A MUTATION-BASED APPROACH TO ALLEVIATE THE CLASS IMBALANCE PROBLEM IN SOFTWARE DEFECT PREDICTION

A Thesis Submitted to
the Graduate School of Engineering and Sciences of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Dinçer GÜNER

June 2023
İZMİR

We approve the thesis of **Dinçer GÜNER**

Examining Committee Members:

Prof. Dr. Onur DEMİRÖRS
Department of Computer Engineering, Izmir Institute of Technology

Assoc. Prof. Dr. Tuğkan TUĞLULAR
Department of Computer Engineering, Izmir Institute of Technology

Prof. Dr. Geylani KARDAŞ
International Computer Institute, Ege University

19 June 2023

Prof. Dr. Onur DEMİRÖRS
Supervisor, Department of Computer Engineering
İzmir Institute of Technology

Dr. Görkem GİRAY
Co-supervisor, Independent Researcher

Prof. Dr. Cüneyt F. BAZLAMAÇCI
Head of the Department of Computer Engineering

Prof. Dr. Mehtap EANES
Dean of the Graduate School of Engineering and Sciences

ACKNOWLEDGMENTS

I would like to thank …

…my supervisor Prof. Dr. Onur Demirörs, not only for his experience and excellent guidance but also for his continued trust and patience through my master’s study.

…my co-supervisor Dr. Görkem Giray, for his support, valuable contributions, and encouragement through this journey. He has been inspiring for me.

…my family for their endless support throughout my whole life.

ABSTRACT

A MUTATION-BASED APPROACH TO ALLEVIATE THE CLASS IMBALANCE PROBLEM IN SOFTWARE DEFECT PREDICTION

Highly imbalanced training datasets considerably degrade the performance of software defect predictors. Software Defect Prediction (SDP) datasets have a general problem, which is class imbalance. Therefore, a variety of methods have been developed to alleviate Class Imbalance Problem (CIP). However, these classical methods, like data-sampling, balance datasets without connecting any relation with SDP. Over-sampling techniques generate synthetic minor class instances, which generalize a small number of minor class instances and result in less diverse instances, whereas under-sampling techniques eliminate major class instances, resulting in significant information loss. In this study, we present an approach that uses software mutations to balance software repositories. Mutation-based Approach (MBA) injects mutants into defect-free instances, causing them to transform into defective instances. In this way, MBA balances datasets with diverse data produced by mutation operators, and there is no loss on instances as in under-sampling.

For recall scores, almost all rebalancing methods outperformed Baseline in Inter-release Defect Prediction (IRDP) scenario but only MBA significantly outperformed Baseline in Cross-project Defect Prediction (CPDP) scenario. The performance increase in recall resulted in the production of more false alarms. We can not generalize that MBA outperforms Baseline and the five over-sampling strategies in terms of AUC scores. In terms of recall values, the MBA performed better in CPDP than IRDP.

For both IRDP and CPDP scenarios, there were significant and positive correlations between SMC (the change percentage of software measures) and recall, and SMC and false alarm but there was no significant correlation between SMC and AUC.

ÖZET

YAZILIM HATA TAHMİNİNDE SINIF DENGESİZLİK PROBLEMİNİ AZALTMAK İÇİN MUTASYON TABANLI BİR YAKLAŞIM

Yüksek düzeyde dengesiz eğitim veri kümeleri, yazılım hatası tahmin edicilerinin performansını önemli ölçüde düşürür. Yazılım Hata Tahmini (SDP) veri kümelerinde genel olarak bulunan problem sınıf dengesizliğidir. Bu nedenle, Sınıf Dengesizliği Probleminin (CIP) getirdiği zorluğu hafifletmek için çeşitli yöntemler geliştirilmiştir. Bununla birlikte, veri örnekleme gibi klasik yöntemler, veri kümelerini SDP ile bir bağlantı kurmadan dengeler. Aşırı örnekleme teknikleri, az sayıda küçük sınıf örneğini genelleştiren ve daha az çeşitli örneklerle sonuçlanan sentetik küçük sınıf örnekleri üretirken, yetersiz örnekleme teknikleri, önemli bilgi kaybına neden olan büyük sınıf örneklerini ortadan kaldırır. Bu çalışmada, yazılım depolarını dengelemek için yazılım mutasyonlarını kullanan bir yaklaşım sunduk. Mutasyon Tabanlı Yaklaşım (MBA), mutantları hatasız örneklere enjekte ederek hatalı örneklere dönüşmelerine neden olur. Bu şekilde MBA, veri kümelerini mutasyon operatörleri tarafından üretilen çeşitli verilerle dengeler ve düşük örneklemede olduğu gibi örneklerde kayıp olmaz.

Duyarlılık (recall) puanlarına göre, Çapraz Versiyon Hata Tahmini (IRDP) senaryosu için hemen hemen tüm yeniden dengeleme yöntemleri Baseline’dan daha iyi bir performans gösterirken yalnızca MBA, Çapraz Proje Hata Tahmini (CPDP) senaryosunda Baseline’dan daha iyi bir performans gösterdi. Duyarlılık puanlarındaki performans artışı daha fazla yanlış alarm üretilmesiyle sonuçlandı. AUC puanlarına göre MBA’nın Baseline’den ve beş aşırı örnekleme yönteminden daha iyi performans gösterdiğini genelleyemeyiz. Duyarlılık değerleri açısından; MBA, IRDP senaryosunda CPDP senaryosundan daha iyi performans gösterdi.

Hem CPDP senaryosunda hem de IRDP senaryosunda, SMC (yazılım ölçülerindeki değişim yüzdesi) ile duyarlılık, ve SMC ile yanlış alarm aralarında anlamlı ve pozitif bir korelasyon mevcuttur ama SMC ile AUC arasında anlamlı ve pozitif bir korelasyon mevcut değildir.

TABLE OF CONTENTS

LIST OF FIGURES viii

LIST OF TABLES x

CHAPTER 1. INTRODUCTION 1

1.1. Software Defect Prediction 2

1.2. Class Imbalance Problem 6

1.3. Sampling Approaches for Class Imbalance Problem 6

1.3.1. Under-Sampling 8

1.3.2. Over-Sampling 8

1.4. Problem Statement 9

1.5. Objective and Research Questions 10

1.6. Research Approach 10

1.7. Overview 12

CHAPTER 2. LITERATURE REVIEW 13

CHAPTER 3. PROPOSED APPROACH 15

3.1. Choice of Mutation Tool 16

3.2. Balancing Software Repository with Mutants 18

CHAPTER 4. EXPERIMENTAL DESIGN 19

4.1. Dataset 20

4.2. Software Measures Calculation 26

4.3. Data Preprocessing 35

4.4. Hyperparameter Tuning 36

4.5. Performance Measure 38

4.6. Summary of Experimental Designs 40

4.6.1. Baseline Experimental Design 41
4.6.2. Over-sampling-based Experimental Design.................................. 42
4.6.3. Mutation-based Experimental Design 43
4.7. Performance Comparison 44

CHAPTER 5. RESULTS AND DISCUSSION.. 45
5.1. Performance Evaluation of Rebalancing Methods for IRDP Scenario46
5.2. Performance Evaluation of Rebalancing Methods for CPDP Scenario 52
5.3. Stability of MBA for IRDP Scenario 59
5.4. Stability of MBA for CPDP Scenario 65
5.5. General Discussion.. 70
5.6. Threats to Validity 74

CHAPTER 6. CONCLUSION AND FUTURE WORK 76

REFERENCES 79

APPENDICES
APPENDIX A. IRDP SCENARIO: AUC, PD AND PF VALUES OF BASELINE AND DIFFERENT DEFECT LEVELS (0.3, 0.4, AND 0.5 DEFECT RATIO) OF MBA FOR EACH DATASET 92
APPENDIX B. IRDP SCENARIO: NUMBER OF CHANGED MEASURES ON DIFFERENT DEFECT LEVELS (0.3, 0.4, AND 0.5 DEFECT RATIO) OF MBA FOR EACH DATASET 97
APPENDIX C. CPDP SCENARIO: NUMBER OF CHANGED MEASURES ON DIFFERENT DEFECT LEVELS (0.3, 0.4, AND 0.5 DEFECT RATIO) OF MBA FOR EACH DATASET 100
APPENDIX D. CPDP SCENARIO: NUMBER OF CHANGED MEASURES ON DIFFERENT DEFECT LEVELS (0.3, 0.4, AND 0.5 DEFECT RATIO) OF MBA FOR EACH DATASET 117

LIST OF FIGURES

Figure **Page**

Figure 1. SDP Process (Source: Giray et al., 2023) 2
Figure 2. WRDP process 3
Figure 3. IRDP process 4
Figure 4. CPDP process 5
Figure 5. Differences between under-sampling and over-sampling (Source: Robles Velasco et al., 2021) 7
Figure 6. Methodological approach 11
Figure 7. MBA for CIP 15
Figure 8. Balancing process with mutants 18
Figure 9. Experimentation process 19
Figure 10. Comparison of software measures presented in Jureczko's study and calculated in our experiment with ckjm-extended 2.1 on ant 1.3 32
Figure 11. Comparison of software measures presented in Jureczko's study and calculated in our experiment with ckjm-extended 2.2 on ant 1.3 32
Figure 12. Comparison of software measures presented in Jureczko's study and calculated in our experiment with ckjm-extended 2.3 on ant 1.3 33
Figure 13. Comparison of software measures calculated by Java SDK 1.6 and calculated by Java SDK 1.7 with ckjm-extended 2.2 on ant 1.3 34
Figure 14. Baseline experimental design 41
Figure 15. Over-sampling-based experimental design 42
Figure 16. Mutation-based experimental design 43
Figure 17. IRDP Scenario: Quartile plots of performance measures (AUC, pd, and pf) for Baseline, MBA, over-sampling techniques per each ML algorithm 47
Figure 18. IRDP Scenario: Wilcoxon test win-tie-loss comparison of MBA vs. Baseline, SMOTE, ROS, SMOTE Nominal, Borderline-SMOTE, SVM SMOTE across all datasets per each ML algorithm and performance measures (AUC, pd, and pf) 48
Figure 19. CPDP Scenario: Quartile plots of performance measures (AUC, pd, and pf) for Baseline, MBA, over-sampling techniques per each ML algorithm 53

Figure 20. CPDP Scenario: Wilcoxon test win-tie-loss comparison of MBA vs. Baseline, SMOTE, ROS, SMOTE Nominal, Borderline-SMOTE, SVM SMOTE across all datasets per each ML algorithm and performance measures (AUC, pd, and pf)........ 55
Figure 21. IRDP Scenario: Quartile plots of performance measures (AUC, pd, and pf) for Baseline, MBA 0.3, MBA 0.4, and MBA 0.5 for each ML algorithm........ 61
Figure 22. IRDP Scenario: The scatter plots of performance measures (AUC, pd, and pf) for MBA 0.3, MBA 0.4, and MBA 0.5 for each ML algorithm........ 64
Figure 23. CPDP Scenario: Quartile plots of performance measures (AUC, pd, and pf) for Baseline, MBA 0.3, MBA 0.4, and MBA 0.5 for each ML algorithm........ 66
Figure 24. CPDP Scenario: The scatter plots of performance measures (AUC, pd, and pf) for MBA 0.3, MBA 0.4, and MBA 0.5 for each ML algorithm........ 69

LIST OF TABLES

Table **Page**

Table 1. Sampling techniques for SDP 7
Table 2. Mutation operators implemented in Major 17
Table 3. Public datasets for SDP 20
Table 4. Selected projects and versions in our dataset 21
Table 5. Defect ratios of the projects 23
Table 6. Training and testing dataset pairs used in experiments 24
Table 7. Software measures definitions (Source: Jureczko & Spinellis, 2010) 26
Table 8. Software measures of four instances of Ant 1.3 35
Table 9. Tuned hyperparameters for ML models 37
Table 10. Confusion Matrix for Binary Classification (Source: Moussa & Sarro, 2022) 39
Table 11. The definition of the performance measures (Source: Moussa & Sarro, 2022) 40
Table 12. IRDP Scenario: Rankings for performance measures in terms of wins (W), losses (L), wins-losses (W-L), and ties (T) 50
Table 13. IRDP Scenario: Median values for AUC, pd, and pf for each dataset 51
Table 14. CPDP Scenario: Rankings for performance measures in terms of wins (W), losses (L), wins-losses (W-L), and ties (T) 57
Table 15. CPDP Scenario: Median values for AUC, pd, and pf for each dataset 58
Table 16. IRDP Scenario: Kendall's Tau correlation analysis between SMC and each performance measure per each ML algorithm 63
Table 17. CPDP Scenario: Kendall's Tau correlation analysis between SMC and each performance measure per each ML algorithm 68

CHAPTER 1

INTRODUCTION

Software quality assurance is the process of observing the software development process in order to achieve the expected software quality at the lowest possible cost. Formal code inspections, code reviews, software testing, and Software Defect Prediction (SDP) may all be used to improve quality (Rathore and Kumar 2019). In software engineering, identifying buggy code sections is a crucial task to enable the development of better-quality software. Decreasing the number of defects or locating some of the defective components before testing and production pipelines may lead to significant resource saving. Hence, it would be possible to utilize an organization's resources, increasing its profit. The potential use of SDP models to identify defective software modules from the beginning of the software development life cycle has generated a lot of interest over the past 20 years (Rathore and Kumar 2019). Previously, studies on SDP used a variety of Machine Learning (ML) techniques to predict buggy software modules. According to the findings of these studies, the techniques did not perform as well as expected, and the suitability of the techniques has been another research topic for SDP (Catal 2011). SDP models had an accuracy of between 70% and 85% but produced more false alarms (Venkata et al. 2006; Elish and Elish 2008; Guo et al. 2003). One of the main problems related to SDP is Class Imbalance Problem (CIP), which causes biased SDP models (Menzies et al. 2010). Data sampling techniques are commonly used to solve the CIP (Bennin et al. 2018). Generally, data sampling approaches are weak at increasing data variety in nature (Chawla et al., 2002; Han Hui and Wang, 2005; H. He et al., 2008). SDP models built with less diverse data perform poorly. In this study, we proposed a Mutation-based Approach (MBA) to alleviate CIP in SDP by injecting software mutants into defect-free modules, which are generally the majority of software prediction datasets. As a result, we were able to balance SDP datasets with highly diverse transformed defective instances, potentially improving SDP model performance.

1.1. Software Defect Prediction

SDP aims to predict software blocks with defects. SDP process is modeled in Figure 1. Firstly, software attributes are required for the prediction process in order to create an ML model or do statistical analysis. These attributes are extracted from a software repository. Software attributes are features such as lines of code, the number of people who contributed to the project, images of code blocks, etc. Software attributes are used to build prediction models. We can identify whether any new software blocks are defective using the prediction model.

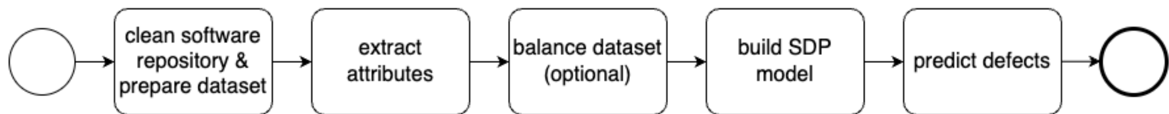

Figure 1. SDP Process (Source: Giray et al., 2023)

As a software attribute, software measures are commonly used features in SDP. Software measures are numerical values used to assess the quality of a code partition quantitatively. In the literature, software measures are studied in two groups: product measures and process measures.

Product measures are attributes of a software repository that are extracted from source codes. These attributes identify a snapshot of the project. Source code is inspected from the perspective of some features like lines of code, complexity, functional aggregation, inheritance, etc. (Subramanyam and Krishnan 2003; Nagappan, Ball, and Zeller 2006; Gyimothy, Ferenc, and Siket 2005). Product measures are different from process measures in that they do not include information about the history of development.

Process measures are attributes extracted from historical information about projects. Process measures can be derived from a source code management system. For instance, the number of code additions and deletions, the number of different developers, the number of modified lines, etc. (Nagappan and Ball 2005; Moser, Pedrycz, and Succi 2008; Hassan 2009).

SDP is divided into three types of scenarios: within-release (intra-release) defect prediction, inter-release (cross-release) defect prediction, and cross-project SDP (Rathore and Kumar 2019).

Within-release defect prediction (WRDP) refers to a scenario of prediction in which training and testing datasets belong to a specific version of a project. The same release is used for model building and performance evaluation.

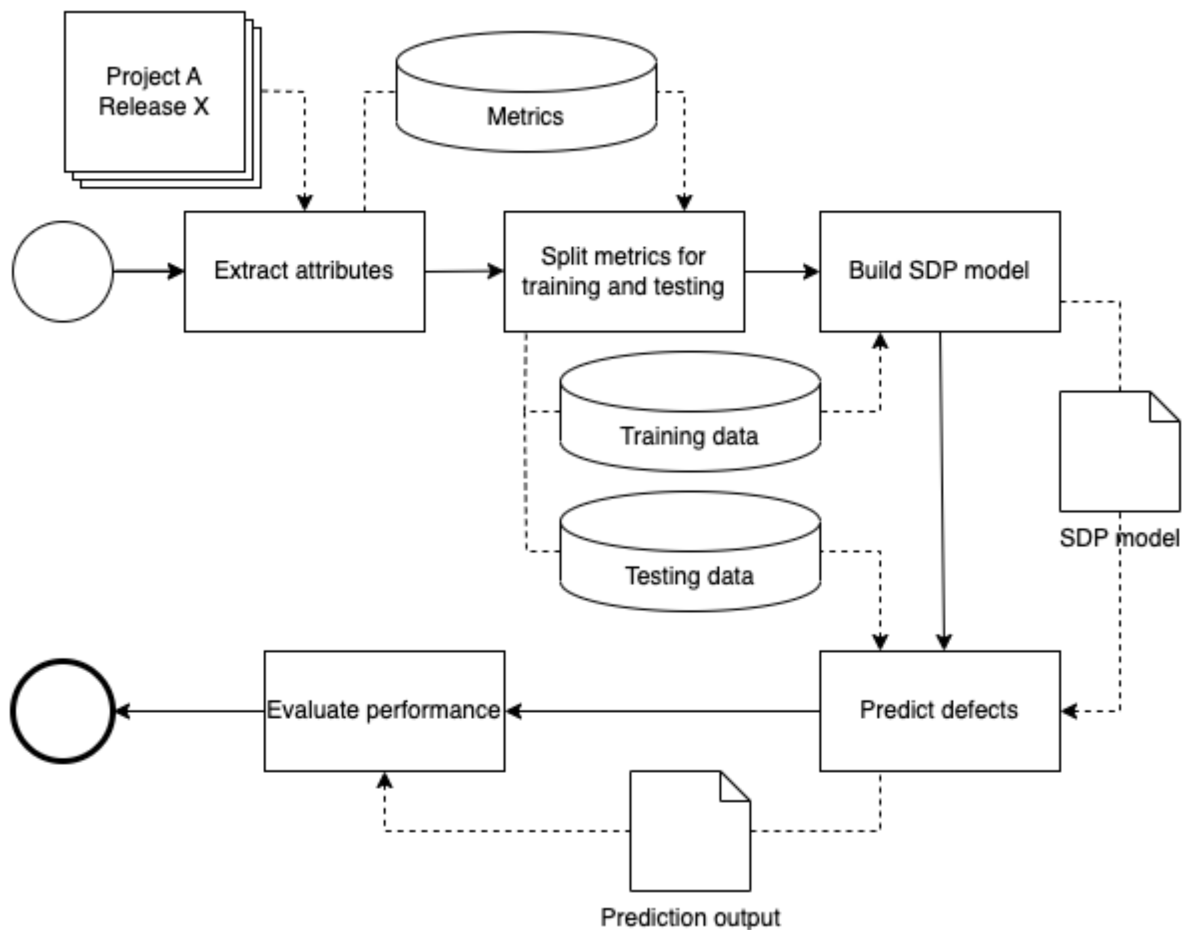

Figure 2. WRDP process

Inter-release defect prediction (IRDP) refers to a scenario of prediction in which the training dataset is chosen from previous releases of a project and the testing dataset is taken from a version of the project released after the one used for the training dataset.

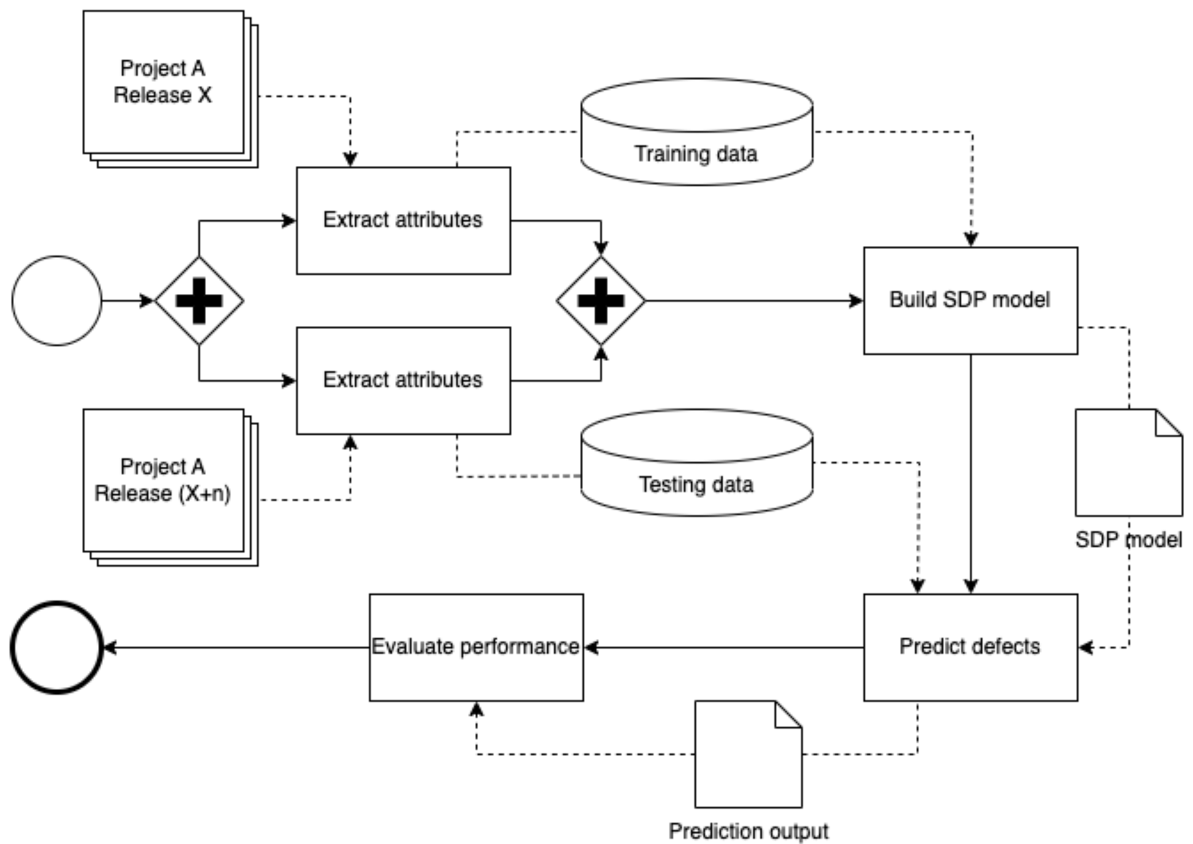

Figure 3. IRDP process

Cross-project defect prediction (CPDP) refers to a scenario of prediction in which the training dataset is created from different software projects and the testing data is created from different projects used for the training dataset.

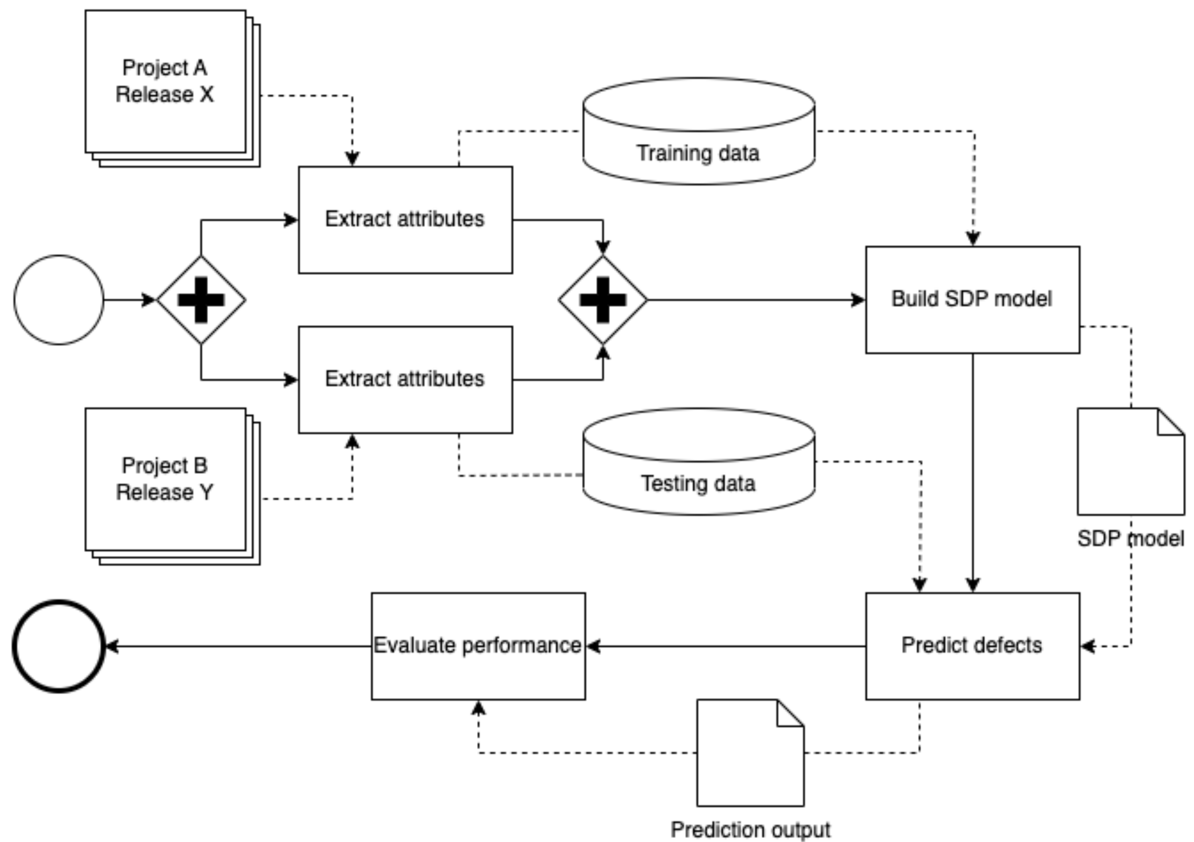

Figure 4. CPDP process

As shown in Figure 2, Figure 3, and Figure 4, for all scenarios, the SDP model is the most important component of the SDP process because all efforts are made to make better predictions, and choosing the right prediction technique is critical to building a better SDP model. Prediction techniques can be categorized as Supervised, Semi-supervised, and Unsupervised. Supervised techniques require a dataset with all instances labeled. The model draws boundaries or defines prediction methodologies with respect to the labels of training instances, and any new testing instances are assigned the label of the training instances to which they are most similar. Supervised techniques are investigated under two groups, which are classification and regression. Classification is the term used to describe a procedure whose output is a category. Regression is the name of the procedure when the result is a continuous variable. For the SDP, if a technique answers whether a software block is defective or not, it is classification; if it responds to how many defects are in a software block, it is regression. Unsupervised techniques do not need a labeled dataset. Unsupervised techniques group dataset instances with respect to the features. Semi-supervised techniques combine a small set of labeled instances with

many unlabeled instances for model building. A model is first built using a small, labeled set, and then unlabeled data is labeled using this model. All instances are used to build a new model that produces better outcomes. In most studies, researchers prefer to use supervised techniques for SDP because supervised techniques perform better than other techniques. Many prediction techniques have the fundamental problem of assuming that all classes in a dataset are equally balanced (Weiss and Provost 2001; Yoon and Kwek 2007). Therefore, prediction models that are trained with imbalanced datasets usually produce inaccurate results (Provost 2008). As a result, CIP is widely acknowledged as one of the main reasons why SDP algorithms underperform (Hall et al. 2012; Arisholm, Briand, and Johannessen 2010).

1.2. Class Imbalance Problem

CIP indicates an unbalanced distribution of a dataset. Major classes refer to those that have more instances than other classes, and minor classes refer to those that have fewer instances than other classes. In SDP, most of the dataset has CIP, and generally, the major class is defect-free whereas the minor class is not (Sayyad Shirabad and Menzies 2005). The prediction techniques commonly fail to identify the minority defective components when predicting the occurrence of software defects if the major class is non-defective. ML models are biased when they are built with an imbalanced dataset. Menzies et al. highlighted that the performances of prediction techniques can be enhanced by using data sampling techniques, as software defect datasets are extremely prone to the CIP (Menzies et al. 2007). In the literature, many methods have been studied to solve CIP. These approaches are evaluated in the subsections on over-sampling and under-sampling, which are covered in Section 1.3.

1.3. Sampling Approaches for Class Imbalance Problem

Data sampling is the process of producing or reducing some of the instances of a class in a dataset. As shown in Figure 5, there are two types of sampling: under-sampling and over-sampling. Under-sampling methods eliminate samples, while over-sampling methods add samples in the reverse way of under-sampling.

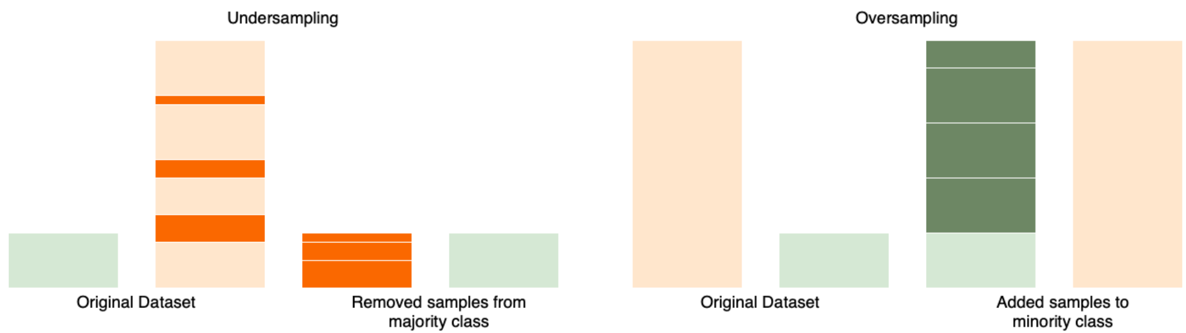

Figure 5. Differences between under-sampling and over-sampling (Source: Robles Velasco et al., 2021)

Sampling is a very common method to solve CIP in SDP (Bennin et al. 2018). There are many sampling approaches. Each technique has a different feature and serves a different purpose. These techniques are used to organize the data distribution. The organization of data distribution is important, but so is the quality of the instances. In the following sub-sections, we discussed some of the most commonly used sampling methods in SDP, which are stated in Table 1.

Table 1. Sampling techniques for SDP

Under-sampling	**Over-sampling**
Near-Miss (NM)	Synthetic Over-sampling Technique (SMOTE)
Instance Hardness Threshold (IHT)	Borderline-SMOTE
Cluster Centroids (CC)	Support Vector Machines (SVM) SMOTE
Random Under-sampling (RUS)	SMOTE Nominal
	Random Over-sampling (ROS)

1.3.1. Under-Sampling

Under-sampling approaches delete instances belonging to major classes from the dataset until the class distribution is fixed to alleviate CIP. Under-sampling causes reduced information. In the literature, many under-sampling techniques have been proposed. Random Under-sampling technique selects a major sample and deletes it until the class distribution is fixed. ROS can not manage which data sample is deleted. It's possible that a more representative data sample is removed, and crucial data is lost in the dataset. Under-sampling methods differ in terms of the selection of instances to be deleted. Preserving valuable data for learning is very important. Zhang and Mani present Near Miss Under-sampling algorithm that uses K-nearest neighbors' algorithm to select major instances to delete with respect to their distance to minor class instances (J. Zhang and Mani 2003). Smith, Martinez, and Giraud-Carrier provide an algorithm called Instance Hardness Threshold in which a classifier is trained on the original dataset, and the major class instances with low probabilities are deleted (Smith, Martinez, and Giraud-Carrier 2014). Cluster Centroids algorithm was proposed by Yen and Lee as a method for under-sampling the majority class by replacing a cluster of majority instances with the cluster centroid of a K-Means algorithm (Yen and Lee 2006).

1.3.2. Over-Sampling

Over-sampling creates new instances that belong to minor classes in the dataset until the class distribution is fixed to balance the dataset. Generating new instances from a small number of samples causes overfitting which may result in poorer prediction performance. In the literature, many over-sampling techniques have been proposed. Over-sampling methods differ in terms of instance generation logic. Strengthening class boundaries, reducing over-fitting, and improving discriminating were all subjects that were taken into consideration to improve the effectiveness of over sampling strategies (Johnson and Khoshgoftaar 2019). Random Over-sampling (ROS) technique randomly selects a minor sample and clones it until the class distribution is fixed. This approach creates samples that are identical to one another, and the ML model starts to overfit specific samples (Van Hulse, Khoshgoftaar, and Napolitano 2007). Synthetic Over-

sampling Technique (SMOTE) was introduced by Chawla et al. (Bowyer et al. 2011). SMOTE generates new minor class instances by interpolating minor instances and their nearest minor class neighbors. SMOTE method has several variants, such as Borderline-SMOTE (Han Hui and Wang, 2005), which applies SMOTE with borderline samples; Support Vector Machines (SVM) SMOTE (Nguyen, Cooper, and Kamei 2011), which applies SMOTE with samples detected with SVM; and SMOTE Nominal (Bowyer et al. 2011), which expects that the data being resampled only contains categorical features.

1.4. Problem Statement

To solve the CIP in SDP, several sampling methods have been used. We discussed some of the sampling methods in section 1.3. The main difficulty with the sampling strategies is creating a balanced dataset with the proper instances to train better defect predictors. Software measures are accepted by sampling methods as a collection of numbers with no explanation of what these numbers mean. These methods balance datasets by using these numbers without connecting any relation with SDP. Even if the numbers come from a different domain, the sampling methods produce the same results. This situation can be interpreted as sampling methods being general-purpose methods that can be applied to any problem. Lack of domain knowledge, on the other hand, reduces the performance of ML models. Because of that, rather than using over-sampling techniques to alleviate CIP, we injected software mutants into defect-free code instances. Therefore, instead of over-sampling for defective classes by using these techniques, we try to create a domain-specific rebalancing method by adding real faults to the dataset. We used software mutation tools to generate real faults. As described in chapter 3, software mutation tools are typically used to improve software testing suites, but we used them to solve CIP in SDP. Since MBA is specific to the SDP domain, we predict that it can improve the performance of ML models. MBA transforms defect-free instances into defective instances by using software mutation operators different from sampling methods, as shown in Figure 9, and CIP is solved by injecting mutants into the defect-free instances as detailed in chapter 3. In order to observe the impact of MBA on software measures, we also calculated the change percentages of software measures (SMC) for datasets, and we detailed SMC in section 5.3.

1.5. Objective and Research Questions

The primary objective of this study is to propose an effective approach to alleviate CIP in SDP.

The research questions of this thesis work are:

RQ1: Does the proposed MBA improve performance over existing over-sampling approaches and Baseline on IRDP?

RQ2: Does the proposed MBA improve performance over existing over-sampling approaches and Baseline on CPDP?

RQ3: How does the change percentage of software measures (SMC) affect performance of MBA on IRDP?

RQ4: How does the change percentage of software measures (SMC) affect performance of MBA on CPDP?

1.6. Research Approach

The methodological approach consists of three stages:

- Literature review, which includes literature surveys about SDP and CIP in SDP, such as SDP scenarios, used techniques to build models, used software measures, suitable performance measures for SDP, and applications.
- Development of rebalancing techniques, which includes the commonly used five over-sampling techniques in the literature, MBA, and Baseline.
- Comparison of rebalancing techniques, which includes three experimental designs (detailed in section 4.6), and statistical analysis of performance measures (detailed in section 4.7).

A graphical representation of the methodological approach is shown in Figure 6.

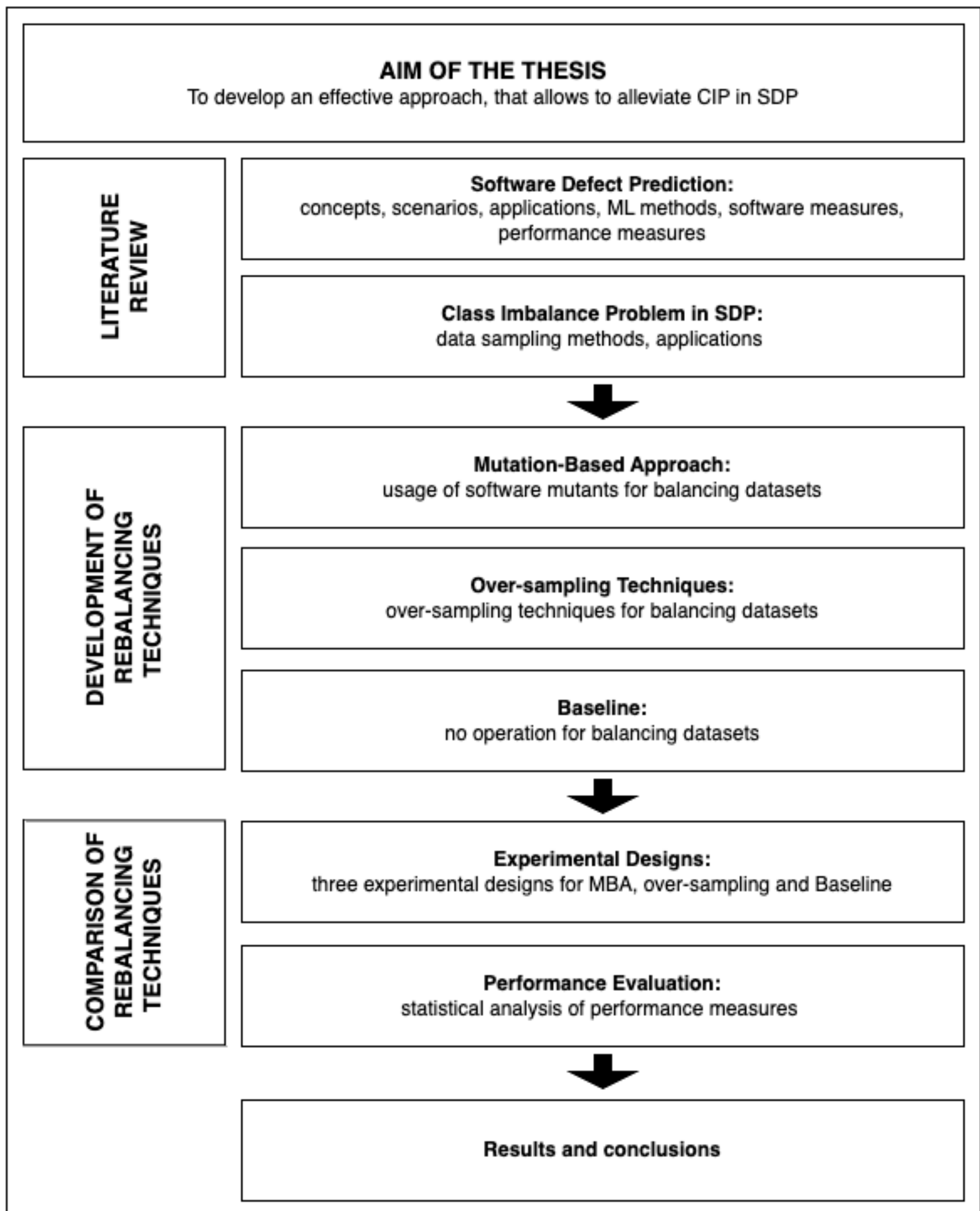

Figure 6. Methodological approach

1.7. Overview

The thesis consists of 6 main chapters in addition to appendices, Organization of the chapters follows as:

- Chapter 1 consists of the introduction, which includes background information, the motivation of the study, and a methodological approach that explains the steps on which this thesis was founded.
- Chapter 2 provides a literature review about SDP. Because class imbalance is a widespread issue with SDP datasets, the research's primary aim has shifted from improving SDP models to balancing the SDP datasets.
- Chapter 3 presents MBA and a mutation tool (Major), as well as the operations of the mutation tool and how a dataset is balanced with MBA.
- Chapter 4 focuses on the experimental design of our study, which includes dataset preparation, software measure calculation, data preprocessing, hyperparameter tuning of ML methods, performance measures, and performance comparison of rebalancing techniques.
- Chapter 5 covered the performance evaluation results of rebalancing techniques, the stability of MBA, and threats to validity.
- In Chapter 6, contributions and the future work of our study are given.

CHAPTER 2

LITERATURE REVIEW

Munson and Khoshgoftaar suggested utilizing prediction models to help in finding program defects after noticing using simple discriminant analysis that there is a significant helpful association between software defects and software complexity measures during development (Munson and Khoshgoftaar 1992). Many conventional classification techniques, such as tree-based techniques (Menzies, Greenwald, and Frank 2007; Guo et al. 2004; Song et al. 2011), analogy-based strategies (Taghi M. Khoshgoftaar and Seliya 2003; Emam et al. 2001), neural networks (Quah and Thwin 2003; T. M. Khoshgoftaar et al. 1997), and Bayes methods (Bouguila, Wang, and Ben Hamza 2008; Turhan and Bener 2009), have been used in SDP. Random Forest (RF) has been suggested in some of the studies because it is simple, quick to train, and more resilient (Monden et al. 2013; Lessmann et al. 2008).

The performance of Naïve Bayes (NB) with a log-filtering preprocessor was empirically demonstrated to be better compared to that of tree-based learning methods by Menzies et al. (Menzies, Greenwald, et al., 2007). They also asserted that the choice of learning method is significantly more crucial than the selection of the data subset to be used for learning. Nevertheless, with a large-scale empirical study, Lessmann et al. (Lessmann et al. 2008) came to an inconsistent conclusion, suggesting that the significance of a particular learning technique may be less than the dataset used for training, as Menzies et al. (Menzies, Greenwald, and Frank 2007) previously stated, and that there were no significant differences between 17 classification techniques. Moreover, Song et al. (Song et al. 2011) proposed a more comprehensive and trustworthy study for SDP in response to Menzies et al.'s (Menzies, Greenwald, and Frank 2007) argument that the study may be biased. Many researchers stated the success of ML techniques such as SVM (Kumar et al. 2018), DT (Y. Zhang et al. 2018), and neural networks (Miholca, Czibula, and Czibula 2018) in SDP (Özakıncı and Tarhan 2018; Goyal and Bhatia 2020; Erturk and Sezer 2015; Rathore and Kumar 2019). The primary objective of previous studies has been to improve the performance of SDP. These studies have considered a variety of methodologies, such as model enhancement, making use of powerful feature

selection, proposing preprocessing methods, and suggesting different measures. However, all these studies resulted in suboptimal solutions.

The success of SDP models worsens when the dataset is imbalanced (Haixiang et al. 2017; Galar et al. 2012; L. Chen et al. 2018). Data resampling techniques have been used to alleviate CIP. There are alternatives to resampling, such as setting weights to the cost of the classes to reduce misclassification costs (Sun et al. 2007; Pazzani et al. 1994; Domingos 1999), and adoption of ensemble methods (Wong, Leung, and Ling 2013; Laradji, Alshayeb, and Ghouti 2015). Resampling is becoming more and more popular because it is simple to separate from the prediction model and easy to see the impacts on prediction performance. Researchers are focused on improving the selection of instances to be removed (Tsai et al. 2019; Vuttipittayamongkol and Elyan 2020; Rao and Reddy 2020; Goyal 2022) and to be used to generate new instances (Qu et al., 2022; Rekha G. and Shailaja, 2022) for under-sampling and over-sampling, respectively. Resampling outperforms other methods, considering recent studies (Bennin et al. 2018). We concentrated on resampling methods, and we provided a review of various resampling studies in section 1.3. We found MBA is more comparable with over-sampling methods because MBA increases defective instances as in over-sampling methods. According to several studies, over-sampling is preferred to under-sampling (Shanab et al. 2012; García, Sánchez, and Mollineda 2012; Japkowicz and Stephen 2002). Under-sampling eliminates some of the instances and causes information loss, so the SDP model does not include the necessary instances to make better predictions. Thus, we considered only over-sampling methods in our study.

CHAPTER 3

PROPOSED APPROACH

In this chapter, we have presented the proposed approach, which is a mutation-based solution for CIP in SDP. We explained how software mutation, which is generally used for software mutation testing, is used for CIP in SDP.

Software mutation is created by injecting artificial faults. These faults, or mutants, are used to evaluate testing methodologies such as fault-finding methods, input value generation models, and oracle solutions (Just, Schweiggert, and Kapfhammer 2011). In this study, we used software mutants to balance imbalanced datasets in SDP. As shown in Figure 7, we applied software mutants to some of the parts of the defect-free code blocks. In this way, the number of defective instances increased while the number of defect-free instances decreased.

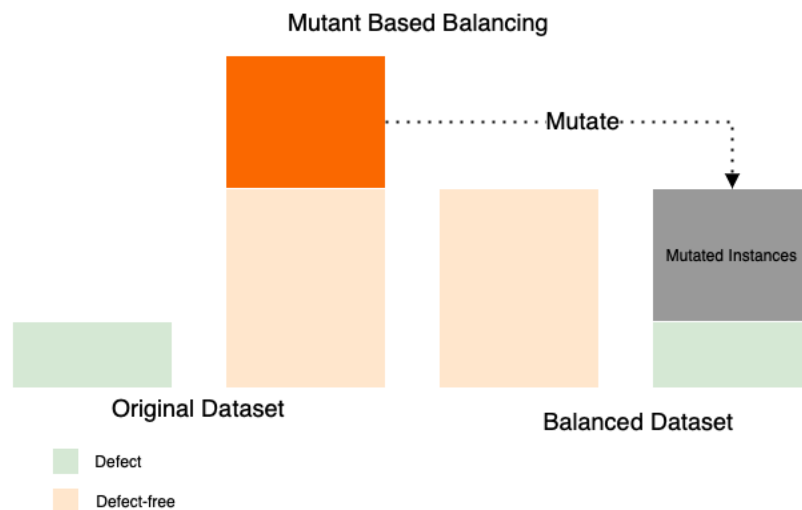

Figure 7. MBA for CIP

Mutants are produced by a tool or framework using mutation operators, which define the type of mutation, such as changing arithmetic or logical operators, modifying conditional operators, or deleting statements (Just et al. 2014). The performance of

mutation is commonly studied and compared with hand-seeded defects and real defects (Andrews, Briand, and Labiche 2005; Just et al. 2014; Namin and Kakarla 2011). The selection of mutation operators is very critical because increasing the similarity of mutants with real-life bugs supports the validity of the testing system.

3.1. Choice of Mutation Tool

Numerous tools and frameworks have been proposed for mutation testing, such as Jumble, MuJava, Javalanche, and Major (Just, Schweiggert, and Kapfhammer 2011). Mutation tools differ with respect to their execution time, the number of available mutation operators, flexibility, and the degree of automation (Just, Schweiggert, and Kapfhammer 2011). In the literature, the following set of mutation operators are recommended: constant replacement, operator replacement, branch condition modification, and statement deletion (Jia and Harman 2011; Just, Kapfhammer, and Schweiggert 2012; Siami Namin, Andrews, and Murdoch 2008; Offutt et al. 1996). We decided to use Major as a mutation tool that covers all suggested mutation operators. Major is integrated into the compiler and does not require any other framework, so it is easy to use. Also, Major generates mutated source codes. We can manage the number of mutations and mutation variety with the source codes of mutations. We used the following mutation operator set listed in Table 2, which is generated by Major.

Table 2. Mutation operators implemented in Major

Mutation operator	**Example**
AOR (Arithmetic Operator Replacement)	$a + b \rightarrow a - b$
LOR (Logical Operator Replacement)	$a\hat{}b \rightarrow a \vert b$
COR (Conditional Operator Replacement)	$a \vert\vert b \rightarrow a\&\&b$
SOR (Shift Operator Replacement)	$a \gg b \rightarrow a \ll b$
ORU (Operator Replacement Unary)	$-a \rightarrow \sim a$
EVR (Expression Value Replacement) Replaces an expression with a default value.	$return\ a\ \rightarrow return\ 0$ $int\ a = b \rightarrow a = 0$
LVR (Literal Value Replacement) Replaces a literal value with a default value: - A numerical literal is replaced with a positive number, a negative number, and zero. - A boolean literal is replaced with its logical complement. - A String literal is replaced with the empty String.	$0 \rightarrow 1$ $1 \rightarrow -1$ $1 \rightarrow 0$ $true \rightarrow false$ $false \rightarrow true$ $"Hello" \rightarrow ""$
STD (Statement Deletion) Deletes a single statement: - return statement - break statement - continue statement - method call - assignment - pre/post increment - pre/post decrement	$return\ a \rightarrow < no\ op >$ $break \rightarrow < no\ op >$ $continue \rightarrow < no\ op >$ $foo(a, b) \rightarrow < no\ op >$ $a = b \rightarrow < no\ op >$ $+ + a \rightarrow < no\ op >$ $- - a \rightarrow < no\ op >$

3.2. Balancing Software Repository with Mutants

We used Major mutation testing tool to increase the number of defects. Major lists multiple mutation operators for a statement. The number of mutants in a file is dependent on the number and type of mutants that Major can apply to the file. We applied mutations starting from the top of the listed mutation operators. We excluded some of the mutation operators for some of the files because some of the mutants caused compilation failures. A failed compilation is not suitable for our study because we produce necessary attributes from compiled object codes. Just et al. removed some of the tests that cause compilation errors because they could not include failed cases (Just et al., 2014).

Since SDP is performed at the file level in our study, every prediction can be used to determine whether a certain file is defective or not. We transformed some of the defect-free files into defective files by injecting mutants to balance the software repository. We started to balancing process presented in Figure 8 by calculating all possible mutants for every file in the repository. And then, we excluded some of the mutants that cause compilation errors. We generated new source code files with mutants for some of the defect-free files. Regarding the order of Major's mutant insertion and the number of files required to balance the repository, we chose the defect-free files. As a result, the balancing process is completed. The SDP process in MBA differs from that depicted in Figure 1. The task of balancing the dataset was moved ahead of attribute extraction because mutants were directly applied to source codes.

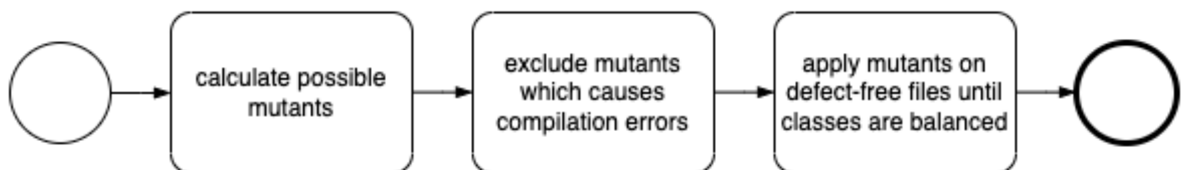

Figure 8. Balancing process with mutants

CHAPTER 4

EXPERIMENTAL DESIGN

The experiment setups were shaped by the research questions. We need to observe the performance difference between Baseline, existing over-sampling approaches, and MBA. Keeping the original dataset for Baseline, data-sampling for over-sampling methods, and mutant injection for MBA are the three different situations for our experimental setup. As we stated in section 3.2, the mutant injection process must be done before software measures calculations. For over-sampling approaches, data-sampling is done after the data preprocessing stage. The difference between MBA and over-sampling approaches resulted in different experiment setups, and Baseline experiment setup has no step for balancing, so three experiment setups are proposed in our study. We carried out our research for two separate SDP scenarios, IRDP and CPDP. Because the nature of the training and testing datasets differentiates the IRDP scenario from the CPDP scenario, we performed three distinct experimental designs twice.

There are six main parts of the experimental setups, which are software repository preparation, software measures calculation, data preprocessing, ML hyperparameter tuning, performance evaluation of the ML models, and performance comparison, as shown in Figure 9.

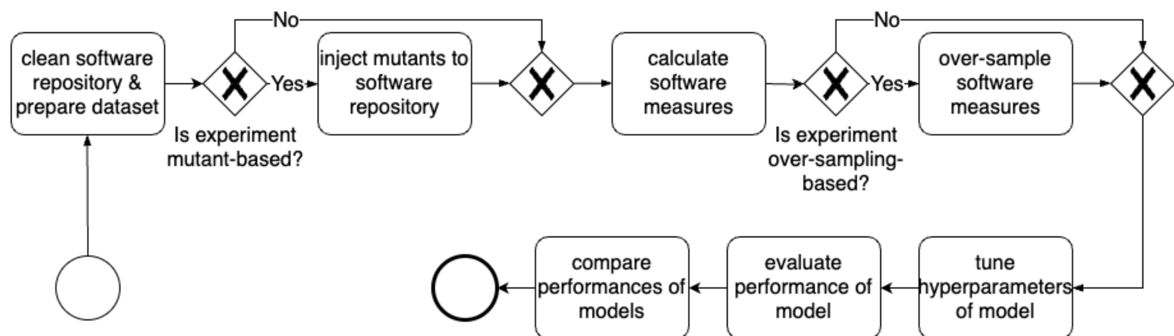

Figure 9. Experimentation process

Below, we described the stages of experimental design in detail. We discussed our actions to increase the validity of our study. We chose commonly used software repositories, over-sampling, data preprocessing, and ML techniques. We used statistical tests to determine the statistical significance of the performance comparison.

4.1. Dataset

The software engineering research community frequently uses a collection of datasets to build models for SDP (Sayyad Shirabad and Menzies 2005). Also, as we pointed out in section 1.1, supervised learning needs a training set to build an ML model. Several datasets, some of which are listed in Table 3, are publicly available (Ferenc et al. 2018). We defined the following criteria to select our training and testing datasets:

1. The source code must be publicly available because our solution includes software mutation, and we must be able to compile mutated source codes.
2. We need to know which part of the code is defective.
3. Defect labels must be available and associated with the relevant code elements.

Table 3. Public datasets for SDP

Dataset	**Source**
PROMISE	(Sayyad Shirabad and Menzies 2005)
Eclipse Bug Dataset	(Zimmermann, Premraj, and Zeller 2007)
Bug Prediction Dataset	(D’Ambros, Lanza, and Robbes 2010)
Bugcatchers Bug Dataset	(Hall et al. 2014)
GitHub Bug Dataset	(Tóth Zoltán and Gyimesi, 2016)

PROMISE is one of the largest datasets for SDP. One of the main datasets in the PROMISE was given by Jurezcko and Madeyski (Jureczko and Madeyski 2010). There are lots of popular projects in the dataset, such as Ant, Camel, Forrest, Ivy, JEdit, Log4j, Lucene, PBeans, Synapse, Velocity, Xalan and Xerces. The advantage of choosing well-

known projects is that it is possible to find lots of available resources which are highly needed while compiling and deciding the dependencies of the projects. It is the main reason why we chose PROMISE to increase the validity of our study. The modules of these projects, which were developed in Java, were subjected to 20 static software measures. Defective modules are labeled with the total number of defects contained in the module, while non-defective modules are labeled with zero. Source codes for datasets are publicly available.

We had to be sure that the source codes that were used to create PROMISE and the publicly available versions that we used to apply for MBA were the same, but Ferenc et al. stated that some of the modules in the source code do not match the published modules by Jureczko (Ferenc et al. 2018). Therefore, we could not include all the versions of projects. We decided to include or exclude projects with respect to the reasons in Table 4.

Table 4. Selected projects and versions in our dataset

Project	**Included Versions**	**Explanation**
Ant	1.3, 1.4, 1.5, 1.6, 1.7	All versions included.
Camel	-	Suitable dependencies not found.
Ckjm	-	In Jureczko's dataset, there is a file that does not exist in the source code (Ferenc et al. 2018).
Forrest	-	Source code contains two distinct filees that appear twice (Ferenc et al. 2018).
Ivy	-	Suitable dependencies not found.
JEdit	3.2.1, 4.0, 4.1, 4.2, 4.3	All versions included.
Log4j	-	There is a contribs directory containing the source code of various contributors. It is unknown which files Jureczko has included (Ferenc et al. 2018).
Lucene	2.0, 2.2, 2.4	All versions included.

(Cont. on next page)

Table 4. (cont.)

PBeans	1.0, 2.0	All versions included.
Poi	1.5, 2.0RC1, 2.5.1, 3.0	All versions included.
Synapse	1.0, 1.1, 1.2	All versions included.
Velocity	1.4, 1.5, 1.6.1	All versions included.
Xalan	2.6, 2.7	2.4 and 2.5 are not included because suitable dependencies not found.
Xerces	1.2, 1.3	1.4 is not included because the number of publicly available source code files does not match up to the number shared by Jureczko (Ferenc et al. 2018).

As we stated in section 1.1, training datasets play an important role in an SDP scenario. WRDP is not possible for MBA because we inject mutations into the software repository before calculating the software measures, and even if we choose testing samples from non-mutated samples, the software measures are affected by mutation because correlated files may have mutations. As a result, we can not evaluate MBA applied models without a proper testing dataset. There are two scenarios available: IRDP and CPDP.

For IRDP, we trained our models on the project versions that have at least one newer version for testing, so the latest versions available are not training dataset candidates. We only considered increasing the defect ratio because of the nature of the MBA and over-sampling methods. We listed the defect ratios of the projects in Table 5. Because some of the projects had a defect ratio of more than 50%, we were unable to use all of the available software repositories for training datasets. For testing datasets, there is no defect ratio restriction.

Table 5. Defect ratios of the projects

Project	**Version**	**Defect Ratio (%)**
Ant	1.3	15.87
Ant	1.4	22.47
Ant	1.5	10.92
Ant	1.6	26.14
Ant	1.7	22.28
JEdit	3.2.1	33.09
JEdit	4.0	24.51
JEdit	4.1	25.32
JEdit	4.2	13.08
JEdit	4.3	2.24
Lucene	2.0	46.67
Lucene	2.2	58.3
Lucene	2.4	59.71
PBeans	1.0	76.92
PBeans	2.0	19.61
Poi	1.5	59.49
Poi	2.0RC1	11.78
Poi	2.5.1	64.42
Poi	3.0	63.57
Synapse	1.0	10.19
Synapse	1.1	27.03
Synapse	1.2	33.59
Velocity	1.4	75.0
Velocity	1.5	66.36
Velocity	1.6.1	34.06
Xalan	2.6	46.44
Xalan	2.7	98.79
Xerces	1.2	16.14
Xerces	1.3	15.23

While adding mutants to projects, some of the files do not contain any elements that the Major mutation operator can change. We could only increase the defect ratio of Synapse 1.0 from 10.19% to 26.11% because the mutation operators we used did not allow us to increase the number of mutated files. We excluded Synapse 1.0 because its mutated defect ratio was significantly lower than 50%.

For CPDP, we used all project versions as a testing dataset. We trained our models on all versions of the projects that remained in the testing dataset, ensuring that no versions of the same project were included in both the testing and training datasets. We included every project stated in Table 5, although we did not include some of the dataset versions with defect ratios greater than 50% in the training dataset.

Finally, as shown in Table 6, we could use 27 dataset pairs for our IRDP experiments, which included 22 versions of seven PROMISE projects. We could use 29 dataset pairs for our CPDP experiments, which included 29 versions of nine PROMISE projects.

Table 6. Training and testing dataset pairs used in experiments

Training → *Testing* Datasets for IRDP Scenario	**Training → *Testing* Datasets for CRDP Scenario**
Ant 1.3 → *1.4, 1.5, 1.6, 1.7* Ant 1.4 → *1.5, 1.6, 1.7* Ant 1.5 → *1.6, 1.7* Ant 1.6 → *1.7*	JEdit 3.2.1, 4.0, 4.1, 4.2, 4.3 + Lucene 2.0 + pBeans 2.0 + Poi 2.0RC1 + Synapse 1.1, 1.2 + Velocity 1.6.1 + Xalan 2.6 + Xerces 1.2, 1.3 → *Ant 1.3, 1.4, 1.5, 1.6, 1.7*
JEdit 3.2.1 → *4.0, 4.1, 4.2, 4.3* JEdit 4.0 → *4.1, 4.2, 4.3* JEdit 4.1 → *4.2, 4.3* JEdit 4.2 → *4.3*	Ant 1.3, 1.4, 1.5, 1.6, 1.7 + Lucene 2.0 + pBeans 2.0 + Poi 2.0RC1 + Synapse 1.1, 1.2 + Velocity 1.6.1 + Xalan 2.6 + Xerces 1.2, 1.3 → *JEdit 3.2.1, 4.0, 4.1, 4.2, 4.3*
Lucene 2.0 → *2.2, 2.4*	Ant 1.3, 1.4, 1.5, 1.6, 1.7 + JEdit 3.2.1, 4.0, 4.1, 4.2, 4.3 + pBeans 2.0 + Poi 2.0RC1 + Synapse 1.1, 1.2 + Velocity 1.6.1 + Xalan 2.6 + Xerces 1.2, 1.3 → *Lucene 2.0, 2.2, 2.4*

(Cont. on next page)

Table 6. (cont.)

	Ant 1.3, 1.4, 1.5, 1.6, 1.7 + JEdit 3.2.1, 4.0, 4.1, 4.2, 4.3 + Lucene 2.0 + Poi 2.0RC1 + Synapse 1.1, 1.2 + Velocity 1.6.1 + Xalan 2.6 + Xerces 1.2, 1.3 → *pBeans 1.0, 2.0*
Poi 2.0RC1 → *2.5.1, 3.0*	Ant 1.3, 1.4, 1.5, 1.6, 1.7 + JEdit 3.2.1, 4.0, 4.1, 4.2, 4.3 + Lucene 2.0 + pBeans 2.0 + Synapse 1.1, 1.2 + Velocity 1.6.1 + Xalan 2.6 + Xerces 1.2, 1.3 → *Poi 1.5, 2.0RC1, 2.5.1, 3.0*
Synapse 1.1 → *1.2*	Ant 1.3, 1.4, 1.5, 1.6, 1.7 + JEdit 3.2.1, 4.0, 4.1, 4.2, 4.3 + Lucene 2.0 + pBeans 2.0 + Poi 2.0RC1 + Velocity 1.6.1 + Xalan 2.6 + Xerces 1.2, 1.3 → *Synapse 1.0, 1.1, 1.2*
	Ant 1.3, 1.4, 1.5, 1.6, 1.7 + JEdit 3.2.1, 4.0, 4.1, 4.2, 4.3 + Lucene 2.0 + pBeans 2.0 + Poi 2.0RC1 + Synapse 1.1, 1.2 + Xalan 2.6 + Xerces 1.2, 1.3 → *Velocity 1.4, 1.5, 1.6.1*
Xalan 2.6 → *2.7*	Ant 1.3, 1.4, 1.5, 1.6, 1.7 + JEdit 3.2.1, 4.0, 4.1, 4.2, 4.3 + Lucene 2.0 + pBeans 2.0 + Poi 2.0RC1 + Synapse 1.1, 1.2 + Velocity 1.6.1 + Xerces 1.2, 1.3 → *Xalan 2.6, 2.7*
Xerces 1.2 → *1.3*	Ant 1.3, 1.4, 1.5, 1.6, 1.7 + JEdit 3.2.1, 4.0, 4.1, 4.2, 4.3 + Lucene 2.0 + pBeans 2.0 + Poi 2.0RC1 + Synapse 1.1, 1.2 + Xalan 2.6 + Velocity 1.6.1 → *Xerces 1.2, 1.3*

We went over the first stage of our experimentation. The software measures calculation process was introduced in section 4.2, which follows the software repository selection process. In contrast to MBA, the mutant injection process is carried out before software measures calculations, as detailed in section 3.2.

4.2. Software Measures Calculation

As we mentioned in section 1.1, we need to represent source code in a suitable format to feed an SDP model. Several researchers used software measures to predict defect-prone code as an input to SDP models as attributes. The majority of the researchers do not concentrate on the measure calculation process. They used calculated measures from previous studies. Jureczko shared a software measures dataset that is mostly used by researchers studying SDP (Jureczko and Spinellis 2010). Jureczko used an extended version of Chidamber and Kemerer Java Metrics (ckjm-extended) for the calculation of measures (Spinellis 2005). Ckjm-extended is a tool that calculates 20 size and structure software measures by processing the object code of compiled Java files (Jureczko and Spinellis 2010). The program calculates measures for each files listed in Table 7.

Table 7. Software measures definitions (Source: Jureczko & Spinellis, 2010)

Measure	**Definition**	**Source**
Weighted methods per class (WMC)	The value of the WMC is equal to the number of methods in the class (assuming unity weights for all methods).	(Chidamber and Kemerer 1994)
Depth of Inherit-ance Tree (DIT)	The DIT measure provides for each class a measure of the inheritance levels from the object hierarchy top.	(Chidamber and Kemerer 1994)
Number of Children (NOC)	The NOC measure simply measures the number of immediate descendants of the class.	(Chidamber and Kemerer 1994)

(Cont. on next page)

Table 7. (cont.)

Coupling between object classes (CBO)	The CBO measure represents the number of classes coupled to a given class (efferent couplings and afferent couplings). These couplings can occur through method calls, field accesses, inheritance, method arguments, return types, and exceptions.	(Chidamber and Kemerer 1994)
Response for a Class (RFC)	The RFC measures the number of different methods that can be executed when an object of that class receives a message. Ideally, we would want to find for each method of the class, the methods that class will call, and repeat this for each called method, calculating what is called the transitive closure of the method call graph. This process can however be both expensive and quite inaccurate. Ckjm calculates a rough approximation to the response set by simply inspecting method calls within the class method bodies. The value of RFC is the sum of number of methods called within the class method bodies and the number of class methods. This simplification was also used in the Chidamber and Kemerer's description of the measure (Chidamber and Kemerer 1994).	(Chidamber and Kemerer 1994)
Lack of cohesion in methods (LCOM)	The LCOM measure counts the sets of methods in a class that are not related through the sharing of some of the class fields. The original definition of this measure (which is the one used in ckjm) considers all pairs of class methods. In some of these pairs both methods access at least one common field of the class, while in other pairs the two methods do not share any common field accesses. The lack of cohesion in methods is then calculated by subtracting from the number of method pairs that do not share a field access the number of method pairs that do.	(Chidamber and Kemerer 1994)

(Cont. on next page)

Table 7. (cont.)

Lack of cohesion in methods (LCOM3)	$LCOM3 = \frac{(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)) - m}{1-m}$ $m: number\ of\ methods\ in\ a\ class$ $a: number\ of\ attributes\ in\ a\ class$ $\mu(A_j): number\ of\ methods\ that\ access\ the\ attribute$	(Henderson -Sellers 1995)
Afferent couplings (Ca)	The Ca measure represents the number of classes that depend upon the measured class.	(Martin 1994)
Efferent couplings (Ce)	The Ce measure represents the number of classes that the measured class is depended upon.	(Martin 1994)
Number of Public Methods (NPM)	The NPM measure simply counts all the methods in a class that are declared as public. The measure is known also as Class Interface Size (CIS).	(Bansiya and Davis 2002)
Data Access Metric (DAM)	This measure is the ratio of the number of private (protected) attributes to the total number of attributes declared in the class.	(Bansiya and Davis 2002)
Measure of Aggregatio n (MOA)	The MAO measures the extent of the part-whole relationship, realized by using attributes. The measure is a count of the number of class fields whose types are user defined classes.	(Bansiya and Davis 2002)
Measure of Functional Abstraction (MFA)	This measure is the ratio of the number of methods inherited by a class to the total number of methods accessible by the member methods of the class. The constructors and the java.lang.Object (as parent) are ignored.	(Bansiya and Davis 2002)

(Cont. on next page)

Table 7. (cont.)

Cohesion Among Methods of Class (CAM)	This measure computes the relatedness among methods of a class based upon the parameter list of the methods. The measure is computed using the summation of number of different types of method parameters in every method di- vided by a multiplication of number of different method parameter types in whole class and number of methods.	(Bansiya and Davis 2002)
Inheritance Coupling (IC)	This measure provides the number of parent classes to which a given class is coupled. A class is coupled to its parent class if one of its inherited methods functionally dependent on the new or redefined methods in the class. A class is coupled to its parent class if one of the following conditions is satisfied: • One of its inherited methods uses an attribute that is defined in a new/redefined method. • One of its inherited methods calls a redefined method. • One of its inherited methods is called by a redefined method and uses a parameter that is defined in the redefined method.	(Tang, Kao, and Chen 1999)
Coupling Between Methods (CBM)	The CBM measures the total number of new/redefined methods to which all the inherited methods are coupled. There is a coupling when at least one of the given in the IC measure definition conditions is held.	(Tang, Kao, and Chen 1999)
Average Method Complexity (AMC)	The AMC measures the average method size for each class. Size of a method is equal to the number of Java binary codes in the method.	(Tang, Kao, and Chen 1999)

(Cont. on next page)

Table 7. (cont.)

Maximum McCabe's cyclomatic complexity (MAX CC) Average McCabe's cyclomatic complexity (AVG CC)	CC is equal to number of different paths in a method (function) plus one. The cyclomatic complexity is defined as: $CC = E - N + P$ $E: the\ number\ of\ edges\ of\ the\ graph$ $N: the\ number\ of\ nodes\ of\ the\ graph$ $P: the\ number\ of\ connected\ components$ CC is the only method size measure. The constructed models make the class size predictions. Therefore, the measure had to be converted to a class size measure. Two measures has been derived: • MAX(CC) - the greatest value of CC among methods of the investigated class. • AVG(CC) - the arithmetic mean of the CC value in the investigated class.	(McCabe 1976)
Lines of Code (LOC)	The LOC measure based on Java binary code. It is the sum of number of fields, number of methods and number of instructions in every method of the investigated class.	

We can not use pre-calculated software measures because we apply mutations to source code, and the software measures calculation process comes after compilation as we mentioned in section 3.2. Therefore, we need to recalculate software measures after balancing the dataset with MBA. We chose ckjm-extended to work with the most studied measures and we were able to compare our software measures with other studies to determine the validity of our study. In this way, we supported the reliability of MBA. In the study by Jureczko, which shared a software measures database, they extended the PROMISE (software repository dataset) with some other private repositories. Software measures from private repositories are not useful for the MBA because we can not reach the source codes. We validated our results with the public software repositories. To increase the reliability of our study, we determined the causes of the differences in our software measures calculation and Jureczko's. We could not produce the same software

measures values as in Jureczko's study. Ferenc et al. mentioned that some tool and environment versions are not clearly defined in Jureczko's study (Ferenc et al. 2018). They attempted to match their software measures calculation with the software measures dataset shared by Jureczko, but they were unable to do so, as we were. We had to guess the following variables while calculating software measures:

1. Versions of dependencies in software repositories
2. Java versions of the compilation of software repositories and their dependencies
3. Version of ckjm-extended
4. Java version to run ckjm-extended

Jureczko did not mention which versions of software repositories, dependencies, and the Java SDK were used. Jureczko most likely calculated software measures using pre-compiled object codes from software repositories; thus, we selected dependencies and Java SDK versions to conduct several trials to obtain the same software measures as the pre-compiled object codes. Each software repository was compiled using a specific version of the Java SDK.

We have tried some of the possible combinations of variables to get the software measures calculated by Jureczko. For the ckjm-extended version, we tried three versions of ckjm-extended which are 2.1, 2.2, and 2.3. We compared software measures published by Jureczko and calculated in our experiment on Ant 1.3 for the three ckjm-extended versions and we plotted the number of different software measures for each software measure type. The ckjm-extended 2.2 gave us the closest results both as shown in Figure 10, Figure 11, and Figure 12 and in the literature (Ferenc et al. 2018).

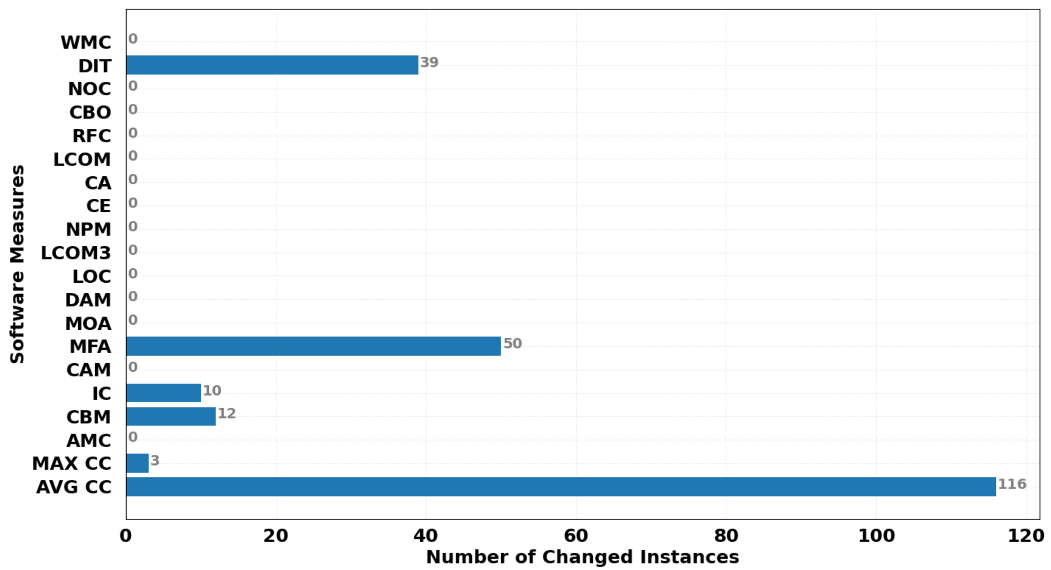

Figure 10. Comparison of software measures presented in Jureczko's study and calculated in our experiment with ckjm-extended 2.1 on ant 1.3

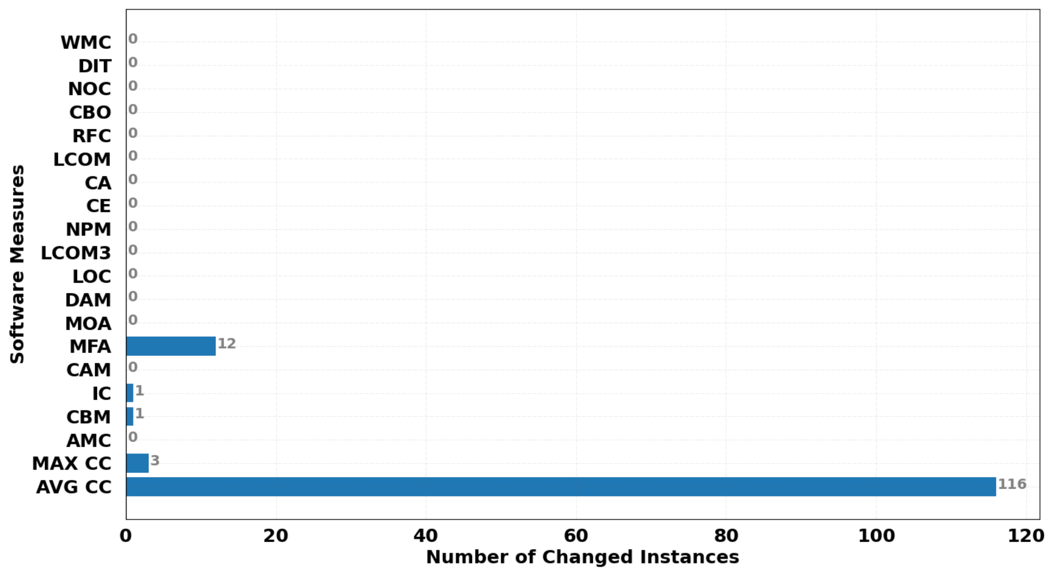

Figure 11. Comparison of software measures presented in Jureczko's study and calculated in our experiment with ckjm-extended 2.2 on ant 1.3

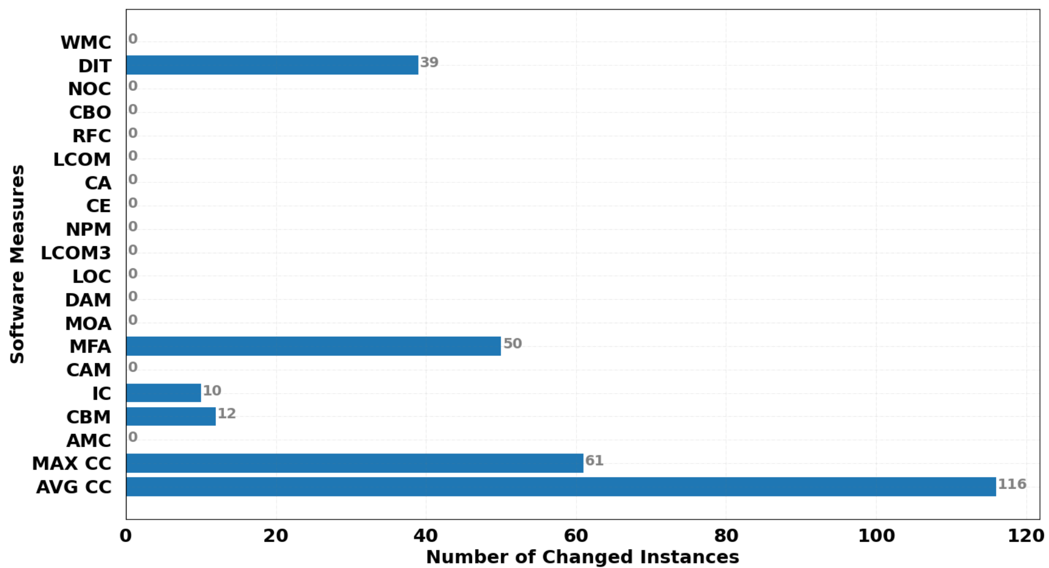

Figure 12. Comparison of software measures presented in Jureczko's study and calculated in our experiment with ckjm-extended 2.3 on ant 1.3

When we consider the measures "DIT", "MFA", "IC", "CBM" and "CC" that do not match, we noticed that these measures are about the inheritance hierarchy of the code. In the documentation of ckjm-extended, by default, Java SDK packages are not considered while calculating the software measures. They also include a flag to optionally enable Java SDK packages. During the inspection of the source code, we realized that ckjm-extended uses a framework called BCEL for these calculations. BCEL tracks the inheritance of objects with Java SDK objects if the related object inherits any Java SDK object. The problem here is the Java SDK version used for software measures calculation. We could produce differences only by changing the version of Java SDK that used to run ckjm-extended; some of the inheritance-dependent measures changed. It clearly seems that ckjm-extended considers Java SDK packages by default. Also, our results are parallel to this claim, as shown in Figure 13 below.

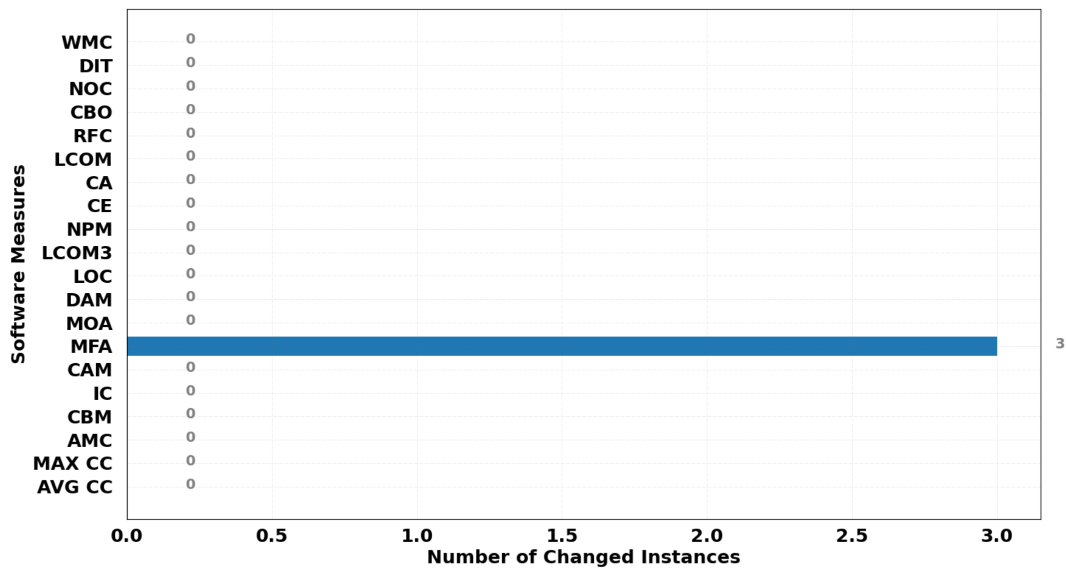

Figure 13. Comparison of software measures calculated by Java SDK 1.6 and calculated by Java SDK 1.7 with ckjm-extended 2.2 on ant 1.3

On the other hand, in terms of backward compatibility, we considered that a project should be compatible with other Java SDK versions. Since the differences in software measures presented by Jureczko and calculated by us are unrelated to the software repository, which is the dataset, we accept these differences as environmental differences and do not expect them to affect an instance's defectiveness. Another problem is that BCEL could not find some of the objects with the Java 1.8 SDK. We generate the results with different Java SDK versions, which are 1.6, 1.7, and 1.8. We compared the generated results with Jureczko's, but there is not a clear difference to select a better version of the Java SDK. Also, 1.5 and lower versions of the Java SDK are not compatible with ckjm-extended 2.2. There are two options to continue: Java SDK 1.6 and Java SDK 1.7. Most of the projects in PROMISE dataset were released before Java SDK 1.6 was released. Java SDK 1.6 will be more compatible with projects. We chose Java SDK 1.6 to run ckjm-extended 2.2. We went over the software measures calculation process in great detail. In section 4.3, we introduced the third stage, data preprocessing.

4.3. Data Preprocessing

Distinctions in scale across input data could make the problem being modeled more challenging. When numerical input parameters are scaled to a normal range, the performance of many ML algorithms increases (Brownlee 2020). Normalization scales each input variable to a value between 0 and 1 (Bharati, Podder, and Hossain Mondal 2020). We applied the MinMax scaler to normalize values as in the literature (Kumar, Rath, and Sureka 2017). We want features that have the same effect on the model.

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{4.1}$$

In the equation above, x denotes the original value of a feature of an instance, x_{min} is minimum value and x_{max} is maximum value of the feature in all instances in the dataset. x' gives us the normalized value of x. In our dataset, we have 20 features for input data. Table 8 contains 4 Ant 1.3 instances. It appears that there are large scale differences between features, and normalization is required for our study. Before building ML models in our study, we applied the MinMax scaler to each data instance, and then we tuned the hyperparameters of ML methods, as detailed in section 4.4.

Table 8. Software measures of four instances of Ant 1.3

WMC	DIT	NOC	CBO	RFC	LCOM	CA	CE	NPM	LCOM3	LOC	DAM	MOA	MFA	CAM	IC	CBM	AMC	MAX CC	AVG CC
11	2	0	7	15	13	5	3	11	.75	97	1	3	.20	.22	0	0	7.3	1	1
8	3	0	4	41	10	0	4	8	.80	236	1	0	.84	.87	1	1	27.8	14	2.75
5	1	0	4	20	0	0	4	4	.33	144	1	1	0	.90	0	0	27.2	5	2.40
3	2	0	8	18	3	1	7	2	1	132	0	0	.92	.50	0	0	42	1	1

4.4. Hyperparameter Tuning

Hyperparameters are the parameters that are used to configure a model to minimize the loss function, and exploring the best combination of hyperparameters is called as hyperparameter tuning (L. Yang and Shami 2020). In the literature, some of the commonly used hyperparameter tuning techniques are Grid Search, Random Search, Gradient-based Optimization, Bayesian Optimization, Multi-fidelity Optimization, Genetic Algorithm. Every technique has weaknesses and strengths. Grid Search is a simple method that tries all combinations of defined parameters. Evaluating every defined combination is a very time-consuming process, so parameter space selection is very important. Random Search evaluates randomly selected hyperparameters. Random search works like Grid Search but the selection of hyperparameter combinations is done randomly, so it does not guarantee any stability. It can hit a better hyperparameter combination than Grid Search. The number of trials increases the success of Random Search. Gradient-based Optimization only works for continuous hyperparameters. Gradient-based Optimization evaluates a parameter, which is generally given by an analyst. Gradient-based Optimization searches for near values of the parameter with a step that is also generally given by the analyst. Gradient-based Optimization converges on a local maximum and stops with a performance criterion. Bayesian Optimization is an iterative algorithm that determines the next hyperparameter configuration (L. Yang and Shami 2020). Bayesian Optimization is generally used for problems whose evaluation takes longer (L. Yang and Shami 2020). Bayesian Optimization has two steps: surrogate model and acquisition function. A surrogate model fits all currently observed points into the objective function. An acquisition function selects the best subset of a dataset to make a surrogate model more representative. Bayesian Optimization builds a surrogate model and iteratively does three steps until an initially set iteration count is reached. First, it detects optimal hyperparameter values on the surrogate model. Applies these hyperparameters to the real objective function to evaluate them and updates the surrogate model with respect to new results. Bayesian Optimization needs fewer resources than Grid Search and Random Search but does not cover hyperparameter ranges as much as Grid Search. Multi-fidelity Optimization combines low-fidelity and high-fidelity. Low-fidelity means evaluating hyperparameters on a small subset of a dataset. High-fidelity means evaluating hyperparameters on a large subset of a dataset. Low-fidelity has a lower

cost but poorer performance than high-fidelity. Multi-fidelity selects hyperparameters by applying high-fidelity to well-performed low-fidelity cases (L. Yang and Shami 2020). Genetic Algorithm simulates a survival race on hyperparameters. Each chromosome represents a hyperparameter, and every chromosome has several genes. Genetic Algorithm applies crossing-over and mutations to randomly initialized individuals, and new individuals are generated. New individuals are evaluated, and worse performing individuals are eliminated. Genetic Algorithm continues this process until the termination condition is met.

In the SDP, Grid search and Random search are used in several studies (Bennin et al. 2018; Li et al. 2020; Bahaweres et al. 2020). As we mentioned, Grid search is a good choice when the hyperparameter search space is defined in optimum ranges. We decided to use Grid search in our study because Hyperparameter Tuning problem is studied for many ML algorithm hyperparameters and their ranges in SDP on many datasets. We considered previous studies and decided on feasible search spaces for our Hyperparameter Tuning setup. Even though Grid Search is not good at reducing unnecessary evaluations, we have enough computational power to cover all evaluations. As a result, the likelihood of missing well hyperparameter setting is reduced. Grid search runs all possible combinations (cartesian product) of the set of parameters and selects the best parameters with respect to an evaluation function. Parameter selection is essential for Grid Search. It is also the only input for this method. We select parameters in Table 9 below. We get the parameters from three studies with respect to the ML method (Bennin et al. 2018; Bahaweres et al. 2020; Li et al. 2020).

Table 9. Tuned hyperparameters for ML models

Method	**Parameter**	**Range**
K-Nearest Neighbor (KNN)	n_neighbours	{1, 3, 5, 7, 9, 11, 13, 15}
Random Forest (RF)	n_estimaators	[10, 100] step:10
	criterion	{‘gini’, ‘entropy’}
	max_features	{‘None’, ‘sqrt’, ‘log2’}

(Cont. on next page)

Table 7. (cont.)

SVM	kernel	{'rbf', 'linear', 'poly', 'sigmoid'}
	degree	[0, 3] step: 1
	coef()	[0, 3] step: 1
	gamma	{'scale', 'auto'}
Naïve Bayes (NB)	-	-
Decision Tree (DT)	criterion	{'gini', 'entropy', 'log_loss'}
	max_features	{'auto', 'sqrt', 'log2'}
	max_depth	[5, 10] step: 1

4.5. Performance Measure

The commonly used performance measures in SDP are AUC (Area Under the Curve), F-measure (F1), recall, precision, false alarm, g-measure, and balance (Moussa and Sarro 2022). There is no agreement on which performance measures should be used to evaluate an ML method (Seliya, Khoshgoftaar, and Van Hulse 2009). Selecting appropriate performance measures is very important, especially in SDP because CIP makes assessing ML models harder. If we do not measure its performance using the proper measurement method, we can not find the right model. For instance, if we select accuracy as the performance measure for an imbalanced dataset, predictions will be biased toward the major class because using accuracy as the performance measure means maximizing the number of true predictions over total predictions. Assume that the minor class ratio is 10%. For such a scenario, predicting all instances as major class results in 90% accuracy, which is a very high score, but the model fails to identify any minor classes. This is unacceptable from the point of view of SDP because the main concern is detecting defective instances.

The base for measuring performance in a binary classification problem is confusion matrix because four performance measures that are also used to calculate other measures are calculated from confusion matrix. True Positive (TP), False Negative (FN), False Positive (FP) and True Negative (TN) measures are calculated from confusion matrix. Confusion Matrix shows the number of predicted and actual classes of instances as in Table 10.

Table 10. Confusion Matrix for Binary Classification (Source: Moussa & Sarro, 2022)

Actual Value	**Predicted Value**	
	Defective	**Non-Defective**
Defective	True Positive (TP)	False Negative (FN)
Non-Defective	False Positive (FP)	True Negative (TN)

Other than the measures defined in the confusion matrix, there are some measures developed in the literature for assessing the performance of ML models from various perspectives. CIP is the most important topic to consider when selecting an performance evaluation measure because it is present in the majority of the dataset. We considered the most commonly used measures in SDP, which are AUC, F1, recall, precision, FPR, g-measure, and balance.

Recall shows the probability of classifying defective instances correctly. Precision shows how well the model classifies instances as defective while misclassifying non-defective instances. For SDP, accurately classifying faulty modules is important, but on the other hand, classifying non-defective instances accurately is a very important task because non-defective instances mostly belong to the major class.

G-measure is the geometric mean between precision and recall, and F1 calculates the harmonic mean of precision and recall. From class imbalance perspective, g-measure and F1 provide more honest evaluation results for SDP models.

Balance measures the distance between FPR and recall. When FPR is equal to the recall, balance gets its higher value, which is 1. The importance of the balance between FPR and recall shows that the success of the classification of defective instances does not come from classifying instances most likely as defective.

Receiver Operator Characteristic (ROC) curve visualizes the relation between TP and FN. In a ROC curve, one axis represents True Positive Rate (TPR), and the other one represents False Positive Rate (FPR). Using rates of TP and FP together makes assessment fair for imbalanced datasets. When we plot this graph, we get a curve. The area under the curve (AUC) gives us an ideal performance measure for SDP problem. We summarized the seven commonly used performance measures in SDP in Table 11.

Table 11. The definition of the performance measures (Source: Moussa & Sarro, 2022)

Performance Measure	Definition
AUC	Area under the Receiver Operating Characteristic Curve
Recall or pd (TPR)	$\frac{TP}{TP+FN}$
Precision	$\frac{TP}{TP+FP}$
F-measure (F1)	$2 \times \frac{Precision \times Recall}{Precision + Recall}$
False Alarm or pf (FPR)	$\frac{FP}{FP+TN}$
G-measure	$\frac{2 \times Recall \times (1-FPR)}{Recall + (1-FPR)}$
Balance	$1-\frac{\sqrt{(0-FPR)^2+(1-Recall)^2}}{\sqrt{2}}$

We chose AUC, pd and pf measures for evaluation of our models because they are the most commonly used performance measure for SDP, and AUC works well for imbalanced datasets, as we need in our problem. Lower pf values and higher AUC and pd values indicate better predictors. Since precision and F1 are unstable for evaluating models trained using imbalance datasets (Menzies et al. 2007), we did not include them. Performance evaluation is the last stage of the experimental design before comparing the performances of the experiments. We summarized three experimental designs in section 4.6.

4.6. Summary of Experimental Designs

In this chapter, we explained our experimental design in detail. In this section, we were interested in demonstrating the overall steps of each of the three experimental designs. The performance comparison stage is excluded from these designs because it

employs statistical analysis techniques that require the aggregated results of all experiments, as detailed in section 4.7.

4.6.1. Baseline Experimental Design

We followed the steps in Figure 14 for each dataset. First, as shown in Table 6, we prepared training and testing software repositories. For both training and testing datasets, we calculated software measures and preprocessed them with the MinMax scaler. The training dataset is divided into two parts: 20% for validation and 80% for training. Because we employed 5-fold stratified cross validation, the defect ratio was preserved across all folds. We chose the best-performing model's parameter as optimal and developed the prediction model with the optimal parameter. Finally, we ran the prediction model against the testing dataset and calculated its performance.

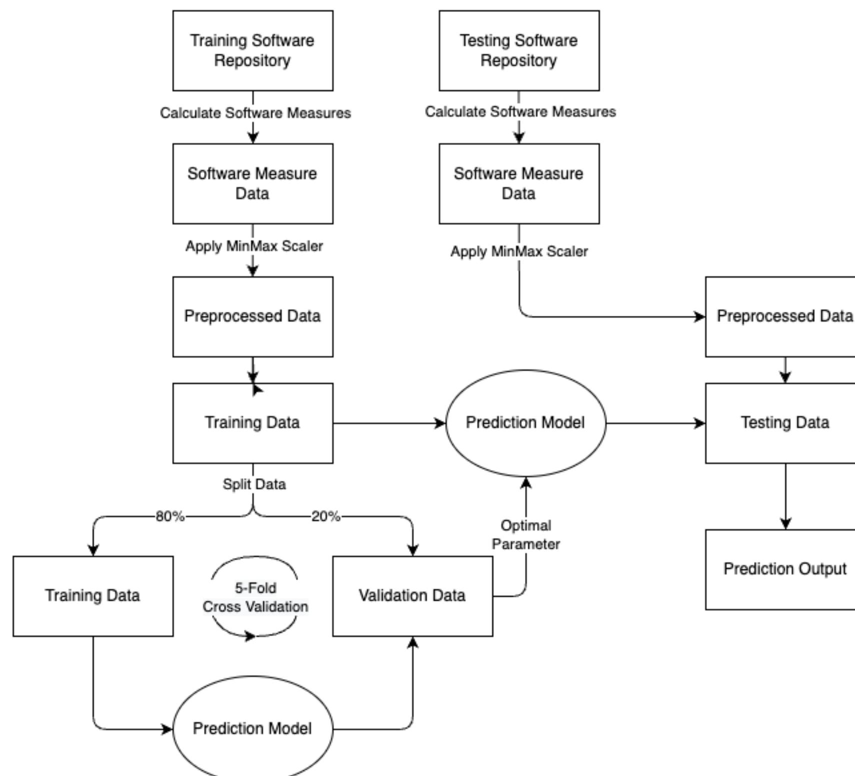

Figure 14. Baseline experimental design

4.6.2. Over-sampling-based Experimental Design

We followed the steps in Figure 15 for each dataset. First, as shown in Table 6, we prepared training and testing software repositories. For both training and testing datasets, we calculated software measures and preprocessed them with the MinMax scaler. We resampled the training dataset with over-sampling methods. The training dataset is divided into two parts: 20% for validation and 80% for training. Because we employed 5-fold stratified cross validation, the defect ratio was preserved across all folds. We chose the best-performing model's parameter as optimal and developed the prediction model with the optimal parameter. Finally, we ran the prediction model against the testing dataset and calculated its performance.

Figure 15. Over-sampling-based experimental design

4.6.3. Mutation-based Experimental Design

We followed the steps in Figure 16 for each dataset. First, as shown in Table 6, we prepared training and testing software repositories as we discussed in chapter 3. We injected mutants into the training software repository. For both training and testing datasets, we calculated software measures and preprocessed them with the MinMax scaler. The training dataset is divided into two parts: 20% for validation and 80% for training. Because we employed 5-fold stratified cross validation, the defect ratio was preserved across all folds. We chose the best-performing model's parameter as optimal and developed the prediction model with the optimal parameter. Finally, we ran the prediction model against the testing dataset and calculated its performance.

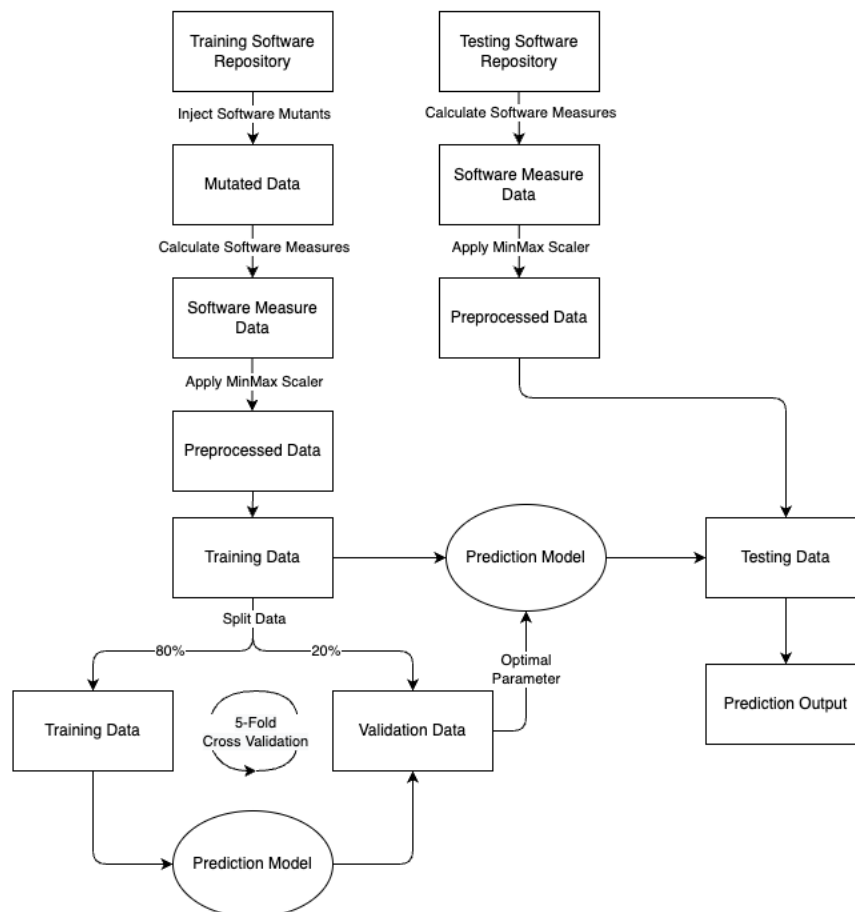

Figure 16. Mutation-based experimental design

4.7. Performance Comparison

To statistically evaluate and compare MBA to the other over-sampling techniques and Baseline, we selected one-way repeated measures ANOVA to determine whether the means of the performance measure values of the seven groups are different. We decided to use a parametric test because parametric tests are more powerful than non-parametric tests if the sample is normally distributed (Pappas and DePuy 2004). One-way repeated measures ANOVA has two preconditions, which are normality and sphericity. To test normality, we used Z-scores of skewness and kurtosis attributes. To check normality, we checked skewness and kurtosis values. These values must be between -1.96 and +1.96 for $\alpha = 0.05$. To check sphericity, we applied Mauchly's test of sphericity. The significance value of Mauchly's test must be greater than 0.05 for the samples that satisfy sphericity. If these tests failed, we decided to use Friedman test, which is the non-parametric alternative to the one-way repeated measures ANOVA. If the Friedman test shows that there is a statistically significant difference in performance measure values depending on the type of over-sampling methods, MBA, and Baseline, we decided to use Wilcoxon signed-rank tests on the groups of two between the seven methods. To quantitatively analyze and compare MBA to the baseline and the other five over-sampling approaches, we employed win-tie-loss statistics, which have been used in earlier research studies (Bennin et al. 2018). The performance values of two predictors were compared using the Wilcoxon signed-rank test, and if the performance distributions were not statistically different, the "ties" counter was increased by one. If there were a statistically significant difference between two predictions, the counters for "wins" and "losses" were increased by one. To compute Friedman and Wilcoxon signed-rank tests, we used the SciPy Python library (Virtanen et al. 2020). To compute the sphericity and normality tests, we used the Pingouin Python library (Vallat 2018). Because every analysis failed the sphericity and normality tests, we did not employ any tool to compute the one-way repeated measures ANOVA.

CHAPTER 5

RESULTS AND DISCUSSION

In this chapter, we presented the results of Baseline, MBA, and each over-sampling technique on the ML methods and compared the performances of MBA with over-sampling techniques and Baseline. We discussed the results of our study. We provided a critical analysis of the results that were obtained.

RQ1: *Does the proposed MBA improve performance over existing over-sampling approaches and Baseline on IRDP?* MBA and over-sampling techniques have better recall values than Baseline, but Baseline has the lowest false alarm values, and there was no significant difference between the AUC values of rebalancing techniques, so we can not conclude that MBA outperforms over-sampling techniques and Baseline.

RQ2: *Does the proposed MBA improve performance over existing over-sampling approaches and Baseline on CPDP?* Only MBA consistently outperformed Baseline for recall values, but Baseline has the lowest false alarm values, and there was no significant difference between the AUC values of rebalancing techniques, so we can not conclude that MBA outperforms over-sampling techniques and Baseline.

RQ3: *How does the change percentage of software measures (SMC) affect performance of MBA on IRDP?* SMC and recall and SMC and false alarm showed a significantly positive association, whereas SMC and AUC had no significant correlation, so we can not draw the conclusion that performance improved as datasets became more balanced.

RQ4: *How does the change percentage of software measures (SMC) affect performance of MBA on CPDP?* SMC and recall and SMC and false alarm showed a significantly positive association, whereas SMC and AUC had no significant correlation, so we can not draw the conclusion that performance improved as datasets became more balanced.

5.1. Performance Evaluation of Rebalancing Methods for IRDP Scenario

For the IRDP scenario, we created 945 SDP models using five ML algorithms, seven rebalancing methods (Baseline, MBA, and five over-sampling techniques), and 27 training and testing dataset pairs, as shown in Table 6.

The quartile plots of performance measures (AUC, pd, and pf) for Baseline, MBA, and five over-sampling methods for each ML algorithm are shown in Figure 17. The quartile bounds represent the 25th and 75th percentiles, respectively, and the solid dots represent the median values. Higher median values show higher AUC and pd measure performance, while lower median values indicate better pf measure performance. MBA and over-sampling techniques did not significantly improve AUC values for all ML algorithms, as shown in the top lane of Figure 17. Rebalancing method and ML algorithm pairs have AUC values around 0.6. MBA and five over-sampling techniques improved the recall (pd) measure compared to the Baseline. According to Figure 17, in the middle lane, MBA beat the Baseline and other five over-sampling techniques for all ML algorithms. MBA and over-sampling techniques, on the other hand, created more false alarms (pf) than the Baseline (bottom lane of Figure 17). In most cases, MBA had the worst pf value.

Figure 18 shows Wilcoxon signed-rank test win-tie-loss comparisons of MBA to the Baseline and five over-sampling techniques. Except for the SMOTE Nominal and NB pairs, MBA did not increase IRDP performance using the AUC measure. MBA reduced AUC values in five cases (shown with orange dots). In terms of pd, MBA yielded statistically significantly better results (23 wins, green dots, and 7 ties, blue dots). Instead of increasing pd values, MBA decreased pf values in 26 of 30 cases (four of which resulted in ties). In terms of pd and pf performance, MBA outperformed "SMOTE & NB" and "SMOTE Nominal & SVM" pairs, as well as tied pf values.

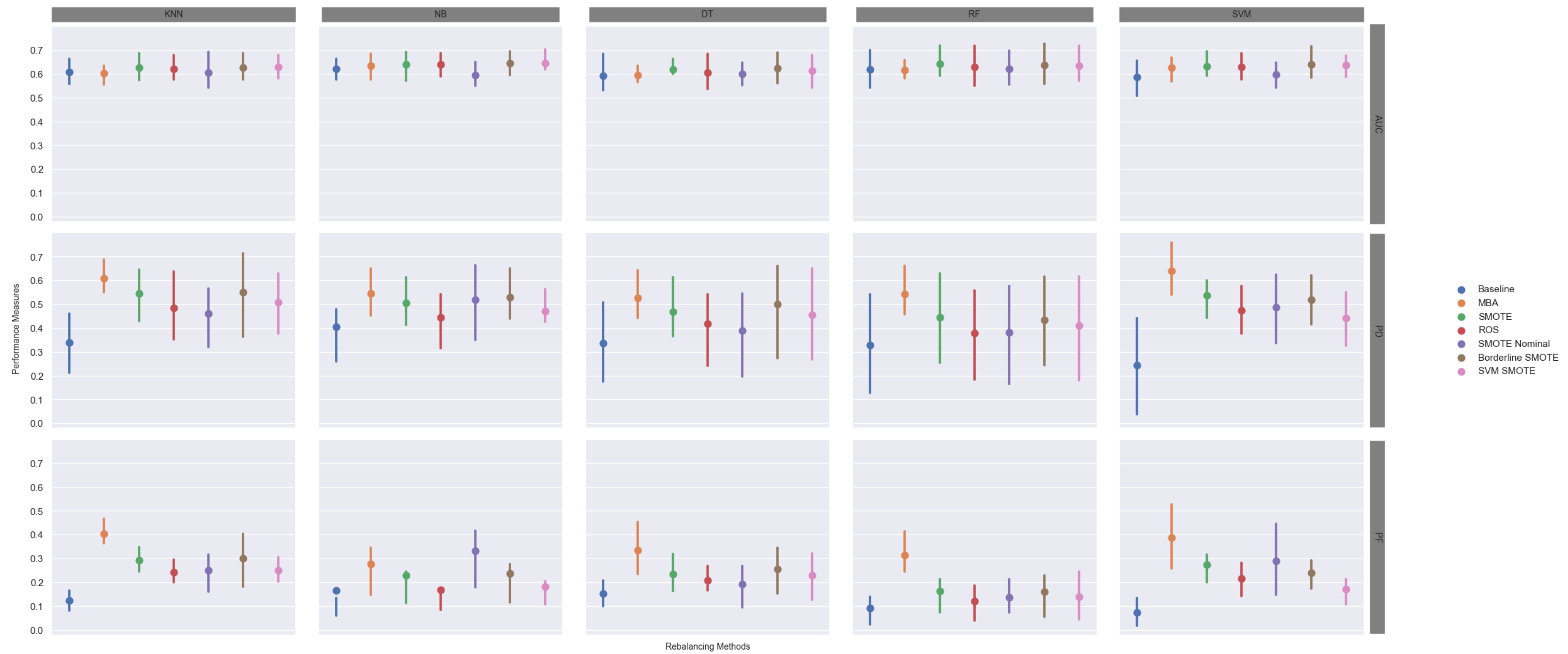

Figure 17. IRDP Scenario: Quartile plots of performance measures (AUC, pd, and pf) for Baseline, MBA, over-sampling techniques per each ML algorithm

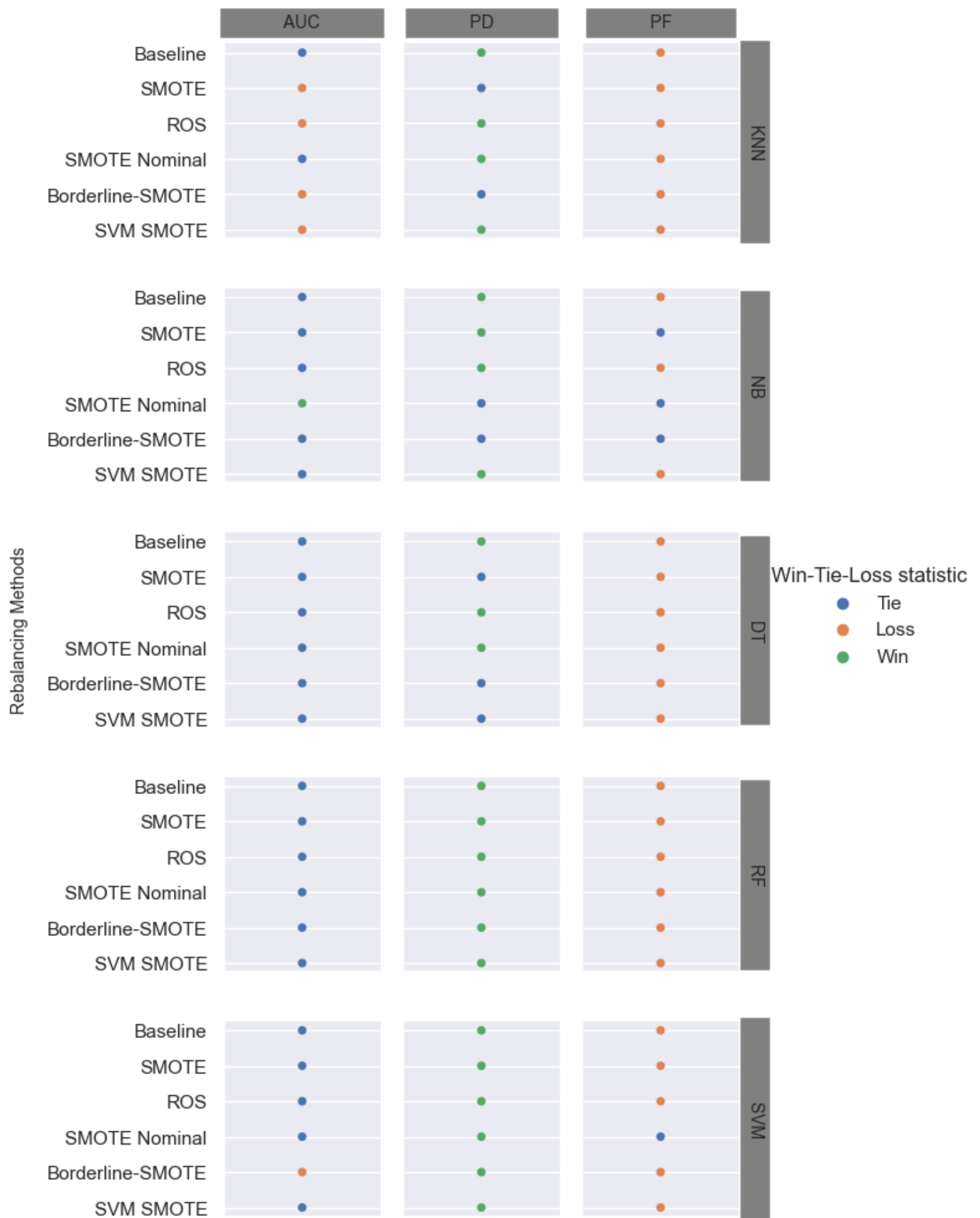

Figure 18. IRDP Scenario: Wilcoxon test win-tie-loss comparison of MBA vs. Baseline, SMOTE, ROS, SMOTE Nominal, Borderline-SMOTE, SVM SMOTE across all datasets per each ML algorithm and performance measures (AUC, pd, and pf)

We used the Wilcoxon signed-rank test to compare AUC, recall, and pf values to find the best performing combination of rebalancing method and ML algorithm. Table 12 displays win-tie-loss values sorted by the difference between wins and losses. Due to space constraints, we give the top and bottom ten performers for each performance measure.

MBA and over-sampling methods provide some improvement in terms of AUC performance measure, as the Baseline combined with ML algorithms is listed in the bottom half of Table 12 (Baseline with RF 20th, NB 24th, KNN 28th, DT 34th, SVM 35th). The first seven positions are occupied by SMOTE, Borderline-SMOTE, and SVM SMOTE. NB appears to be the best performing ML algorithm, with five positions in the top ten. MBA, paired with NB, was ranked ninth. In terms of AUC values, we can not find a dominant sampling method and ML algorithm pair. Even the RF and SMOTE combination improved AUC values in 16 cases but not in 18. In terms of recall performance, MBA combined with SVM and KNN beat the over-sampling and ML algorithm pairs, with wins-losses scores of 32 and 28 out of 34 comparisons, respectively. MBA was ranked in the top ten when combined with RF and DT, although with substantially lower wins-losses scores of 18 and nine, respectively. SMOTE and Borderline-SMOTE were the other sampling methods seen in the top ranks. MBA and over-sampling techniques often underperform compared to Baseline in terms of the pf performance measure. SVM, RF, NB, and KNN algorithms trained on identical datasets provided much fewer false alarms, with 34, 31, 24, and 24 wins-losses scores out of 34, respectively. MBA, when paired with KNN and SVM, produced the most false alarms, with -32 and -31 wins-losses values, respectively

Table 12. IRDP Scenario: Rankings for performance measures in terms of wins (W), losses (L), wins-losses (W-L), and ties (T)

	AUC						pd						pf					
#	**ML**	**Sampl. Tech.**	**W**	**L**	**W-L**	**T**	**ML**	**Sampl. Tech.**	**W**	**L**	**W-L**	**T**	**ML**	**Sampl. Tech.**	**W**	**L**	**W-L**	**T**
1	RF	SMOTE	16	0	16	18	SVM	MBA	32	0	32	2	SVM	Baseline	34	0	34	0
2	NB	B-SMOTE	15	0	15	19	KNN	MBA	28	0	28	6	RF	Baseline	32	1	31	1
3	SVM	B-SMOTE	14	0	14	20	KNN	SMOTE	21	1	20	12	NB	Baseline	25	1	24	8
4	RF	B-SMOTE	12	0	12	22	RF	MBA	19	1	18	14	KNN	Baseline	26	2	24	6
5	NB	SVM SMOTE	12	0	12	22	KNN	B-SMOTE	18	0	18	16	RF	ROS	26	2	24	6
6	RF	SVM SMOTE	12	0	12	22	SVM	SMOTE	18	2	16	14	RF	SMOTE Nom	24	2	22	8
7	NB	SMOTE	11	0	11	23	SVM	B-SMOTE	17	2	15	15	RF	SVM-SMOTE	23	3	20	8
8	NB	ROS	11	0	11	23	KNN	SVM SMOTE	14	3	11	17	DT	Baseline	21	3	18	10
9	NB	MBA	10	0	10	24	NB	B-SMOTE	11	1	10	22	NB	ROS	20	3	17	11
10	SVM	SVM SMOTE	10	0	10	24	DT	MBA	11	2	9	21	RF	SMOTE	19	6	13	9
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
26	DT	MBA	0	10	-10	24	DT	ROS	4	13	-9	17	KNN	SVM-SMOTE	6	17	-11	11
27	DT	ROS	0	11	-11	23	NB	ROS	3	13	-10	18	SVM	SMOTE	3	15	-12	16
28	KNN	Baseline	1	14	-13	19	DT	SMOTE Nom	2	18	-16	14	SVM	SMOTE Nom	1	17	-16	16
29	KNN	SMOTE Nom	0	16	-16	18	NB	Baseline	1	18	-17	15	NB	SMOTE Nom	0	19	-19	15
30	SVM	SMOTE Nom	0	16	-16	18	RF	ROS	2	22	-20	10	KNN	B-SMOTE	2	24	-22	8
31	KNN	MBA	0	17	-17	17	RF	SMOTE Nom	2	22	-20	10	DT	MBA	2	25	-23	7
32	NB	SMOTE Nom	0	18	-18	16	DT	Baseline	1	26	-25	7	RF	MBA	2	25	-23	7
33	DT	SMOTE Nom	0	20	-20	14	KNN	Baseline	1	27	-26	6	KNN	SMOTE	2	25	-23	7
34	DT	Baseline	0	21	-21	13	RF	Baseline	1	28	-27	5	SVM	MBA	0	31	-31	3
35	SVM	Baseline	0	23	-23	11	SVM	Baseline	0	34	-34	0	KNN	MBA	0	32	-32	2

Table 13. IRDP Scenario: Median values for AUC, pd, and pf for each dataset

	Project	**Baseline**	**SMOTE**	**B-SMOTE**	**SVM SMOTE**	**SMOTE Nom**	**ROS**	**MBA**
AUC	Ant	0.56	**0.61**	**0.61**	**0.61**	0.58	0.60	**0.61**
	jEdit	0.68	**0.69**	0.71	0.68	0.67	**0.69**	0.63
	Lucene	0.58	0.53	0.51	0.53	0.49	0.52	**0.59**
	Poi	0.52	0.53	0.51	0.53	0.49	0.52	**0.69**
	Synapse	**0.61**	0.60	0.59	0.60	0.58	**0.61**	0.51
	Xalan	0.66	0.67	0.66	0.68	0.68	**0.69**	0.65
	Xerces	0.54	0.55	0.56	0.53	0.51	0.54	**0.63**
pd	Ant	0.18	0.44	0.42	0.36	0.39	0.35	**0.63**
	jEdit	0.48	0.63	**0.68**	0.61	0.60	0.61	0.64
	Lucene	0.34	0.42	0.45	0.44	**0.47**	0.40	0.35
	Poi	0.09	0.34	0.25	0.19	0.12	0.25	**0.60**
	Synapse	0.28	0.41	0.41	0.42	0.35	0.44	**0.50**
	Xalan	0.32	0.36	**0.46**	0.37	0.37	0.38	0.37
	Xerces	0.20	0.39	0.32	0.26	0.22	0.29	**0.61**
pf	Ant	**0.06**	0.19	0.15	0.16	0.15	0.15	0.43
	jEdit	**0.14**	0.26	0.29	0.24	0.27	0.24	0.39
	Lucene	0.18	0.22	0.21	0.23	0.25	**0.16**	0.18
	Poi	**0.02**	0.20	0.18	0.15	0.14	0.19	0.18
	Synapse	**0.11**	0.23	0.24	0.16	0.21	0.24	0.52
	Xalan	**0.00**	**0.00**	0.09	**0.00**	**0.00**	**0.00**	**0.00**
	Xerces	**0.09**	0.34	0.17	0.19	0.17	0.32	0.38

Table 13 shows the medians of AUC, pd, and pf for each dataset in the IRDP scenario. MBA improved AUC values for the Lucene, Poi, and Xerces datasets and performed similarly to SMOTE, Borderline-SMOTE, and SVM SMOTE for the Ant dataset. MBA outperformed pd on the Ant, Poi, Synapse, and Xerces datasets. Except for the Lucene dataset, MBA and over-sampling techniques provided more false alarms than the Baseline. ROS reduced false alarms for Lucene significantly. The Xalan false alarm rate is 0% because the only testing dataset, Xalan 2.7, has a defect ratio of 99% (see Table 5), and hence none of the models produced a false signal. MBA outperformed the other sample methods on the Poi dataset, significantly improving AUC and pd values while providing as few false alarms as possible.

5.2. Performance Evaluation of Rebalancing Methods for CPDP Scenario

For the CPDP scenario, we created 1015 SDP models using five ML algorithms, seven rebalancing techniques (Baseline, MBA, and five over-sampling techniques), and 29 training and testing dataset pairs, as shown in Table 6.

The quartile plots of performance measures (AUC, pd, and pf) for Baseline, MBA, and over-sampling methods for each ML algorithm are shown in Figure 19. MBA and over-sampling techniques did not significantly increase AUC values for all ML algorithms, similar to the IRDP scenario (see Figure 19's top lane). Seven rebalancing techniques and ML algorithm pairs have AUC values around 0.6. Only MBA produced more false alarms against the Baseline (bottom lane of Figure 19) but improved pd values for all ML methods (middle lane of Figure 19).

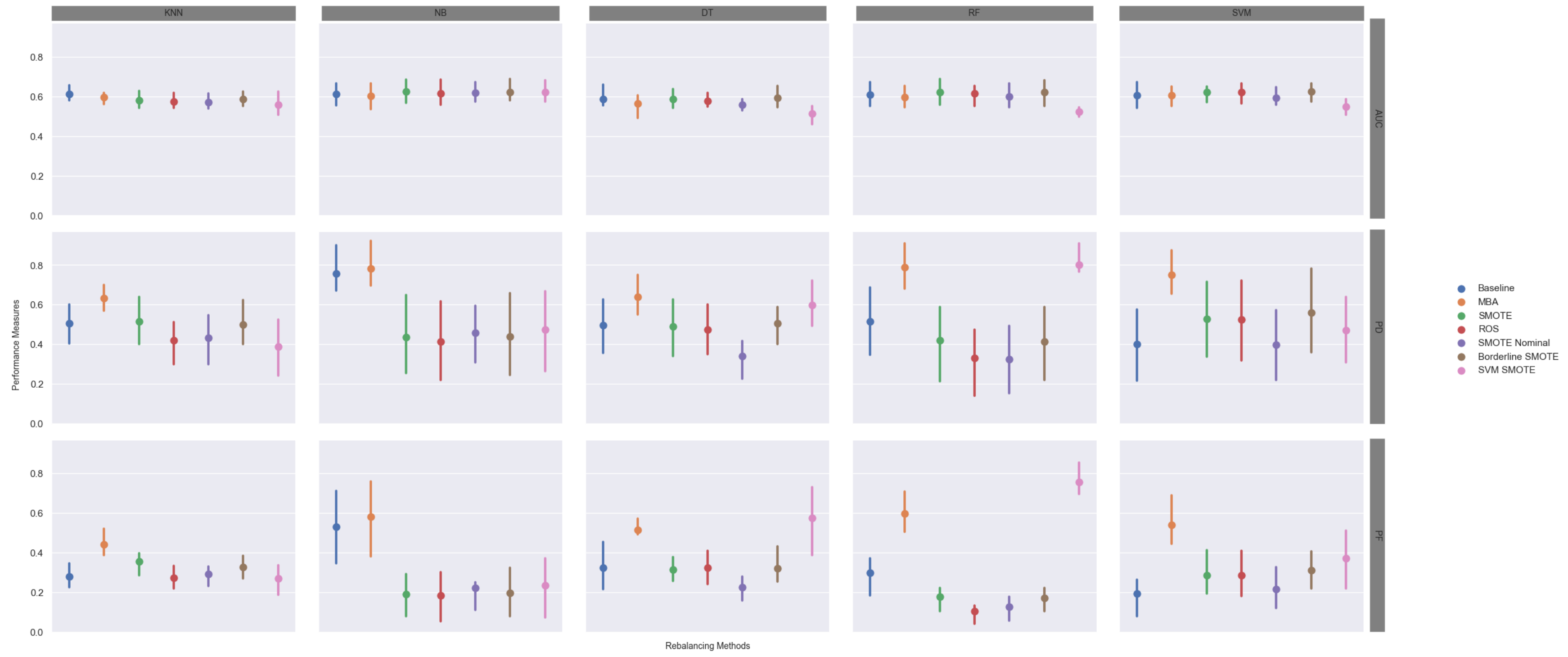

Figure 19. CPDP Scenario: Quartile plots of performance measures (AUC, pd, and pf) for Baseline, MBA, over-sampling techniques per each ML algorithm

Figure 20 uses Wilcoxon signed-rank test win-tie-loss comparisons to show how the MBA performed when compared to the Baseline and five over-sampling techniques. MBA performed better than SVM SMOTE when DT, RF, and SVM algorithms were combined. AUC value for the MBA and NB pair decreased compared to Baseline. In general, MBA did not increase AUC when compared to Baseline and five over-sampling techniques. MBA resulted in a decrease in AUC values in five instances (denoted by orange dots). MBA yielded statistically significantly better outcomes in terms of pd (27 wins, green dots, and 3 ties, blue dots) than Baseline and five over-sampling methods. In 28 out of 30 cases (one tie and one loss), MBA worsened pf values at the expense of higher pd values. In terms of three performance measures, MBA outperformed “SVM SMOTE & DT” by improving AUC with tied pd and pf values and “SVM SMOTE & RF” with better AUC and pf, and tied pd.

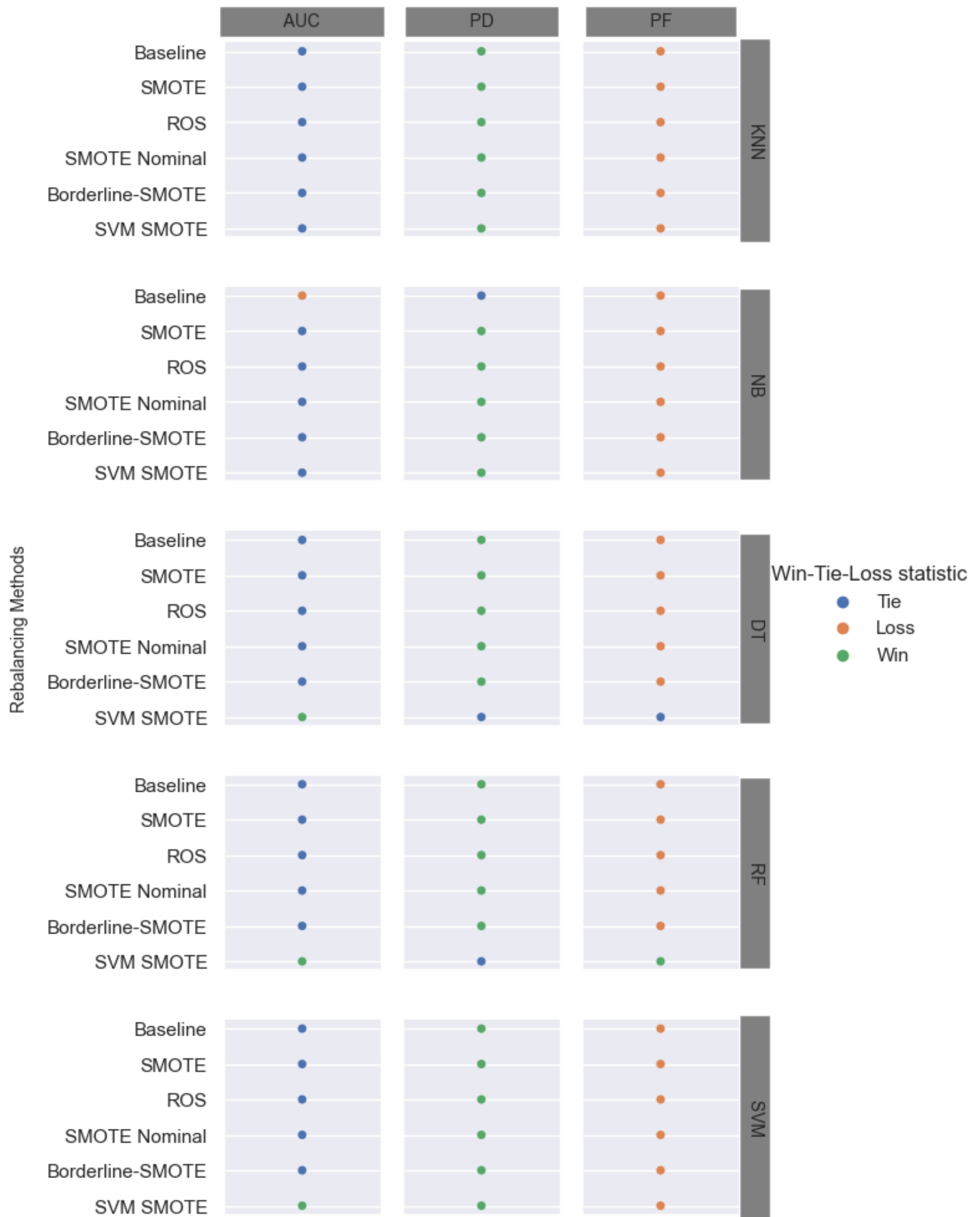

Figure 20. CPDP Scenario: Wilcoxon test win-tie-loss comparison of MBA vs. Baseline, SMOTE, ROS, SMOTE Nominal, Borderline-SMOTE, SVM SMOTE across all datasets per each ML algorithm and performance measures (AUC, pd, and pf)

In terms of AUC, pd, and pf performance measures, Table 14 displays the top and worst performing combinations of rebalancing techniques and ML algorithms. Wilcoxon signed-rank test was used to calculate win-tie-loss results, which were then displayed in descending order by the difference between wins and losses. Due to space restrictions, we only provide the top and lowest 10 performers for each performance measure.

None of the ML algorithm and sampling method combinations dominated in terms of the AUC measure. Despite being ranked as the top performers, SVM and RF combined with SMOTE and Borderline SMOTE were unable to outperform more than half of the other combinations (18 wins and 16 ties for SVM; 16 wins and 18 ties for RF). In terms of AUC, MBA was unable to perform well (MBA combined with SVM 16th, RF 17th, KNN 18th, and NB 19th). With two wins, 18 losses, and 14 ties, MBA together with DF underperformed. MBA paired with NB, RF, SVM, KNN, and DT placed between third and seventh, respectively, in the recall measure. The values for the wins-losses out of 34 range from 30 to 21. The combination of RF and SVM SMOTE scored the best in terms of recall but had the worst AUC and pf values. MBA's relatively good recall values were at the cost of worsened pf values.

Table 14. CPDP Scenario: Rankings for performance measures in terms of wins (W), losses (L), wins-losses (W-L), and ties (T)

	AUC						pd						pf					
#	**ML**	**Sampl. Tech.**	**W**	**L**	**W-L**	**T**	**ML**	**Sampl. Tech.**	**W**	**L**	**W-L**	**T**	**ML**	**Sampl. Tech.**	**W**	**L**	**W-L**	**T**
1	SVM	SMOTE	18	0	18	16	RF	SVM SMOTE	31	0	31	3	RF	ROS	34	0	34	0
2	SVM	B-SMOTE	18	0	18	16	NB	Baseline	30	0	30	4	RF	SMOTE Nom	31	1	30	2
3	RF	SMOTE	16	0	16	18	NB	MBA	30	0	30	4	NB	ROS	27	1	26	6
4	RF	B-SMOTE	16	0	16	18	RF	MBA	30	0	30	4	NB	SMOTE	26	2	24	6
5	SVM	ROS	15	0	15	19	SVM	MBA	30	1	29	3	SVM	Baseline	25	2	23	7
6	NB	SMOTE	13	0	13	21	KNN	MBA	26	5	21	3	RF	B-SMOTE	25	2	23	7
7	NB	SMOTE Nom	13	0	13	21	DT	MBA	26	5	21	3	RF	SMOTE	24	2	22	8
8	RF	ROS	12	0	12	22	SVM	B-SMOTE	20	5	15	9	NB	B-SMOTE	25	4	21	5
9	KNN	Baseline	11	0	11	23	DT	SVM SMOTE	19	5	14	10	SVM	SMOTE Nom	22	3	19	9
10	NB	Baseline	11	0	11	23	SVM	SMOTE	15	8	7	11	DT	SMOTE Nom	20	4	16	10
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
26	KNN	SMOTE	4	13	-9	17	KNN	ROS	4	19	-15	11	KNN	SMOTE	8	20	-12	6
27	KNN	ROS	3	16	-13	15	RF	SMOTE	3	18	-15	13	SVM	SVM SMOTE	7	21	-14	6
28	DT	ROS	2	15	-13	17	RF	B-SMOTE	3	18	-15	13	KNN	MBA	6	26	-20	2
29	KNN	SMOTE Nom	2	16	-14	16	SVM	SMOTE Nom	2	20	-18	12	NB	Baseline	3	27	-24	4
30	KNN	SVM-SMOTE	2	16	-14	16	SVM	Baseline	2	21	-19	11	DT	MBA	2	28	-26	4
31	DT	MBA	2	18	-16	14	NB	ROS	2	22	-20	10	SVM	MBA	1	28	-27	5
32	DT	SMOTE Nom	2	25	-23	7	KNN	SVM SMOTE	2	23	-21	9	DT	SVM SMOTE	1	28	-27	5
33	SVM	SVM SMOTE	2	27	-25	5	DT	SMOTE Nom	0	28	-28	6	NB	MBA	1	29	-28	4
34	DT	SVM SMOTE	0	33	-33	1	RF	ROS	0	32	-32	2	RF	MBA	1	30	-29	3
35	RF	SVM SMOTE	0	33	-33	1	RF	SMOTE Nom	0	32	-32	2	RF	SVM SMOTE	0	34	-34	0

Table 15. CPDP Scenario: Median values for AUC, pd, and pf for each dataset

	Project	Baseline	SMOTE	B-SMOTE	SVM SMOTE	SMOTE Nom	ROS	MBA
AUC	Ant	0.66	**0.67**	0.62	0.58	0.63	0.65	0.58
	jEdit	**0.67**	**0.67**	**0.67**	0.57	0.62	0.63	0.63
	Lucene	**0.61**	0.58	0.58	0.53	0.56	0.59	0.56
	pBeans	0.54	0.60	**0.64**	0.52	0.59	0.63	0.59
	Poi	0.59	0.54	0.55	0.52	0.52	0.54	**0.60**
	Synapse	0.62	0.65	**0.66**	0.58	0.63	0.65	0.58
	Velocity	0.56	0.56	0.56	0.52	0.54	0.56	**0.57**
	Xalan	0.56	0.57	0.59	0.51	0.56	0.58	**0.67**
	Xerces	0.54	0.55	**0.58**	0.55	0.55	0.54	0.54
pd	Ant	0.72	0.69	0.65	0.69	0.59	0.60	**0.83**
	jEdit	0.53	0.55	0.59	0.58	0.49	0.45	**0.70**
	Lucene	0.46	0.33	0.37	0.33	0.32	0.31	**0.72**
	pBeans	0.50	0.48	0.45	**0.70**	0.30	0.43	**0.70**
	Poi	0.48	0.31	0.35	0.39	0.27	0.35	**0.79**
	Synapse	0.65	0.67	0.69	0.73	0.55	0.62	**0.90**
	Velocity	0.42	0.34	0.36	0.35	0.22	0.28	**0.62**
	Xalan	0.29	0.30	0.31	0.39	0.21	0.21	**0.64**
	Xerces	0.34	0.32	0.34	0.30	0.26	0.29	**0.53**
pf	Ant	0.40	0.41	0.43	0.52	**0.31**	0.36	0.63
	jEdit	0.22	0.22	0.26	0.40	0.24	**0.21**	0.45
	Lucene	0.26	0.17	0.22	0.33	0.18	**0.16**	0.56
	pBeans	0.33	0.29	0.30	0.47	**0.04**	0.27	0.46
	Poi	0.31	0.20	0.25	0.28	**0.19**	0.27	0.54
	Synapse	0.38	0.39	0.37	0.52	**0.31**	0.35	0.69
	Velocity	0.34	0.22	0.25	0.37	**0.14**	0.18	0.53
	Xalan	0.09	0.09	0.04	0.28	0.09	**0.02**	0.32
	Xerces	0.27	0.20	**0.16**	0.22	0.22	0.17	0.39

AUC, pd, and pf median values for each dataset for the CPDP scenario are listed in Table 15. AUC values for the Poi, Velocity, and Xalan datasets were improved using MBA. The most effective method for pBeans, Synapse, and Xerces was B-SMOTE. SMOTE and B-SMOTE displayed the same performance as Baseline for the jEdit dataset. No rebalancing method raised the AUC value over the baseline. MBA fared better than all other strategies for all datasets in terms of the pd measure, with the exception of a tie for the pBeans dataset. With the exception of Xerces, SMOTE Nominal and ROS generated the fewest false alarms across all datasets. B-SMOTE produced the best median value for Xerces. For all datasets, MBA generated the most false alarms.

5.3. Stability of MBA for IRDP Scenario

To assess the stability of MBA for IRDP scenario, we investigated software measure changes and performance changes when datasets are balanced. Additionally, to test if balancing a dataset with a 50% defect ratio is the right approach, as described in the literature, we injected mutants with 30%, 40%, and 50% steps. In this way, we were able to see the trend of performance change with respect to defect ratio change. We provided the number of changed software measures and the number of new defects by MBA for each dataset in Appendix B to illustrate the impact of MBA on software measures. We investigated the impact of increasing the number of mutants on the performance of SDP models in Appendix A to show how MBA performed in detail. For the IRDP scenario, we created 400 SDP models using five ML algorithms, four different defect levels (Baseline, MBA 0.3, MBA 0.4, and MBA 0.5), and 20 training and testing dataset pairs. For IRDP scenario, there are 27 dataset pairs, as shown in Table 6 but we could not include seven dataset pairs (JEdit 3.2.1 → *4.0, 4.1, 4.2, 4.3,* Lucene 2.0 → *2.2, 2.4,* Xalan 2.6 → *2.7*) because their training datasets (JEdit 3.2.1, Lucene 2.0, Xalan 2.6) have a defect ratio more than 30%. The quartile plots of performance measures (AUC, pd, and pf) for Baseline, MBA 0.3, MBA 0.4, and MBA 0.5 for each ML algorithm are shown in Figure 21. For 48 of 100 cases (20 dataset pairs x 5 ML algorithms), each increase in defect ratio resulted in higher recall values. The median recall values are 0.39, 0.46, and 0.63 for 30%, 40%, and 50% defect ratios, respectively (see Figure 21's middle lane). For 63 of 100 cases, each increase in defect ratio resulted in higher false alarm values. The median false alarm values are 0.14, 0.23, and 0.41 for 30%, 40%, and 50%

defect ratios, respectively (see Figure 21's bottom lane). In general, we can observe that increasing defect ratio with MBA increased both recall and false alarm values. For 11 of 100 cases, each increase in defect ratio resulted in higher AUC values. The median AUC values are 0.62, 0.62, and 0.61 for 30%, 40%, and 50% defect ratios, respectively (see Figure 21's top lane).

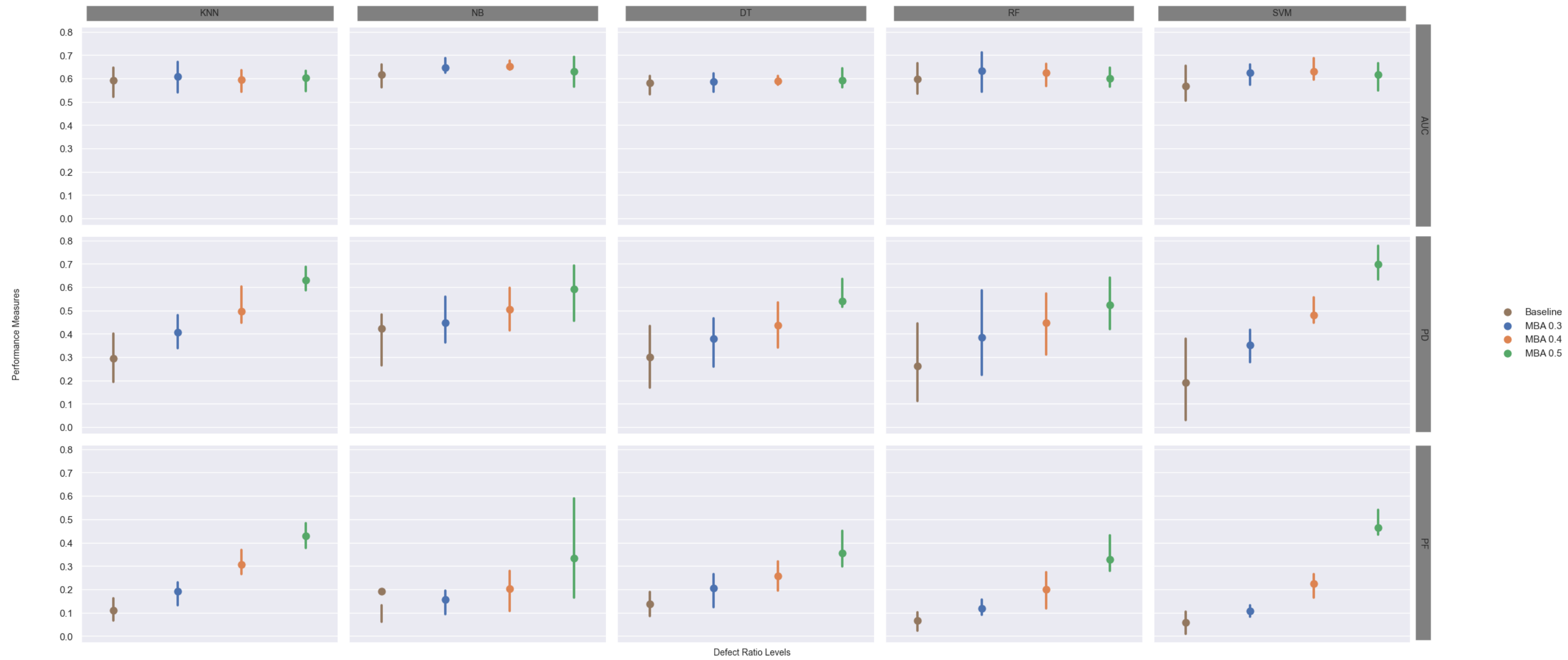

Figure 21. IRDP Scenario: Quartile plots of performance measures (AUC, pd, and pf) for Baseline, MBA 0.3, MBA 0.4, and MBA 0.5 for each ML algorithm

We also investigated the impact of changing software measures on prediction performance of SDP models. For each training dataset, we calculated the change of software measures between Baseline and three different defect levels (MBA 0.3, MBA 0.4, and MBA 0.5) and normalized them to enable proper analysis. The change percentage of software measures (SMC) is calculated using the formula below to compare two datasets:

$$SMC\ (\%) = \frac{\sum_{i=1}^{f} \sum_{j=1}^{s} (M_{i,j})}{f \times s} \times 100 \tag{5.1}$$

where f is the number of files in a dataset, s is the number of performance measures which is 20 (see Table 7) in our study, $M_{i,j}$ is 1 if i^{th} file of jth software measure differs from Baseline, otherwise 0. The scatter plots of performance measures (AUC, pd, and pf) for MBA 0.3, MBA 0.4, and MBA 0.5 for each ML algorithm are shown in Figure 22. To check the correlation between SMC and performance measures, we decided to calculate Pearson correlation coefficient if normality is not violated otherwise, we chose Kendall's Tau correlation coefficient which is non-parametric alternative of Pearson correlation test. 10 of 15 ML algorithm and performance measure pairs satisfied normality but SMC violated (the same SMC is used for all cases), so we used Kendall's Tau correlation coefficients to show the relation between performance measures and SMC for all ML algorithms (KNN, NB, DT, RF, and SVM) as shown in Table 16. Low and insignificant correlation values of less than 0.09% were observed for AUC. SMC and the other performance measures (pd and pf) were positively and significantly correlated for all ML techniques. However, a positive and significant correlation for pf measure indicates that increasing SMC causes producing more false alarms which is undesirable for better performance. The software measures' change has little impact on NB, DT, and RF because they are the most resistant to SMC.

Table 16. IRDP Scenario: Kendall's Tau correlation analysis between SMC and each performance measure per each ML algorithm

		Software Measure Change	
Performance Measure	**ML Method**	SMC	
		Correlation	**Significance (p=0.05)**
AUC	KNN	-0.02	0.85
	NB	0.01	0.89
	DT	0.08	0.35
	RF	-0.07	0.46
	SVM	-0.03	0.72
pd	KNN	**0.39**	**0.00**
	NB	**0.24**	**0.01**
	DT	**0.28**	**0.00**
	RF	**0.18**	**0.04**
	SVM	**0.46**	**0.00**
pf	KNN	**0.50**	**0.00**
	NB	**0.25**	**0.00**
	DT	**0.29**	**0.00**
	RF	**0.36**	**0.00**
	SVM	**0.51**	**0.00**

Figure 22. IRDP Scenario: The scatter plots of performance measures (AUC, pd, and pf) for MBA 0.3, MBA 0.4, and MBA 0.5 for each ML algorithm

5.4. Stability of MBA for CPDP Scenario

To assess the stability of MBA for CPDP scenario, we investigated software measure changes and performance changes when datasets are balanced. Additionally, to test if balancing a dataset with a 50% defect ratio is the right approach, as described in the literature, we injected mutants with 30%, 40%, and 50% steps. In this way, we were able to see the trend of performance change with respect to defect ratio change. We provided the number of changed software measures and the number of new defects by MBA for each dataset in Appendix D to illustrate the impact of MBA on software measures. We investigated the impact of increasing the number of mutants on the performance of SDP models in Appendix C to show how MBA performed in detail. For the CPDP scenario, we created 1540 SDP models using five ML algorithms, four different defect levels (Baseline, MBA 0.3, MBA 0.4, and MBA 0.5), and 77 training and testing dataset pairs. There are 29 dataset pairs for the CPDP scenario, but we were unable to build the training datasets as shown in Table 6 since some of the training datasets (JEdit 3.2.1, Lucene 2.0, Xalan 2.6) have a defect ratio more than 30%. All of these three projects' versions might be used as a testing dataset because we did not utilize them in a training dataset. For all of the resting training datasets, it led to a drop of three training datasets but an increase of eight testing datasets, so in total 77 training and testing dataset pairs are created as shown in Appendix C. The quartile plots of performance measures (AUC, pd, and pf) for Baseline, MBA 0.3, MBA 0.4, and MBA 0.5 for each ML algorithm are shown in Figure 23. For 192 of 385 cases (77 dataset pairs x 5 ML algorithms), each increase in defect ratio resulted in higher recall values. The median recall values are 0.34, 0.48, and 0.55 for 30%, 40%, and 50% defect ratios, respectively (see Figure 23's middle lane). For 203 of 385 cases, each increase in defect ratio resulted in higher false alarm values. The median false alarm values are 0.23, 0.34, and 0.43 for 30%, 40%, and 50% defect ratios, respectively (see Figure 23's bottom lane). In general, we can observe that increasing defect ratio with MBA increased both recall and false alarm values. For 24 of 385 cases, each increase in defect ratio resulted in higher AUC values. The median AUC values are 0.56, 0.58, and 0.56 for 30%, 40%, and 50% defect ratios, respectively (see Figure 23's top lane).

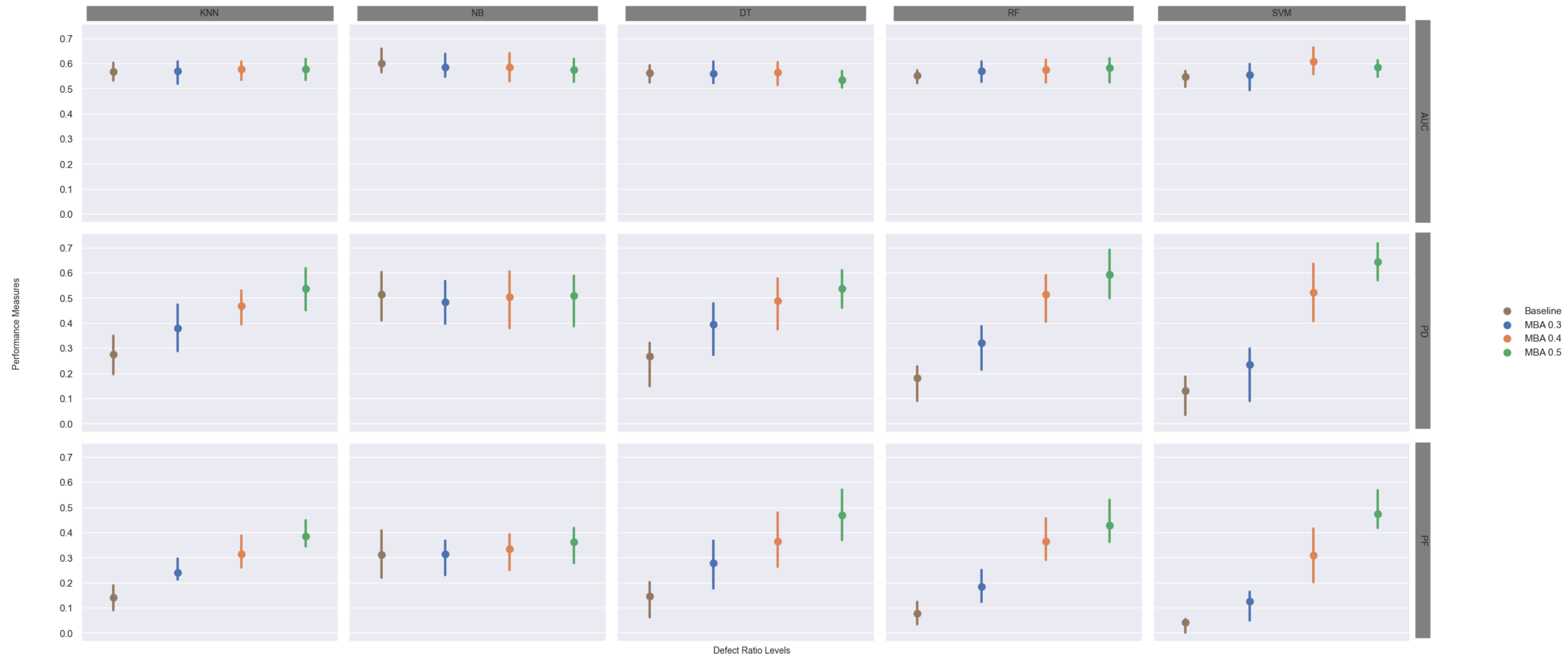

Figure 23. CPDP Scenario: Quartile plots of performance measures (AUC, pd, and pf) for Baseline, MBA 0.3, MBA 0.4, and MBA 0.5 for each ML algorithm

We also investigated the impact of changing software measures on prediction performance of SDP models. For each training dataset, we calculated the change of software measures between Baseline and three different defect levels (MBA 0.3, MBA 0.4, and MBA 0.5) and normalized them to enable proper analysis. The scatter plots of performance measures (AUC, pd, and pf) for MBA 0.3, MBA 0.4, and MBA 0.5 for each ML algorithm are shown in Figure 24. To check the correlation between SMC and performance measures, we decided to calculate Pearson correlation coefficient if normality is not violated otherwise, we chose Kendall's Tau correlation coefficient which is non-parametric alternative of Pearson correlation test. 4 of 15 ML algorithm and performance measure pairs satisfied normality but SMC violated (the same SMC is used for all cases), so we used Kendall's Tau correlation coefficients to show the relation between performance measures and SMC for all ML algorithms (KNN, NB, DT, RF, and SVM) as shown in Table 177. Low correlation values of less than 0.11% were observed for AUC. On a rare occasion did the SMC have significant negative correlation with AUC for DT. SMC and the other performance measures (pd and pf) were positively and significantly correlated for all ML techniques. However, a positive and significant correlation for pf measure indicates that increasing SMC causes producing more false alarms which is undesirable for better performance. The software measures' change has little impact on NB because it is the most resistant to SMC. The software measures' change has remarkable impact on DT and RF different than IRDP scenario.

Table 17. CPDP Scenario: Kendall's Tau correlation analysis between SMC and each performance measure per each ML algorithm

		Software Measure Change	
Performance Measure	**ML Method**	SMC	
		Correlation	**Significance (p=0.05)**
AUC	KNN	-0.02	0.67
	NB	0.04	0.43
	DT	**-0.10**	**0.02**
	RF	0.03	0.52
	SVM	0.10	0.10
pd	KNN	**0.35**	**0.00**
	NB	**0.13**	**0.00**
	DT	**0.30**	**0.00**
	RF	**0.44**	**0.00**
	SVM	**0.69**	**0.00**
pf	KNN	**0.44**	**0.00**
	NB	**0.18**	**0.00**
	DT	**0.36**	**0.00**
	RF	**0.48**	**0.00**
	SVM	**0.71**	**0.00**

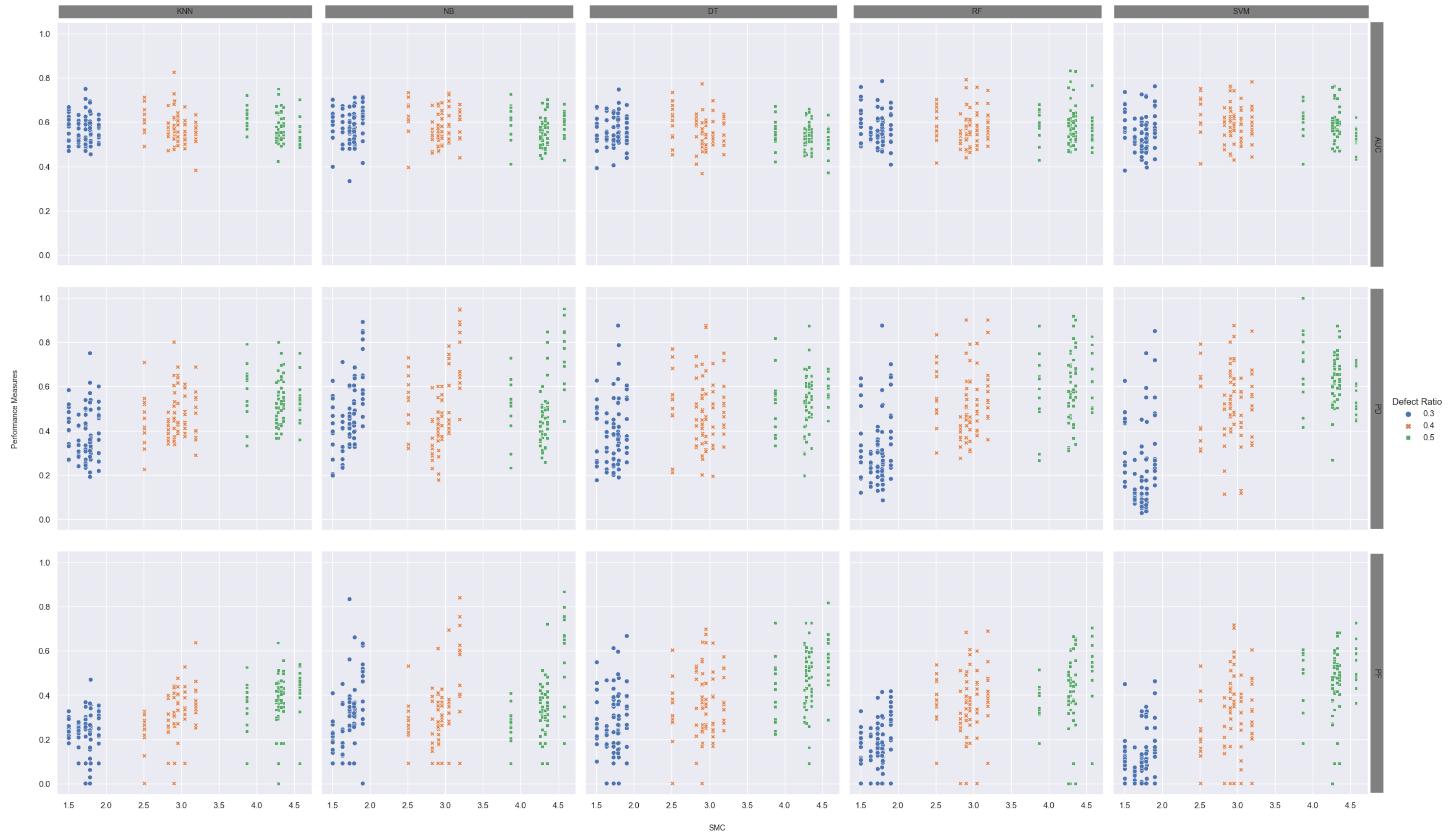

Figure 24. CPDP Scenario: The scatter plots of performance measures (AUC, pd, and pf) for MBA 0.3, MBA 0.4, and MBA 0.5 for each ML algorithm

5.5. General Discussion

The optimal predictor should have a high probability of finding a defect (pd/recall) and produce few false alarms (pf). According to earlier studies, this ideal condition is extremely uncommon (Menzies, Greenwald, and Frank 2007). High detection probabilities can be achieved at the expense of more false alarms (Menzies, Greenwald, and Frank 2007; Turhan et al. 2009). A comprehensive meta-analysis confirms the difficulty of achieving high recall results without reducing precision (Hosseini, Turhan, and Gunarathna 2019). However, they found that when the factors affecting performance are addressed, CPDP techniques can reach comparable predictive performance to WRDP (Hosseini, Turhan, and Gunarathna 2019). In our trials, we encountered identical results for the IRDP and CPDP scenarios. Only MBA consistently beat the Baseline in the CPDP scenario, despite the fact that nearly all rebalancing approaches increased recall compared to Baseline in the IRDP scenario. According to reports in the literature, the increase in false alarms for both scenarios was caused by the improvement in recall (Hosseini, Turhan, and Gunarathna 2019; Menzies, Greenwald, and Frank 2007; Turhan et al. 2009). Only three datasets—Lucene, Poi, and Xerces for the IRDP scenario and Poi, Velocity, and Xalan for the CPDP scenario—were used by MBA to get the best AUC values for each scenario. Therefore, we can not draw the conclusion that an MBA always raises AUC. The median AUC values range between 0.53 and 0.90 for the CPDP scenario and 0.35 - 0.63 for the IRDP scenario, indicating that the recall values for the CPDP scenario were greater than the ones for IRDP.

We had to understand how MBA affected software measures in our experiment because we used software measures to build SDP models. In order to achieve this, we counted the number of datasets affected by mutation for each software measure. All training datasets mentioned in Table 6 had their RFC, LOC, AMC, MAX CC, and AVG CC measures modified. In the majority of datasets (eight to twelve), LCOM, LCOM3, CBO, CA, CE, CBM, and IC measures were influenced. In a small number of datasets (two to four), WMC, DAM, MFA, and CAM indicators were impacted. DIT, NOC, NPM, and MOA measures were unaffected by the mutation operators described in Table 2 for any dataset. As each software measure's contribution to SDP may differ depending on the project (Esteves et al. 2020). Software measures that were left unchanged or that were not

changed sufficiently may have degraded MBA's performance. For instance, CAM and WMC measures, which were impacted by mutations for three and four datasets respectively, were reported as effective predictors of defects (Al Dallal and Briand 2010; Radjenović et al. 2013). LOC, AMC, DAM, RFC, and NPM are mentioned as important features for SDP (Esteves et al. 2020). Mutations had no effect on NPM and had an impact on DAM for only two datasets. In order to have an impact on WMC, it is worthwhile to expand the collection of mutation operators to include mutation operators such as overriding and overloading method deletion (Ma and Offutt 2005). Other mutation operators, such as the access modifier change operator for NPM, can be used to affect unchanged software measures (Ma and Offutt 2005). Additionally, measures that have been shown to be successful for SDP in the literature, including Similarity-based Class Cohesion (Al Dallal and Briand 2010), can be included in the software measure suite. We examined the correlation between the number of defects in a file and performance measures (AUC, pd, and pf) to see how the number of defects in a file affected the performance of SDP models. No performance measure is statistically significantly correlated with the quantity of defects in a file, as per Kendall’s Tau correlation test. On the performance of SDP models, the change percentage of software measures (SMC) contributes more than the quantity of defects. In other words, the quality of the mutation is more important than the amount.

We can not conclude that increasing the defect ratio with MBA increases performance, as mentioned in many other studies (Bennin et al., 2018). One of the biggest problems for MBA is locating the exact defect or defects at the same time and getting the right software measures changed for the right dataset. In some cases, more mutant addition causes a decrease in the performance of SDP models for some of the ML methods. We examined these examples and found that the software measures had not changed enough. Since the ML process only understands measures, this is equivalent to converting labels to defective without mutant addition. For instance, Synapse 1.1 has a significant number of changes only on RFC, LOC, and AMC and its performance decreased when the defect ratio increased. On the other hand, Xerces 1.2 has a significant number of changes to RFC, LCOM, LCOM3, LOC, CBM, AMC, MAX CC, and AVG CC and its performance increased when the defect ratio increased. Also, two SDP models of Poi 2.0RC1 have an increase in both recall and AUC values of 0.5 defect ratio. When we inspected the number of changed software measures, we noticed that there were a notable number of CBM changes. Because all training datasets share similar projects with

one another, the CPDP scenario did not see the same variation in software measures change as the IRDP scenario. Since SDP models use software measures as predictors of defects, the performance of MBA depends on how much mutants impact software measures. With the set of mutation operators used in this study, we are not able to conclude that MBA constantly improves defect prediction performance as training datasets are more balanced. Therefore, we propose to investigate a better set of mutation operators that lead to more meaningful changes in software measures to improve prediction performance.

Due to the need to execute a test suite on each mutation, mutation testing is seen as expensive (Jahangirova and Tonella 2020). MBA consists solely of applying mutation operators to source code, as seen in Figure 9 and Figure 16. A test suite's execution and accessibility are unimportant to MBA. As a result, we contrasted how long it took to apply mutation operators to balancing via sampling techniques. A personal computer running the Ubuntu 18.04 LTS operating system and one physical Intel Core i7-10875H processor running at 2.30 GHz with eight cores and sixteen threads was used to carry out the experiment. Over-sampling techniques required no more than 1 msec per dataset. On Xalan 2.6, however, mutating a dataset took an average of 0.89 seconds and as long as 1.9 seconds. Since an SDP model is only trained after prediction performance has significantly declined, the additional time required for mutation is therefore negligible. Furthermore, the mutation procedure adds only 1-2 seconds of time to the model construction process for each dataset. Although this time cost depends on the tool used for the mutation, we do not expect a significant increase as a result of the tool change.

The significance of repeatability and replication of ML studies in software engineering research was highlighted by the fact that several researchers do not publish their artifacts, such as source code, datasets, parameters used to develop models, and other information for reproducing experiments (Giray et al. 2023; Liu et al. 2021). Due to the unavailability of (1) project source codes, (2) project dependencies, (3) the Java SDK version used to build source codes, and (4) the details of the software measure extraction tool (in

our experiment, the version of the ckjm-extended tool and the Java version to run ckjm-extended), we encountered several problems. We were able to acquire the source codes and dependencies for the projects listed in Table 4 as a result of these problems. We performed many evaluations to determine the ideal Java SDK version and the ideal version of ckjm-extended (versions 2.1, 2.2, and 2.3 were used in the evaluations), as detailed in section 4.2. We could reproduce the measures MFA, IC, CBM, MAX CC, and AVG CC with some minor differences reported by (Jureczko and Madeyski 2010). We publish all artifacts and information publicly to clear the way for solving the problem of reproducibility and replication.

The wide variety of performance measures provided in the research presents another challenge in utilizing the SDP literature that already exists (Y. Yang et al. 2022). Multiple-number assessment criteria make it more difficult to evaluate ML models (Ng 2019). Therefore, using a suitable number of measures rather than a single-number evaluation measure is recommended whenever that is not practicable. Unfortunately, researchers have not come to an agreement on a standard set of performance measures for evaluating SDP models (Giray et al. 2023). To evaluate different aspects of the models, we used three performance measures. We were able to compare our findings with certain research that accurately reported performance measures. To address this issue, we publish the values for a variety of performance measures that were obtained throughout the experiment in the online repository[1].

Despite the challenges with SDP, there is a lot of interest in defect predictors because alternative testing approaches are more expensive and time-consuming. Out of 395 practitioners, more than 90% said that they would be willing to embrace SDP models (Wan et al. 2020). Interest in SDP models and calls for further research have been reported in the literature (Zimmermann et al. 2009; Lewis et al. 2013). Therefore, we think that MBA may have additional advantages for creating better predictors. The prospective MBA enhancement opportunities are detailed in chapter 6 and reserved for future investigation.

[1] https://github.com/dincerguner/Mutant-based-Approach-to-Alleviate-CIP-in-SDP

5.6. Threats to Validity

The data that are used to produce our experimental results is a threat to validity because there are some differences between our software measure dataset and other studies, as we discussed and stated the possible reasons in section 4.2. We must exclude some of the projects of PROMISE dataset as we discussed in section 4.1. All the projects used were from the PROMISE repository and were heavily utilized in numerous defect prediction studies. Because we used a limited number of repositories as datasets, our findings might not be appropriate for all software projects.

The collection of software measures we utilized to create SDP models is one of the potential threats to our results. Our findings can not be generalized to different software measures in general. Static code measures appear to perform well, according to earlier research (Menzies et al. 2010). However, it might be beneficial to add measures to the measure suite that have been shown to be useful in the literature, such as Similarity-based Class Cohesion (Al Dallal and Briand 2010). Additionally, as we discussed in section 5.3, variety and combinations of mutants are important for MBA.

The ML methods that we used in our setup are another threat to validity. In our study, we only used five different ML methods. We could not validate our results with many other algorithms because of time limitations, but we chose the five ML methods with respect to popularity and success in SDP. Data science is a very large and dynamic field with many different algorithms. State-of-the-art approaches, in particular, are built with deep learning models, but deep learning models require a large number of instances in training datasets (Giray et al. 2023). Because the dataset in our study was limited, we did not use deep learning techniques.

We randomly divided the training datasets into training (80%) and validation (20%) sets as part of the ML model training process. We used a five-fold cross-validation procedure and presented the average findings to lessen the impact of this random split. The selection of hyperparameters during model training presents another threat. We utilized grid search to optimize a pool of hyperparameters that we had chosen from the literature, as given in Table 9. This is justified by the fact that meta-heuristic methods can assist in computing the ideal values and that random selection of hyperparameters may result in lower prediction performance (N. Zhang et al. 2022).

The variety of over-sampling methods that we used in our analysis is a threat to validity. We used five different over-sampling methods with MBA and Baseline in our experiments. Even if we choose these methods with respect to popularity in SDP, there are other sampling strategies that we did not include. Therefore, we could not validate our results with many other over-sampling algorithms because of space and time limitations.

CHAPTER 6

CONCLUSION AND FUTURE WORK

The high degree of class imbalance in most real-world defect datasets makes resampling techniques necessary to alleviate CIP. Synthetic minority data instances are produced to balance the distribution between minority and majority class samples in over-sampling techniques. It has been stated that these synthetic approaches improve prediction performance, but they occasionally produce duplicate or inaccurate data instances. Additionally, these over-sampling techniques are domain-agnostic, so the only source is the instances that are minor in the dataset. Therefore, new instances must be similar to the minor instances, which already have less diversity. Exploiting these challenges, we proposed an alternative approach that transforms major class instances into minor class instances with software mutants. Our motivation is balancing the class ratio in the dataset with synthetic mutants and strengthening minor class instance diversity from different than current over-sampling techniques.

We empirically evaluated MBA by comparing it to five other over-sampling approaches (SMOTE, ROS, SMOTE Nominal, Borderline-SMOTE, and SVM SMOTE) and Baseline using five ML methods (KNN, NB, DT, RF, and SVM). We used 13 and 19 imbalanced datasets, whose minor class is defective, for IRDP and CPDP scenarios, respectively. In total, 945 and 1015 different experiment instances were produced for IRDP and CPDP scenarios respectively. In the IRDP scenario, almost all rebalancing techniques increased recall compared to Baseline; however, in the CPDP scenario, only MBA consistently outperformed the Baseline. According to reports in the literature, the improvement in recall resulted in the production of more false alarms (Menzies, Greenwald, and Frank 2007; Turhan et al. 2009; Hosseini, Turhan, and Gunarathna 2019). Only three datasets—Lucene, Poi, and Xerces for the IRDP scenario and Poi, Velocity, and Xalan for the CPDP scenario—were used by MBA to get the best AUC values for each scenario. Therefore, we can not draw the conclusion that an MBA always improves AUC. The median AUC values range between 0.53 and 0.90 for the CPDP scenario and 0.35 - 0.63 for the IRDP scenario, indicating that the recall values for the CPDP scenario were greater than those for IRDP.

In terms of Wilcoxon signed-rank tests, our experimental results show that:

- For recall scores, almost all rebalancing methods outperformed Baseline in Inter-release Defect Prediction (IRDP) scenario but only MBA significantly outperformed Baseline in Cross-project Defect Prediction (CPDP) scenario.
- As stated in literature, the performance increase in recall resulted in the production of more false alarms for both scenarios.
- Only three datasets—Lucene, Poi, and Xerces for the IRDP scenario and Poi, Velocity, and Xalan for the CPDP scenario— used by MBA resulted in the best AUC values, so we can not generalize that MBA outperforms Baseline and the five over-sampling strategies in terms of AUC scores.
- In terms of recall values, the MBA performed better in CPDP than IRDP; specifically, the CPDP scenario's range for median recall values is between 0.53 and 0.90, whereas the IRDP scenario's range is between 0.35 and 0.63.

We also investigated the correlation between the change percentage of software measures (SMC) and performance measures. For both IRDP and CPDP scenarios, in terms of Kendall's Tau correlation analysis, our results show that:

- There was a significant and positive correlation between SMC and recall.
- There was also a significant and positive correlation between SMC and false alarm.
- On the other hand, there was no significant correlation between SMC and AUC.
- NB is the less impacted ML method from software measure changes by MBA.

We see the following areas of future investigation for ourselves and other possible researchers based on the limitations and threats to the validity of our study:

- To increase prediction performance, the set of mutation operators can be expanded to simulate a greater variety of software defects. Overriding, overloading, method deletion, and access modifier change operators are examples of additional mutation operators (Ma and Offutt 2005).
- With the use of datasets created using several programming languages, this experiment can be repeated. These programming languages must provide a mutation tool, such as Python's MutPy (Hałas 2011).
- Additional software measures, such as Similarity-based Class Cohesion, can be added to the existing collection of software measures (Al Dallal and Briand 2010).

- Source code can be represented by other schemes, such as AST (Liang et al. 2019) or image (J. Chen et al. 2020).
- Researching the effectiveness of MBA for small-scale and large-scale projects can be another interesting direction (Majumder, Mody, and Menzies 2022).

REFERENCES

Andrews, J. H., L. C. Briand, and Y. Labiche. 2005. “Is Mutation an Appropriate Tool for Testing Experiments? [Software Testing].” In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, 402–11. https://doi.org/10.1109/ICSE.2005.1553583.

Arisholm, Erik, Lionel C. Briand, and Eivind B. Johannessen. 2010. “A Systematic and Comprehensive Investigation of Methods to Build and Evaluate Fault Prediction Models.” *Journal of Systems and Software* 83 (1): 2–17. https://doi.org/https://doi.org/10.1016/j.jss.2009.06.055.

Bahaweres, Rizal Broer, Fajar Agustian, Irman Hermadi, Arif Imam Suroso, and Yandra Arkeman. 2020. “Software Defect Prediction Using Neural Network Based SMOTE.” In *2020 7th International Conference on Electrical Engineering, Computer Sciences and Informatics (EECSI)*, 71–76. https://doi.org/10.23919/EECSI50503.2020.9251874.

Bansiya, J., and C. G. Davis. 2002. “A Hierarchical Model for Object-Oriented Design Quality Assessment.” *IEEE Transactions on Software Engineering* 28 (1): 4–17. https://doi.org/10.1109/32.979986.

Bennin, Kwabena Ebo, Jacky Keung, Passakorn Phannachitta, Akito Monden, and Solomon Mensah. 2018. “MAHAKIL: Diversity Based Oversampling Approach to Alleviate the Class Imbalance Issue in Software Defect Prediction.” *IEEE Transactions on Software Engineering* 44 (6): 534–50. https://doi.org/10.1109/TSE.2017.2731766.

Bharati, Subrato, Prajoy Podder, and M. Rubaiyat Hossain Mondal. 2020. “Diagnosis of Polycystic Ovary Syndrome Using Machine Learning Algorithms.” In *2020 IEEE Region 10 Symposium (TENSYMP)*, 1486–89. https://doi.org/10.1109/TENSYMP50017.2020.9230932.

Bouguila, Nizar, Jian Han Wang, and A. Ben Hamza. 2008. “A Bayesian Approach for Software Quality Prediction.” In *2008 4th International IEEE Conference Intelligent Systems*, 2:11-49-11–54. https://doi.org/10.1109/IS.2008.4670508.

Bowyer, Kevin W., Nitesh V. Chawla, Lawrence O. Hall, and W. Philip Kegelmeyer. 2011. "SMOTE: Synthetic Minority Over-Sampling Technique." *CoRR* abs/1106.1813. http://arxiv.org/abs/1106.1813.

Brownlee, Jason. 2020. *Data Preparation for Machine Learning: Data Cleaning, Feature Selection, and Data Transforms in Python*. Machine Learning Mastery.

Chawla, N. V., K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. 2002. "SMOTE: Synthetic Minority Over-Sampling Technique." *Journal of Artificial Intelligence Research* 16 (June): 321–57. https://doi.org/10.1613/jair.953.

Chen, Jinyin, Keke Hu, Yue Yu, Zhuangzhi Chen, Qi Xuan, Yi Liu, and Vladimir Filkov. 2020. "Software Visualization and Deep Transfer Learning for Effective Software Defect Prediction." In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 578–89. ICSE '20. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3377811.3380389.

Chen, Lin, Bin Fang, Zhaowei Shang, and Yuanyan Tang. 2018. "Tackling Class Overlap and Imbalance Problems in Software Defect Prediction." *Software Quality Journal* 26 (1): 97–125. https://doi.org/10.1007/s11219-016-9342-6.

Chidamber, S. R., and C. F. Kemerer. 1994. "A Metrics Suite for Object Oriented Design." *IEEE Transactions on Software Engineering* 20 (6): 476–93. https://doi.org/10.1109/32.295895.

Dallal, Jehad Al, and Lionel C. Briand. 2010. "An Object-Oriented High-Level Design-Based Class Cohesion Metric." *Information and Software Technology* 52 (12): 1346–61. https://doi.org/https://doi.org/10.1016/j.infsof.2010.08.006.

D'Ambros, Marco, Michele Lanza, and Romain Robbes. 2010. "An Extensive Comparison of Bug Prediction Approaches." In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, 31–41. https://doi.org/10.1109/MSR.2010.5463279.

Domingos, Pedro. 1999. "MetaCost: A General Method for Making Classifiers Cost-Sensitive." In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 155–64. KDD '99. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/312129.312220.

Emam, Khaled El, Saida Benlarbi, Nishith Goel, and Shesh N. Rai. 2001. "Comparing Case-Based Reasoning Classifiers for Predicting High Risk Software Components." *J. Syst. Softw.* 55 (3): 301–20. https://doi.org/10.1016/S0164-1212(00)00079-0.

Erturk, Ezgi, and Ebru Akcapinar Sezer. 2015. “A Comparison of Some Soft Computing Methods for Software Fault Prediction.” *Expert Systems with Applications* 42 (4): 1872–79. https://doi.org/https://doi.org/10.1016/j.eswa.2014.10.025.

Esteves, Geanderson, Eduardo Figueiredo, Adriano Veloso, Markos Viggiato, and Nivio Ziviani. 2020. “Understanding Machine Learning Software Defect Predictions.” *Automated Software Engineering* 27 (3): 369–92. https://doi.org/10.1007/s10515-020-00277-4.

Ferenc, Rudolf, Zoltán Tóth, Gergely Ladányi, István Siket, and Tibor Gyimóthy. 2018. “A Public Unified Bug Dataset for Java.” In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, 12–21. PROMISE’18. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3273934.3273936.

Galar, Mikel, Alberto Fernandez, Edurne Barrenechea, Humberto Bustince, and Francisco Herrera. 2012. “A Review on Ensembles for the Class Imbalance Problem: Bagging-, Boosting-, and Hybrid-Based Approaches.” *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42 (4): 463–84. https://doi.org/10.1109/TSMCC.2011.2161285.

García, V., J. S. Sánchez, and R. A. Mollineda. 2012. “On the Effectiveness of Preprocessing Methods When Dealing with Different Levels of Class Imbalance.” *Knowledge-Based Systems* 25 (1): 13–21. https://doi.org/https://doi.org/10.1016/j.knosys.2011.06.013.

Giray, Görkem, Kwabena Ebo Bennin, Ömer Köksal, Önder Babur, and Bedir Tekinerdogan. 2023. “On the Use of Deep Learning in Software Defect Prediction.” *Journal of Systems and Software* 195: 111537. https://doi.org/https://doi.org/10.1016/j.jss.2022.111537.

Goyal, Somya. 2022. “Handling Class-Imbalance with KNN (Neighbourhood) Under-Sampling for Software Defect Prediction.” *Artificial Intelligence Review* 55 (3): 2023–64. https://doi.org/10.1007/s10462-021-10044-w.

Goyal, Somya, and Pradeep Bhatia. 2020. “Comparison of Machine Learning Techniques for Software Quality Prediction.” *International Journal of Knowledge and Systems Science* 11 (May): 20–40. https://doi.org/10.4018/IJKSS.2020040102.

Guo, L., Y. Ma, B. Cukic, and Harshinder Singh. 2004. “Robust Prediction of Fault-Proneness by Random Forests.” In *15th International Symposium on Software Reliability Engineering*, 417–28. https://doi.org/10.1109/ISSRE.2004.35.

Gyimothy, T., R. Ferenc, and I. Siket. 2005. “Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction.” *IEEE Transactions on Software Engineering* 31 (10): 897–910. https://doi.org/10.1109/TSE.2005.112.

Haixiang, Guo, Li Yijing, Jennifer Shang, Gu Mingyun, Huang Yuanyue, and Gong Bing. 2017. “Learning from Class-Imbalanced Data: Review of Methods and Applications.” *Expert Systems with Applications* 73: 220–39. https://doi.org/https://doi.org/10.1016/j.eswa.2016.12.035.

Hałas, Konrad. 2011. “Mutation Testing in Python.” Doctoral dissertation, Instytut Informatyki.

Hall, Tracy, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2012. “A Systematic Literature Review on Fault Prediction Performance in Software Engineering.” *IEEE Transactions on Software Engineering* 38 (6): 1276–1304. https://doi.org/10.1109/TSE.2011.103.

Hall, Tracy, Min Zhang, David Bowes, and Yi Sun. 2014. “Some Code Smells Have a Significant but Small Effect on Faults.” *ACM Trans. Softw. Eng. Methodol.* 23 (4). https://doi.org/10.1145/2629648.

Han Hui and Wang, Wen-Yuan and Mao Bing-Huan. 2005. “Borderline-SMOTE: A New Over-Sampling Method in Imbalanced Data Sets Learning.” In *Advances in Intelligent Computing*, edited by Xiao-Ping and Huang Guang-Bin Huang De-Shuang and Zhang, 878–87. Berlin, Heidelberg: Springer Berlin Heidelberg.

Hassan, Ahmed E. 2009. “Predicting Faults Using the Complexity of Code Changes.” In *Proceedings of the 31st International Conference on Software Engineering*, 78–88. ICSE ’09. USA: IEEE Computer Society. https://doi.org/10.1109/ICSE.2009.5070510.

He, Haibo, Yang Bai, Edwardo A. Garcia, and Shutao Li. 2008. “ADASYN: Adaptive Synthetic Sampling Approach for Imbalanced Learning.” In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, 1322–28. https://doi.org/10.1109/IJCNN.2008.4633969.

Henderson-Sellers, Brian. 1995. *Object-Oriented Metrics: Measures of Complexity*. USA: Prentice-Hall, Inc.

Hosseini, Seyedrebvar, Burak Turhan, and Dimuthu Gunarathna. 2019. “A Systematic Literature Review and Meta-Analysis on Cross Project Defect Prediction.” *IEEE Transactions on Software Engineering* 45 (2): 111–47. https://doi.org/10.1109/TSE.2017.2770124.

Hulse, Jason Van, Taghi M. Khoshgoftaar, and Amri Napolitano. 2007. “Experimental Perspectives on Learning from Imbalanced Data.” In *Proceedings of the 24th International Conference on Machine Learning*, 935–42. ICML ’07. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1273496.1273614.

Jahangirova, Gunel, and Paolo Tonella. 2020. “An Empirical Evaluation of Mutation Operators for Deep Learning Systems.” In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 74–84. https://doi.org/10.1109/ICST46399.2020.00018.

Japkowicz, Nathalie, and Shaju Stephen. 2002. “The Class Imbalance Problem: A Systematic Study.” *Intell. Data Anal.* 6 (5): 429–49.

Jia, Yue, and Mark Harman. 2011. “An Analysis and Survey of the Development of Mutation Testing.” *IEEE Transactions on Software Engineering* 37 (5): 649–78. https://doi.org/10.1109/TSE.2010.62.

Johnson, Justin M., and Taghi M. Khoshgoftaar. 2019. “Survey on Deep Learning with Class Imbalance.” *Journal of Big Data* 6 (1): 27. https://doi.org/10.1186/s40537-019-0192-5.

Jureczko, Marian, and Lech Madeyski. 2010. “Towards Identifying Software Project Clusters with Regard to Defect Prediction.” In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. PROMISE ’10. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1868328.1868342.

Jureczko, Marian, and Diomidis Spinellis. 2010. “Using Object-Oriented Design Metrics to Predict Software Defects.” In *Models and Methods of System Dependability*.

Just, René, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. “Are Mutants a Valid Substitute for Real Faults in Software Testing?” In *Proceedings of the 22nd ACM SIGSOFT International Symposium on*

Foundations of Software Engineering, 654–65. FSE 2014. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2635868.2635929.

Just, René, Gregory M. Kapfhammer, and Franz Schweiggert. 2012. "Using Non-Redundant Mutation Operators and Test Suite Prioritization to Achieve Efficient and Scalable Mutation Analysis." In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, 11–20. https://doi.org/10.1109/ISSRE.2012.31.

Just, René, Franz Schweiggert, and Gregory M. Kapfhammer. 2011. "MAJOR: An Efficient and Extensible Tool for Mutation Analysis in a Java Compiler." In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 612–15. https://doi.org/10.1109/ASE.2011.6100138.

Khoshgoftaar, T. M., E. B. Allen, J. P. Hudepohl, and S. J. Aud. 1997. "Application of Neural Networks to Software Quality Modeling of a Very Large Telecommunications System." *IEEE Transactions on Neural Networks* 8 (4): 902–9. https://doi.org/10.1109/72.595888.

Khoshgoftaar, Taghi M., and Naeem Seliya. 2003. "Analogy-Based Practical Classification Rules for Software Quality Estimation." *Empirical Software Engineering* 8 (4): 325–50. https://doi.org/10.1023/A:1025316301168.

Kumar, Lov, Sai Krishna Sripada, Ashish Sureka, and Santanu Ku. Rath. 2018. "Effective Fault Prediction Model Developed Using Least Square Support Vector Machine (LSSVM)." *Journal of Systems and Software* 137: 686–712. https://doi.org/https://doi.org/10.1016/j.jss.2017.04.016.

Kumar, Santanu Rath, and Ashish Sureka. 2017. "Using Source Code Metrics and Ensemble Methods for Fault Proneness Prediction," February.

Laradji, Issam H., Mohammad Alshayeb, and Lahouari Ghouti. 2015. "Software Defect Prediction Using Ensemble Learning on Selected Features." *Information and Software Technology* 58: 388–402. https://doi.org/https://doi.org/10.1016/j.infsof.2014.07.005.

Lessmann, Stefan, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings." *IEEE Transactions on Software Engineering* 34 (4): 485–96. https://doi.org/10.1109/TSE.2008.35.

Lewis, Chris, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead. 2013. "Does Bug Prediction Support Human Developers? Findings from

a Google Case Study." In *2013 35th International Conference on Software Engineering (ICSE)*, 372–81. https://doi.org/10.1109/ICSE.2013.6606583.

Li, Ke, Zilin Xiang, Tao Chen, Shuo Wang, and Kay Chen Tan. 2020. "Understanding the Automated Parameter Optimization on Transfer Learning for Cross-Project Defect Prediction: An Empirical Study." In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 566–77. ICSE '20. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3377811.3380360.

Liang, Hongliang, Yue Yu, Lin Jiang, and Zhuosi Xie. 2019. "Seml: A Semantic LSTM Model for Software Defect Prediction." *IEEE Access* 7: 83812–24. https://doi.org/10.1109/ACCESS.2019.2925313.

Liu, Chao, Cuiyun Gao, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. 2021. "On the Reproducibility and Replicability of Deep Learning in Software Engineering." *ACM Trans. Softw. Eng. Methodol.* 31 (1). https://doi.org/10.1145/3477535.

Ma, Yu-Seung, and Jeff Offutt. 2005. "Description of Class Mutation Mutation Operators for Java," April.

Majumder, Suvodeep, Pranav Mody, and Tim Menzies. 2022. "Revisiting Process versus Product Metrics: A Large Scale Analysis." *Empirical Software Engineering* 27 (3): 60. https://doi.org/10.1007/s10664-021-10068-4.

Martin, Robert. 1994. "OO Design Quality Metrics." *An Analysis of Dependencies* 12 (1): 151–70.

McCabe, T. J. 1976. "A Complexity Measure." *IEEE Transactions on Software Engineering* SE-2 (4): 308–20. https://doi.org/10.1109/TSE.1976.233837.

Menzies, Tim, Alex Dekhtyar, Justin Distefano, and Jeremy Greenwald. 2007. "Problems with Precision: A Response to 'Comments on "Data Mining Static Code Attributes to Learn Defect Predictors."'" *IEEE Transactions on Software Engineering* 33 (9): 637–40. https://doi.org/10.1109/TSE.2007.70721.

Menzies, Tim, Jeremy Greenwald, and Art Frank. 2007. "Data Mining Static Code Attributes to Learn Defect Predictors." *IEEE Transactions on Software Engineering* 33 (1): 2–13. https://doi.org/10.1109/TSE.2007.256941.

Menzies, Tim, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. 2010. "Defect Prediction from Static Code Features: Current Results, Limitations,

New Approaches." *Automated Software Engineering* 17 (4): 375–407. https://doi.org/10.1007/s10515-010-0069-5.

Miholca, Diana-Lucia, Gabriela Czibula, and Istvan Gergely Czibula. 2018. "A Novel Approach for Software Defect Prediction through Hybridizing Gradual Relational Association Rules with Artificial Neural Networks." *Information Sciences* 441: 152–70. https://doi.org/https://doi.org/10.1016/j.ins.2018.02.027.

Monden, Akito, Takuma Hayashi, Shoji Shinoda, Kumiko Shirai, Junichi Yoshida, Mike Barker, and Kenichi Matsumoto. 2013. "Assessing the Cost Effectiveness of Fault Prediction in Acceptance Testing." *IEEE Transactions on Software Engineering* 39 (10): 1345–57. https://doi.org/10.1109/TSE.2013.21.

Moser, Raimund, Witold Pedrycz, and Giancarlo Succi. 2008. "A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction." In *Proceedings of the 30th International Conference on Software Engineering*, 181–90. ICSE '08. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1368088.1368114.

Moussa, Rebecca, and Federica Sarro. 2022. "On the Use of Evaluation Measures for Defect Prediction Studies." In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 101–13. ISSTA 2022. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3533767.3534405.

Munson, J. C., and T. M. Khoshgoftaar. 1992. "The Detection of Fault-Prone Programs." *IEEE Transactions on Software Engineering* 18 (5): 423–33. https://doi.org/10.1109/32.135775.

Nagappan, Nachiappan, and Thomas Ball. 2005. "Use of Relative Code Churn Measures to Predict System Defect Density." In *Proceedings of the 27th International Conference on Software Engineering*, 284–92. ICSE '05. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1062455.1062514.

Nagappan, Nachiappan, Thomas Ball, and Andreas Zeller. 2006. "Mining Metrics to Predict Component Failures." In *Proceedings of the 28th International Conference on Software Engineering*, 452–61. ICSE '06. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1134285.1134349.

Namin, Akbar Siami, and Sahitya Kakarla. 2011. "The Use of Mutation in Testing Experiments and Its Sensitivity to External Threats." In *Proceedings of the 2011*

International Symposium on Software Testing and Analysis, 342–52. ISSTA '11. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2001420.2001461.

Ng, Andrew. 2019. "Machine Learning Yearning Technical Strategy for Ai Engineers in the Era of Deep Learning." https://www.mlyearning.org.

Nguyen, Hien M., Eric W. Cooper, and Katsuari Kamei. 2011. "Borderline Over-Sampling for Imbalanced Data Classification." *Int. J. Knowl. Eng. Soft Data Paradigm.* 3 (1): 4–21. https://doi.org/10.1504/IJKESDP.2011.039875.

Offutt, A. Jefferson, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. 1996. "An Experimental Determination of Sufficient Mutant Operators." *ACM Trans. Softw. Eng. Methodol.* 5 (2): 99–118. https://doi.org/10.1145/227607.227610.

Özakıncı, Rana, and Ayça Tarhan. 2018. "Early Software Defect Prediction: A Systematic Map and Review." *Journal of Systems and Software* 144: 216–39. https://doi.org/https://doi.org/10.1016/j.jss.2018.06.025.

Pappas, Paul A., and Venita DePuy. 2004. "An Overview of Non-Parametric Tests in SAS: When, Why, and How." *Paper TU04. Duke Clinical Research Institute, Durham*, 1–5.

Pazzani, Michael, Christopher Merz, Patrick Murphy, Kamal Ali, Timothy Hume, and Clifford Brunk. 1994. "Reducing Misclassification Costs." In *Machine Learning Proceedings 1994*, edited by William W Cohen and Haym Hirsh, 217–25. San Francisco (CA): Morgan Kaufmann. https://doi.org/https://doi.org/10.1016/B978-1-55860-335-6.50034-9.

Provost, Foster J. 2008. "Machine Learning from Imbalanced Data Sets 101." In .

Qu, Yang, Zhenming Li, Jiaoru Zhao, and Hui Li. 2022. "Unbalanced Data Processing for Software Defect Prediction." In *2022 4th International Conference on Data-Driven Optimization of Complex Systems (DOCS)*, 1–6. https://doi.org/10.1109/DOCS55193.2022.9967755.

Quah, Tong-Seng, and Mie Mie Thet Thwin. 2003. "Application of Neural Networks for Software Quality Prediction Using Object-Oriented Metrics." In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, 116–25. https://doi.org/10.1109/ICSM.2003.1235412.

Radjenović, Danijel, Marjan Heričko, Richard Torkar, and Aleš Živkovič. 2013. "Software Fault Prediction Metrics: A Systematic Literature Review." *Information and Software Technology* 55 (8): 1397–1418. https://doi.org/https://doi.org/10.1016/j.infsof.2013.02.009.

Rao, K. Nitalaksheswara, and Ch. Satyananda Reddy. 2020. "A Novel under Sampling Strategy for Efficient Software Defect Analysis of Skewed Distributed Data." *Evolving Systems* 11 (1): 119–31. https://doi.org/10.1007/s12530-018-9261-9.

Rathore, Santosh S., and Sandeep Kumar. 2019. "A Study on Software Fault Prediction Techniques." *Artificial Intelligence Review* 51 (2): 255–327. https://doi.org/10.1007/s10462-017-9563-5.

Rekha G. and Shailaja, K. and Jatoth Chandrashekar. 2022. "Informative Software Defect Data Generation and Prediction: INF-SMOTE." In *Advances in Computing and Data Sciences*, edited by Vipin and Gupta P K. and Flusser Jan and Ören Tuncer Singh Mayank and Tyagi, 179–91. Cham: Springer International Publishing.

Sayyad Shirabad, J., and T. J. Menzies. 2005. "The PROMISE Repository of Software Engineering Databases." http://promise.site.uottawa.ca/SERepository.

Seliya, Naeem, Taghi M. Khoshgoftaar, and Jason Van Hulse. 2009. "A Study on the Relationships of Classifier Performance Metrics." In *2009 21st IEEE International Conference on Tools with Artificial Intelligence*, 59–66. https://doi.org/10.1109/ICTAI.2009.25.

Shanab, Ahmad Abu, Taghi M. Khoshgoftaar, Randall Wald, and Amri Napolitano. 2012. "Impact of Noise and Data Sampling on Stability of Feature Ranking Techniques for Biological Datasets." In *2012 IEEE 13th International Conference on Information Reuse & Integration (IRI)*, 415–22. https://doi.org/10.1109/IRI.2012.6303039.

Siami Namin, Akbar, James Andrews, and Duncan Murdoch. 2008. "Sufficient Mutation Operators for Measuring Test Effectiveness." In *2008 ACM/IEEE 30th International Conference on Software Engineering*, 351–60. https://doi.org/10.1145/1368088.1368136.

Smith, Michael R., Tony Martinez, and Christophe Giraud-Carrier. 2014. "An Instance Level Analysis of Data Complexity." *Machine Learning* 95 (2): 225–56. https://doi.org/10.1007/s10994-013-5422-z.

Song, Qinbao, Zihan Jia, Martin Shepperd, Shi Ying, and Jin Liu. 2011. "A General Software Defect-Proneness Prediction Framework." *IEEE Transactions on Software Engineering* 37 (3): 356–70. https://doi.org/10.1109/TSE.2010.90.

Spinellis, D. 2005. "Tool Writing: A Forgotten Art? (Software Tools)." *IEEE Software* 22 (4): 9–11. https://doi.org/10.1109/MS.2005.111.

Subramanyam, R., and M. S. Krishnan. 2003. "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects." *IEEE Transactions on Software Engineering* 29 (4): 297–310. https://doi.org/10.1109/TSE.2003.1191795.

Sun, Yanmin, Mohamed S. Kamel, Andrew K. C. Wong, and Yang Wang. 2007. "Cost-Sensitive Boosting for Classification of Imbalanced Data." *Pattern Recognition* 40 (12): 3358–78. https://doi.org/https://doi.org/10.1016/j.patcog.2007.04.009.

Tang, Mei-Huei, Ming-Hung Kao, and Mei-Hwa Chen. 1999. "An Empirical Study on Object-Oriented Metrics." In *Proceedings Sixth International Software Metrics Symposium (Cat. No.PR00403)*, 242–49. https://doi.org/10.1109/METRIC.1999.809745.

Tóth Zoltán and Gyimesi, Péter and Ferenc Rudolf. 2016. "A Public Bug Database of GitHub Projects and Its Application in Bug Prediction." In *Computational Science and Its Applications – ICCSA 2016*, edited by Beniamino and Misra Sanjay and Rocha Ana Maria A.C. and Torre Carmelo M. and Taniar David and Apduhan Bernady O. and Stankova Elena and Wang Shangguang Gervasi Osvaldo and Murgante, 625–38. Cham: Springer International Publishing.

Tsai, Chih-Fong, Wei-Chao Lin, Ya-Han Hu, and Guan-Ting Yao. 2019. "Under-Sampling Class Imbalanced Datasets by Combining Clustering Analysis and Instance Selection." *Information Sciences* 477: 47–54. https://doi.org/https://doi.org/10.1016/j.ins.2018.10.029.

Turhan, Burak, and Ayse Bener. 2009. "Analysis of Naive Bayes' Assumptions on Software Fault Data: An Empirical Study." *Data & Knowledge Engineering* 68 (2): 278–90. https://doi.org/https://doi.org/10.1016/j.datak.2008.10.005.

Turhan, Burak, Tim Menzies, Ayşe B. Bener, and Justin Di Stefano. 2009. "On the Relative Value of Cross-Company and within-Company Data for Defect Prediction." *Empirical Software Engineering* 14 (5): 540–78. https://doi.org/10.1007/s10664-008-9103-7.

Vallat, Raphael. 2018. “Pingouin: Statistics in Python.” *The Journal of Open Source Software* 3 (31): 1026.

Virtanen, Pauli, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, et al. 2020. “SciPy 1.0: Fundamental Algorithms for Scientific in Python.” *Nature Methods* 17: 261–72. https://doi.org/10.1038/s41592-019-0686-2.

Vuttipittayamongkol, Pattaramon, and Eyad Elyan. 2020. “Neighbourhood-Based Undersampling Approach for Handling Imbalanced and Overlapped Data.” *Information Sciences* 509: 47–70. https://doi.org/https://doi.org/10.1016/j.ins.2019.08.062.

Wan, Zhiyuan, Xin Xia, Ahmed E. Hassan, David Lo, Jianwei Yin, and Xiaohu Yang. 2020. “Perceptions, Expectations, and Challenges in Defect Prediction.” *IEEE Transactions on Software Engineering* 46 (11): 1241–66. https://doi.org/10.1109/TSE.2018.2877678.

Weiss, Gary, and Foster Provost. 2001. “The Effect of Class Distribution on Classifier Learning: An Empirical Study.” *Tech Rep*, February.

Wong, Ginny Y., Frank H. F. Leung, and Sai-Ho Ling. 2013. “A Novel Evolutionary Preprocessing Method Based on Over-Sampling and under-Sampling for Imbalanced Datasets.” In *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society*, 2354–59. https://doi.org/10.1109/IECON.2013.6699499.

Yang, Li, and Abdallah Shami. 2020. “On Hyperparameter Optimization of Machine Learning Algorithms: Theory and Practice.” *Neurocomputing* 415: 295–316. https://doi.org/https://doi.org/10.1016/j.neucom.2020.07.061.

Yang, Yanming, Xin Xia, David Lo, Tingting Bi, John Grundy, and Xiaohu Yang. 2022. “Predictive Models in Software Engineering: Challenges and Opportunities.” *ACM Trans. Softw. Eng. Methodol.* 31 (3). https://doi.org/10.1145/3503509.

Yen, Show-Jane, and Yue-Shi Lee. 2006. “Cluster-Based Under-Sampling Approaches for Imbalanced Data Distributions.” *Expert Systems with Applications* 36 (October): 5718–27. https://doi.org/10.1016/j.eswa.2008.06.108.

Yoon, Kihoon, and Stephen Kwek. 2007. “A Data Reduction Approach for Resolving the Imbalanced Data Issue in Functional Genomics.” *Neural Computing and Applications* 16 (3): 295–306. https://doi.org/10.1007/s00521-007-0089-7.

Zhang, J., and I. Mani. 2003. "KNN Approach to Unbalanced Data Distributions: A Case Study Involving Information Extraction." In *Proceedings of the ICML'2003 Workshop on Learning from Imbalanced Datasets*.

Zhang, Nana, Shi Ying, Kun Zhu, and Dandan Zhu. 2022. "Software Defect Prediction Based on Stacked Sparse Denoising Autoencoders and Enhanced Extreme Learning Machine." *IET Software* 16 (April). https://doi.org/10.1049/sfw2.12029.

Zhang, Yun, David Lo, Xin Xia, and Jianling Sun. 2018. "Combined Classifier for Cross-Project Defect Prediction: An Extended Empirical Study." *Frontiers of Computer Science* 12 (2): 280–96. https://doi.org/10.1007/s11704-017-6015-y.

Zimmermann, Thomas, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. "Cross-Project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process." In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 91–100. ESEC/FSE '09. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1595696.1595713.

Zimmermann, Thomas, Rahul Premraj, and Andreas Zeller. 2007. "Predicting Defects for Eclipse." In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, 9. https://doi.org/10.1109/PROMISE.2007.10.

APPENDIX A

IRDP SCENARIO: AUC, PD AND PF VALUES OF BASELINE AND DIFFERENT DEFECT LEVELS (0.3, 0.4, AND 0.5 DEFECT RATIO) OF MBA FOR EACH DATASET

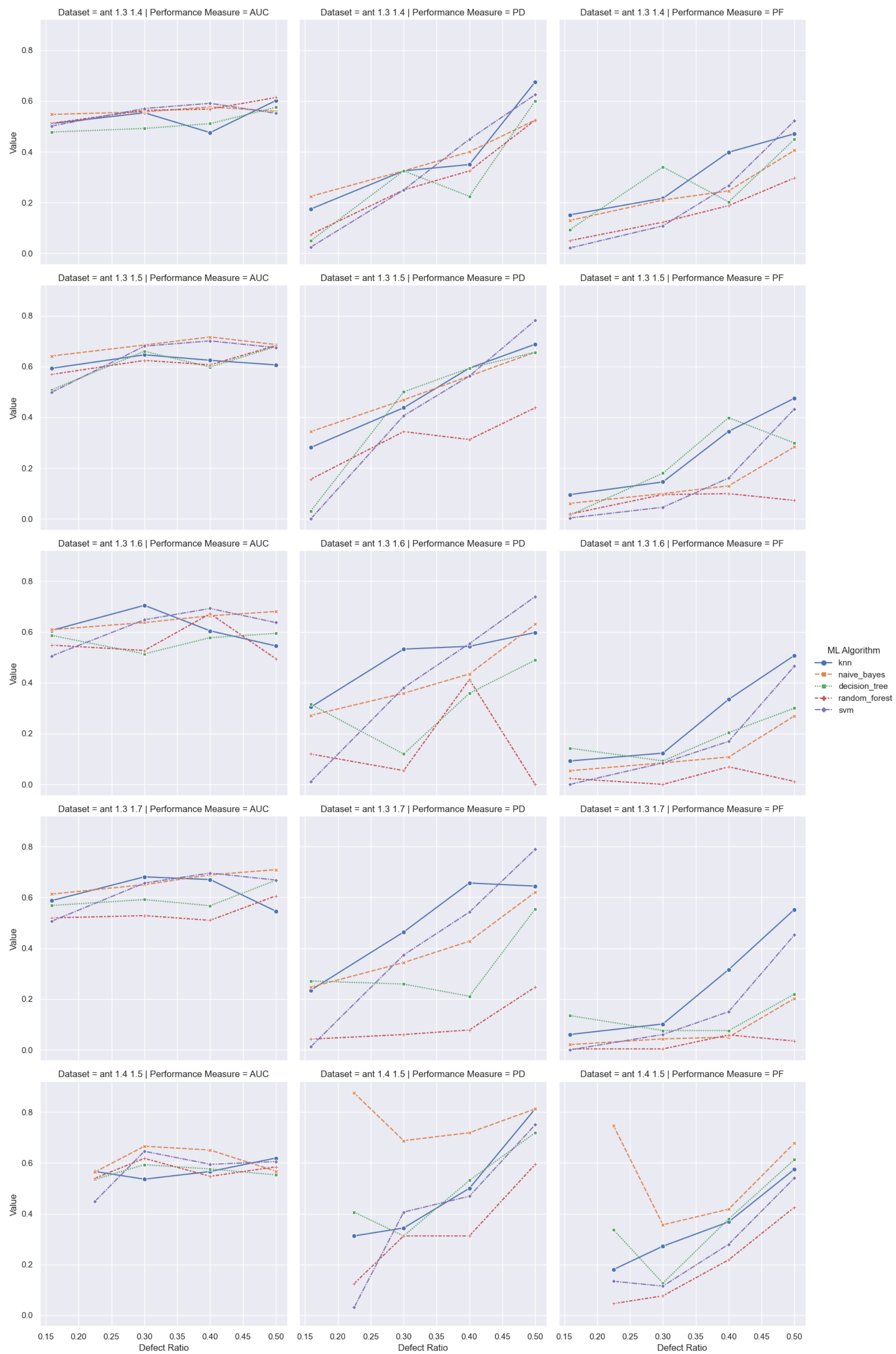

Figure A.1. AUC, pd and pf values of Baseline and different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (1/4)

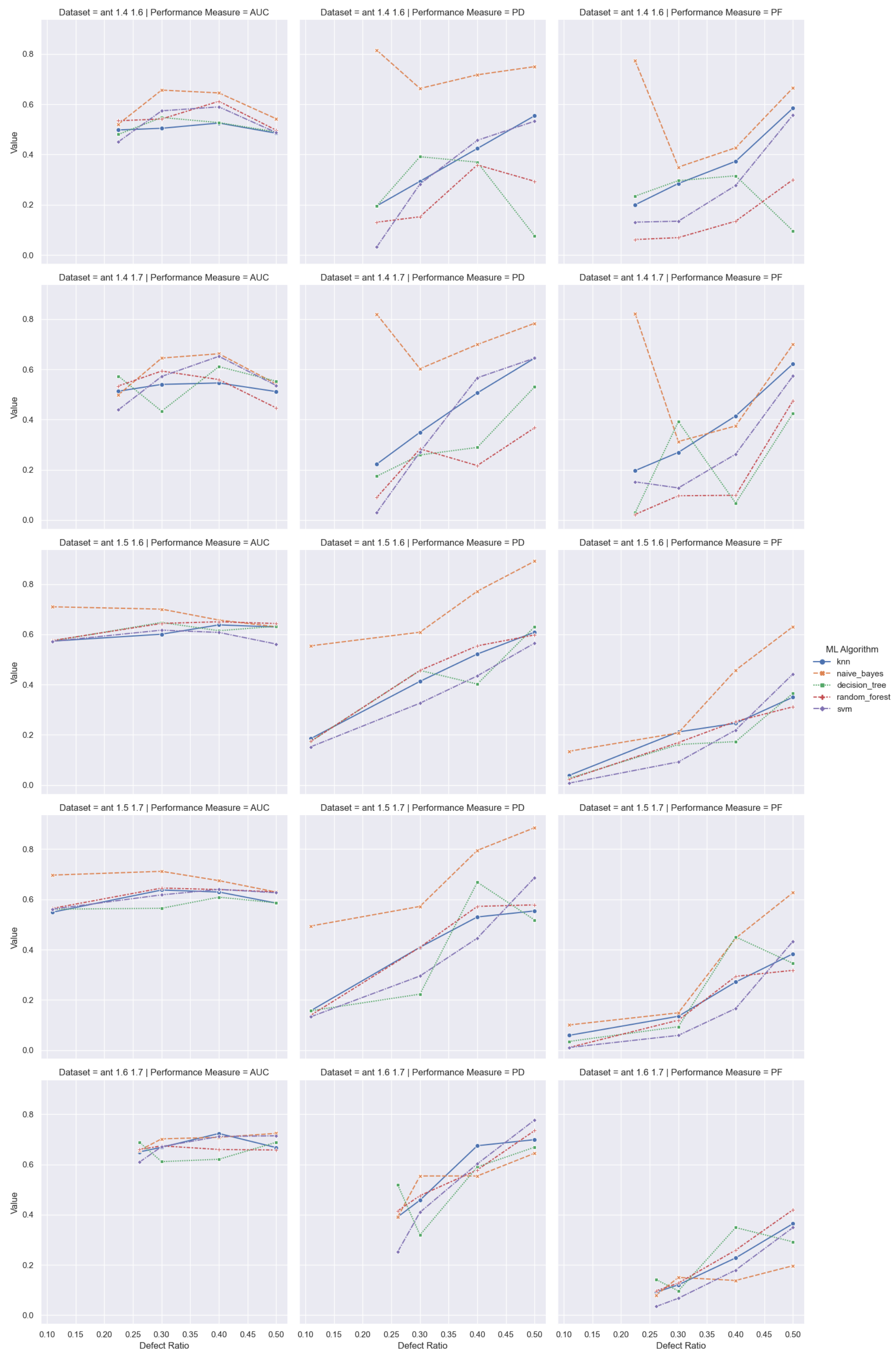

Figure A.2. AUC, pd and pf values of Baseline and different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (2/4)

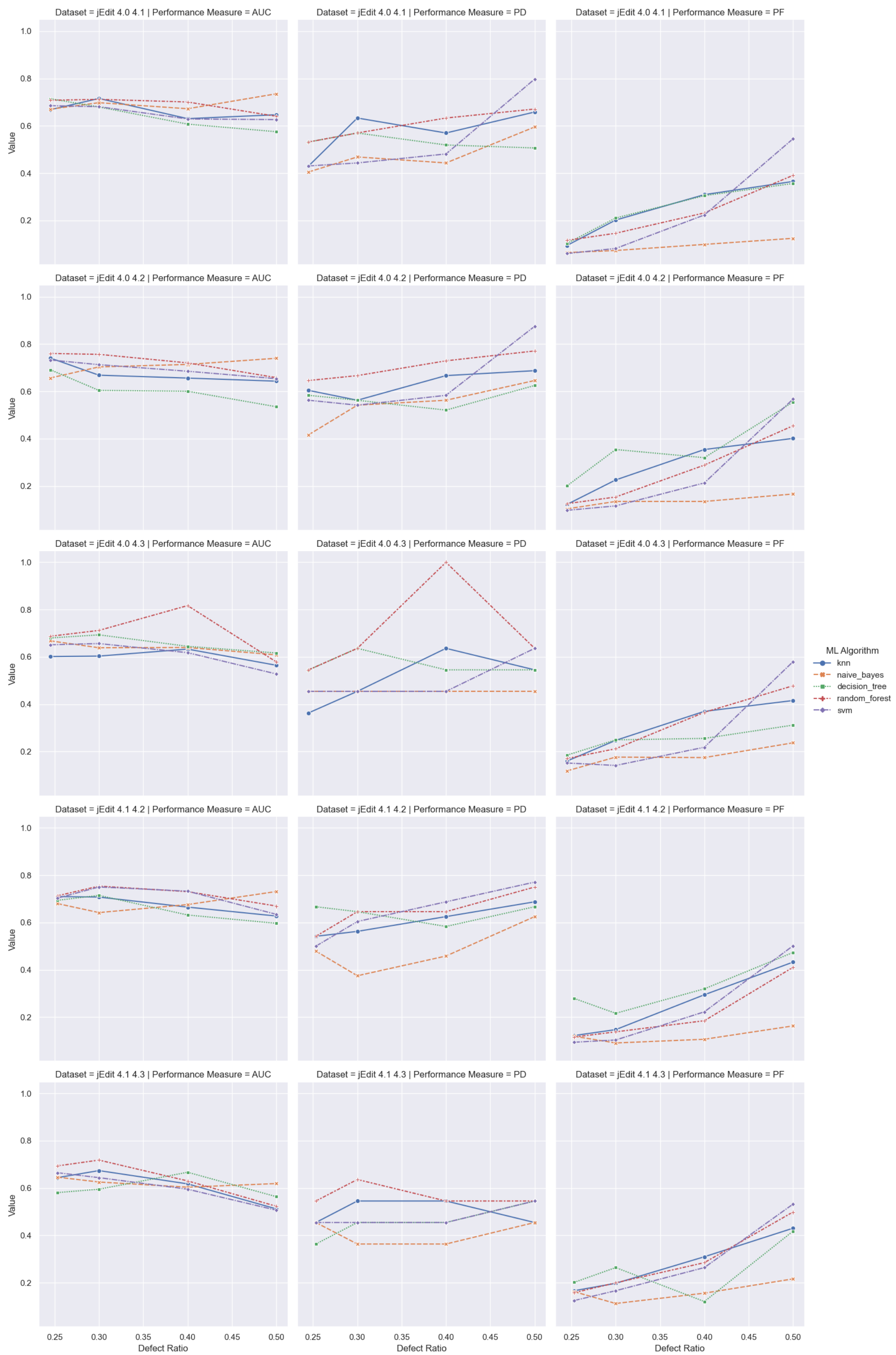

Figure A.3. AUC, pd and pf values of Baseline and different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (3/4)

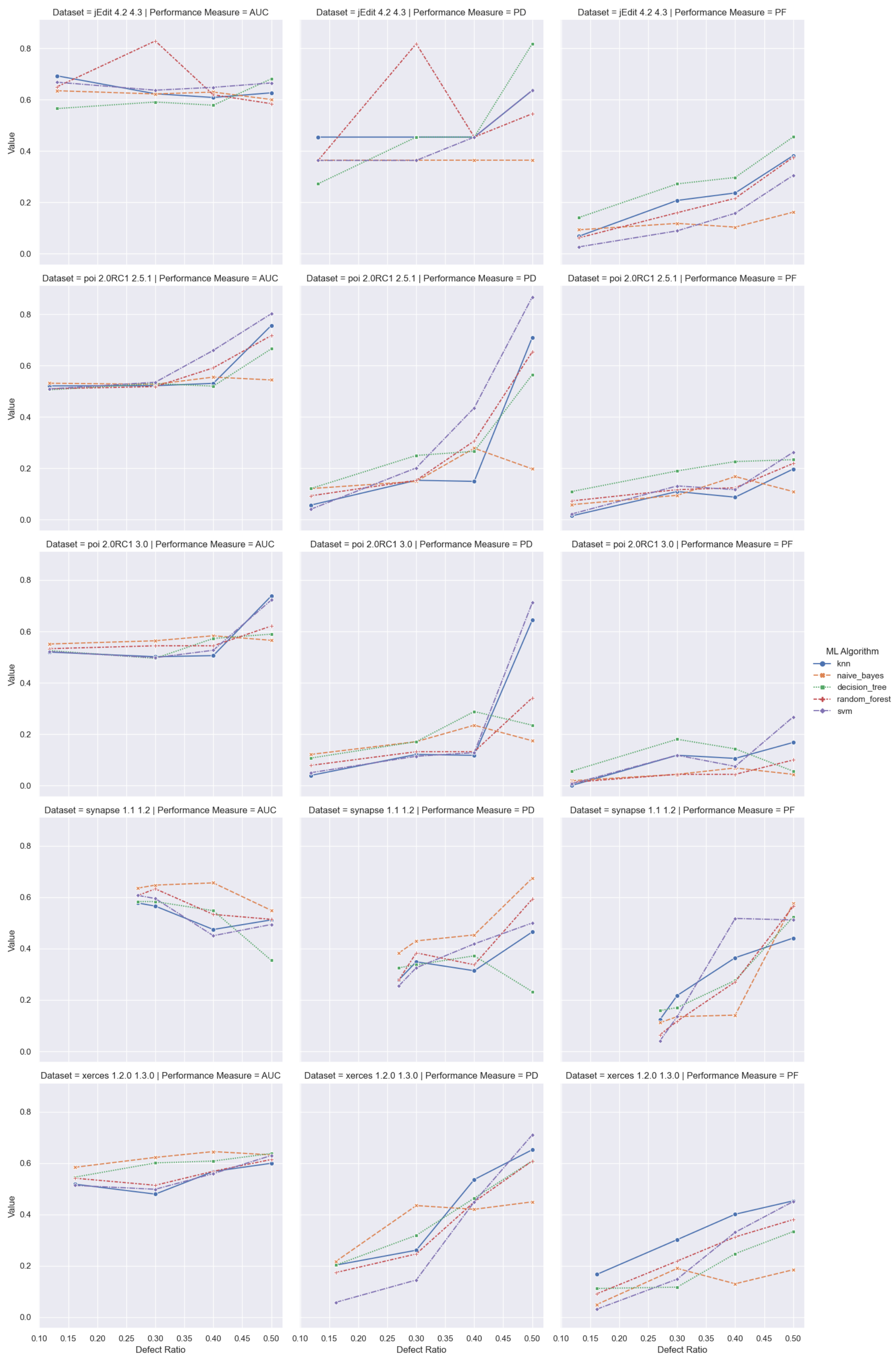

Figure A.4. AUC, pd and pf values of Baseline and different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (4/4)

APPENDIX B

IRDP SCENARIO: NUMBER OF CHANGED MEASURES ON DIFFERENT DEFECT LEVELS (0.3, 0.4, AND 0.5 DEFECT RATIO) OF MBA FOR EACH DATASET

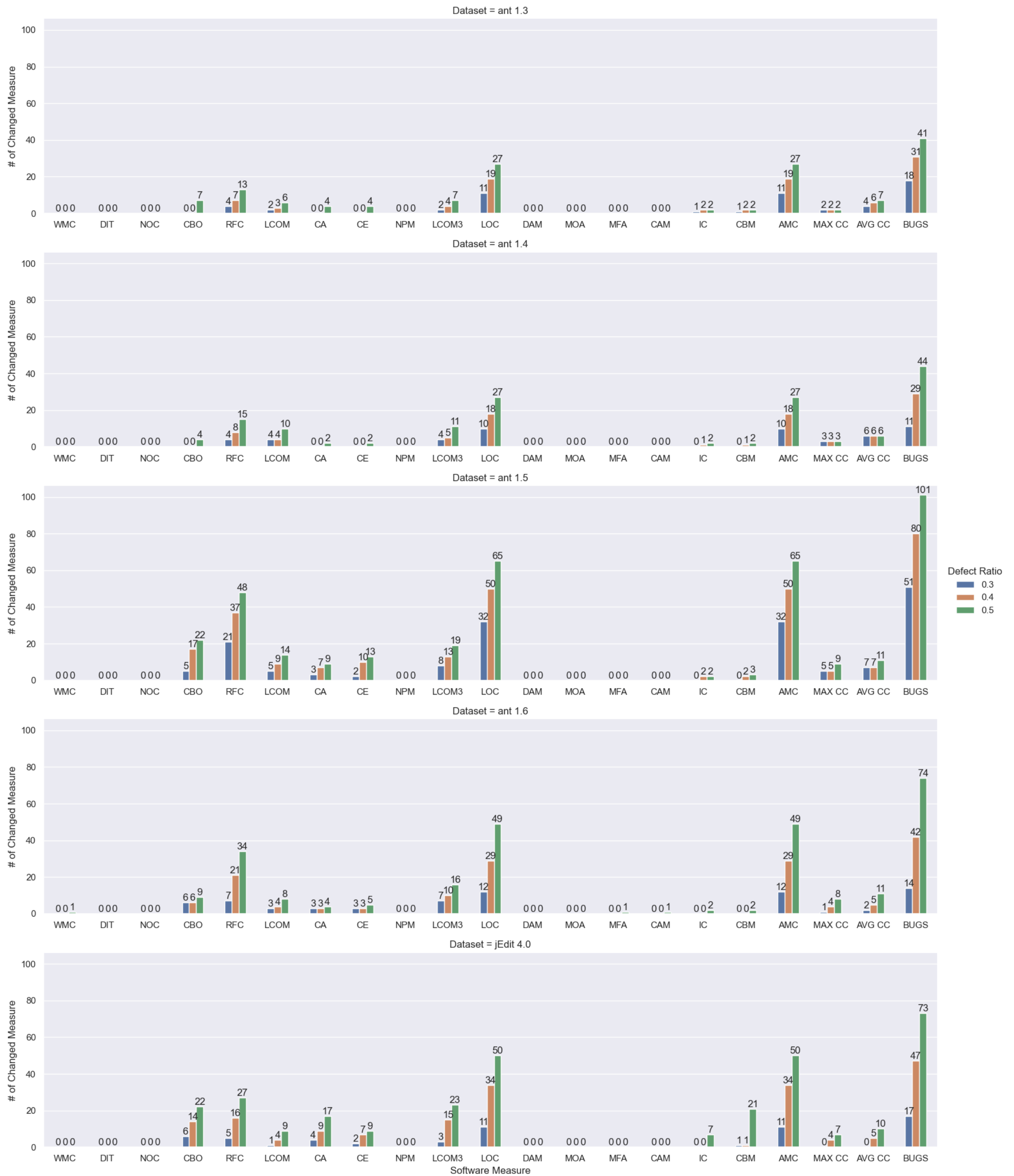

Figure B.1. Number of changed measures on different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (1/2)

Figure B.2. Number of changed measures on different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (2/2)

APPENDIX C

CPDP SCENARIO: NUMBER OF CHANGED MEASURES ON DIFFERENT DEFECT LEVELS (0.3, 0.4, AND 0.5 DEFECT RATIO) OF MBA FOR EACH DATASET

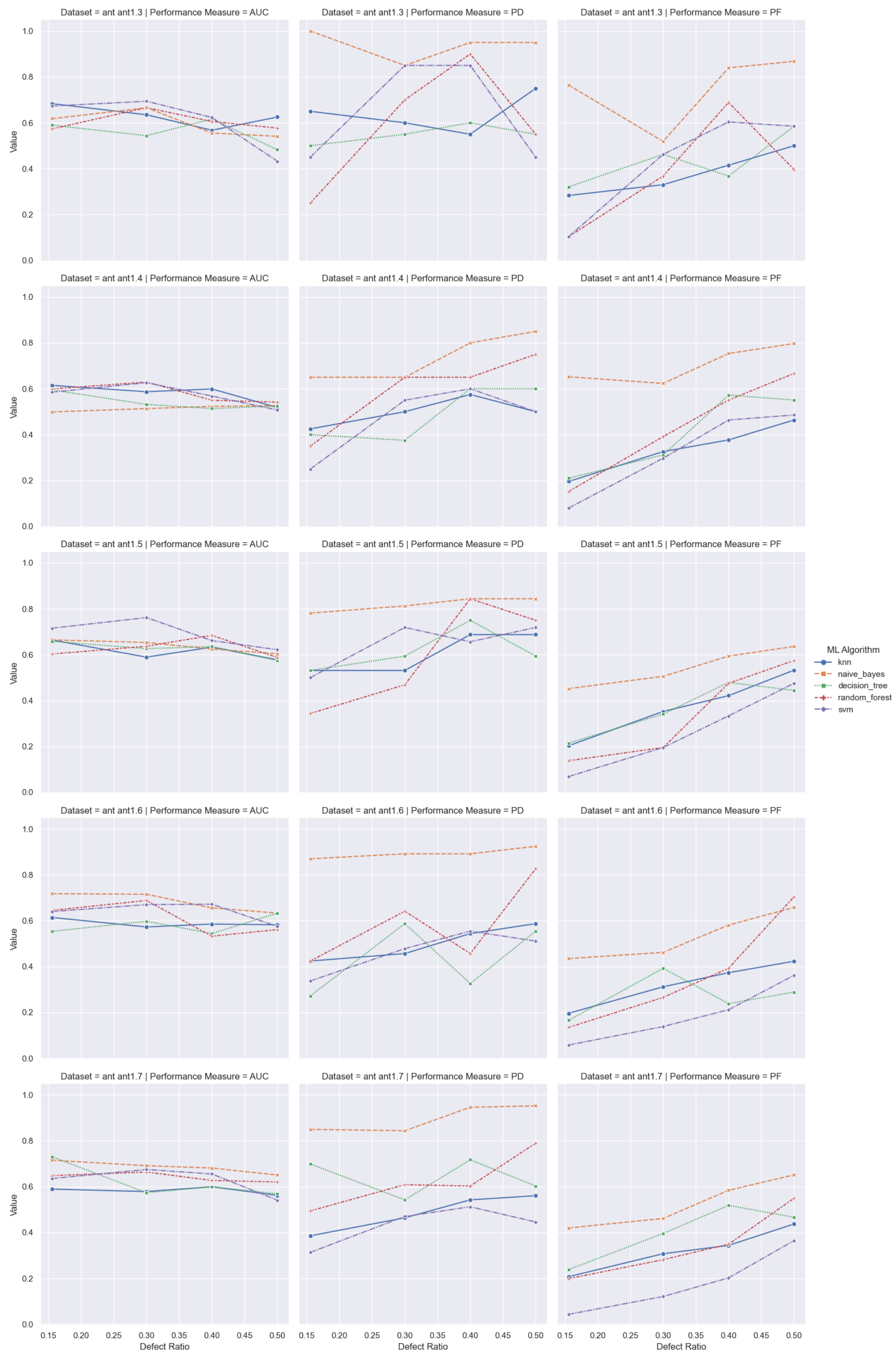

Figure C.1. AUC, pd and pf values of Baseline and different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (1/16)

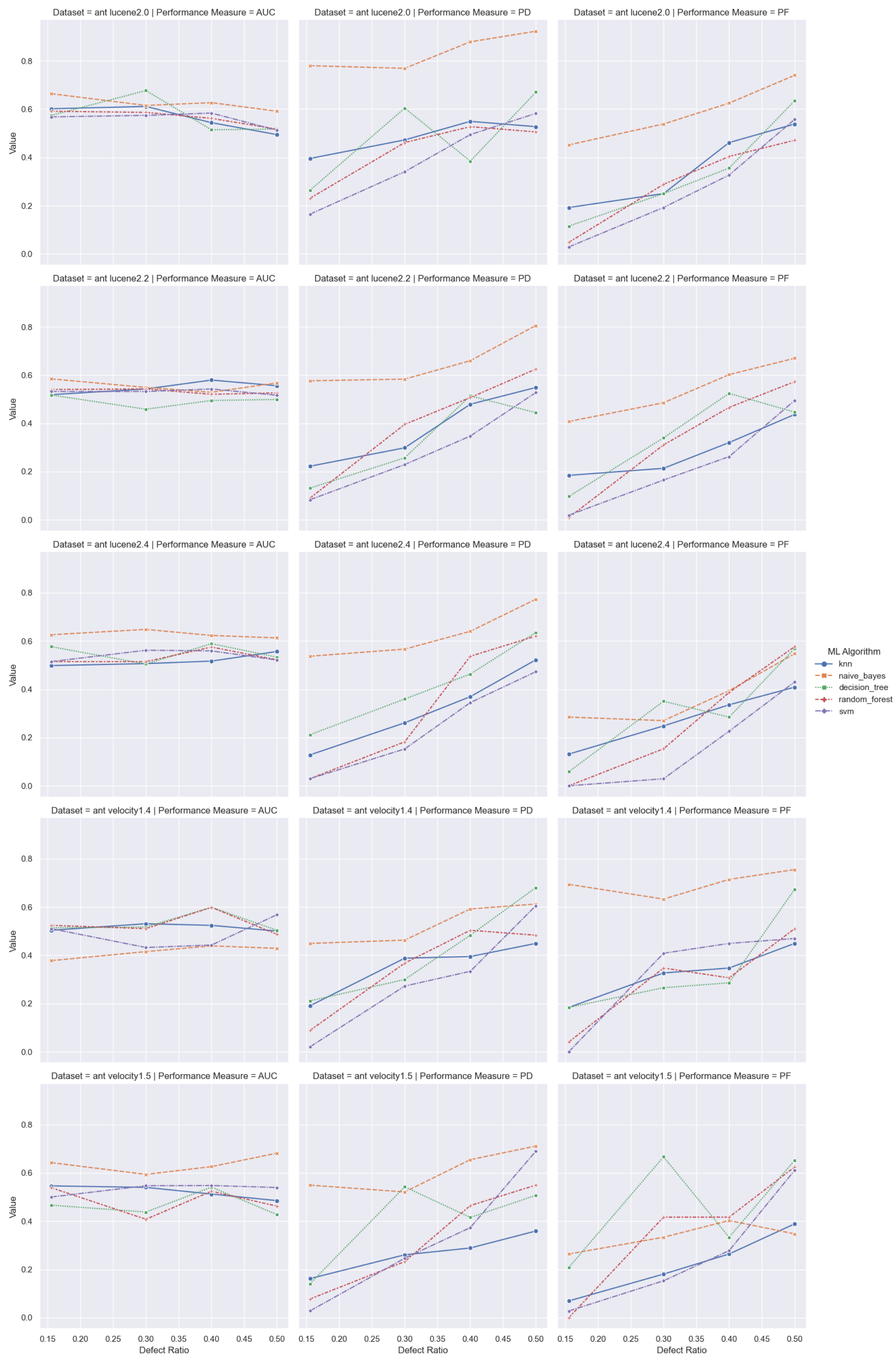

Figure C.2. AUC, pd and pf values of Baseline and different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (2/16)

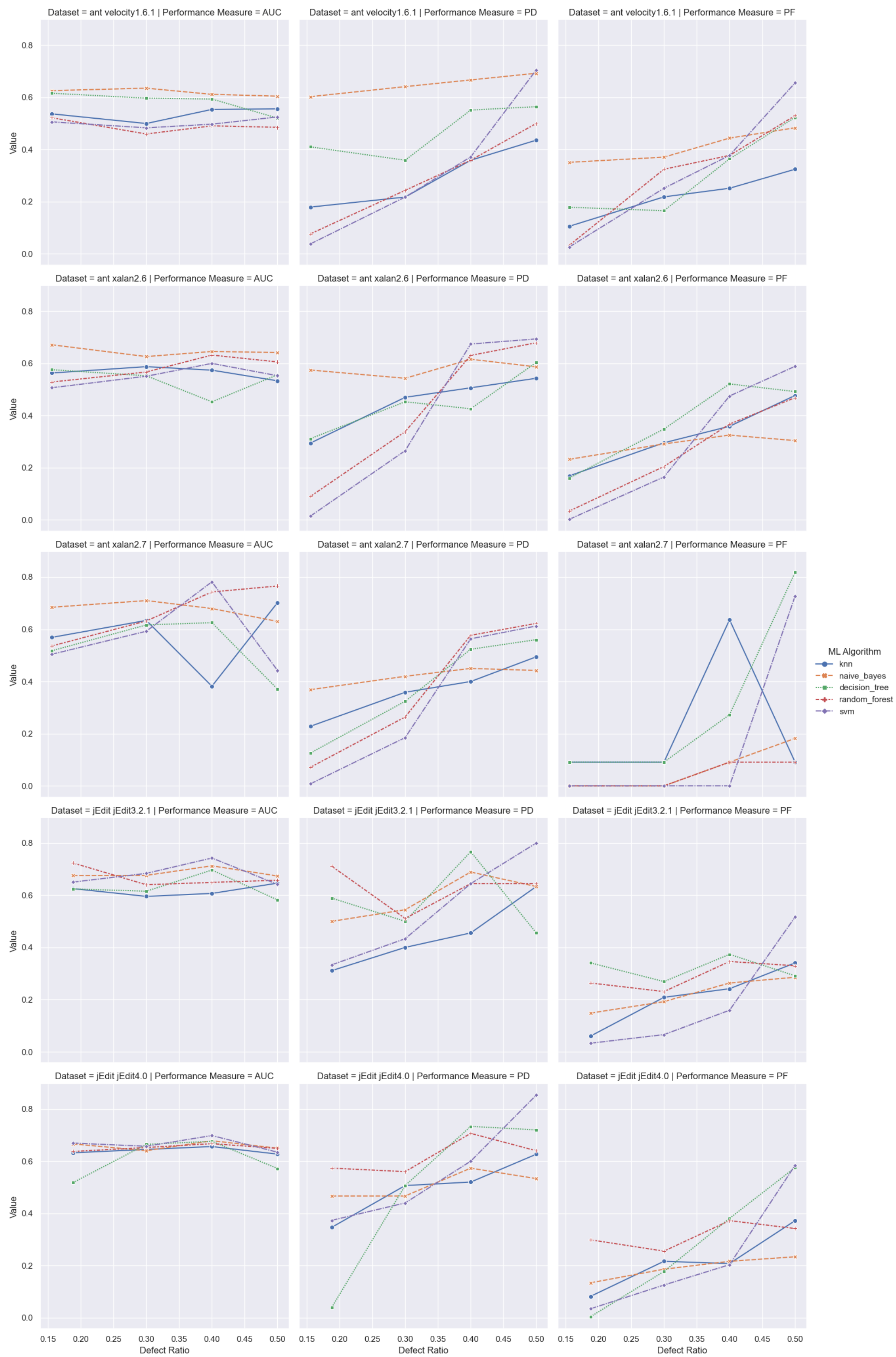

Figure C.3. AUC, pd and pf values of Baseline and different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (3/16)

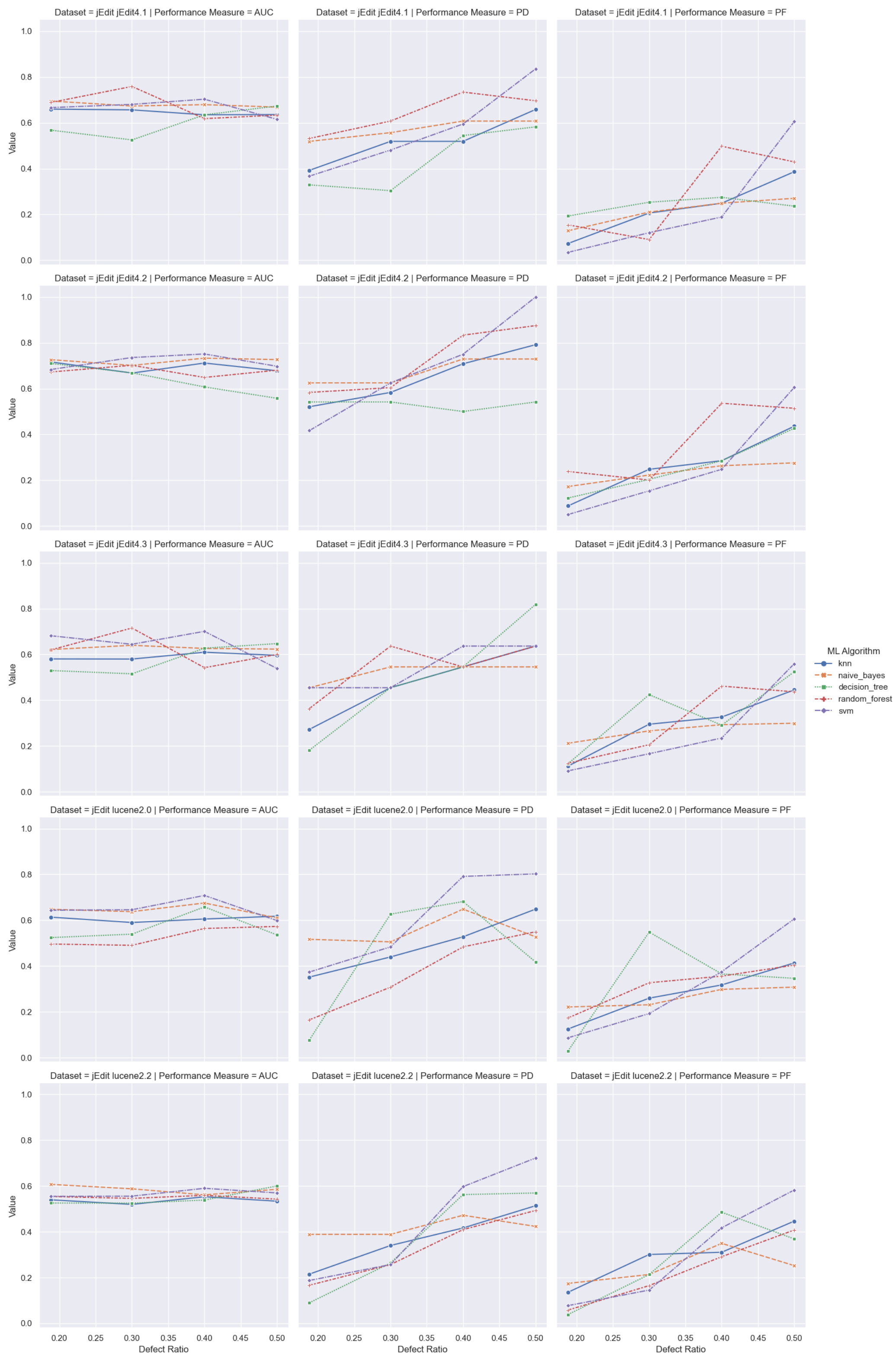

Figure C.4. AUC, pd and pf values of Baseline and different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (4/16)

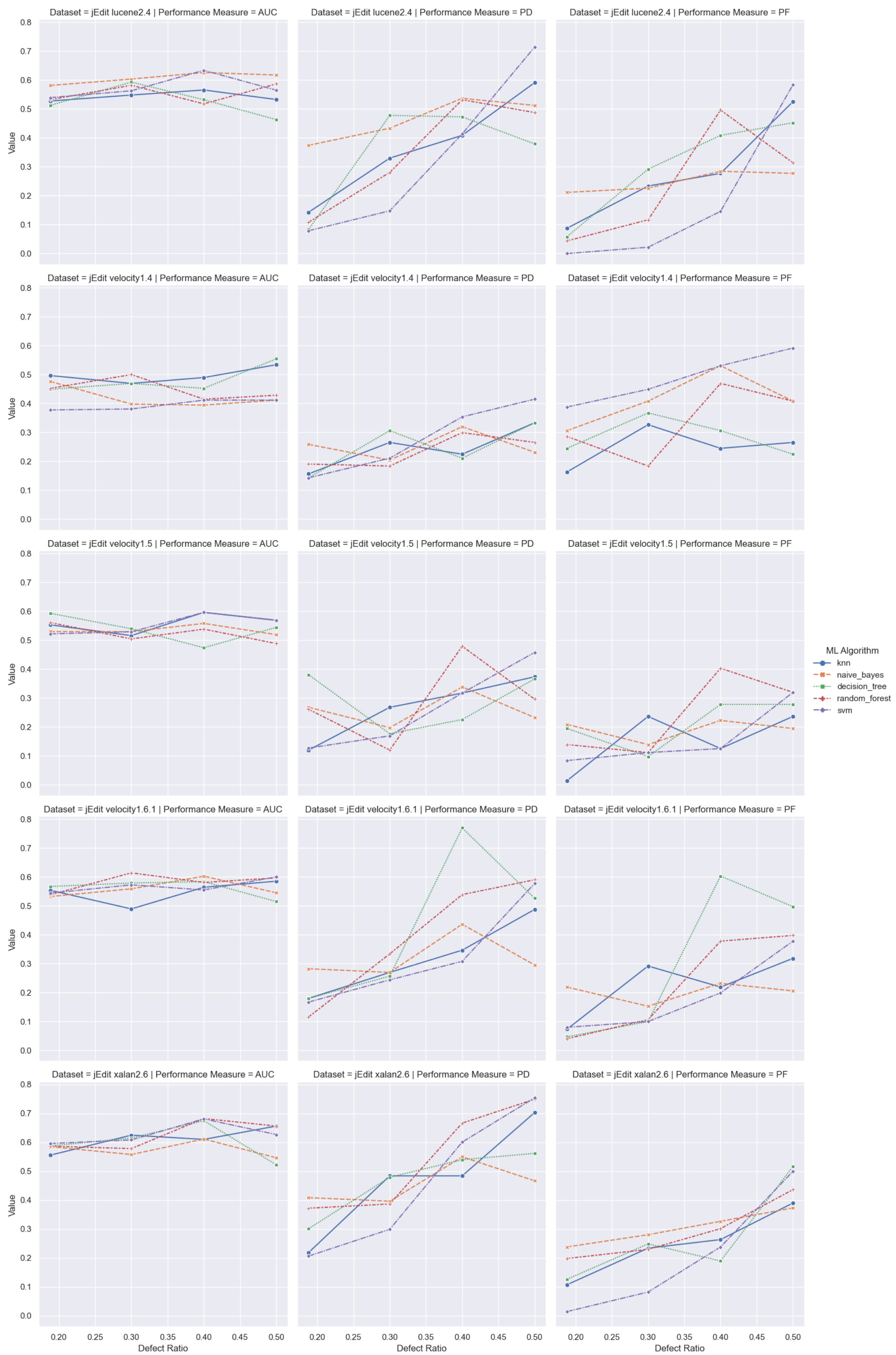

Figure C.5. AUC, pd and pf values of Baseline and different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (5/16)

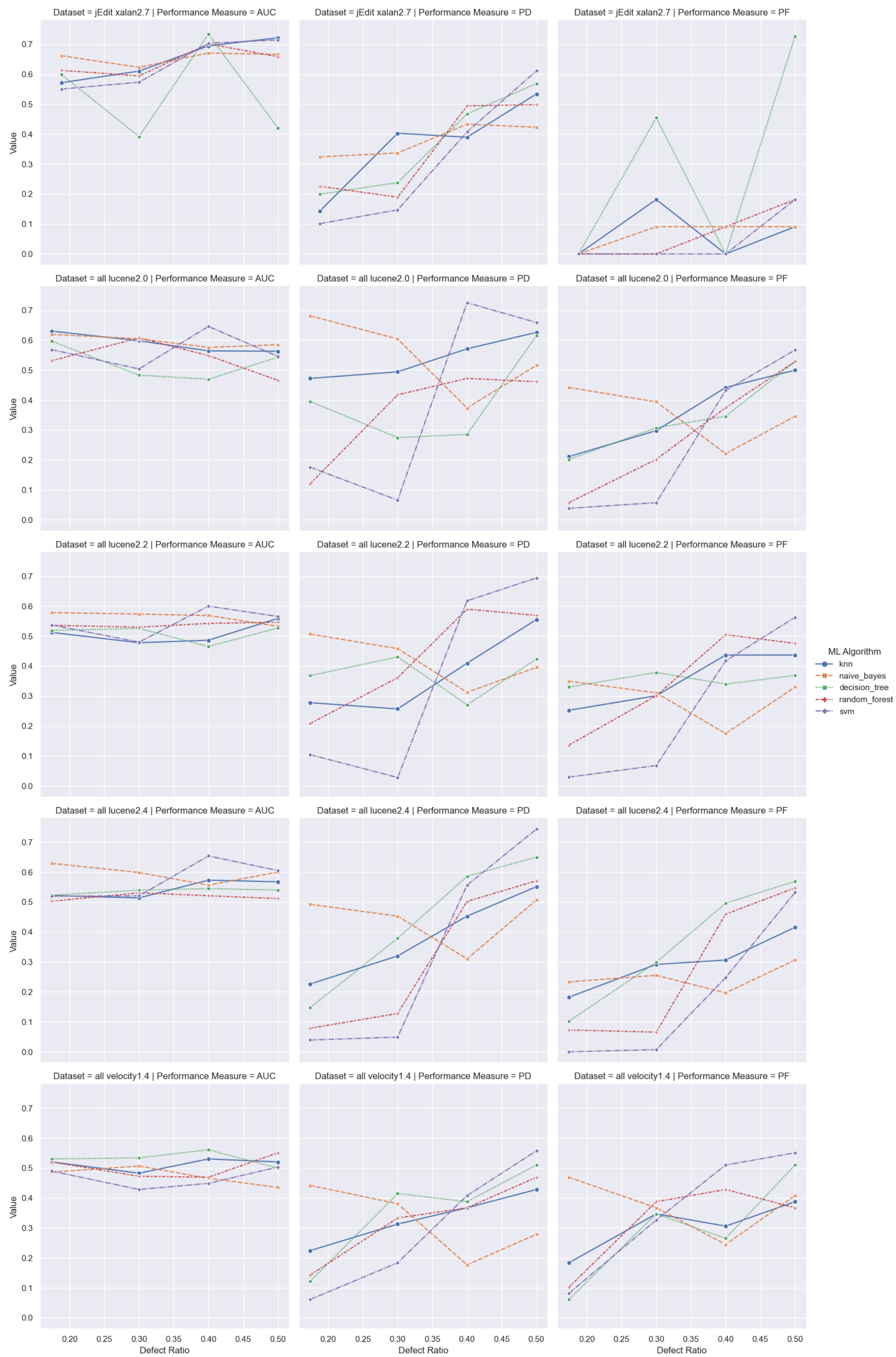

Figure C.6. AUC, pd and pf values of Baseline and different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (6/16)

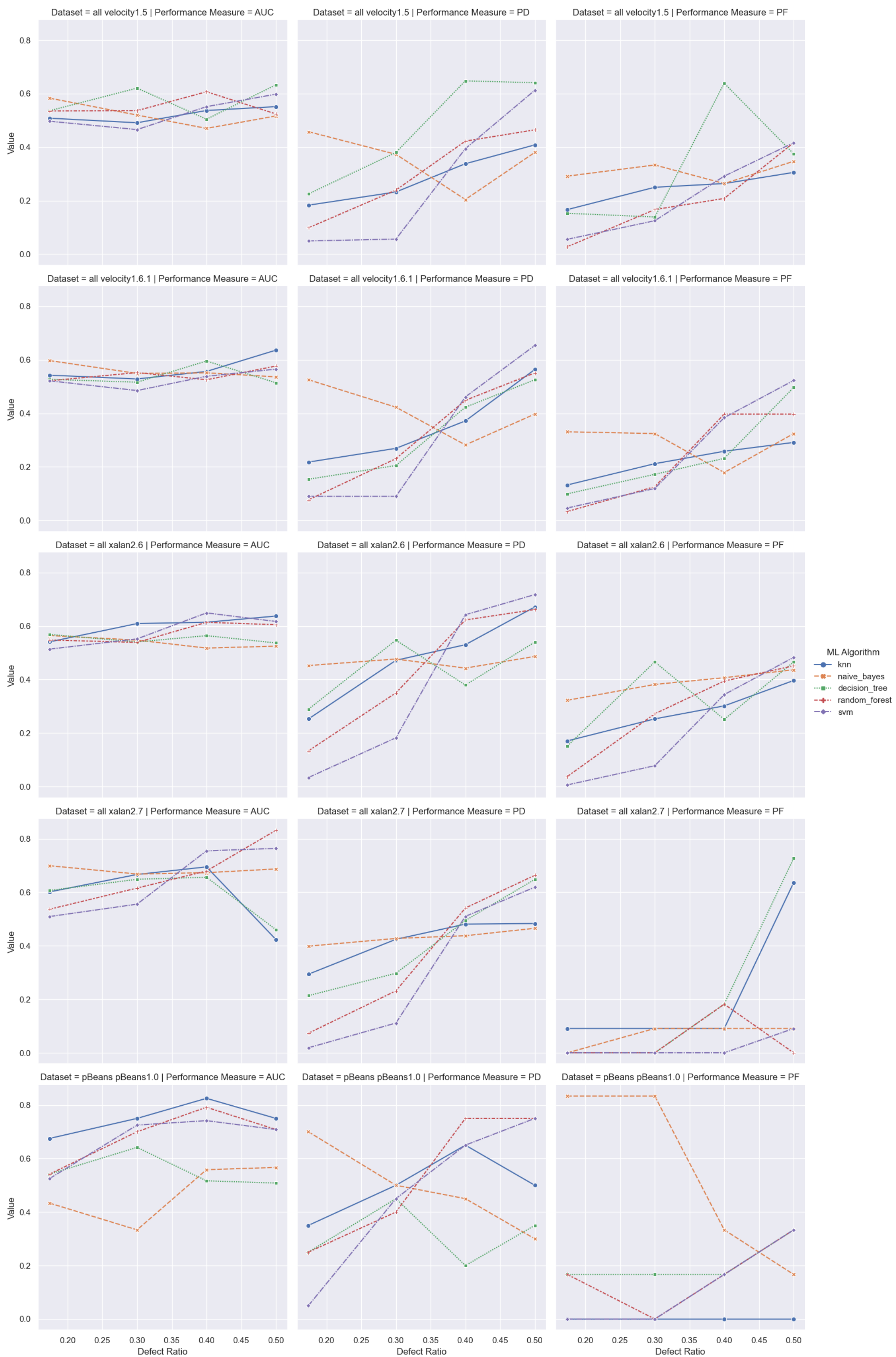

Figure C.7. AUC, pd and pf values of Baseline and different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (7/16)

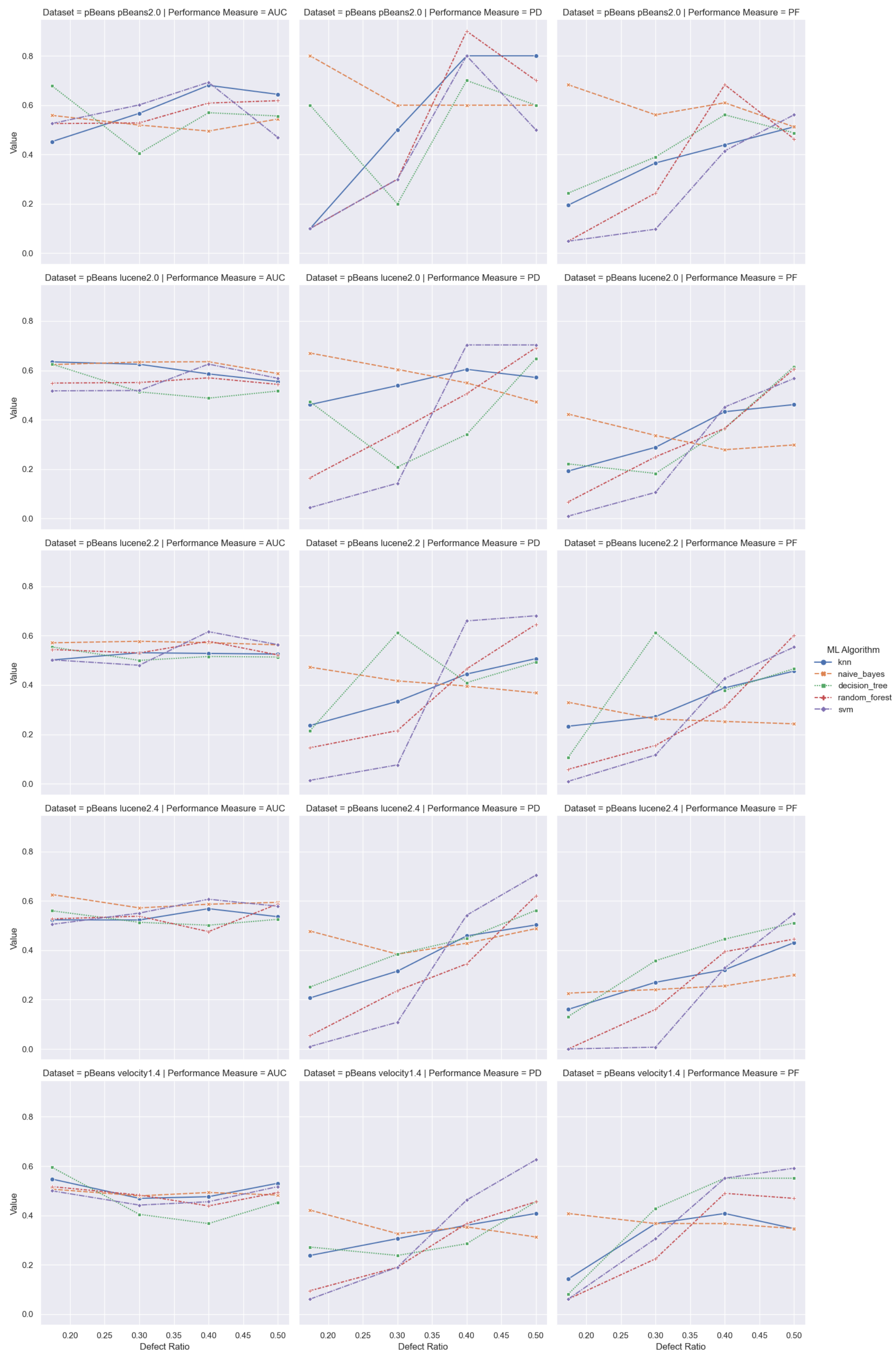

Figure C.8. AUC, pd and pf values of Baseline and different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (8/16)

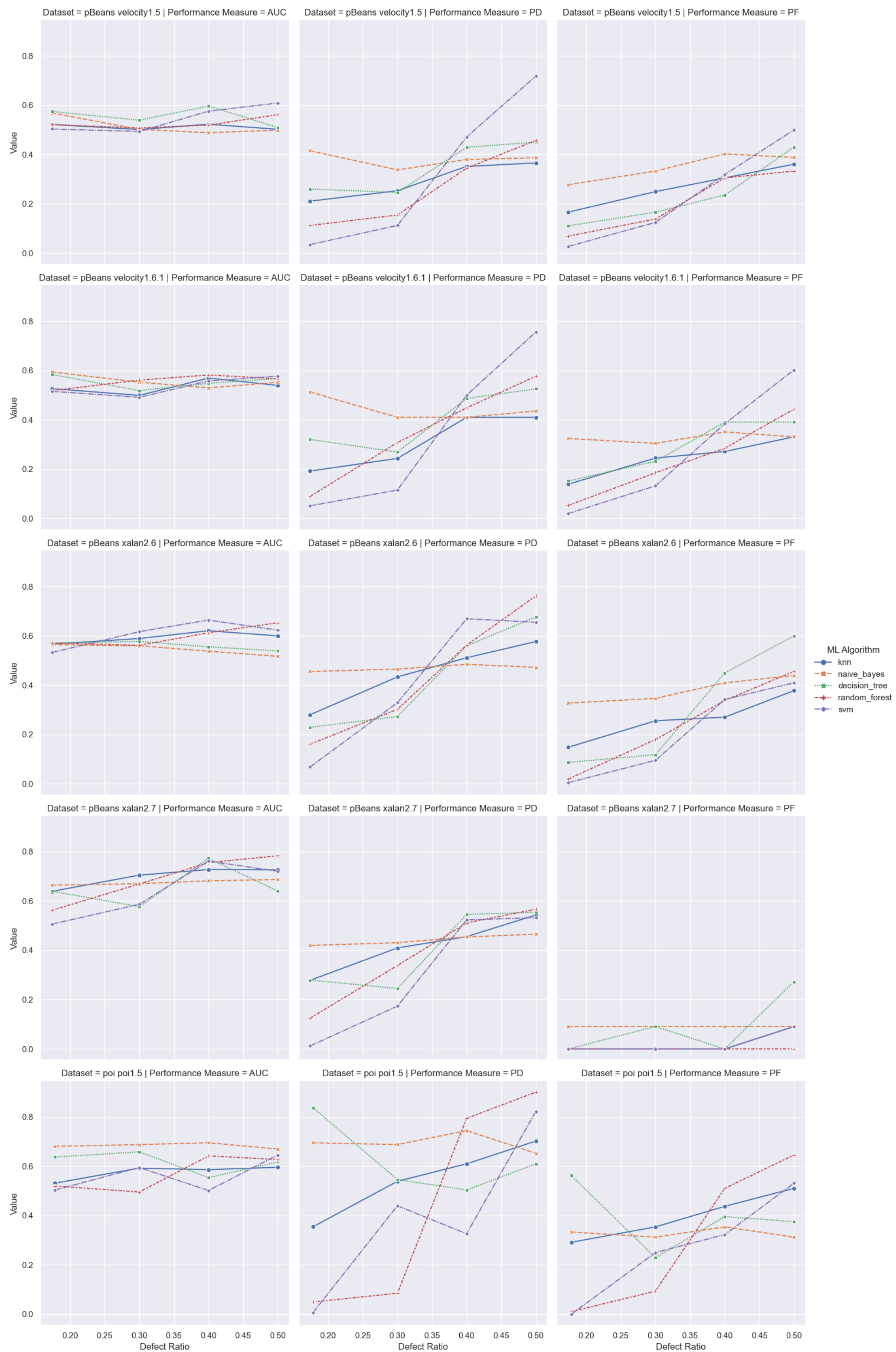

Figure C.9. AUC, pd and pf values of Baseline and different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (9/16)

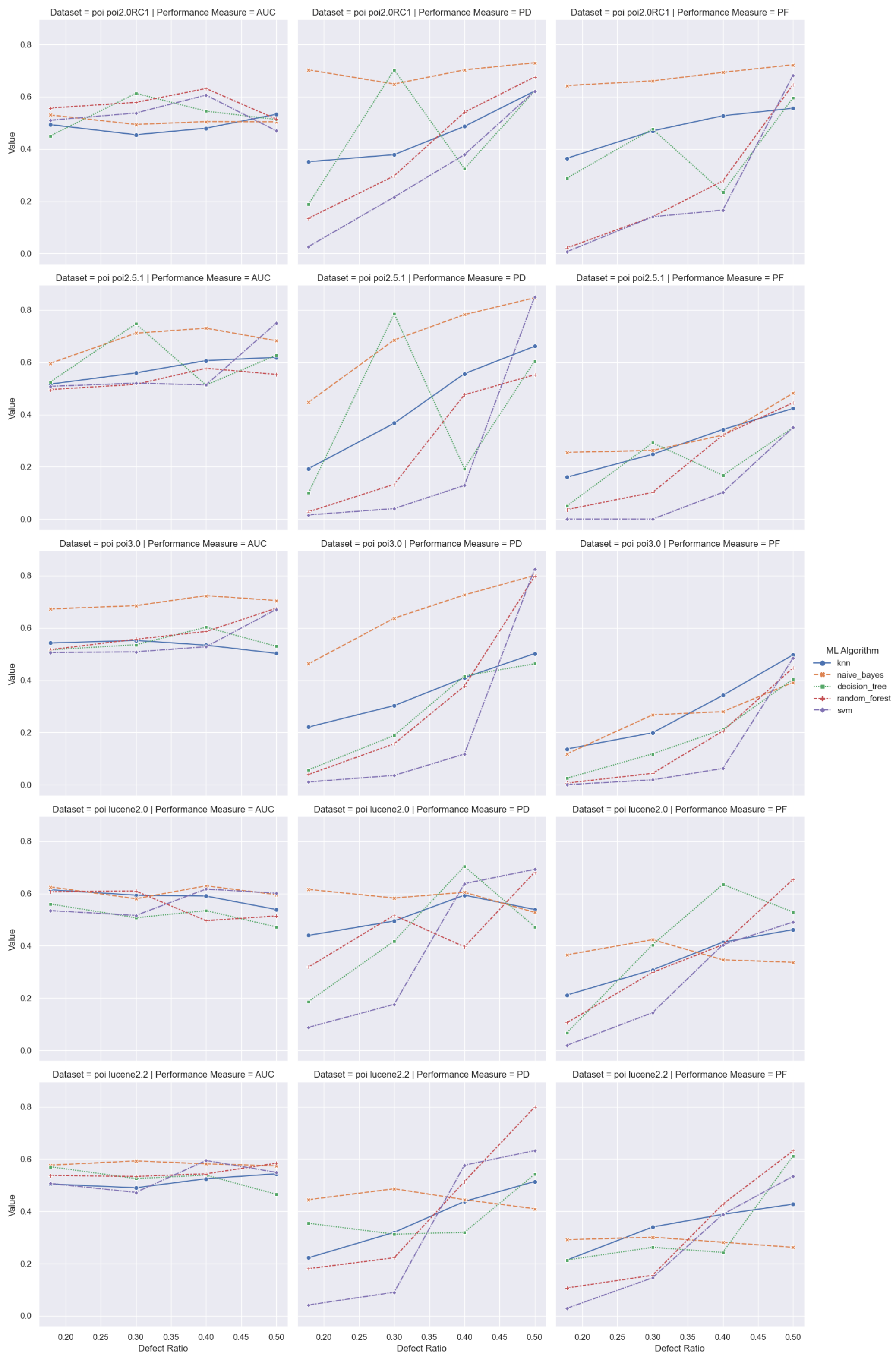

Figure C.10. AUC, pd and pf values of Baseline and different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (10/16)

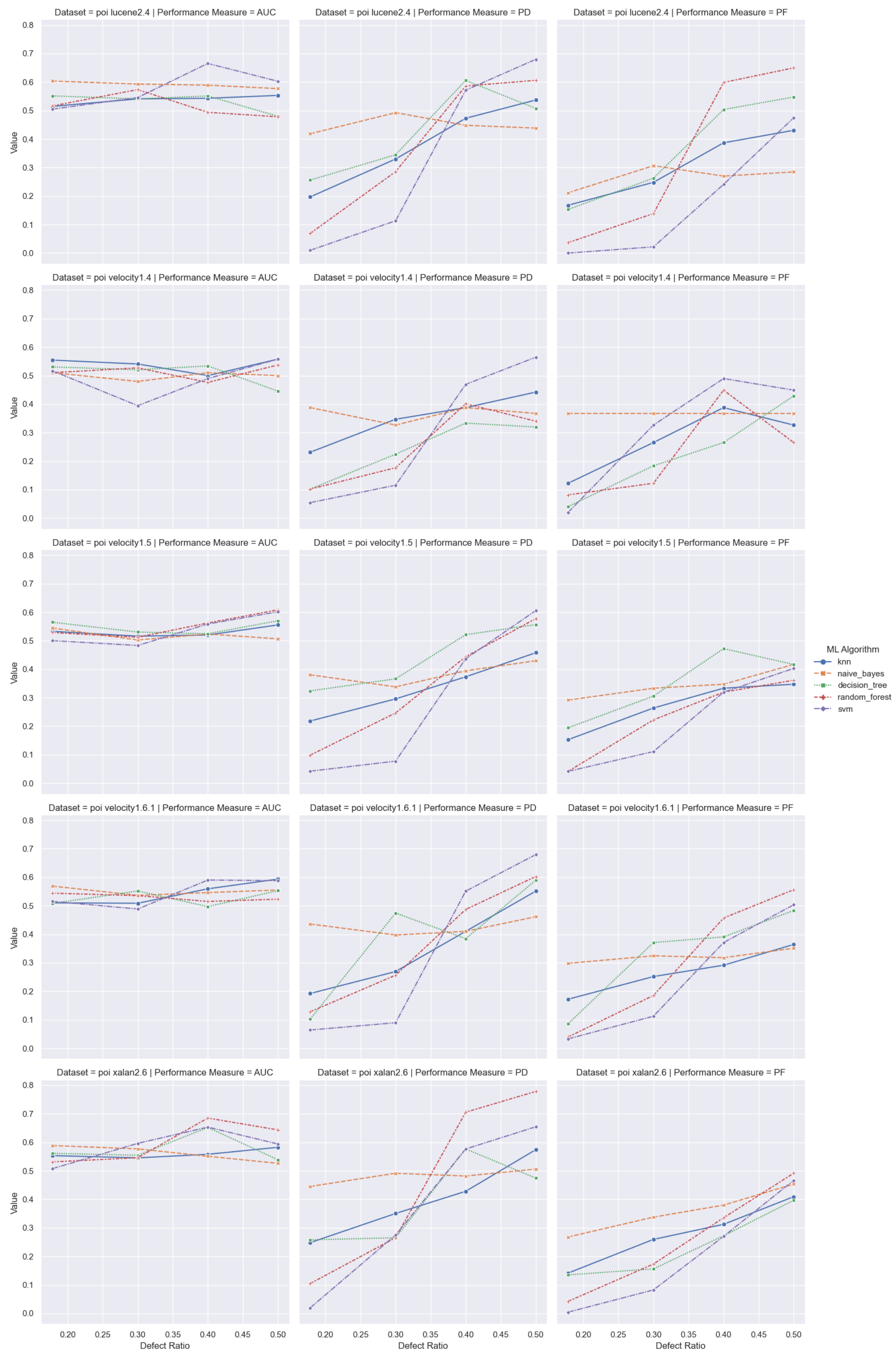

Figure C.11. AUC, pd and pf values of Baseline and different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (11/16)

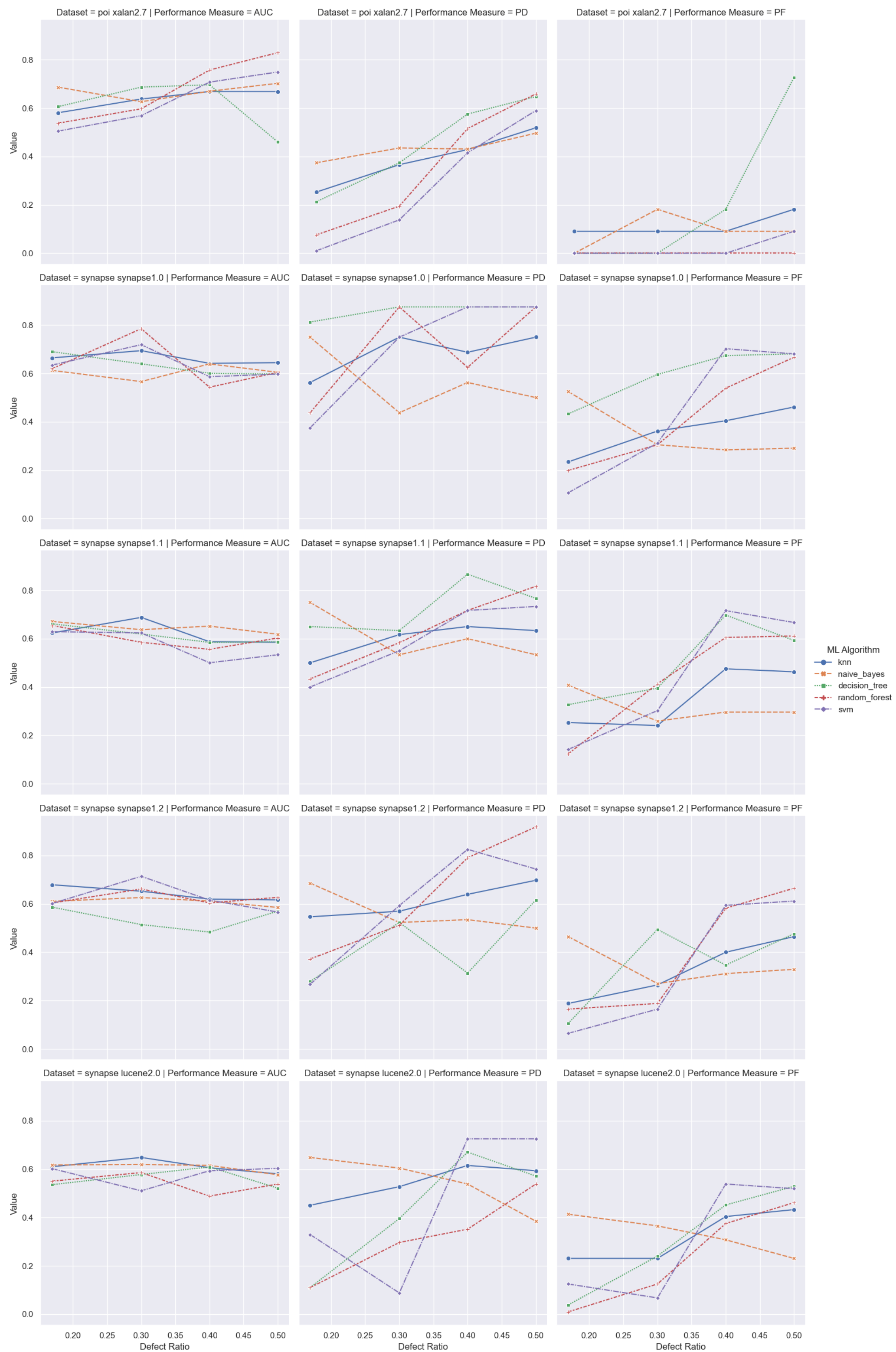

Figure C.12. AUC, pd and pf values of Baseline and different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (12/16)

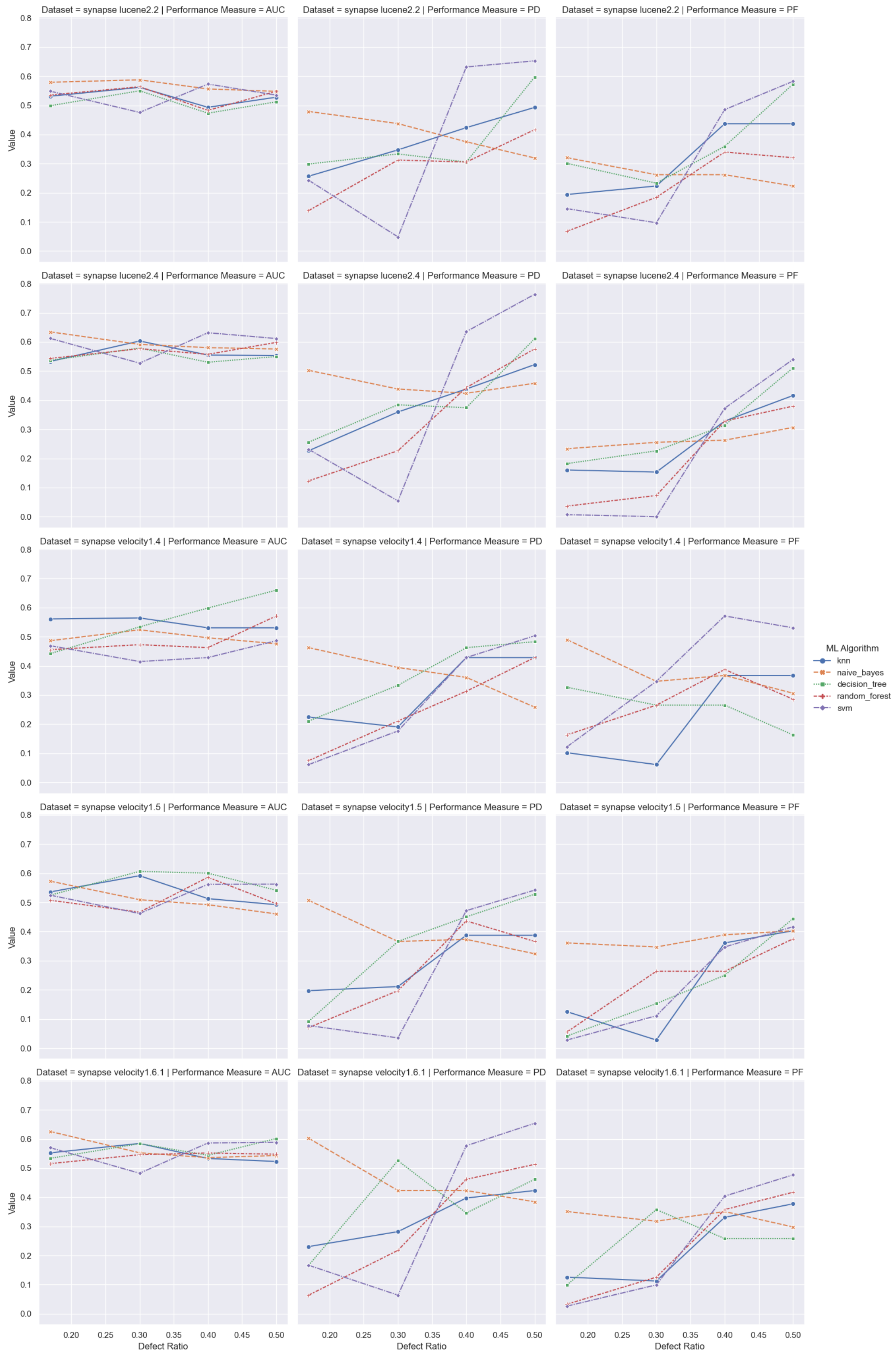

Figure C.13. AUC, pd and pf values of Baseline and different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (13/16)

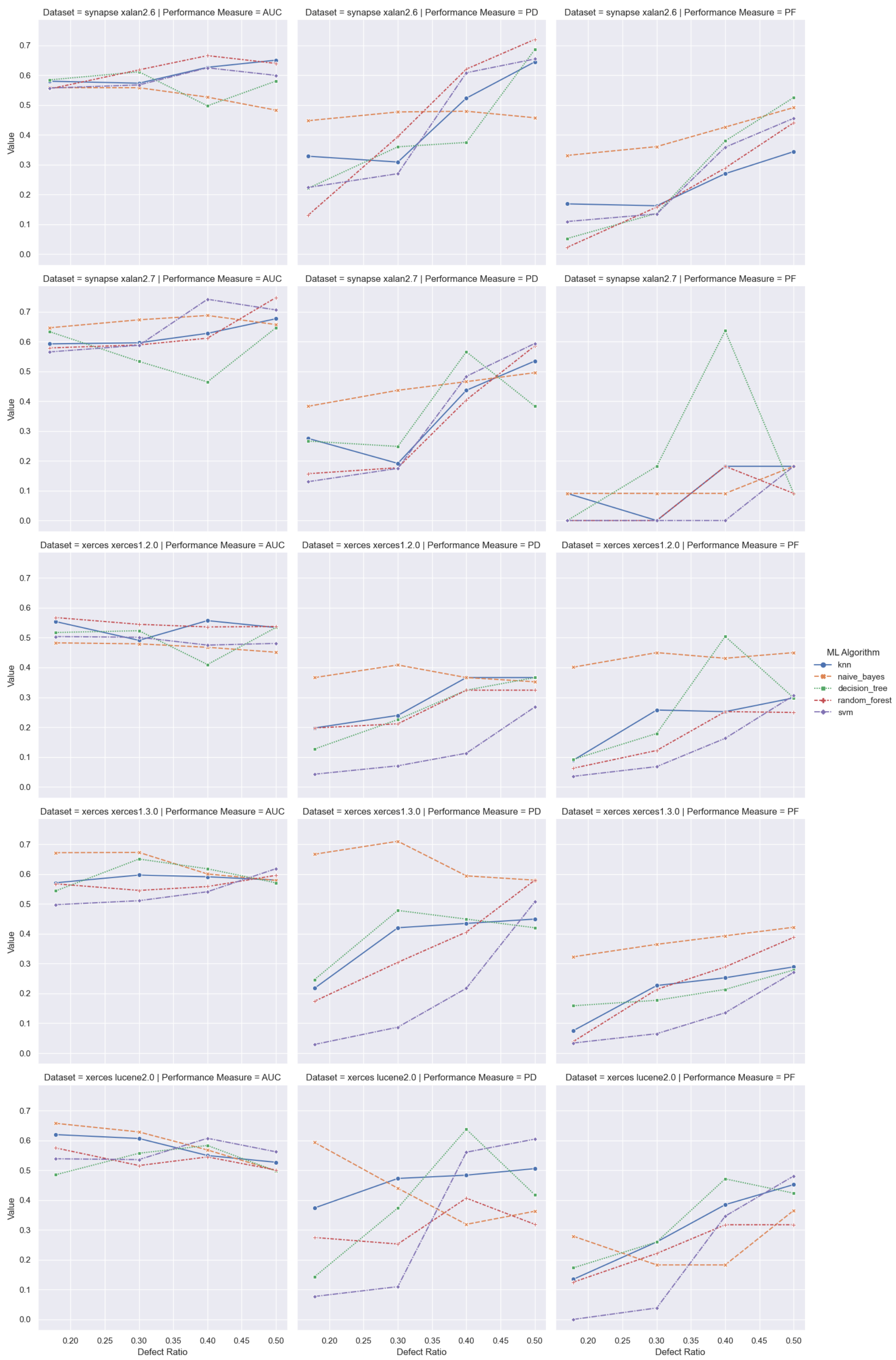

Figure C.14. AUC, pd and pf values of Baseline and different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (14/16)

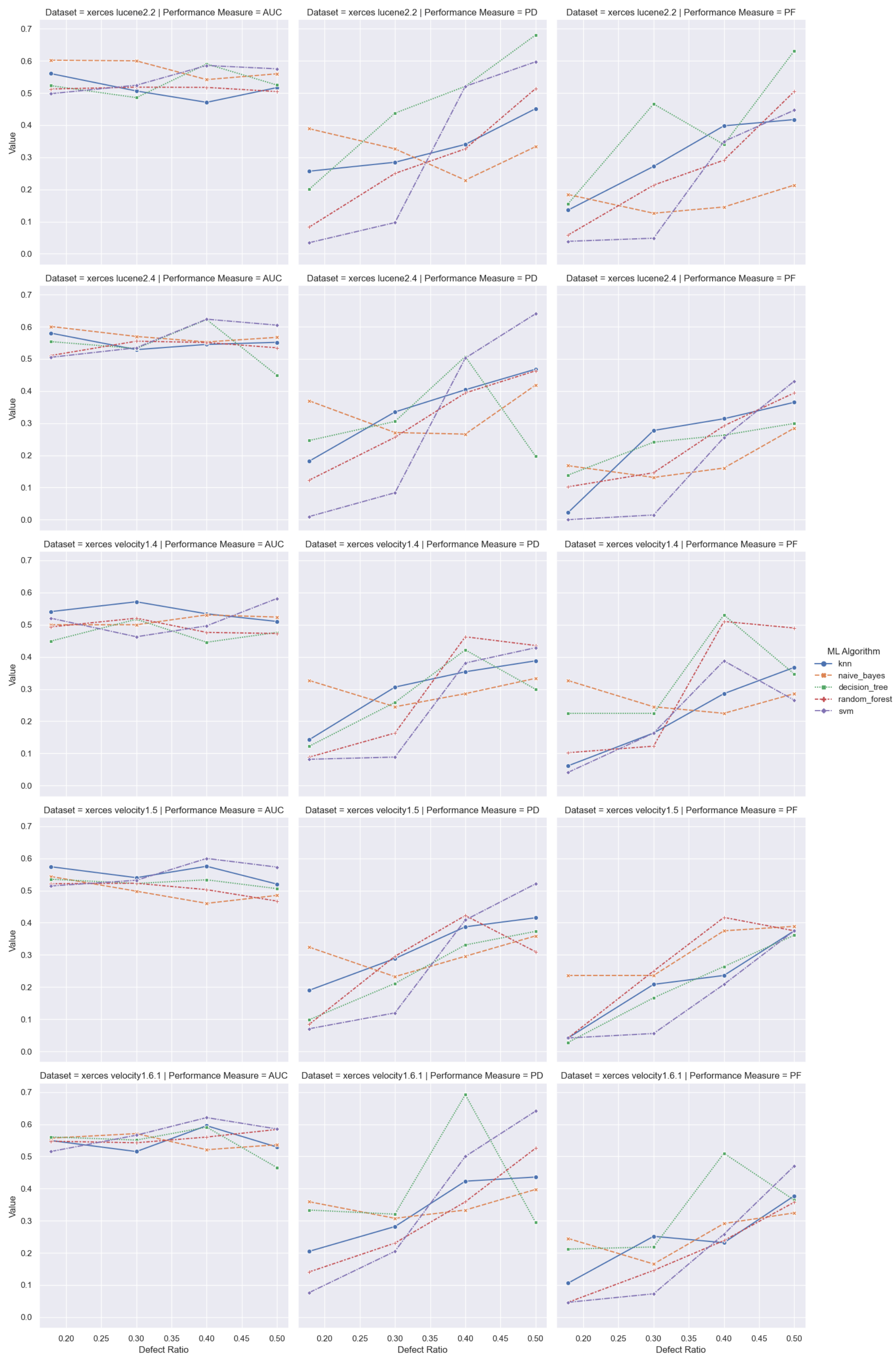

Figure C.15. AUC, pd and pf values of Baseline and different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (15/16)

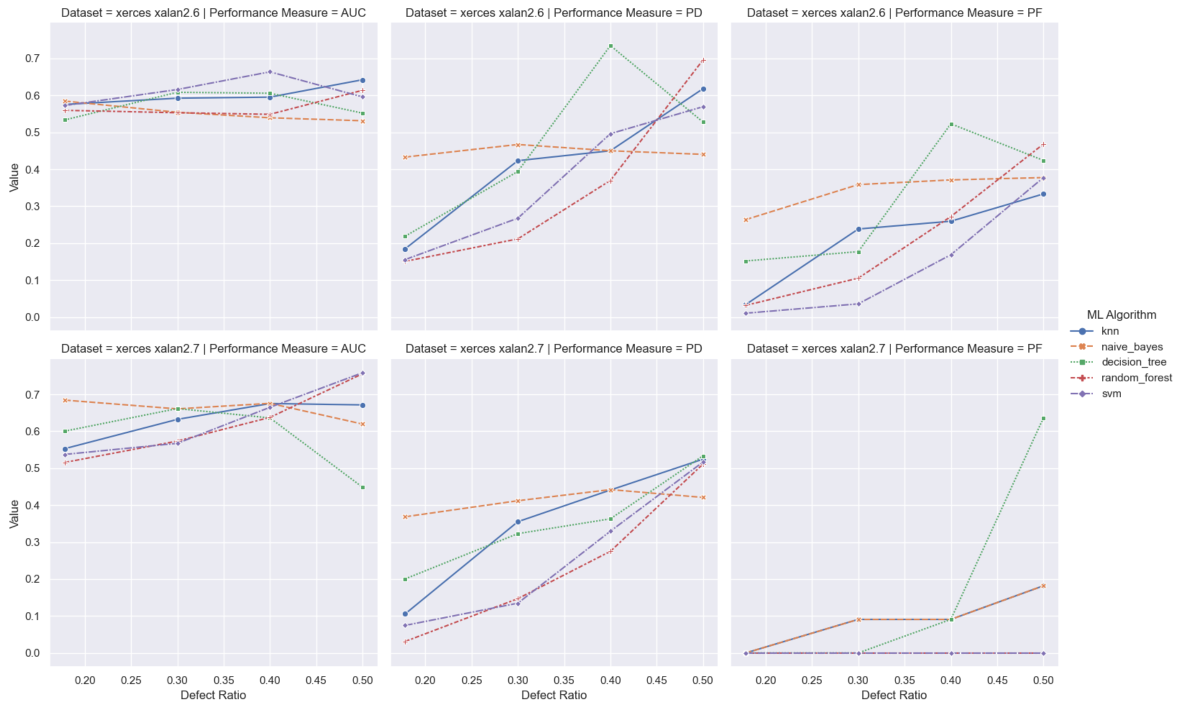

Figure C.16. AUC, pd and pf values of Baseline and different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (16/16)

APPENDIX D

CPDP SCENARIO: NUMBER OF CHANGED MEASURES ON DIFFERENT DEFECT LEVELS (0.3, 0.4, AND 0.5 DEFECT RATIO) OF MBA FOR EACH DATASET

Figure D.1. Number of changed measures on different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (1/2)

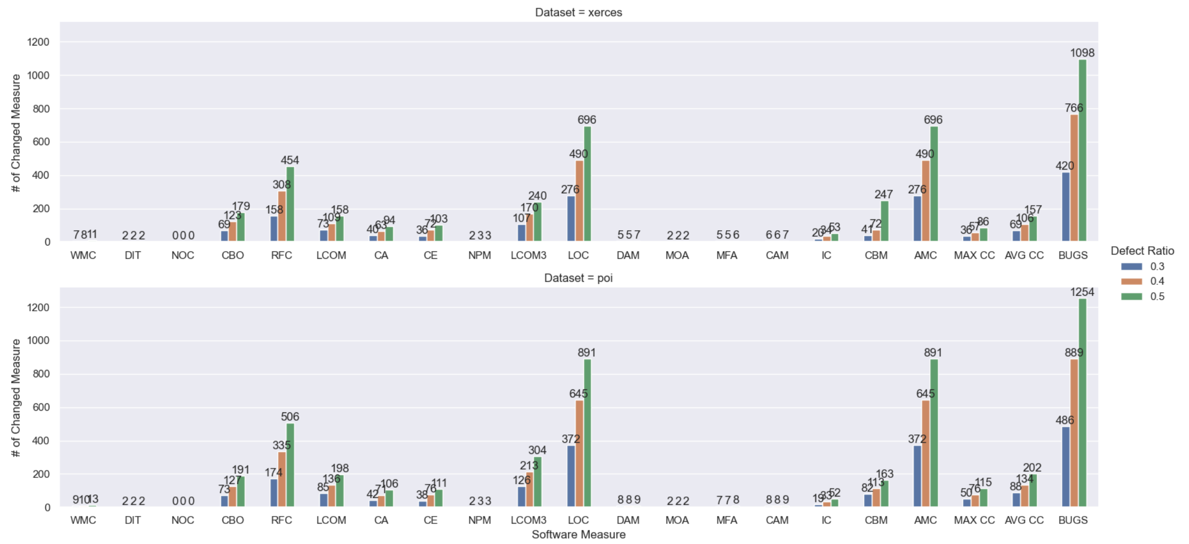

Figure D.2. Number of changed measures on different defect levels (0.3, 0.4, and 0.5 defect ratio) of MBA for each dataset (2/2)