF	SMOTE	40	10	30	RF	MAHAKIL	38	15	23	NNET	RUS	35	6	29	NNET	MAHAKIL	59	26	33
KNN	SAFE-LEVEL	36	10	26	RF	NONE	45	24	21	KNN	ROS	65	40	25	NNET	RUS	55	23	32
KNN	SMOTE	36	10	26	SVM	ROS	30	9	21	C45	RUS	45	24	21	KNN	ROS	68	37	31
NNET	RUS	34	8	26	KNN	ROS	40	20	20	RF	SMOTE	43	32	11	RF	MAHAKIL	56	26	30
NNET	ADASYN	26	12	14	RF	BORDERLINE	32	15	17	SVM	SAFE-LEVEL	27	16	11	C45	RUS	60	32	28
KNN	BORDERLINE	19	9	10	SVM	RUS	26	9	17	SVM	SMOTE	28	17	11	NNET	SAFE-LEVEL	56	29	27
KNN	ROS	22	13	9	NNET	MAHAKIL	41	26	15	RF	ADASYN	40	30	10	KNN	RUS	47	27	20
NNET	ROS	20	11	9	RF	SMOTE	27	13	14	RF	SAFE-LEVEL	44	34	10	NNET	ROS	50	38	12
NNET	BORDERLINE	21	13	8	NNET	SAFE-LEVEL	26	15	11	NNET	SMOTE	38	30	8	C45	MAHAKIL	47	45	2
SVM	ROS	14	12	2	SVM	SAFE-LEVEL	22	11	11	RF	NONE	49	46	3	RF	BORDERLINE	41	41	0
NNET	SAFE-LEVEL	16	15	1	NNET	ADASYN	20	11	9	SVM	RUS	16	14	2	RF	SMOTE	46	52	-6
NNET	SMOTE	16	16	0	C45	RUS	30	22	8	NNET	ADASYN	22	22	0	SVM	SAFE-LEVEL	20	28	-8
SVM	RUS	41	43	-2	NNET	SMOTE	22	14	8	NNET	MAHAKIL	38	40	-2	RF	NONE	48	57	-9
RF	ROS	24	29	-5	NNET	ROS	20	18	2	RF	ROS	38	42	-4	SVM	SMOTE	20	30	-10
C45	ADASYN	18	34	-16	NNET	BORDERLINE	21	20	1	SVM	ADASYN	25	31	-6	RF	ADASYN	43	54	-11
SVM	SMOTE	26	42	-16	RF	ROS	26	25	1	RF	BORDERLINE	36	44	-8	RF	SAFE-LEVEL	44	55	-11
SVM	SAFE-LEVEL	26	44	-18	SVM	SMOTE	21	20	1	NNET	BORDERLINE	21	33	-12	NNET	BORDERLINE	47	61	-14
C45	RUS	24	44	-20	KNN	BORDERLINE	26	29	-3	NNET	ROS	22	34	-12	NNET	ADASYN	27	48	-21
SVM	NONE	45	65	-20	C45	ADASYN	21	31	-10	NNET	SAFE-LEVEL	22	36	-14	SVM	ADASYN	33	56	-23
SVM	BORDERLINE	16	37	-21	C45	MAHAKIL	31	48	-17	C45	MAHAKIL	37	58	-21	RF	ROS	37	66	-29
C45	BORDERLINE	36	58	-22	SVM	ADASYN	18	36	-18	C45	BORDERLINE	20	42	-22	C45	SAFE-LEVEL	24	57	-33
C45	ROS	35	59	-24	C45	SAFE-LEVEL	14	39	-25	SVM	BORDERLINE	15	51	-36	C45	SMOTE	24	57	-33
C45	SAFE-LEVEL	21	46	-25	C45	SMOTE	14	39	-25	KNN	RUS	32	69	-37	C45	ROS	22	72	-50
C45	SMOTE	21	46	-25	C45	BORDERLINE	17	49	-32	C45	ADASYN	21	63	-42	C45	BORDERLINE	23	74	-51
C45	MAHAKIL	38	66	-28	C45	RUS	12	59	-47	C45	NONE	27	75	-48	C45	ADASYN	31	83	-52
C45	NONE	8	36	-28	C45	NONE	19	68	-49	SVM	NONE	48	101	-53	C45	NONE	27	80	-53
KNN	ADASYN	19	48	-29	KNN	RUS	37	91	-54	C45	ROS	13	77	-64	SVM	NONE	46	99	-53
SVM	ADASYN	21	55	-34	SVM	NONE	45	103	-58	KNN	NONE	50	115	-65	NNET	NONE	36	92	-56
KNN	RUS	42	77	-35	NNET	NONE	27	96	-69	NNET	NONE	33	98	-65	KNN	NONE	47	113	-66
NNET	NONE	18	62	-44	SVM	BORDERLINE	14	83	-69	C45	SAFE-LEVEL	13	79	-66	SVM	BORDERLINE	16	83	-67
KNN	NONE	42	112	-70	KNN	NONE	47	118	-71	C45	SMOTE	13	79	-66	SVM	RUS	15	128	-113

of the table, suggesting that prediction models perform worse when no sampling (NONE) methods are applied to imbalanced datasets.

2. A stable predictor could not be observed across all four *Pfp* rates. The best predictor varied across the *Pfp* rates.
3. Sampling methods significantly improved the KNN prediction model whilst the C4.5 prediction model is the least impacted by sampling methods.

The win-tie-loss results indicate sampling methods are more effective in improving performance of prediction models. However, the best sampling method to use differs with respect to the *Pfp* rate applied and evaluation metric. It is thus difficult to find a stable *Pfp* rate to use for defect prediction research and we recommend using different predictors based on the evaluation metric used. For each dataset and sampling method, several experiments should be conducted with different *Pfp* values where (default $>Pfp \leq 50$). The *Pfp* value resulting in the highest performance should be used. Our results is in agreement with the observations made by Bowes et al. [13] that there are uncertainties in the predictions made by classifiers. The instability in the prediction performances among the classifiers caused by the *Pfp* rates provides practical implications for practitioners. In safety critical scenarios where false alarms are costly, *Pfp* rates

similar to the default distribution rate of the dataset is recommended for most sampling methods. From our results, we recommend KNN prediction model in combination with MAHAKIL or Safe-level SMOTE since it performed very well statistically irrespective of the Pfp rate across all performance measures. The results suggest that the number of minority samples introduced into the training data has significant effect on prediction performance.

8.5 Threats to Validity

As an empirical study, there are several potential limitations. We considered only 10 open-source projects with static metrics for the experiments. The results may not generalize for commercial projects and a replication study using commercial projects is left for a future study. The default imbalance ratio and project sizes span from low to high values. These imbalance values were considered so as to reduce any bias of specific datasets dominating and influencing the results. The effect of sampling methods on very large project sizes will be explored in a future work. We carefully selected a large number of performance measures and applied the robust statistical test recommended by Kitchenham et al. [64] for the experiment. The threat to statistical conclusion is thus limited in our study. We considered a limited number of prediction models. These models are however widely used in defect prediction studies. We acknowledge different prediction models could have different implications. The configuration parameters chosen for the prediction models could also have an impact on the prediction results.

8.6 Chapter Summary

Sampling methods applied to training datasets prior to model construction are controlled by configurable parameters. Exploring all the accessible parameter for the various sampling methods is however impractical. Recent studies conclude that performance of prediction models are influenced by the imbalance ratio of the training datasets. This imbalance ratio (Pfp) can be controlled when sampling methods are applied. To assess the effect of Pfp on sampling and prediction performance, we examine the statistical significant performance of seven sampling methods on five prediction models across 20 releases of 10 open source projects. Sampling methods had a significant and positive influence on prediction models across all Pfp values. AUC values were not significantly improved by sampling methods whilst sampling methods significantly increases

the false alarms (pf) as Pfp increased. The experimental results indicate that Pfp has significant effect on prediction performance. This suggest that not only is the sampling method important but the Pfp selected is important as well. It is however difficult to find a stable Pfp value to use for defect prediction studies. This depends on several factors with the default imbalance ratio of the dataset being the major factor. We suggest researchers apply different Pfp values and select the rate that produces the best performance.

8.6.1 Concluding Remarks

Sampling methods are known to impact defect prediction performance. These sampling methods have configurable parameters that can significantly affect the prediction performance. It is however, impractical to assess the effect of all the possible different settings in the parameter space for all the several existing sampling methods. A constant and easy to tweak parameter present in all sampling methods is the distribution of the defective and non-defective modules in the dataset known as Pfp (% of fault-prone modules). In this chapter, we investigate and assess the performance of defect prediction models where the Pfp parameter of sampling methods are tweaked. An empirical experiment and assessment of seven sampling methods on five prediction models over 20 releases of 10 static metric projects indicate that (1) Area Under the Receiver Operating Characteristics Curve (AUC) performance is not improved after tweaking the Pfp parameter, (2) pf (false alarms) performance degrades as the Pfp is increased. (3) a stable predictor is difficult to achieve across different Pfp rates. Hence, we conclude that the Pfp parameter setting can have a large impact on the performance (except AUC) of defect prediction models. We thus recommend researchers experiment with the Pfp parameter of the sampling method since the distribution of training datasets vary.

Chapter 9

Conclusion

9.1 Introduction

Software defect data sets are typically characterized by an unbalanced class distribution where the defective modules are fewer than the non-defective modules. Prediction performances of defect prediction models are detrimentally affected by the skewed distribution of the faulty minority modules in the data set since most algorithms assume both classes in the data set to be equally balanced. Resampling approaches address this concern by modifying the class distribution to balance the minority and majority class distribution. However, very little is known about the best distribution for attaining high performance especially in a more practical scenario. There are still inconclusive results pertaining to the suitable ratio of defect and clean instances (Pfp), the statistical and practical impacts of resampling approaches on prediction performance and the more stable resampling approach across several performance measures. The impact of data resampling methods on prediction performance has not been studied in detail considering robust statistical tests. This thesis examines the significant and practical impacts of resampling methods on prediction models, the best resampling rate (Pfp) that should be used per each resampling method and the evaluation measure adopted and the most stable and high performing resampling method suitable for software defect prediction studies. Furthermore, this thesis proposes a novel data sampling approach to tackle the main drawback of existing sampling approaches. The main findings and contributions of this thesis are discussed in details in the sections below.

9.2 Research Findings and Implications

From the studies conducted in this thesis, we conclude that resampling methods do not improve AUC performance values since there were no practical significance difference between resampling methods and no sampling method. The

results is in agreement with findings observed by Shatnawi [118] that sampling methods do not improve the AUC performance of prediction models. Also, no sampling (NONE) significantly outperformed sampling methods when evaluated with the pf performance measure. Sampling methods significantly improved the performance (g -mean, pd , $balance$) of all prediction models. The improvements are however accompanied with a high cost of false alarms (pf). With regards to the best and more stable Pfp rate, the experimental results indicate that it is difficult to attain a stable Pfp to be used during sampling for the AUC measure. Sampling with a Pfp rate of 50% produces higher pd values whereas a Pfp rate of 20% produces less pf values. From our stability ranking experiments in Chapters 7 and 8, MAHAKIL sampling method attained the highest rank across four out of five performance measures. Additionally, the Borderline-SMOTE outperformed the other oversampling methods. By conducting a comprehensive empirical experiments with more performance measures and robust statistical tests, we discuss in detail the overall findings from our analysis.

Finding 1: *The Pfp rate applied to resampling approaches significantly impacts performance of prediction models.* There was a positive relationship between Pfp and g -mean, $balance$ and pd performance measures. A negative relationship was observed between Pfp and pf . This implies prediction performance is affected by the amount of minority samples. Most machine learning algorithms have been intuitively made to favor an equally balanced dataset for the best prediction performance. The effect of Pfp on prediction performance is related to the information present in the dataset since the increase of minority instances tend to increase the information within the minority instances. Prediction models are known to work well when the amount of information present in the dataset is sufficient. Class imbalance due to rare minority instances or small sample size makes it difficult for learning algorithms of classifiers to generalize inductive rules [47]. Menzies et al. [91] noted that the information content within the training data is improved when sampling methods are applied and we confirm their assertion based on our experimental results. Oversampling methods aim to increase the defective information within a dataset. In contrast, undersampling methods aim to reduce non-defective information present in a dataset. Accordingly, both type of sampling methods ensure that the defective modules dominate which subsequently shifts the decision boundary of the classifier towards the region of the minority (defective) class. Synthetic oversampling methods introduce new minority instances into the dataset that provides new information to the classifier. Interestingly, simple methods such as ROS and RUS outperformed more complex and synthetic generating methods.

Both methods do not provide "new" information to the classifier. However, they modify the distribution of the data which tends to positively improve performance of classifiers. ROS which tend to randomly duplicate the minority instances could duplicate instances which are very close to the decision boundary. This automatically shifts the decision boundary of any classifier trained on the dataset hence increasing the performance on the classifier on the minority instances. Similarly, RUS could randomly delete more majority instances which are close to the decision boundary thus shifting the bias of the decision boundary towards the minority instances.

Implications: *Pfp* parameter setting does have a large impact on the performance (except AUC) of defect prediction models. Researchers should experiment with the *Pfp* parameter of the sampling method until the highest performance is obtained since the distribution of training datasets vary.

Finding 2: *Intrinsic algorithm make-up of sampling approaches impacts prediction performance.* Considering the oversampling methods, the assessment based on the *win-tie-loss* reveals that the strategy adopted by the sampling methods in generating new samples does impact prediction performance. Specifically, the individual minority data instances selected for oversampling can significantly improve prediction performance. The Borderline-SMOTE sampling method proved to be more stable across all performance measures. This method consistently ranked in the top three sampling methods for all performance measures except the *pf* according to the win-tie-loss statistical analysis. Additionally, the Safe-Level SMOTE outperformed the original SMOTE method. As an upgrade on SMOTE, both Borderline and Safe-Level SMOTE make significant modifications to the original SMOTE sampling method. These methods specifically focus on subsets of minority instances and applies the SMOTE method to generate more samples.

Implications: This finding suggest the importance of analyzing the minority instances and selecting the more relevant instances that will generate informative instances and thus provide more information to the classifier.

Finding 3: *Sampling approaches increase false alarms (pf).* Sampling methods significantly increased the false alarms with RUS and ADASYN being the worse performing sampling methods. RUS method deletes majority or non-defective modules as *Pfp* increases thus information on the majority modules provided to the training model is limited or even non-existent. RUS does not provide any "new" minority instance information to classifiers similar to ROS which provides duplicated information on the minority instances. RUS and ROS make the samples limited in information leading to the rules formed by

the learning algorithms to be too specific. This causes overfitting as reported by Phung et al. [108] and thus increases the probability of false alarms. With a significant loss of information, the trained classifier lacks information on the non-defective modules and will accordingly classify non-defective modules as defective. For the synthetic oversampling methods, the high false alarms could be attributed to the intrinsic characteristics of the algorithms specifically the selection of parent modules used for new samples generation and positioning of the new samples generated. Most of the algorithms of synthetic oversampling methods focus on modules (parents) very close to decision boundary for new modules generation. Consequently, the positioning or placement of the new samples generated are usually ignored by these algorithms and these samples are randomly positioned based on the parent modules. As such, modules could be accidentally positioned in the region of the non-defective modules. Additionally, based on the parent selection criteria of these algorithms, if the minority class is intra-class imbalanced, that is, should there exist several or very similar modules within a specific cluster of the defective modules, the newly generated modules will most likely be a duplicate and will provide no new information to the prediction model. The proposed MAHAKIL approach considers this issue and thus performs better than the other synthetic based oversampling approaches.

Implications: Sampling approaches should not aim at basically making the minority samples dominate. Oversampling methods should consider the relevance of the new minority data being introduced into the data. This includes the selection of neighbors and parents instances used to generate the new data and the position or placement of the new data instance. Additionally, noise or outliers are inherent with defect prediction datasets [40]. To significantly reduce the false alarm rates, sampling methods should consider noise detection and elimination techniques to ensure only relevant and important minority instances remain or are added to the training data.

Finding 4: *Sampling approaches are not beneficial for defect prioritization.* Software quality teams are mostly interested in prediction models that can aid in the efficient allocation of scarce testing resources. Prioritization of defective modules is thus necessary. The use of the area under the ROC curve (AUC) to assess the performance of prediction models does help in prioritizing the top-n defective modules which require more attention. The AUC performance values after applying sampling methods were insignificant irrespective of prediction model or dataset type. The empirical assessment based on win-tie-loss and effect sizes does reveal that sampling methods cannot be used for defect prioritization.

The insignificant effect sizes computed from the Cliff's method with Hochberg's method for prediction models trained on resampled datasets demonstrate that sampling methods cannot improve the prediction of defect-proneness itself.

Implications: Due to the high and significant pf values, data resampling methods should be not considered for defect prioritization activities.

Finding 5: *Sampling approaches are useful for defect classification:* The significant and practical results observed for the performance measures computed from confusion matrix (g -mean, $balance$, pd and pf) indicate the importance of sampling methods for defect classification. When software quality teams want to consider all defective modules for testing, sampling methods could be of great help. With very large effect sizes observed for the g -mean, $balance$ and pd values, we have demonstrated that sampling methods can find all defective modules. The sampling methods considered in this study explicitly set the threshold between the defective and non-defective modules. However, extra cost will be incurred in testing modules which may not be defective (false alarms) since significant pf values were also observed.

Implications: Existing sampling approaches can properly set the threshold between buggy and clean groups. Practitioners can adopt sampling approaches for defect classification purposes.

9.3 Research Contributions

The contributions resulting from the studies in this thesis is listed in this section.

1. Provide better clarifications on the impact of data sampling approaches on the performance of defect prediction models. In this thesis, we conducted several empirical experiments across several datasets and performance measures using robust statistical tests to validate the benefits and drawbacks of most commonly used data sampling approaches. We explain in detail how and why some sampling methods perform better than others
2. Conducted an in-depth analysis on the best rate of sampling (Pfp) to use for sampling methods. We find that a Pfp of 20% results in the best pf values and a Pfp of 50% or higher results in best recall(pd) values. For other performance values, it was still a challenge to obtain specific values. AUC performance values were the most robust and indicate that future research should evaluate sampling methods using AUC performance measure.

3. We highlight the possible shortcomings of current data sampling approaches. Although we investigated only seven data sampling approaches in this thesis, we notice that the strategy adopted by these approaches affect performance. Most of these approaches increase the false alarms(pf) as they improve pd values. This leads us to our next contribution.
4. We provide a novel oversampling approach called MAHAKIL which significantly outperformed all the oversampling approaches considered in this thesis. To facilitate learning from class imbalanced datasets, practitioners can adopt our novel robust and diversity-based oversampling approach called MAHAKIL which improves the performance of the prediction models by generating new and diverse synthetic data and strengthening the decision boundary of the minority class.
5. Provide practical recommendations and guidelines on which data resampling approach to use or avoid for software quality teams and researchers.

9.4 Future Research

The research reported in the thesis provided answers for several questions regarding the impacts and practical benefits of applying data sampling approaches during the training of software prediction models. Nonetheless, this research has brought to mind several other questions and provided several avenues for future work. These future research avenues could possibly address some of these questions.

We present a summary of the likely future research areas regarding our approach. The development of a robust diversity based data sampling approach could be extended and improved for other research areas of defect prediction.

9.4.1 More Analysis of sampling approaches

The study in this thesis considered as much as seven sampling approaches and five prediction models for the impact analysis. For a very large comprehensive study, empirical investigations using more simple and complex sampling methods and analysis of the effect on the intra- and inter- release prediction performances using more prediction models and other metric based datasets could be conducted.

9.4.2 Improving the MAHAKIL oversampling approach

In the domain of cross project or company prediction, where historical datasets are not readily available for training by new projects and thus acquires projects of similar characteristics and domains for training classifiers, MAHAKIL could prove to be of great value when appropriately applied with similar projects to generate synthetic data, which is close to the new project under consideration. This could be applied after using state of the art techniques to filter and select the appropriate datasets for the new project. This technique could also be used to generate both majority and minority classes of datasets for cross project prediction. It would be interesting to investigate this area further. To significantly reduce the pf , our approach could be modified to identify multi-clusters and work in local patches. The MAHAKIL algorithm could thus be applied to work within each cluster of minority instances found after applying clustering algorithms.

Moreover, several aspects of the theory of inheritance were not considered in the proposed method, such as fitness evaluation, mutation and recombination. Future research could leverage other existing information of the inheritance theory to significantly improve the method. We will also consider comparing the performance of MAHAKIL to that of the ensemble models and other complex sampling techniques.

9.4.3 Considering the impact of other parameters of sampling approaches

This thesis considered the impacts of data sampling approaches and the effect of the resampling rate parameter(Pfp). Parameters such as the number of nearest-neighbors, the considered data samples for undersampling or oversampling and others could be examined and their impacts on prediction performance evaluated.

Bibliography

- [1] Amritanshu Agrawal and Tim Menzies. " " Better Data" is Better than " Better Data Miners"(Benefits of Tuning SMOTE for Defect Prediction)". In: *arXiv preprint arXiv:1705.03697* (2017).
- [2] Tatu Aida. "On the inheritance of color in a fresh-water fish, *Aplocheilichthys latipes* Temmick and Schlegel, with special reference to sex-linked inheritance". In: *Genetics* 6.6 (1921), p. 554.
- [3] Erik Arisholm and Lionel C Briand. "Predicting fault-prone components in a java legacy system". In: *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. ACM. 2006, pp. 8–17.
- [4] Erik Arisholm, Lionel C Briand, and Eivind B Johannessen. "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models". In: *Journal of Systems and Software* 83.1 (2010), pp. 2–17.
- [5] Ricardo Barandela et al. "Strategies for learning in class imbalance problems". In: *Pattern Recognition* 36.3 (2003), pp. 849–851.
- [6] Sukarna Barua et al. "MWMOTE–majority weighted minority oversampling technique for imbalanced data set learning". In: *IEEE Transactions on Knowledge and Data Engineering* 26.2 (2014), pp. 405–425.
- [7] Victor R Basili, Lionel C Briand, and Walcélio L Melo. "A validation of object-oriented design metrics as quality indicators". In: *IEEE Transactions on Software Engineering* 22.10 (1996), pp. 751–761.
- [8] K.E. Bennin et al. "Empirical Evaluation of Cross-Release Effort-Aware Defect Prediction Models". In: *Proceedings - 2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016*. 2016. ISBN: 9781509041275. DOI: 10.1109/QRS.2016.33.
- [9] Kwabena Ebo Bennin et al. "Investigating the Effects of Balanced Training and Testing Datasets on Effort-Aware Fault Prediction Models". In: *in Proceedings of the 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)* 1 (2016), pp. 154–163.

- [10] Kwabena Ebo Bennin et al. “MAHAKIL: Diversity based Oversampling Approach to Alleviate the Class Imbalance Issue in Software Defect Prediction”. In: *IEEE Transactions on Software Engineering* (2017). ISSN: 00985589. DOI: 10.1109/TSE.2017.2731766.
- [11] Kwabena Ebo Bennin et al. “The Significant Effects of Data Sampling Approaches on Software Defect Prioritization and Classification”. In: *Proceedings of the Eleventh International Symposium on Empirical Software Engineering and Measurement (ESEM)* ().
- [12] Christopher M Bishop and Michael E Tipping. “Variational relevance vector machines”. In: *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc. 2000, pp. 46–53.
- [13] David Bowes, Tracy Hall, and Jean Petrić. “Software defect prediction: do different classifiers find the same defects?” In: *Software Quality Journal* (2017). ISSN: 1573-1367. DOI: 10.1007/s11219-016-9353-3. URL: <https://doi.org/10.1007/s11219-016-9353-3>.
- [14] Andrew P Bradley. “The use of the area under the ROC curve in the evaluation of machine learning algorithms”. In: *Pattern recognition* 30.7 (1997), pp. 1145–1159.
- [15] Leo Breiman. “Random forests”. In: *Machine learning* 45.1 (2001), pp. 5–32.
- [16] Lionel C Briand, V R Brasili, and Christopher J Hetmanski. “Developing interpretable models with optimized set reduction for identifying high-risk software components”. In: *IEEE Transactions on Software Engineering* 19.11 (1993), pp. 1028–1044.
- [17] Edgar Brunner, Ullrich Munzel, and Madan L Puri. “The multivariate nonparametric Behrens–Fisher problem”. In: *Journal of Statistical Planning and Inference* 108.1 (2002), pp. 37–53.
- [18] Michael K Buckland and Fredric C Gey. “The relationship between recall and precision”. In: *JASIS* 45.1 (1994), pp. 12–19.
- [19] Peter Bühlmann and Torsten Hothorn. “Boosting algorithms: Regularization, prediction and model fitting”. In: *Statistical Science* (2007), pp. 477–505.

- [20] Chumphol Bunkhumpornpat, Krung Sinapiromsaran, and Chidchanok Lursinsap. “Safe-level-smote: Safe-level-synthetic minority over-sampling technique for handling the class imbalanced problem”. In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2009, pp. 475–482.
- [21] Nitesh V Chawla. “Data mining for imbalanced datasets: An overview”. In: *Data Mining and Knowledge Discovery Handbook*. Springer, 2005, pp. 875–886.
- [22] Nitesh V Chawla et al. “SMOTE: synthetic minority over-sampling technique”. In: *Journal of Artificial Intelligence Research* (2002), pp. 321–357.
- [23] Nitesh V Chawla et al. “SMOTEBoost: Improving prediction of the minority class in boosting”. In: *Knowledge Discovery in Databases: PKDD 2003*. Springer, 2003, pp. 107–119.
- [24] John G Cleary, Leonard E Trigg, et al. “K*: An instance-based learner using an entropic distance measure”. In: *in Proceedings of the 12th International Conference on Machine Learning* 5 (1995), pp. 108–114.
- [25] Norman Cliff. “Dominance statistics: Ordinal analyses to answer ordinal questions.” In: *Psychological Bulletin* 114.3 (1993), p. 494.
- [26] Thomas Cover and Peter Hart. “Nearest neighbor pattern classification”. In: *IEEE Transactions on Information Theory* 13.1 (1967), pp. 21–27.
- [27] Marco D’Ambros, Michele Lanza, and Romain Robbes. “An extensive comparison of bug prediction approaches”. In: *Proceedings of 2010 7th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE. 2010, pp. 31–41.
- [28] Marco D’Ambros, Michele Lanza, and Romain Robbes. “Evaluating defect prediction approaches: a benchmark and an extensive comparison”. In: *Empirical Software Engineering* 17.4-5 (2012), pp. 531–577.
- [29] Pedro Domingos. “Metacost: A general method for making classifiers cost-sensitive”. In: *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 1999, pp. 155–164.
- [30] Dennis J Drown, Taghi M Khoshgoftaar, and Naeem Seliya. “Evolutionary sampling and software quality modeling of high-assurance systems”. In: *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans* 39.5 (2009), pp. 1097–1107.

- [31] Bradley Efron et al. “Least angle regression”. In: *The Annals of Statistics* 32.2 (2004), pp. 407–499.
- [32] Andrew Estabrooks, Taeho Jo, and Nathalie Japkowicz. “A multiple re-sampling method for learning from imbalanced data sets”. In: *Computational Intelligence* 20.1 (2004), pp. 18–36.
- [33] Tom Fawcett and Foster Provost. “Adaptive fraud detection”. In: *Data Mining and Knowledge Discovery* 1.3 (1997), pp. 291–316.
- [34] Norman E Fenton and Niclas Ohlsson. “Quantitative analysis of faults and failures in a complex software system”. In: *IEEE Transactions on Software Engineering* 26.8 (2000), pp. 797–814.
- [35] Alberto Fernández, María José del Jesus, and Francisco Herrera. “On the 2-tuples based genetic tuning performance for fuzzy rule based classification systems in imbalanced data-sets”. In: *Information Sciences* 180.8 (2010), pp. 1268–1291.
- [36] Eibe Frank et al. “Using model trees for classification”. In: *Machine Learning* 32.1 (1998), pp. 63–76.
- [37] Salvador García and Francisco Herrera. “Evolutionary undersampling for classification with imbalanced datasets: Proposals and taxonomy”. In: *Evolutionary computation* 17.3 (2009), pp. 275–306.
- [38] Vicente García, Javier Salvador Sánchez, and Ramón Alberto Mollineda. “On the effectiveness of preprocessing methods when dealing with different levels of class imbalance”. In: *Knowledge-Based Systems* 25.1 (2012), pp. 13–21.
- [39] Trygve Gjerdem. “Genetic variation in quantitative traits and selective breeding in fish and shellfish”. In: *Aquaculture* 33.1-4 (1983), pp. 51–72.
- [40] David Gray et al. “The misuse of the NASA metrics data program data sets for automated software defect prediction”. In: *in Proceedings of the 15th Annual Conference on Evaluation & Assessment in Software Engineering (EASE 2011)* (2011), pp. 96–103.
- [41] Lan Guo et al. “Robust prediction of fault-proneness by random forests”. In: *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*. IEEE. 2004, pp. 417–428.
- [42] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. “Empirical validation of object-oriented metrics on open source software for fault prediction”. In: *IEEE Transactions on Software engineering* 31.10 (2005), pp. 897–910.

- [43] Tracy Hall et al. “A systematic literature review on fault prediction performance in software engineering”. In: *IEEE Transactions on Software Engineering* 38.6 (2012), pp. 1276–1304.
- [44] Hui Han, Wen-Yuan Wang, and Bing-Huan Mao. “Borderline-SMOTE: a new over-sampling method in imbalanced data sets learning”. In: *Advances in Intelligent Computing*. Springer, 2005, pp. 878–887.
- [45] Xiaoshu Hang and Honghua Dai. “Applying both positive and negative selection to supervised learning for anomaly detection”. In: *Proceedings of the 7th annual conference on Genetic and evolutionary computation*. ACM. 2005, pp. 345–352.
- [46] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. “Bug prediction based on fine-grained module histories”. In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press. 2012, pp. 200–210.
- [47] Haibo He and Edwardo A Garcia. “Learning from imbalanced data”. In: *IEEE Transactions on knowledge and data engineering* 21.9 (2009), pp. 1263–1284.
- [48] Haibo He et al. “ADASYN: Adaptive synthetic sampling approach for imbalanced learning”. In: *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*. IEEE. 2008, pp. 1322–1328.
- [49] Zhimin He et al. “An investigation on the feasibility of cross-project defect prediction”. In: *Automated Software Engineering* 19.2 (2012), pp. 167–199.
- [50] Nathalie Japkowicz and Shaju Stephen. “The class imbalance problem: A systematic study”. In: *Intelligent data analysis* 6.5 (2002), pp. 429–449.
- [51] Yue Jiang, Bojan Cukic, and Yan Ma. “Techniques for evaluating fault prediction models”. In: *Empirical Software Engineering* 13.5 (2008), pp. 561–595.
- [52] Wattana Jindaluang, Varin Chouvatut, and Sanpawat Kantabutra. “Under-sampling by algorithm with performance guaranteed for class-imbalance problem”. In: *Computer Science and Engineering Conference (ICSEC), 2014 International*. IEEE. 2014, pp. 215–221.
- [53] ZHAO Jiu-ran. “New Method of Elite Inbred Line Breeding in Corn [J]”. In: *Journal of Maize Sciences* 2 (2005), p. 9.

- [54] Mahesh V Joshi, Vipin Kumar, and Ramesh C Agarwal. “Evaluating boosting algorithms to classify rare classes: Comparison and improvements”. In: *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*. IEEE. 2001, pp. 257–264.
- [55] Lou Jost. “Entropy and diversity”. In: *Oikos* 113.2 (2006), pp. 363–375.
- [56] Marian Jureczko and Lech Madeyski. “Towards identifying software project clusters with regard to defect prediction”. In: *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. ACM. 2010, p. 9.
- [57] Marian Jureczko and Diomidis Spinellis. “Using object-oriented design metrics to predict software defects”. In: *Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki Wrocławskiej* (2010), pp. 69–81.
- [58] Yasutaka Kamei et al. “Revisiting common bug prediction findings using effort-aware models”. In: *Proceedings of 2010 IEEE International Conference on Software Maintenance (ICSM)*. IEEE. 2010, pp. 1–10.
- [59] Yasutaka Kamei et al. “The effects of over and under sampling on fault-prone module detection”. In: *First International Symposium on Empirical Software Engineering and Measurement, 2007. ESEM 2007*. IEEE. 2007, pp. 196–204.
- [60] Taghi M Khoshgoftaar, Abhijit S Pandya, and David L Lanning. “Application of neural networks for predicting program faults”. In: *Annals of Software Engineering* 1.1 (1995), pp. 141–154.
- [61] Taghi M Khoshgoftaar and Naeem Seliya. “Comparative assessment of software quality classification techniques: An empirical case study”. In: *Empirical Software Engineering* 9.3 (2004), pp. 229–257.
- [62] Sunghun Kim et al. “Dealing with noise in defect prediction”. In: *in Proceedings of the 2011 33rd International Conference on Software Engineering (ICSE)* (2011), pp. 481–490.
- [63] Barbara Kitchenham. “Robust statistical methods: why, what and how: keynote”. In: *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. ACM. 2015, p. 1.
- [64] Barbara Kitchenham et al. “Robust Statistical Methods for Empirical Software Engineering”. In: *Empirical Software Engineering* (2016), pp. 1–52.

- [65] Patrick Knab, Martin Pinzger, and Abraham Bernstein. “Predicting defect densities in source code files with decision tree learners”. In: *Proceedings of the 2006 international workshop on Mining software repositories*. ACM. 2006, pp. 119–125.
- [66] Ekrem Kocaguneli, Tim Menzies, and Jacky W Keung. “On the value of ensemble effort estimation”. In: *IEEE Transactions on Software Engineering* 38.6 (2012), pp. 1403–1416.
- [67] Ekrem Kocaguneli et al. “Active learning and effort estimation: Finding the essential content of software effort estimation data”. In: *IEEE Transactions on Software Engineering* 39.8 (2013), pp. 1040–1053.
- [68] Ekrem Kocaguneli et al. “Exploiting the essential assumptions of analogy-based effort estimation”. In: *IEEE Transactions on Software Engineering* 38.2 (2012), pp. 425–438.
- [69] Helena Chmura Kraemer and David J Kupfer. “Size of treatment effects and their importance to clinical research and practice”. In: *Biological psychiatry* 59.11 (2006), pp. 990–996.
- [70] Jeffrey D Kromrey et al. “Robustness in meta-analysis: An empirical comparison of point and interval estimates of standardized mean differences and Cliffs delta”. In: *Joint Statistical Meetings. Minneapolis, MN*. 2005.
- [71] Miroslav Kubat, Stan Matwin, and Others. “Addressing the curse of imbalanced training sets: one-sided selection”. In: *ICML*. Vol. 97. Nashville, USA. 1997, pp. 179–186.
- [72] M Kuhn et al. *caret: classification and regression training. R package version 6.0-24*. 2014.
- [73] Issam H Laradji, Mohammad Alshayeb, and Lahouari Ghouti. “Software defect prediction using ensemble learning on selected features”. In: *Information and Software Technology* 58 (2015), pp. 388–402.
- [74] Sauchi Stephen Lee. “Noisy replication in skewed binary classification”. In: *Computational statistics & data analysis* 34.2 (2000), pp. 165–191.
- [75] Stefan Lessmann et al. “Benchmarking classification models for software defect prediction: A proposed framework and novel findings”. In: *IEEE Transactions on Software Engineering* 34.4 (2008), pp. 485–496.

- [76] Paul Luo Li et al. “Experiences and results from initiating field defect prediction and product test prioritization efforts at ABB Inc.” In: *Proceedings of the 28th international conference on Software engineering*. ACM. 2006, pp. 413–422.
- [77] Andy Liaw and Matthew Wiener. “Classification and regression by randomForest”. In: *R news* 2.3 (2002), pp. 18–22.
- [78] Alexander Liu, Joydeep Ghosh, and Cheryl E Martin. “Generative Over-sampling for Mining Imbalanced Datasets.” In: *DMIN*. 2007, pp. 66–72.
- [79] Mingxia Liu, Linsong Miao, and Daoqiang Zhang. “Two-stage cost-sensitive learning for software defect prediction”. In: *IEEE Transactions on Reliability* 63.2 (2014), pp. 676–686.
- [80] Z J Liu and J F Cordes. “DNA marker technologies and their applications in aquaculture genetics”. In: *Aquaculture* 238.1 (2004), pp. 1–37.
- [81] Yan Ma, Lan Guo, and Bojan Cukic. “A statistical framework for the prediction of fault-proneness”. In: *Advances in Machine Learning Application in Software Engineering*, Idea Group Inc (2006), pp. 237–265.
- [82] Ying Ma et al. “Transfer learning for cross-company software defect prediction”. In: *Information and Software Technology* 54.3 (2012), pp. 248–256.
- [83] Lech Madeyski. *Test-driven development: An empirical evaluation of agile practice*. Springer Science & Business Media, 2010.
- [84] Lech Madeyski and Marian Jureczko. “Which process metrics can significantly improve defect prediction models? An empirical study”. In: *Software Quality Journal* 23.3 (2015), pp. 393–422.
- [85] Prasanta Chandra Mahalanobis. “On the generalized distance in statistics”. In: *Proceedings of the National Institute of Sciences (Calcutta)* 2 (1936), pp. 49–55.
- [86] T Mende, R Koschke Software Maintenance And, and undefined 2010. “Effort-aware defect prediction models”. In: *ieeexplore.ieee.org* (). URL: <http://ieeexplore.ieee.org/abstract/document/5714425/>.
- [87] Thilo Mende and Rainer Koschke. “Revisiting the evaluation of defect prediction models”. In: *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*. ACM. 2009, p. 7.

- [88] Tim Menzies, Jeremy Greenwald, and Art Frank. “Data mining static code attributes to learn defect predictors”. In: *IEEE Transactions on Software Engineering* 33.1 (2007), pp. 2–13.
- [89] Tim Menzies et al. “Assessing predictors of software defects”. In: *Proc. Workshop Predictive Software Models*. 2004.
- [90] Tim Menzies et al. “Defect prediction from static code features: current results, limitations, new approaches”. In: *Automated Software Engineering* 17.4 (2010), pp. 375–407.
- [91] Tim Menzies et al. “Implications of ceiling effects in defect predictors”. In: *Proceedings of the 4th international workshop on Predictor models in software engineering*. ACM. 2008, pp. 47–54.
- [92] Tim Menzies et al. “Problems with precision: A response to "comments on 'data mining static code attributes to learn defect predictors'"”. In: *IEEE Transactions on Software Engineering* 33.9 (2007), p. 637.
- [93] Audris Mockus and David M Weiss. “Predicting risk of software changes”. In: *Bell Labs Technical Journal* 5.2 (2000), pp. 169–180.
- [94] Akito Monden et al. “Assessing the cost effectiveness of fault prediction in acceptance testing”. In: *IEEE Transactions on Software Engineering* 39.10 (2013), pp. 1345–1357.
- [95] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction”. In: *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE. 2008, pp. 181–190.
- [96] John C Munson and Taghi M Khoshgoftaar. “The detection of fault-prone programs”. In: *IEEE Transactions on Software Engineering* 18.5 (1992), pp. 423–433.
- [97] Ingunn Myrtveit, Erik Stensrud, and Martin Shepperd. “Reliability and validity in comparative studies of software prediction models”. In: *IEEE Transactions on Software Engineering* 31.5 (2005), pp. 380–391.
- [98] Nachiappan Nagappan and Thomas Ball. “Use of relative code churn measures to predict system defect density”. In: *Proceedings of the 27th international conference on Software engineering*. ACM. 2005, pp. 284–292.
- [99] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. “Mining metrics to predict component failures”. In: *Proceedings of the 28th international conference on Software engineering*. ACM. 2006, pp. 452–461.

- [100] Jaechang Nam. “Survey on Software Defect Prediction”. In: *Master’s Thesis* (2009).
- [101] Adam Nickerson, Nathalie Japkowicz, and Evangelos Milios. “Using unsupervised learning to guide resampling in imbalanced data sets”. In: *Proceedings of the Eighth International Workshop on AI and Statistics*. 2001, pp. 261–265.
- [102] Niclas Ohlsson and Hans Alberg. “Predicting fault-prone software modules in telephone switches”. In: *IEEE Transactions on Software Engineering* 22.12 (1996), pp. 886–894.
- [103] Ganesh J Pai and Joanne Bechta Dugan. “Empirical analysis of software fault content and fault proneness using Bayesian methods”. In: *IEEE Transactions on software Engineering* 33.10 (2007), pp. 675–686.
- [104] Jeff Papows. *Glitch: the hidden impact of faulty software*. Pearson Education, 2010.
- [105] Michael Pazzani et al. “Reducing misclassification costs”. In: *Proceedings of the Eleventh International Conference on Machine Learning*. 1994, pp. 217–225.
- [106] Lourdes Pelayo and Scott Dick. “Applying novel resampling strategies to software defect prediction”. In: *Fuzzy Information Processing Society, 2007. NAFIPS’07. Annual Meeting of the North American*. IEEE. 2007, pp. 69–72.
- [107] Fayola Peters, Tim Menzies, and Andrian Marcus. “Better cross company defect prediction”. In: *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*. IEEE. 2013, pp. 409–418.
- [108] Son Lam Phung, Abdessalam Bouzerdoun, and Giang Hoang Nguyen. “Learning pattern classification tasks with imbalanced data sets”. In: (2009).
- [109] Andrea Dal Pozzolo, Olivier Caelen, and Gianluca Bontempi. *unbalanced: Racing for Unbalanced Methods Selection*. 2015. URL: <http://cran.r-project.org/package=unbalanced>.
- [110] Foster Provost. “Machine learning from imbalanced data sets 101”. In: *Proceedings of the AAAI\ 2000 workshop on imbalanced data sets*. 2000, pp. 1–3.
- [111] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

- [112] Danijel Radjenović et al. “Software fault prediction metrics: A systematic literature review”. In: *Information and Software Technology* 55.8 (2013), pp. 1397–1418.
- [113] J C Riquelme et al. “Finding defective modules from highly unbalanced datasets”. In: *Actas de los Talleres de las Jornadas de Ingenier{\\^i}a del Software y Bases de Datos 2.1* (2008), pp. 67–74.
- [114] Felix Salfner, Maren Lenk, and Mirosław Malek. “A survey of online failure prediction methods”. In: *ACM Computing Surveys (CSUR)* 42.3 (2010), p. 10.
- [115] J. Sayyad Shirabad and T.J. Menzies. *The PROMISE Repository of Software Engineering Databases*. School of Information Technology and Engineering, University of Ottawa, Canada. 2005. URL: <http://promise.site.uottawa.ca/SERepository>.
- [116] Chris Seiffert et al. “RUSBoost: A hybrid approach to alleviating class imbalance”. In: *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans* 40.1 (2010), pp. 185–197.
- [117] Ahmad Abu Shanab et al. “Impact of noise and data sampling on stability of feature ranking techniques for biological datasets”. In: *Information Reuse and Integration (IRI), 2012 IEEE 13th International Conference on*. IEEE. 2012, pp. 415–422.
- [118] Raed Shatnawi. “The application of ROC analysis in threshold identification, data imbalance and metrics selection for software fault prediction”. In: *Innovations in Systems and Software Engineering* (2017), pp. 1–17.
- [119] Martin Shepperd and Gada Kadoda. “Comparing software prediction techniques using simulation”. In: *IEEE Transactions on Software Engineering* 27.11 (2001), pp. 1014–1022.
- [120] Janet Siegmund, Norbert Siegmund, and Sven Apel. “Views on internal and external validity in empirical software engineering”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)* 1 (2015), pp. 9–19.
- [121] Qinbao Song et al. “Software defect association mining and defect correction effort prediction”. In: *IEEE Transactions on Software Engineering* 32.2 (2006), pp. 69–82.
- [122] Gail M Sullivan and Richard Feinn. “Using effect size or why the p value is not enough”. In: *Journal of graduate medical education* 4.3 (2012), pp. 279–282.

- [123] Yanmin Sun et al. “Cost-sensitive boosting for classification of imbalanced data”. In: *Pattern Recognition* 40.12 (2007), pp. 3358–3378.
- [124] Zhongbin Sun, Qinbao Song, and Xiaoyan Zhu. “Using coding-based ensemble learning to improve software defect prediction”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.6 (2012), pp. 1806–1817.
- [125] Zhongbin Sun et al. “A novel ensemble method for classifying imbalanced data”. In: *Pattern Recognition* 48.5 (2015), pp. 1623–1637.
- [126] Walter S Sutton. “The chromosomes in heredity”. In: *The Biological Bulletin* 4.5 (1903), pp. 231–250.
- [127] Xi Tan et al. “Assessing software quality by program clustering and defect prediction”. In: *Reverse Engineering (WCRE), 2011 18th Working Conference on*. IEEE. 2011, pp. 244–248.
- [128] Wei Tang and Taghi M Khoshgoftaar. “Noise identification with the k-means algorithm”. In: *in Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)* (2004), pp. 373–378.
- [129] R Core Team. “R: A language and environment for statistical computing”. In: *Vienna, Austria: R Foundation for Statistical Computing* (2012).
- [130] MACIEJ Tomczak and E Tomczak. “The need to report effect size estimates revisited. An overview of some recommended measures of effect size”. In: *Trends in Sport Sciences* 21.1 (2014), pp. 19–25.
- [131] Ivan Tomek. “Two modifications of CNN”. In: *IEEE Trans. Syst. Man Cybern.* 6 (1976), pp. 769–772.
- [132] L Torgo. *Data Mining with R, learning with case studies*. Chapman and Hall/CRC, 2010. URL: <http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR>.
- [133] Burak Turhan et al. “On the relative value of cross-company and within-company data for defect prediction”. In: *Empirical Software Engineering* 14.5 (2009), pp. 540–578.
- [134] Olivier Vandecruys et al. “Mining software repositories for comprehensible software fault prediction models”. In: *Journal of Systems and software* 81.5 (2008), pp. 823–839.

- [135] Shuo Wang and Xin Yao. “Using class imbalance learning for software defect prediction”. In: *IEEE Transactions on Reliability* 62.2 (2013), pp. 434–443.
- [136] Liyang Wei et al. “A study on several machine-learning methods for classification of malignant and benign clustered microcalcifications”. In: *IEEE transactions on medical imaging* 24.3 (2005), pp. 371–380.
- [137] Qiong Wei and Roland L Dunbrack Jr. “The role of balanced training and testing data sets for binary classifiers in bioinformatics”. In: *PloS one* 8.7 (2013), e67863.
- [138] Gary M Weiss and Foster Provost. “Learning when training data are costly: the effect of class distribution on tree induction”. In: *Journal of Artificial Intelligence Research* (2003), pp. 315–354.
- [139] Gary M Weiss and Foster Provost. “The effect of class distribution on classifier learning: an empirical study”. In: *Rutgers Univ* (2001).
- [140] R R Wilcox and F Schoenbrodt. “WRS: A compiled package of RR Wilcoxrobust statistics functions”. In: *R package version 0 1* (2009).
- [141] Rand Wilcox. *Modern statistics for the social and behavioral sciences: A practical introduction*. CRC press, 2011.
- [142] Claes Wohlin et al. *Experimentation in software engineering*. Springer Science & Business Media, 2012.
- [143] Ginny Y Wong, Frank H F Leung, and Sai-Ho Ling. “A novel evolutionary preprocessing method based on over-sampling and under-sampling for imbalanced datasets”. In: *Industrial Electronics Society, IECON 2013-39th Annual Conference of the IEEE*. IEEE. 2013, pp. 2354–2359.
- [144] Fei Xing, Ping Guo, and Michael R Lyu. “A novel method for early software quality prediction based on support vector machine”. In: *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*. IEEE. 2005, 10–pp.
- [145] Kihoon Yoon and Stephen Kwek. “A data reduction approach for resolving the imbalanced data issue in functional genomics”. In: *Neural Computing and Applications* 16.3 (2007), pp. 295–306.
- [146] Jun Zheng. “Cost-sensitive boosting neural networks for software defect prediction”. In: *Expert Systems with Applications* 37.6 (2010), pp. 4537–4543.

Appendix A

Appendix

A.1 Extra Results for MAHAKIL (Chapter 7)

In the tables below, we report the AUC and G-mean results for the sampling techniques per each prediction model on the 20 imbalanced datasets across different percentage of fault-prone modules (Pfp) values.

TABLE A.1: AUC Performance values of SMOTE, ADASYN, ROS, Borderline-SMOTE and MAHAKIL on Real World Data sets using C4.5 at different Pfp rates.

Project	Default	$Pfp-30$					$Pfp-40$					$Pfp-50$				
		AD	BS	MA	ROS	SM	AD	BS	MA	ROS	SM	AD	BS	MA	ROS	SM
ant-1.3	52.6	33.1	52.6	33.3	48.8	48.6	55.5	68.8	33.3	51.9	67.1	80.0	64.5	72.9	58.6	63.3
ant-1.4	50.0	47.3	51.6	50.0	43.6	56.2	45.8	63.9	63.0	42.6	62.2	69.6	46.4	57.3	50.8	62.0
ant-1.5	54.8	85.5	57.5	48.3	57.9	61.1	72.0	62.5	51.9	67.7	69.3	61.1	51.6	51.9	67.1	54.4
ant-1.6	76.4	68.8	69.5	76.4	71.5	68.1	68.5	50.3	78.2	75.0	68.3	73.1	62.4	75.5	60.2	74.2
arc	50.0	55.9	70.8	70.2	54.8	58.3	55.2	72.8	69.0	49.5	80.9	52.1	56.8	75.8	67.3	75.9
camel-1.4	50.0	58.3	63.8	62.8	56.8	48.5	51.6	55.1	55.3	54.9	63.3	65.2	62.4	62.4	63.7	62.7
camel-1.6	52.5	67.7	56.7	56.7	55.7	62.8	62.0	67.5	63.9	67.5	65.6	55.0	65.1	64.4	54.4	64.7
ivy-1.4	50.0	56.7	26.5	59.3	53.5	45.6	43.6	45.9	59.3	55.9	63.2	60.3	64.3	76.5	44.7	58.7
ivy-2.0	52.4	61.0	50.2	70.5	42.3	62.1	59.8	76.6	83.7	53.7	77.2	50.7	55.1	85.6	53.7	69.0
jedit-4.0	66.8	67.7	66.8	66.8	67.8	75.6	69.1	57.5	66.9	62.2	62.5	63.1	73.8	61.4	66.0	66.6
jedit-4.1	74.1	56.5	71.7	74.1	70.0	72.0	70.3	75.6	76.0	70.9	72.3	70.2	75.2	68.7	70.7	74.5
jedit-4.2	59.8	70.7	48.9	59.7	66.1	71.7	78.3	66.5	62.2	56.0	53.3	73.9	63.0	62.6	61.5	56.9
log4j-1.0	72.7	73.3	86.8	72.7	86.8	71.5	81.3	64.0	86.9	72.9	71.5	81.3	66.9	85.5	62.7	60.1
pbeans2	75.6	57.7	88.5	96.2	61.5	84.6	53.8	96.2	88.5	73.1	92.3	65.4	84.6	88.5	73.1	92.3
redaktor	65.6	80.8	75.9	65.6	68.0	65.3	71.2	61.0	61.8	70.5	70.1	68.1	68.1	74.4	70.3	65.6
synapse-1.0	54.0	45.7	65.1	69.4	63.6	52.6	43.6	65.1	68.9	61.1	48.3	43.6	62.6	53.6	52.6	59.4
systemdata	63.0	44.4	61.1	58.3	79.6	55.6	61.1	71.3	36.1	44.4	19.4	61.1	44.4	66.7	36.1	27.8
tomcat	70.1	64.6	46.7	73.8	56.1	63.5	65.4	65.3	54.6	56.8	73.2	67.9	71.2	52.5	56.2	76.0
xerces-1.2	50.0	64.2	63.1	66.3	58.2	58.8	60.8	70.6	65.5	66.3	75.0	57.7	68.4	63.4	62.1	69.2
xerces-1.3	61.3	77.5	73.9	66.7	60.5	71.4	77.5	73.9	70.2	65.1	69.0	75.1	73.4	71.1	56.9	67.3

*** NONE-No Sampling | AD-ADASYN | BS-Borderline-SMOTE | MA- MAHAKIL | ROS- Random Oversampling | SM- SMOTE

TABLE A.2: AUC Performance values of SMOTE, ADASYN, ROS, Borderline-SMOTE and MAHAKIL on Real World Data sets using KNN at different Pfp rates.

Project	Default	Pfp -30					Pfp -40					Pfp -50				
	NONE	AD	BS	MA	ROS	SM	AD	BS	MA	ROS	SM	AD	BS	MA	ROS	SM
ant-1.3	75.0	78.6	72.1	80.5	57.9	72.6	82.4	74.5	76.2	61.9	74.3	79.5	72.9	84.0	75.7	58.8
ant-1.4	41.7	53.6	46.7	54.6	44.6	60.6	55.4	50.0	62.5	45.2	52.3	51.6	53.4	57.2	48.0	31.4
ant-1.5	70.0	79.2	69.4	69.0	73.3	69.7	79.8	72.5	70.4	73.7	69.6	79.3	72.5	70.2	74.3	70.7
ant-1.6	85.4	80.3	85.2	85.4	83.7	82.6	75.6	84.8	86.5	85.6	81.3	76.9	82.9	87.3	86.7	85.8
arc	69.2	49.2	58.5	78.6	66.9	68.4	50.3	60.9	81.1	67.9	73.6	47.5	62.9	80.5	62.3	67.1
camel-1.4	67.9	58.1	64.9	68.4	63.3	66.6	58.7	62.1	66.5	65.1	64.7	59.5	63.6	66.7	65.9	63.7
camel-1.6	60.6	62.6	58.2	63.3	60.9	66.3	63.9	59.6	65.3	60.1	64.0	65.7	60.5	66.6	63.5	61.4
ivy-1.4	66.7	70.3	64.1	69.6	56.9	70.0	68.3	61.7	70.7	51.3	74.3	72.0	56.5	70.0	52.0	68.5
ivy-2.0	89.3	83.1	89.4	88.2	88.4	88.8	81.6	87.6	86.9	86.8	88.8	78.4	86.5	86.3	85.2	87.5
jedit-4.0	68.5	70.8	69.7	68.5	68.1	75.2	71.0	70.4	68.1	69.0	73.1	72.1	73.2	72.5	69.9	67.3
jedit-4.1	73.0	75.3	73.0	73.0	73.9	75.6	75.2	69.8	75.1	72.0	75.3	76.6	68.2	75.2	71.3	78.0
jedit-4.2	69.9	81.7	69.4	70.3	68.4	67.7	81.5	66.9	69.5	65.3	69.8	82.1	65.6	69.8	65.9	72.1
log4j-1.0	91.9	69.0	91.7	91.9	88.3	75.8	69.3	88.6	87.9	90.8	82.4	71.1	84.3	85.1	83.2	81.8
pbeans2	87.2	57.7	87.2	84.6	80.8	65.4	62.8	76.9	79.5	65.4	70.5	67.9	85.9	83.3	84.6	79.5
redaktor	60.7	74.6	66.0	72.0	66.1	71.4	76.2	65.4	71.7	63.8	60.1	76.8	76.1	69.6	64.2	52.7
synapse-1.0	56.2	72.8	63.6	70.4	54.7	62.3	71.7	65.1	70.9	56.8	71.5	73.4	67.2	70.9	57.0	74.7
systemdata	65.7	63.0	57.4	72.2	59.3	60.2	63.9	67.6	71.3	61.1	74.1	75.0	59.3	73.1	55.6	1.9
tomcat	79.8	68.9	81.2	80.8	78.8	80.4	69.2	80.4	81.0	79.1	78.5	66.8	79.7	82.0	77.9	78.7
xerces-1.2	45.5	51.9	59.8	69.7	51.0	53.8	54.7	59.2	70.7	51.2	49.6	58.1	55.6	71.4	53.1	46.3
xerces-1.3	71.9	79.9	75.7	68.4	70.4	70.6	77.7	77.6	67.9	71.2	70.5	82.9	78.1	66.0	71.4	69.1

*** NONE-No Sampling | AD-ADASYN | BS-Borderline-SMOTE | MA- MAHAKIL | ROS- Random Oversampling | SM- SMOTE

TABLE A.3: AUC Performance values of SMOTE, ADASYN, ROS, Borderline-SMOTE and MAHAKIL on Real World Data sets using NNET at different Pfp rates.

Project	Default	Pfp -30					Pfp -40					Pfp -50				
	NONE	AD	BS	MA	ROS	SM	AD	BS	MA	ROS	SM	AD	BS	MA	ROS	SM
ant-1.3	49.0	79.0	76.2	66.2	52.9	69.0	77.6	64.0	76.2	80.0	77.9	78.1	61.9	58.1	56.4	60.2
ant-1.4	76.8	70.7	59.9	79.6	77.3	80.9	67.1	79.6	70.1	72.4	55.5	65.6	72.6	76.6	73.7	66.9
ant-1.5	70.0	82.3	65.4	60.7	70.3	60.1	74.9	70.1	69.5	70.2	67.2	83.9	71.7	68.2	60.7	71.1
ant-1.6	78.7	80.2	79.0	82.8	82.5	85.8	76.9	79.2	82.4	82.9	77.4	76.6	76.1	79.6	83.9	77.4
arc	64.4	65.1	70.7	82.9	84.9	79.2	62.7	62.0	78.2	74.9	77.1	61.7	55.6	84.4	68.9	77.5
camel-1.4	74.9	70.3	65.1	76.0	65.6	68.4	68.5	64.4	70.5	66.2	67.2	70.0	73.4	73.8	70.5	68.5
camel-1.6	67.6	65.0	72.4	71.5	73.3	70.5	67.0	67.6	73.6	68.9	66.9	68.7	58.7	73.0	63.2	74.2
ivy-1.4	56.8	62.3	75.1	68.4	64.5	67.7	68.0	63.2	60.8	65.6	64.4	72.8	55.5	69.3	64.0	54.4
ivy-2.0	87.5	77.2	89.8	83.6	86.9	88.2	76.4	74.8	82.5	76.4	85.1	75.9	77.5	75.5	88.6	72.0
jedit-4.0	63.5	71.1	68.0	68.6	71.8	66.8	65.8	72.9	73.4	69.0	68.4	76.1	72.5	68.7	73.8	69.0
jedit-4.1	81.1	79.2	83.7	78.4	82.5	84.5	81.8	83.9	84.2	77.5	86.0	75.9	85.3	84.4	86.2	87.2
jedit-4.2	66.7	74.9	63.6	76.6	68.6	67.5	79.2	67.5	68.3	65.0	77.0	71.8	74.2	61.6	70.2	73.6
log4j-1.0	92.6	81.8	78.8	86.2	81.3	90.4	83.5	88.7	83.7	91.0	84.6	76.6	86.8	87.3	83.2	84.3
pbeans2	82.1	61.5	85.9	84.6	78.2	85.9	69.2	83.3	87.2	87.2	71.8	61.5	74.4	79.5	76.9	51.3
redaktor	68.9	75.7	71.1	70.5	74.4	67.8	86.6	68.0	71.8	71.9	68.0	90.0	74.3	69.2	73.5	76.6
synapse-1.0	65.1	38.3	65.1	64.7	62.6	55.7	44.3	59.6	64.7	54.9	60.2	50.2	60.4	63.8	58.7	58.3
systemdata	66.7	72.2	51.9	68.5	70.4	74.1	61.1	66.7	72.2	64.8	81.5	66.7	50.0	68.5	61.1	53.7
tomcat	80.6	82.3	68.7	80.0	74.9	78.4	81.7	77.3	76.8	78.4	79.3	78.7	71.9	76.9	80.9	82.1
xerces-1.2	74.7	64.7	71.2	74.8	78.0	72.6	65.4	81.1	71.9	73.9	65.0	67.7	79.1	73.2	80.8	69.8
xerces-1.3	78.4	77.6	82.3	80.6	77.5	72.9	71.4	72.6	74.8	72.0	70.0	77.5	72.5	78.4	80.1	78.7

*** NONE-No Sampling | AD-ADASYN | BS-Borderline-SMOTE | MA- MAHAKIL | ROS- Random Oversampling | SM- SMOTE

TABLE A.4: AUC Performance values of SMOTE, ADASYN, ROS, Borderline-SMOTE and MAHAKIL on Real World Data sets using RF at different Pfp rates.

Project	Default	Pfp -30					Pfp -40					Pfp -50				
	NONE	AD	BS	MA	ROS	SM	AD	BS	MA	ROS	SM	AD	BS	MA	ROS	SM
ant-1.3	73.3	76.7	70.7	72.9	62.6	69.0	77.6	71.9	72.1	62.9	81.2	78.1	74.5	77.1	61.4	69.0
ant-1.4	60.9	67.7	69.7	65.4	67.1	64.5	71.6	70.4	76.4	70.3	70.2	69.2	70.2	76.3	67.1	77.7
ant-1.5	66.8	82.0	67.2	66.2	66.0	61.6	80.9	66.0	66.7	66.4	67.4	80.6	65.2	66.0	69.0	67.1
ant-1.6	86.5	83.3	85.6	86.7	86.1	81.3	83.2	82.0	85.8	84.3	79.1	82.0	79.3	84.8	84.0	80.1
arc	71.3	46.9	71.1	82.4	67.5	78.3	45.6	70.6	82.8	67.1	70.5	49.1	72.6	81.3	72.5	77.4
camel-1.4	75.4	73.4	73.2	72.1	68.8	70.9	74.6	73.6	71.4	71.5	67.6	73.9	73.1	69.8	72.6	69.2
camel-1.6	74.6	73.0	75.8	72.8	75.1	72.5	71.8	75.0	73.2	75.3	73.5	71.3	74.2	72.6	75.8	73.6
ivy-1.4	63.5	69.7	62.8	67.5	60.5	57.7	72.1	70.3	66.7	57.7	66.4	73.3	68.1	63.7	54.7	62.9
ivy-2.0	85.3	84.7	82.2	83.5	80.3	81.7	84.3	79.4	86.2	79.8	84.4	84.7	80.3	87.3	77.2	83.1
jedit-4.0	73.6	77.6	78.1	75.0	78.8	76.2	79.0	76.6	76.6	79.9	75.1	81.0	73.4	75.0	78.8	74.5
jedit-4.1	85.2	84.1	84.6	84.6	87.9	87.0	85.3	86.3	86.6	84.5	83.4	83.8	86.3	87.0	84.6	87.5
jedit-4.2	77.0	83.1	74.4	75.1	74.2	75.0	82.9	74.8	75.5	74.0	72.6	84.1	73.8	75.1	73.5	71.6
log4j-1.0	88.2	84.2	91.2	88.2	89.5	88.3	81.5	92.3	91.0	87.3	87.3	83.2	90.6	89.4	87.9	90.4
pbeans2	89.7	71.8	84.6	87.2	89.7	92.3	76.9	84.6	89.7	84.6	84.6	71.8	89.7	89.7	84.6	88.5
redaktor	62.1	83.0	65.0	67.3	62.0	65.8	79.7	66.9	67.5	65.9	66.8	82.8	65.0	66.6	66.6	68.3
synapse-1.0	58.3	65.5	65.7	71.9	60.2	59.8	61.7	66.0	71.9	62.8	60.4	64.3	66.6	72.3	59.8	58.1
systemdata	79.6	63.0	79.6	75.9	79.6	79.6	66.7	77.8	74.1	75.9	74.1	57.4	65.7	79.6	68.5	51.9
tomcat	87.2	83.0	87.8	83.2	85.5	87.9	80.8	85.8	82.9	86.5	89.5	80.4	84.4	81.7	85.2	87.1
xerces-1.2	80.7	71.4	83.1	81.4	77.5	80.2	70.2	84.0	80.1	77.9	75.6	70.2	82.1	78.2	78.1	80.3
xerces-1.3	78.6	86.3	80.0	80.1	77.8	75.8	85.1	80.1	80.0	79.4	79.3	84.2	80.7	80.1	79.0	78.2

*** NONE-No Sampling | AD-ADASYN | BS-Borderline-SMOTE | MA- MAHAKIL | ROS- Random Oversampling | SM- SMOTE

TABLE A.5: AUC Performance values of SMOTE, ADASYN, ROS, Borderline-SMOTE and MAHAKIL on Real World Data sets using SVM at different Pfp rates.

Project	Default	Pfp -30					Pfp -40					Pfp -50				
	NONE	AD	BS	MA	ROS	SM	AD	BS	MA	ROS	SM	AD	BS	MA	ROS	SM
ant-1.3	74.3	73.8	72.9	76.7	70.5	74.8	79.5	67.6	73.8	77.1	74.3	77.1	66.2	74.8	78.1	74.8
ant-1.4	56.7	57.7	58.0	57.5	57.7	57.7	73.4	57.4	64.0	58.7	57.9	69.2	60.5	63.2	66.7	54.3
ant-1.5	66.9	73.0	64.6	64.6	68.9	66.3	73.9	68.6	67.8	70.1	64.8	73.0	69.5	68.6	72.2	69.9
ant-1.6	83.1	83.4	83.2	82.4	80.4	80.0	82.1	82.8	83.9	82.9	80.5	79.2	75.5	82.0	82.5	83.8
arc	62.3	52.8	72.0	74.2	78.9	72.3	49.3	79.9	74.6	80.0	70.7	41.9	74.2	70.7	75.8	65.7
camel-1.4	82.5	71.0	72.4	74.0	75.4	73.2	65.7	64.3	66.7	72.1	63.0	59.6	61.0	61.5	66.3	58.9
camel-1.6	71.5	72.1	70.7	71.7	71.0	73.0	73.4	67.9	66.2	69.9	70.4	67.0	62.7	65.7	66.2	67.0
ivy-1.4	60.0	62.7	63.2	59.2	65.3	64.0	62.4	62.1	65.1	64.0	63.2	65.3	61.3	64.3	67.5	60.0
ivy-2.0	76.8	81.8	81.4	88.3	82.7	88.5	78.1	83.7	88.2	87.3	88.1	63.2	67.4	84.9	86.7	72.8
jedit-4.0	74.0	80.8	74.0	73.8	76.5	74.1	75.9	75.8	72.2	74.2	73.8	73.2	68.2	75.0	78.5	76.9
jedit-4.1	84.7	80.3	84.6	84.2	86.1	85.3	80.0	85.2	85.4	85.2	84.7	80.1	84.4	86.6	87.5	87.2
jedit-4.2	69.3	82.4	68.5	69.8	67.7	68.9	83.9	69.4	70.0	70.0	72.6	77.8	68.9	74.3	74.6	73.9
log4j-1.0	79.1	80.4	84.8	82.1	78.5	88.2	82.1	89.8	91.7	90.9	89.5	78.5	91.7	92.8	91.5	93.1
pbeans2	82.1	53.8	79.5	82.1	71.8	71.8	66.7	74.4	82.1	74.4	66.7	38.5	38.5	23.1	20.5	56.4
redaktor	55.6	72.8	60.3	62.4	65.3	56.7	70.7	64.4	65.3	62.6	61.9	77.3	60.3	60.5	61.2	61.2
synapse-1.0	51.5	65.1	71.1	79.1	69.4	71.1	71.5	71.5	80.9	70.6	71.5	71.5	74.9	78.3	73.2	64.3
systemdata	80.6	81.5	75.9	75.9	77.8	74.1	75.9	72.2	75.9	81.5	72.2	79.6	72.2	79.6	72.2	50.0
tomcat	74.2	76.2	78.7	79.4	84.2	79.6	76.1	79.6	80.3	84.5	77.6	76.0	80.3	80.7	81.6	80.1
xerces-1.2	73.9	65.7	71.2	65.6	65.0	67.0	59.8	68.5	63.4	68.9	58.2	63.2	56.5	55.1	60.9	58.6
xerces-1.3	78.7	83.1	77.0	74.4	73.5	75.0	82.1	75.0	73.5	73.9	73.5	83.4	72.2	71.0	71.7	70.4

*** NONE-No Sampling | AD-ADASYN | BS-Borderline-SMOTE | MA- MAHAKIL | ROS- Random Oversampling | SM- SMOTE

TABLE A.6: G-mean values of SMOTE, ADASYN, ROS, Borderline-SMOTE and MAHAKIL on Real World Data sets using C4.5 at different Pfp rates.

	Default	Pfp -30					Pfp -40					Pfp -50				
Project	NONE	AD	BS	MA	ROS	SM	AD	BS	MA	ROS	SM	AD	BS	MA	ROS	SM
ant-1.3	14.3	18.2	14.3	16.7	16.7	33.3	37.5	13.3	16.7	22.2	42.1	47.1	33.3	53.3	30.0	25.0
ant-1.4	0.0	19.0	18.2	33.1	19.4	34.5	28.6	26.1	30.0	8.3	50.0	23.1	28.6	31.8	27.6	47.4
ant-1.5	37.5	57.1	31.6	26.1	25.0	18.2	42.1	27.6	31.6	42.1	27.9	22.2	23.1	31.6	35.7	22.2
ant-1.6	57.1	53.6	62.5	57.1	66.7	47.5	56.7	44.1	69.1	54.5	50.0	62.9	43.1	66.7	40.6	63.3
arc	0.0	11.1	36.4	44.4	18.2	34.8	7.4	30.0	41.1	18.2	41.4	8.0	26.7	40.0	34.5	39.0
camel-1.4	0.0	38.3	37.2	28.6	29.4	25.2	28.3	34.4	28.0	26.0	33.6	37.6	33.0	30.9	37.5	34.0
camel-1.6	18.7	43.3	31.1	31.5	34.8	40.7	29.6	40.3	35.3	47.2	44.3	28.6	36.8	39.7	34.9	40.5
ivy-1.4	0.0	28.6	0.0	18.2	11.8	8.0	11.1	15.4	40.0	15.4	22.2	25.0	26.7	14.3	0.0	13.3
ivy-2.0	0.0	10.0	16.7	30.3	0.0	29.3	16.7	27.0	44.4	18.2	36.4	7.4	18.7	53.3	18.2	33.8
jedit-4.0	50.0	40.0	50.0	50.0	51.3	57.1	44.9	44.0	45.0	40.9	44.1	54.9	47.1	43.5	50.0	47.6
jedit-4.1	60.5	41.0	68.0	60.5	60.0	61.0	55.9	66.7	64.0	53.6	55.6	59.2	66.7	59.6	53.6	61.3
jedit-4.2	16.7	50.0	28.6	41.2	38.7	25.0	41.4	24.4	39.0	22.2	32.4	42.9	29.4	36.4	33.3	35.0
log4j-1.0	62.5	63.6	62.5	62.5	62.5	47.6	66.7	66.7	70.0	58.8	48.0	66.7	42.1	69.6	41.7	60.0
pbeans2	57.1	28.6	75.0	85.7	57.1	60.0	40.0	85.7	66.7	44.4	75.0	40.0	60.0	75.0	44.4	75.0
redaktor	46.2	50.0	50.0	46.2	41.7	42.9	52.6	42.1	36.4	53.3	50.0	45.5	40.0	52.6	53.3	40.0
synapse-1.0	20.0	0.0	36.4	16.7	30.8	15.4	0.0	28.6	16.7	26.7	11.8	0.0	28.6	16.7	15.4	23.5
systemdata	33.3	0.0	33.3	22.2	57.1	50.0	33.3	44.4	33.3	0.0	0.0	33.3	28.6	28.6	0.0	0.0
tomcat	0.0	25.5	25.5	29.2	20.4	37.7	32.7	36.0	29.8	21.7	25.0	20.3	40.6	27.1	20.8	34.2
xerces-1.2	0.0	30.0	25.6	38.1	25.0	36.7	26.7	46.2	38.3	39.4	45.2	29.0	42.4	25.5	35.8	37.8
xerces-1.3	37.8	44.4	48.0	45.3	33.3	46.7	49.0	48.0	47.3	38.5	45.9	46.2	48.0	47.3	28.6	42.6

*** NONE-No Sampling | AD-ADASYN | BS-Borderline-SMOTE | MA- MAHAKIL | ROS- Random Oversampling | SM- SMOTE

TABLE A.7: G-mean values of SMOTE, ADASYN, ROS, Borderline-SMOTE and MAHAKIL on Real World Data sets using KNN at different Pfp rates.

	Default	Pfp -30					Pfp -40					Pfp -50				
Project	NONE	AD	BS	MA	ROS	SM	AD	BS	MA	ROS	SM	AD	BS	MA	ROS	SM
ant-1.3	66.7	57.1	50.0	42.1	40.0	37.5	50.0	43.5	40.0	34.8	38.1	44.4	34.5	43.5	40.0	38.1
ant-1.4	0.0	0.0	16.0	11.8	0.0	16.0	37.5	20.0	33.3	26.7	37.0	33.3	23.5	33.3	27.3	15.4
ant-1.5	16.7	40.0	33.3	31.2	38.7	29.4	35.3	33.3	27.3	38.7	27.8	29.8	30.4	27.9	30.2	24.0
ant-1.6	65.5	56.7	70.2	65.5	69.8	66.7	58.8	64.8	72.7	68.7	64.7	55.7	61.9	70.4	64.0	66.7
arc	0.0	25.0	23.5	45.2	26.1	30.8	24.2	20.0	50.0	27.6	40.0	19.0	23.3	40.0	24.0	27.6
camel-1.4	8.0	16.4	34.0	35.0	30.1	37.6	25.0	37.3	37.4	34.2	33.1	31.6	32.8	35.7	34.9	34.4
camel-1.6	0.0	16.1	20.3	25.0	30.2	29.3	36.3	30.7	38.2	30.8	31.3	39.1	33.3	36.9	37.6	31.9
ivy-1.4	0.0	21.1	18.2	26.1	16.7	16.7	12.9	18.2	23.1	13.3	26.1	21.1	15.0	23.1	12.8	20.0
ivy-2.0	0.0	43.2	50.0	55.0	51.2	50.0	36.4	44.0	44.0	50.0	50.0	31.4	39.3	38.1	36.6	42.9
jedit-4.0	41.0	45.8	44.4	41.0	46.5	52.2	49.1	55.6	46.8	48.9	52	48.3	54.5	50.0	56.6	50.0
jedit-4.1	59.6	40.0	61.2	59.6	58.8	61.5	42.6	51.7	64.0	59.3	65.4	59.4	52.2	65.5	50.8	62.3
jedit-4.2	18.2	50.0	34.6	40.0	28.6	16.7	47.6	34.5	34.1	26.7	25.5	44.9	36.7	36.0	23.3	34.0
log4j-1.0	53.3	44.4	63.2	53.3	63.6	43.5	60.9	72.0	72.0	69.2	64.0	53.8	74.1	74.1	74.1	60.0
pbeans2	0.0	0.0	0.0	19.7	0.0	20.0	44.4	44.4	40.0	0.0	40.0	44.4	50.0	42.9	42.9	57.1
redaktor	0.0	44.4	0.0	40.0	40.0	10.5	35.7	27.3	40.0	45.5	21.1	35.3	42.9	41.4	34.5	31.6
synapse-1.0	0.0	30.8	14.3	15.4	14.3	15.4	40.0	26.7	15.4	14.3	26.7	40.0	26.7	23.5	25.0	26.7
systemdata	0.0	28.6	0.0	28.2	0.0	0.0	30.8	0.0	32.5	25.0	57.1	26.7	0.0	36.4	0.0	9.1
tomcat	0.0	21.6	41.8	37.7	39.4	28.0	25.0	35.0	38.0	42.9	38.6	22.4	29.6	38.7	29.0	41.1
xerces-1.2	0.0	10.0	26.1	0.0	27.6	25.0	22.8	30.0	29.2	25.9	30.1	23.7	25.0	27.9	26.8	17.9
xerces-1.3	0.0	43.3	41.4	30.3	39.3	34.5	40.5	43.5	35.6	38.8	35.6	46.2	44.4	38.6	40.0	43.7

*** NONE-No Sampling | AD-ADASYN | BS-Borderline-SMOTE | MA- MAHAKIL | ROS- Random Oversampling | SM- SMOTE

TABLE A.8: G-mean values of SMOTE, ADASYN, ROS, Borderline-SMOTE and MAHAKIL on Real World Data sets using NNET at different Pfp rates.

Project	Default	Pfp -30					Pfp -40					Pfp -50				
	NONE	AD	BS	MA	ROS	SM	AD	BS	MA	ROS	SM	AD	BS	MA	ROS	SM
ant-1.3	28.6	50.0	20.0	36.4	16.7	30.8	44.4	15.4	28.6	42.9	30.8	47.1	26.7	20.0	23.5	25.0
ant-1.4	13.3	37.5	31.6	31.7	50.0	52.4	51.4	44.4	41.2	53.8	23.1	45.7	48.3	44.3	43.2	36.4
ant-1.5	33.3	54.5	13.3	17.4	35.3	20.0	25.0	24.0	17.4	25.0	27.6	44.4	30.0	21.4	16.0	18.2
ant-1.6	56.1	51.9	65.5	61.3	61.5	71.0	51.0	54.8	56.7	58.8	54.1	49.3	52.6	59.7	66.7	57.5
arc	16.7	28.6	30.0	63.2	47.1	55.6	26.1	18.2	38.1	41.7	36.4	0.0	29.6	50.0	16.0	40.0
camel-1.4	46.2	38.5	35.1	46.7	32.2	34.6	30.6	33.6	38.2	36.7	37.6	43.1	45.6	44.1	36.8	34.9
camel-1.6	11.6	35.7	33.9	40.7	42.9	38.2	40.2	37.4	43.7	38.0	33.8	40.0	35.0	42.9	34.5	42.2
ivy-1.4	0.0	28.6	42.9	27.3	33.3	17.4	20.0	12.5	20.0	18.2	20.0	20.0	14.3	37.5	23.5	16.0
ivy-2.0	21.1	35.0	54.5	32.3	48.5	48.8	25.8	24.4	37.5	27.3	42.6	30.8	37.5	43.5	42.9	27.0
jedit-4.0	41.2	51.0	37.2	37.2	39.0	54.2	36.1	50.0	54.2	34.1	50.0	47.5	51.5	50.0	40.0	40.0
jedit-4.1	43.1	61.5	64.3	64.4	62.7	64.3	62.9	58.6	64.2	57.1	62.5	55.4	60.0	63.0	61.8	61.8
jedit-4.2	22.2	40.0	24.4	33.3	26.3	30.3	40.0	35.6	42.4	33.3	28.6	36.8	33.3	37.8	27.3	36.7
log4j-1.0	50.0	60.9	47.1	53.3	50.0	73.7	58.3	66.7	62.6	83.3	61.5	54.5	72.7	60.0	63.6	66.7
pbeans2	0.0	57.1	57.1	60.0	44.4	57.1	33.3	60.0	60.0	0.0	60.0	28.6	28.6	60.0	44.4	25.0
redaktor	57.1	57.1	53.3	44.4	55.6	50.0	55.6	44.4	33.3	50.0	42.1	60.0	47.1	44.4	45.5	45.5
synapse-1.0	14.3	0.0	12.5	14.3	13.3	14.3	0.0	14.3	15.4	14.3	13.3	0.0	14.3	15.4	13.3	13.3
systemdata	28.6	80.0	28.6	28.6	33.3	28.6	44.4	28.6	28.6	25.0	28.6	80.0	28.6	28.6	28.6	25.0
tomcat	16.2	37.1	25.0	35.6	25.9	22.7	40.0	40.0	36.0	41.7	39.0	35.1	30.3	34.7	33.0	42.9
xerces-1.2	13.8	26.9	20.0	40.0	44.4	34.9	25.4	46.7	36.7	43.8	19.2	33.8	43.1	35.3	47.1	33.7
xerces-1.3	30.8	42.4	51.2	46.8	40.9	37.9	24.4	35.7	39.3	36.1	40.6	45.0	41.3	38.3	42.5	42.5

*** NONE-No Sampling | AD-ADASYN | BS-Borderline-SMOTE | MA- MAHAKIL | ROS- Random Oversampling | SM- SMOTE

TABLE A.9: G-mean values of SMOTE, ADASYN, ROS, Borderline-SMOTE and MAHAKIL on Real World Data sets using RF at different Pfp rates.

Project	Default	Pfp -30					Pfp -40					Pfp -50				
	NONE	AD	BS	MA	ROS	SM	AD	BS	MA	ROS	SM	AD	BS	MA	ROS	SM
ant-1.3	42.9	20.0	25.0	28.6	30.8	37.5	57.1	37.5	26.7	37.5	37.5	53.3	44.4	37.5	30.8	26.7
ant-1.4	23.5	37.0	22.2	33.3	30.0	20.0	46.7	22.2	28.6	30.0	31.6	38.7	31.6	20.0	20.0	46.7
ant-1.5	33.3	35.3	33.3	33.3	37.5	28.6	28.6	33.3	22.2	35.3	30.8	37.5	31.6	22.2	37.5	27.6
ant-1.6	64.2	61.0	69.1	69.1	60.0	62.9	64.5	62.3	68.9	67.7	59.7	65.6	58.1	67.7	67.7	64.8
arc	18.2	0.0	35.3	66.7	13.3	38.1	0.0	33.3	40.0	14.3	26.7	0.0	31.6	50.0	23.5	33.3
camel-1.4	35.3	37.1	20.9	31.2	23.3	29.2	39.4	22.9	28.6	23.2	32.0	40.0	27.2	32.6	29.7	32.0
camel-1.6	31.5	34.7	28.6	38.0	40.0	39.6	36.0	34.7	42.6	39.6	43.1	35.5	39.2	46.0	33.7	44.6
ivy-1.4	0.0	33.3	0.0	22.0	0.0	10.0	28.6	0.0	40.0	0.0	18.2	25.0	0.0	33.3	0.0	23.1
ivy-2.0	14.3	11.1	26.1	18.2	24.0	36.4	8.7	26.1	32.0	23.1	40.0	9.1	25.0	30.8	16.7	42.3
jedit-4.0	42.6	55.8	43.5	41.7	51.2	50.0	55.3	45.8	44.9	51.2	48.1	56.0	46.2	46.2	51.2	45.6
jedit-4.1	60.9	51.2	58.3	56.5	59.1	64.0	64.2	65.4	68.0	53.1	61.8	67.9	69.2	66.7	56.0	69.0
jedit-4.2	19.0	53.8	23.1	34.5	29.6	37.5	56.0	22.2	32.3	32.0	27.6	44.4	28.6	33.3	33.3	34.1
log4j-1.0	62.5	66.7	58.8	62.5	66.7	58.8	72.0	70.0	70.0	55.6	60.9	66.7	73.7	70.0	63.2	70.0
pbeans2	57.1	40.0	75.0	75.0	75.0	75.0	33.3	75.0	75.0	75.0	75.0	33.3	75.0	75.0	75.0	66.7
redaktor	46.2	57.1	46.2	57.1	46.2	42.9	57.1	57.1	50.0	42.9	37.5	53.3	57.1	50.0	42.9	37.5
synapse-1.0	18.2	0.0	15.4	15.4	14.3	14.3	0.0	16.7	16.7	14.3	14.3	0.0	30.8	16.7	16.7	14.3
systemdata	40.0	0.0	40.0	33.3	33.3	33.3	40.0	40.0	28.6	33.3	28.6	40.0	28.6	50.0	0.0	25.0
tomcat	6.9	20.0	25.6	19.5	11.4	36.7	14.3	20.0	24.4	11.4	47.1	14.3	27.3	18.2	11.1	43.1
xerces-1.2	47.6	35.6	51.0	47.8	40.7	43.6	32.7	51.9	44.0	38.5	37.0	36.7	50.9	40.8	39.2	49.4
xerces-1.3	46.5	53.3	48.0	41.7	40.0	40.7	53.1	44.0	39.2	52.0	50.0	52.0	40.8	39.2	49.0	45.5

*** NONE-No Sampling | AD-ADASYN | BS-Borderline-SMOTE | MA- MAHAKIL | ROS- Random Oversampling | SM- SMOTE

TABLE A.10: G-mean values of SMOTE, ADASYN, ROS, Borderline-SMOTE and MAHAKIL on Real World Data sets using SVM at different Pfp rates.

Project	Default	<i>Pfp</i> -30					<i>Pfp</i> -40					<i>Pfp</i> -50				
	NONE	AD	BS	MA	ROS	SM	AD	BS	MA	ROS	SM	AD	BS	MA	ROS	SM
ant-1.3	36.4	42.9	28.6	42.9	18.2	42.9	40.0	25.0	37.5	37.5	41.7	42.1	26.7	35.3	40.0	42.9
ant-1.4	0.0	28.6	19.0	32.0	11.8	32.0	34.5	32.0	32.0	32.0	32.0	45.7	38.5	37.0	30.8	38.7
ant-1.5	30.8	44.4	36.4	38.1	47.6	32.0	38.5	27.6	35.3	30.0	29.4	25.8	30.8	33.3	33.3	33.3
ant-1.6	48.0	60.3	61.8	39.1	47.3	55.4	65.7	65.8	68.5	67.6	58.0	59.0	52.2	59.4	66.7	59.7
arc	0.0	10.5	37.5	42.9	18.2	42.9	14.8	40.0	40.0	34.8	37.5	8.7	48.0	34.5	41.2	28.6
camel-1.4	7.4	31.0	29.6	40.0	29.4	33.0	27.2	32.3	32.7	44.4	29.3	32.3	35.1	34.1	34.4	35.3
camel-1.6	14.1	28.8	6.0	33.0	33.7	39.1	37.2	33.1	38.8	33.3	42.7	37.4	36.8	37.7	37.3	40.5
ivy-1.4	0.0	0.0	0.0	8.0	22.2	8.0	0.0	22.9	20.0	18.8	21.1	0.0	14.8	15.4	16.7	17.1
ivy-2.0	14.3	27.3	34.5	41.2	44.4	50.0	47.6	30.0	52.4	48.8	46.2	23.3	16.7	36.7	43.1	34.4
jedit-4.0	41.2	45.0	45.0	47.4	42.9	46.5	46.4	46.2	49.0	50.9	49.2	49.3	41.8	45.3	50.7	50.9
jedit-4.1	63.6	51.0	65.2	65.2	66.7	67.9	51.7	67.9	63.5	65.5	63.5	51.2	61.1	61.5	61.5	60.0
jedit-4.2	0.0	64.5	30.0	40.0	43.2	35.0	45.5	28.0	38.3	38.3	36.7	32.8	25.5	36.8	40.0	35.9
log4j-1.0	0.0	0.0	73.7	58.8	58.8	73.7	64.3	63.6	63.6	63.6	74.1	44.4	73.7	66.7	63.6	78.3
pbeans2	0.0	0.0	0.0	25.0	0.0	28.6	33.3	33.3	33.3	33.3	25.0	25.0	26.7	25.0	0.0	46.2
redaktor	46.2	61.5	46.2	46.2	46.2	42.9	50.0	42.1	42.1	38.1	35.3	42.9	35.3	35.3	38.1	37.5
synapse-1.0	25.0	0.0	35.3	37.5	35.3	23.5	22.2	33.3	35.3	33.3	31.6	0.0	40.0	42.9	33.3	35.3
systemdata	0.0	0.0	0.0	28.6	0.0	28.6	26.7	0.0	28.6	44.4	20.0	37.5	22.2	50.0	22.2	22.2
tomcat	0.0	34.7	38.7	37.9	45.1	37.5	30.9	35.5	28.4	39.6	32.5	30.2	26.1	29.0	37.9	27.9
xerces-1.2	26.7	20.0	30.2	25.5	26.7	25.0	24.1	31.4	26.7	36.1	28.9	32.4	25.3	22.0	27.2	30.8
xerces-1.3	42.4	37.5	41.8	39.5	40.0	39.5	50.0	40.4	40.9	40.4	40.0	50.6	39.6	39.6	40.0	39.6

*** NONE-No Sampling | AD-ADASYN | BS-Borderline-SMOTE | MA- MAHAKIL | ROS- Random Oversampling | SM- SMOTE

Appendix B

Appendix

B.1 Appendix B: Process Metrics Data Used This Study (Chapter 8)

The static metric datasets are available on the PROMISE repository. The process metric datasets were manually extracted. The keywords used for matching the commits from the commit logs during the data extraction process are "errors, bugs, fix, fixes, issues, bugfixes, refactoring(s)". Below is a sample code used for the commit matching process.

```
[(bugs?|fix(es|ed)?)[\s:_#]*(\d+)],  
[#(\d+)(issue|bug)s?[\s#-]*(\d+)show_bug\.cgi\?id=(\d+)PR:?( \d+)].
```

Table B.1 presents the metrics and fault collection period for the extracted datasets. The process metric datasets are available at <http://analytics.jpn.org/SEdata/>

TABLE B.1: Summary of versions and collection period for the 10 OSS projects used.

Project Name	Summary of Version A (Older)		
	Version	Metrics recorded (Date)	Faults recorded (Date)
Clam Antivirus	0.96	2009/10/02 - 2010/03/31	2010/03/31 - 2010/09/27
eCos	2.0	2002/11/21 - 2003/05/20	2003/05/20 - 2006/05/04
Helma	1.3.1	2003/03/01 - 2003/08/28	2003/08/28 - 2004/02/24
NetBSD	4.0	2007/06/22 - 2007/12/19	2007/12/19 - 2008/06/16
OpenBSD	4.5	2009/01/31 - 2009/05/01	2009/05/01 - 2009/07/30
OpenCMS	7.0.0	2007/01/06 - 2007/07/05	2007/07/05 - 2008/01/01
OpenNMS	1.7.0	2008/10/17 - 2009/01/15	2009/01/15 - 2009/04/15
Scilab	5.2.0	2009/06/18 - 2009/12/15	2009/12/15 - 2010/06/13
Spring Security	2.0.0	2007/10/17 - 2008/04/14	2008/04/14 - 2008/10/11
XORP	1.0	2004/01/10 - 2004/07/08	2004/07/08 - 2005/01/04
Summary of Version B (New)			
Clam Antivirus	0.97	2010/08/11 - 2011/02/07	2011/02/07 - 2011/08/06
eCos	3.0	2008/10/01 - 2009/03/30	2009/03/30 - 2011/08/25
Helma	1.4.0	2003/10/04 - 2004/04/01	2004/04/01 - 2004/09/28
NetBSD	5.0	2008/10/31 - 2009/04/29	2009/04/29 - 2009/10/26
OpenBSD	4.6	2009/07/20 - 2009/10/18	2009/10/18 - 2010/01/16
OpenCMS	8.0.0	2010/11/09 - 2011/05/08	2011/05/08 - 2011/10/13
OpenNMS	1.8.0	2010/03/09 - 2010/06/07	2010/06/07 - 2010/09/05
Scilab	5.3.0	2010/06/18 - 2010/12/15	2010/12/15 - 2011/06/13
Spring Security	3.0.0	2009/06/25 - 2009/12/22	2009/12/22 - 2010/06/20
XORP	1.1	2004/10/15 - 2005/04/13	2005/04/13 - 2005/10/10

Appendix C

Publications

List of Major Publications

Journal Publications

1. (Chapter 7) Kwabena Ebo Bennin, Jacky Keung, Passakorn Phannachitta, Akito Monden, and Solomon Mensah. "**MAHAKIL: Diversity based Oversampling Approach to Alleviate the Class Imbalance Issue in Software Defect Prediction.**" IEEE Transactions on Software Engineering(TSE),2017. *Impact Factor: 3.27-2017*

2. (Chapter 1 and Chapter 8) Kwabena Ebo Bennin, Jacky Keung, and Akito Monden. "**On the relative value of data re-sampling approaches for Software Defect Prediction.**" Empirical Software Engineering (EMSE),2018. *Impact Factor: 3.27-2017*

Conference Publications

1. (Chapter 6) Kwabena Ebo Bennin, Jacky Keung, Akito Monden, Passakorn Phannachitta, and Solomon Mensah. "**The**

- significant effects of data sampling approaches on software defect prioritization and classification."** In Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 364-373. IEEE Press, 2017. Acceptance Rate: 19%
2. (Chapter 5) Kwabena Ebo Bennin, Jacky Keung, Akito Monden, Yasutaka Kamei, and Naoyasu Ubayashi. "**Investigating the effects of balanced training and testing datasets on effort-aware fault prediction models.**" In Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual, vol. 1, pp. 154-163. IEEE, 2016.
 3. (Chapter 3) Kwabena Ebo Bennin, Koji Toda, Yasutaka Kamei, Jacky Keung, Akito Monden, and Naoyasu Ubayashi. "**Empirical Evaluation of Cross-Release Effort-Aware Defect Prediction Models.**" In Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on, pp. 214-221. IEEE, 2016.
 4. (Chapter 8) Kwabena Ebo Bennin, Jacky Keung, and Akito Monden. "**Impact of the Distribution Parameter of Data Sampling Approaches on Software Defect Prediction Models.**" In Asia-Pacific Software Engineering Conference (APSEC), 2017 24th, pp. 630-635. IEEE, 2017.

Other relevant publications produced during the research period.

Journal Publications

1. Limsettho, Nachai, Kwabena Ebo Bennin, Jacky W. Keung, Hideaki Hata, and Kenichi Matsumoto. "**Cross Project Defect Prediction Using Class Distribution Estimation and Oversampling.**" Information and Software Technology (2018). *Impact Factor: 2.69-2017*
2. Yu, Xiao, Man Wu, Yiheng Jian, Kwabena Ebo Bennin, Mandi Fu, and Chuanxiang Ma. "**Cross-company defect prediction via semi-supervised clustering-based data filtering and MStrA-based transfer learning.**" Soft Computing (2018): 1-12. *Impact Factor: 2.47-2017*
3. Shi, Zhendong, Jacky Keung, Kwabena Ebo Bennin, and Xingjun Zhang. "**Comparing learning to rank techniques in hybrid bug localization.**" Applied Soft Computing 62 (2018): 636-648. *Impact Factor: 3.54-2017*
4. Xiao, Yan, Jacky Keung, Kwabena Ebo Bennin, and Qing Mi. "**Machine translation-based bug localization technique for bridging lexical gap.**" Information and Software Technology (2018). *Impact Factor: 2.69-2017*