SOURCE FILE LEVEL SOFTWARE DEFECT PREDICTION FRAMEWORK

by

Melih Çelik

B.S., in Computer Engineering, Marmara University, 2005

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering
Boğaziçi University
2008

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Assistant Professor Ayşe Başar Bener for her guidance and support during the development of this research. Her patience and understanding helped me solve the problems and continue with the work harder.

I would also like to thank Professor Emin Anarım and and Professor Oğuz Tosun for kindly accepting to be in my thesis jury.

I thank to each and every member of Software Engineering Research Laboratory for their support and being helpful to me all the times. Having the support of my colleagues in the SoftLab, especially of Burak Turhan, was invaluable for producing creative ideas and the right solutions.

I give my best regards and special thanks to my family. They were always my key supporters throughout my graduate studies and particularly this research. Without their endless support and unconditional belief in me, it would have been so hard to pass through all the hard times.

I would also like to thank to my dear friends Ozan Ünay, Samet Çokpınar, Cemalettin Koç, Erdinç Yılmazel, Ömer Aktepe and Esra Eryol for their endless support, understanding and giving me courage to finish this thesis.

Finally I would like to thank Boğaziçi University Research Fund that support this research in part under the grant number BAP 06HA104.

ABSTRACT

SOURCE FILE LEVEL SOFTWARE DEFECT PREDICTION FRAMEWORK

Defect prediction techniques are used to address defective sections of source code in software products. Applying a defect prediction technique before proceeding to testing phase of software development helps the managers to allocate their resources more efficiently and most core effectively such as time and effort to test certain sections of the code. Defect predictors are useful tools to help project managers to plan test stage during the software development life cycle without compromising on the product quality.

In this research we have taken software defect prediction as a two way classification problem. We have used machine learning techniques to construct our prediction model. One of the challenges in learning based models is the collection of data. In software engineering domain data collection is a major problem. Companies and researchers often struggle to find out the right level of granularity in data collection: i.e. module / function level versus file / class level. In this research we have been motivated by the problem of right level of granularity. Our proposed models use the hierarchical structure information about the source code of the software product, in order to perform defect prediction for high level granularity such as source files (also called classes).

We have run experiments on NASA, SoftLab and Eclipse datasets to validate our proposed model. Additionally we have also performed cost-benefit analysis to evaluate the net effect of using our proposed model.

ÖZET

KAYNAK KOD SEVİYESİNDE YAZILIM HATA KESTİRİMİ YAPISI

Hata kestirim teknikleri yazılım ürünlerindeki hatalı kod parçacıklarının tespit edilmesinde kullanılır. Yazılım geliştirme sürecinin test aşamasına geçilmeden önce hata kestirimi teknikleri uygulanması, yöneticilerin zaman ve iş gücü gibi kaynaklarını kodun belirli bölümlerinin test edilmesi için ayırmasına yardımcı olarak kaynakların verimli ve etkili kullanılmasını sağlar. Hata kestirim araları proje yöneticilerine, yazılım geliştirme sürecinin test kısmını ürün kalitesinden ödün vermeden planlamalarında yardımcı olur.

Bu araştırmada yazılım hata kestirimini iki sınıflı bir sınıflandırma problemi olarak ele aldık. Kestirim modelimizi oluşturmak için otomatik öğrenme teknikleri kullandık. Öğrenme temelli modellerin en uğraştırıcı bölümleri veri toplamaktır. Yazılım mühendisliğinde alan verileri ciddi bir problemdir. Şirketler ve araştırmacılar sıklıkla doğru seviyede veri toplama ile boğuşurlar: Örneğin modül / fonksiyon bazı ile dosya / sınıf bazı seviyeleri. Bu araştırmada doğru veri seviyesi probleminden yola çıktık. Sunduğumuz modeller yazılım ürününün kaynak koduyla ilgili hiyerarşi bilgisini kullanarak kaynak dosya (ya da sınıf) gibi daha üst parçacık seviyelerinde hata kestirimi yapmaktadır.

Sunduğumuz modeli geçerli kılmak için NASA, SoftLab ve Eclipse veri setleri üzerinde deneyler gerçekleştirdik. Ayrıca sunduğumuz modelin net etkisini ortaya çıkarmak için kar-zarar analizi de gerçekleştirdik.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . iii
ABSTRACT . . . iv
ÖZET . . . v
LIST OF FIGURES . . . viii
LIST OF TABLES . . . xi
LIST OF SYMBOLS/ABBREVIATIONS . . . xiv
1. INTRODUCTION . . . 1
1.1. Introduction . . . 1
1.2. Motivation . . . 2
1.3. Outline . . . 3
2. BACKGROUND . . . 4
2.1. Overview . . . 4
2.2. Defect Prediction . . . 4
2.2.1. Defect Prediction with Naïve Bayes Classifier . . . 6
2.2.2. Defect Prediction with Decision Tree Classifier . . . 8
2.3. Naïve Bayes Classifier . . . 9
2.4. Decision Tree Classifier . . . 12
2.5. Software Code Metrics . . . 15
3. PROBLEM STATEMENT . . . 17
4. PROPOSED MODEL . . . 19
4.1. Naïve Bayes Classifier . . . 19
4.1.1. Log-filtering . . . 23
4.2. Decision Tree Classifier . . . 24
4.3. Dataset Processor . . . 27
4.4. Model Output . . . 30
5. EXPERIMENTAL DESIGN . . . 31
5.1. Experimental Setup . . . 31
5.1.1. Datasets . . . 31
5.1.2. Stratified K-fold Cross Validation . . . 34

5.1.3. Statistical Testing: T-Test 37
5.2. Experiments 39
5.2.1. Experiments for Model Validation 39
5.2.2. Experiments for Cost-Benefit Analysis 40
5.2.3. Experiments for Multi-level Prediction 41
5.3. Threats to Validity 41
6. RESULTS 42
6.1. Performance Criteria 42
6.2. Performance 49
6.2.1. Model Validation 49
6.2.2. Decision Tree Classifier 58
6.2.3. Multi-level Prediction 66
6.3. Discussion of the Results 74
7. CONCLUSIONS AND FUTURE WORK 77
7.1. Conclusions 77
7.2. Contributions 78
7.3. Future Work 80
APPENDIX A: SOFTWARE CODE METRICS 81
APPENDIX B: DETAILED ANALYSIS OF DATASETS 83
APPENDIX C: BOX PLOTS OF NAÏVE BAYES EXPERIMENTS 86
APPENDIX D: BOX PLOTS OF DECISION TREE EXPERIMENTS 95
REFERENCES 104

LIST OF FIGURES

Figure 4.1. Overview of the proposed model 20

Figure 4.2. Flowchart of the proposed model 22

Figure 4.3. Data distribution before log-filter 23

Figure 4.4. Data distribution after log-filter 24

Figure 4.5. Decision tree model - generate tree procedure 26

Figure 4.6. Decision tree model - split attribute procedure 26

Figure 4.7. Flowchart of dataset processor 28

Figure 6.1. Cost-benefit Analysis Plot 47

Figure C.1. Comparison of Naïve Bayes box plots for CM1 dataset 86

Figure C.2. Comparison of Naïve Bayes box plots for MC1 dataset 86

Figure C.3. Comparison of Naïve Bayes box plots for PC1 dataset 87

Figure C.4. Comparison of Naïve Bayes box plots for PC2 dataset 87

Figure C.5. Comparison of Naïve Bayes box plots for PC3 dataset 88

Figure C.6. Comparison of Naïve Bayes box plots for AR3 dataset 88

Figure C.7. Comparison of Naïve Bayes box plots for AR4 dataset 89

Figure C.8. Comparison of Naïve Bayes box plots for AR5 dataset 89

Figure C.9. Comparison of Naïve Bayes box plots for jdi dataset 90

Figure C.10. Comparison of Naïve Bayes box plots for jdt_apt dataset 90

Figure C.11. Comparison of Naïve Bayes box plots for jdt_common dataset . . . 91

Figure C.12. Comparison of Naïve Bayes box plots for jdt_core dataset 91

Figure C.13. Comparison of Naïve Bayes box plots for jdt_internal dataset . . . 92

Figure C.14. Comparison of Naïve Bayes box plots for jdt_ui dataset 92

Figure C.15. Comparison of Naïve Bayes box plots for jface dataset 93

Figure C.16. Comparison of Naïve Bayes box plots for pde dataset 93

Figure C.17. Comparison of Naïve Bayes box plots for swt dataset 94

Figure C.18. Comparison of Naïve Bayes box plots for team dataset 94

Figure D.1. Comparison of Decision Tree box plots for CM1 dataset 95

Figure D.2. Comparison of Decision Tree box plots for MC1 dataset 95

Figure D.3. Comparison of Decision Tree box plots for PC1 dataset 96

Figure D.4. Comparison of Decision Tree box plots for PC2 dataset 96

Figure D.5. Comparison of Decision Tree box plots for PC3 dataset 97

Figure D.6. Comparison of Decision Tree box plots for AR3 dataset 97

Figure D.7. Comparison of Decision Tree box plots for AR4 dataset 98

Figure D.8. Comparison of Decision Tree box plots for AR5 dataset 98

Figure D.9. Comparison of Decision Tree box plots for jdi dataset 99

Figure D.10. Comparison of Decision Tree box plots for jdt_apt dataset 99

Figure D.11. Comparison of Decision Tree box plots for jdt_common dataset . . 100

Figure D.12. Comparison of Decision Tree box plots for jdt_core dataset 100

Figure D.13. Comparison of Decision Tree box plots for jdt_internal dataset . . 101

Figure D.14. Comparison of Decision Tree box plots for jdt_ui dataset 101

Figure D.15. Comparison of Decision Tree box plots for jface dataset 102

Figure D.16. Comparison of Decision Tree box plots for pde dataset 102

Figure D.17. Comparison of Decision Tree box plots for swt dataset 103

Figure D.18. Comparison of Decision Tree box plots for team dataset 103

LIST OF TABLES

Table 5.1. Overview of NASA datasets 32

Table 5.2. Overview of SoftLab datasets 33

Table 5.3. Overview of eclipse datasets 34

Table 6.1. Confusion matrix 42

Table 6.2. Measurements used in cost-benefit analysis 48

Table 6.3. Module level vs source file level NB results (NASA with SLOC metrics) 51

Table 6.4. Cost-benefit analysis of NB on NASA datasets 51

Table 6.5. Module level vs source file level NB results (NASA without SLOC metrics) 52

Table 6.6. Module level vs source file level NB results (SoftLab) 53

Table 6.7. Cost-benefit analysis of NB on SoftLab datasets 54

Table 6.8. Module level vs source file level NB results (eclipse) 56

Table 6.9. Cost-benefit analysis of NB on eclipse datasets 57

Table 6.10. Optimum set of E_{max} values 59

Table 6.11. Module level vs source file level DT results (NASA) 60

Table 6.12. Cost-benefit analysis of DT on NASA datasets 60

Table 6.13. Module level vs source file level DT results (SoftLab) 61

Table 6.14. Cost-benefit analysis of DT on SoftLab datasets 62

Table 6.15. Module level vs source file level DT results (eclipse) 63

Table 6.16. Cost-benefit analysis of DT on eclipse datasets 65

Table 6.17. Source file level vs multi-level NB results (NASA) 66

Table 6.18. Cost-benefit analysis of NB on NASA datasets 67

Table 6.19. Source file level vs multi-level NB results (Eclipse) 68

Table 6.20. Cost-benefit analysis of NB on eclipse datasets 70

Table 6.21. Source file level vs multi-level DT results (NASA) 70

Table 6.22. Cost-benefit analysis of DT on NASA datasets 71

Table 6.23. Source file level vs multi-level DT results (eclipse) 72

Table 6.24. Cost-benefit analysis of DT on eclipse datasets 73

Table A.1. Base source code metrics 81

Table A.2. Composite source code metrics 82

Table A.3. Object-oriented source code metrics 82

Table B.1. Detailed dataset analysis - module level (part 1) 83

Table B.2. Detailed dataset analysis - source file level (part 1) 84

Table B.3. Detailed dataset analysis - module level (part 2) 84

Table B.4. Detailed dataset analysis - source file level (part 2) 85

LIST OF SYMBOLS/ABBREVIATIONS

a, A	Static code attribute
CL	Decrease in total lines of codes to be inspected
D	Defectiveness of the modules
d	Defect information for a module
$d - loc$	Rate of defective lines of code
GE	Decrease in verification effort to predict defective modules
IL	Number of lines of code inspected by the model
$nd - loc$	Rate of non-defective lines of code
NL	Number of lines of codes required for random selection of classes
O	Output class
o	Output elements
TL	Total lines of codes of project
σ	Standard Deviation
μ	Mean
CBA	Cost-Benefit Analysis
DT	Decision Tree
fn	False Negative
fp	False Positive
LOC	Lines of Code
M	Module
MAE	Mean Absolute Error
ML	Module Level
MLP	Module Level Predictor
MuL	Multi-level Predictor
NB	Naïve Bayes
NBC	Naïve Bayes Classifier

$NIST$	National Institute of Standards and Technology
p, P	Probability
pd	Probability of Detection
pf	Probability of False-alarm
SF	Source File
SFL	Source File Level
$SFLP$	Source File Level Predictor
SCA	Static Code Attribute
tn	True Negative
tp	True Positive

1. INTRODUCTION

1.1. Introduction

Quality of the software product can be considered in various ways. Roger S. defines software quality as *"how well software is designed and how well the software conforms to that design"* [29]. Measuring the rate of software product's conformance to its design is achieved by performing "testing". Testing is the final step of a general software development life cycle which aims at finding the defects before the product is released. During the testing phase any defects that exist in the software system is addressed and fixed. A defect is defined as "nonconformance to the requirements" [57]. However, performing a thorough testing for the entire software product to address all possible defects is a long-lasting and challenging task [58]. Thus, improvements in testing phase would not only affect the quality of the software but also increase the profitability of the software company as well as increasing the customer satisfaction.

Defectiveness, or defect rate of the software is important in finding out the level of quality of the software product is. Products having a high rate of defects are more prone to contain missing functionalities, perform unexpected behaviors or even end up with crashes. Such a product would definitely cause its customers and stakeholders to be unsatisfied. Thus it is crucial to detect the defects and solve them during the testing phase. One way to tackle this problem is to be able to predict defect before the testing phase [2, 3, 7, 11, 12, 14, 15, 16]. These studies in the literature also confirm that defect prediction models help finding the defective sections of the software and as a result the software testing team can concentrate on the defective sections by optimizing their resources to capture as more defects as possible while spending as less effort as possible.

Static Code Attributes (SCA) are widely used in the literature for defect prediction purposes [2, 3, 14, 15, 17, 25]. Static code attributes are collected from the source code of the software product and can be extracted easily with a tool support. Source

lines of code attributes (SLOC), McCabe attributes and Halstead's attributes are the main types of attributes that are used as SCA. Details of these attributes and some more code attributes can be found in Appendix A.

1.2. Motivation

Defect prediction studies using static code attributes in the literature [2, 3, 7, 15, 25] have produced good results in terms of prediction accuracy. The prediction performance of a machine learning model depends on the algorithm used and the data at hand. In this research we mainly focused on the data side rather than the algorithm In particular we aimed at tackling the issue of "granularity" in collecting software metrics. We have used decision tree classifier [15, 30, 31, 34, 37] and Naïve Bayes classifier [2, 3, 7, 37, 54] as they are the two most commonly used machine learning algorithms used in the literature for predicting the defective modules.

To the best of our knowledge, most studies in the literature so far have focused on defect prediction based on a certain level of software product structure, namely module level or source file level. Module based software code attributes, in other words intra-module metrics are widely used to identify the effects of module structures to the defectiveness of the software and module level software code attributes are available in public domain datasets [18, 64]. Some of those studies propose certain data mining operations to be performed on the module level data to improve prediction performance of their model [31]. We have been motivated in this research to challenge the level of granularity that we collect metric data from. We argue that module level metrics capture the general structure of the units of functions in a software product. Inspecting the software from a higher granularity than module level, would let us capture the general structure of the software and hence leading to better prediction performance without increasing the inspection effort compared to less granular level of metric collection.

1.3. Outline

The remainder of this document is organized as follows:

Background information about the techniques and models that is used will be presented in Chapter 2. Chapter 3 will give brief information about the related studies in the literature for defect prediction.

Chapter 4 will contain the details of the machine learning algorithm that are proposed and also the inputs and outputs of the algorithms will be presented in that chapter.

Chapter 5 will explain the experimental setups that are used to validate the proposed models and also detail the experiments that are performed.

Chapter 6 will present the results of the experiments and provide the comparisons of this study's results to the results of other similar studies in the literature.

Chapter 7 will summarize the study and present the conclusions driven from the results of this study. Possible improvements to the proposed models and future works will also be addressed in that chapter.

2. BACKGROUND

2.1. Overview

In this chapter, some brief information about the methods and models used in this study will be presented. Next section will give information about the Defect Prediction Technique, which aims to increase the quality of the software by addressing the defects in a software product and eliminating them. Section 2.3 and 2.4 will provide information on the specific types of defect prediction methods that are used in this study, namely Naïve Bayes classifier and decision tree classifier, respectively. And the final section will provide necessary background on software code metrics.

2.2. Defect Prediction

According to two independent surveys in 2004 [59] and 2007 [60], one in three IT projects run over their budget. Having high ratio of failures in meeting project requirements, point out a need of improvement for the software development phases. Testing is one of the key phases that effect software development time and budget since testing the functionality of the software, integration of the software with other system components and ensuring the quality of the final product is an extensive task to accomplish [5, 48]. Performing an exhaustive test on the software to detect the defects is a way of ensuring the quality, but is also a costly operation that requires half of the entire product development time to be allocated [5, 34]. Especially when the software grows in both size and complexity, testing phase becomes more difficult and even requires sophisticated procedures to be followed and tool support to be used. Thus, it can be concluded that testing is the most expensive, and challenging phase among software development phases, and needs to be handled with most attention. One possible solution to this problem is to use oracles in guiding test managers to answer questions like "When to stop testing?" and "How much testing is enough?".

Defect prediction is a process of addressing defective sections of a software prod-

uct. There are two ways of performing defect prediction: First, estimating the particular defects in given software product, and second, predicting the number of defects. First prediction technique is not applicable for most of the software products, since it requires extensive documentation of both the structure and the flow of data for the product [48, 49, 61]. For this reason, the second approach is given more importance in both the academic studies and by the managers of software product development teams. Another important feature of the second approach is that it helps the managers to have enough information to plan their testing phase and allocate only necessary amount of labor and effort for testing. Throughout this document, defect prediction term will refer to the second type of defect prediction.

Theoretically, according to the accuracy of the predictor, the effort needed to test the software for defects can be decreased substantially and project managers can benefit from this in allocating resources effectively. The effort necessary to extract the data needed by the machine learning algorithm from the source code and to evaluate the results of the algorithm can be neglected when compared to the effort necessary to test the entire software.

There exist numerous studies on defect prediction in the literature using many different approaches. Some of the studies have combined principles of software engineering with principles of other domains, e.g. biology [47]. Chang et. al. have implemented the "Capture-Recapture (CR)" model from biology which is used to estimate the size of an animal population. Main principle of a CR model is to use the number of animals captured for the first time, and recaptured afterwards to estimate the total size of the animals. Authors applied this principle to defect prediction domain by defining two major factors affecting defect detection, which are the ability of the reviewers to find the defects and the level of difficulty in detecting the defects. Using the principles from CR model, authors developed a sequential re-inspection model which performs two consecutive inspections where second level inspection uses the data from the output of first level inspection. Resulting model turned out to provide more accurate results than single level inspection and variance of the results also were more stable in terms of having less outlier.

Some other studies in the literature have applied several machine learning techniques such as Bayesian networks, neural networks, linear regression analysis, decision trees, and Naïve Bayes classification [7, 14, 15, 16, 30, 31]. Defect prediction methods generally use software metrics [44, 46] and defect information of previous software products to train one (or more) of the listed machine learning algorithms. The trained algorithm is then used to predict the defectiveness of newly implemented software code sections. Results of the predictions might be used as a guideline during the testing phase, giving an insight on the possibly defective sections of software thus letting the testing team to concentrate on those sections rather then entire software.

Before explaining the machine learning models that are used in this research, we'll present some of the studies in the literature that applied the same models in following subsections.

2.2.1. Defect Prediction with Naïve Bayes Classifier

Naïve Bayes is one of the widely used machine learning technique in the defect prediction domain [2, 3, 6, 7, 18, 26, 34, 37, 52]. Tosun et. al. used Naïve Bayes classifier and two other machine learning techniques (neural network and voting feature intervals) in parallel to perform defect prediction where all three models are used to perform defect prediction separately and a module is assumed to be defective only if majority of the models predict that module as defective [18]. Since they applied no weighting among the models, "majority of the models" mean at least two of three models. Authors have compared the results of their study with existing Naïve Bayes classifier studies in the literature and ensemble of defect predictors have turned out to provide improvements in terms of probability of detection and verification (testing) size.

Another study in the literature by Turhan et. al. also used Naïve Bayes to perform defect prediction [34]. In the reference study, authors have provided a model that performs both ways of defect prediction that were addressed in the beginning of this section. They used Naïve Bayes classifier and decision tree classifier in order to

address defective modules in the system and used the output of the classifier model as input to a regression model which tries to predict the number of defects that any defectively predicted module actually has. Authors claim that this approach lets the managers to allocate their resources better since they have the information on density of defects within the software. Results of the study have shown that classification models predicted defective modules as good as manual inspections and also regression models could predict defect densities better since regression models are trained with modules that are more likely to be defective, while many studies in the literature used entire dataset to train their regression models [34].

An interesting study that uses Naïve Bayes to perform defect prediction is performed by Turhan and Bener [26]. Authors have considered modifying one of the main principles of Naïve Bayes classifier, which is "independence assumption". Independence assumption states that the input attributes are not correlated to each other and that each of them are normally distributed. Assuming a normal distribution, $x \sim N(\mu, \sigma^2)$, then the probability density function can be written as in Equation 2.1 [26]:

$$p(x) = \frac{1}{\sqrt{(2\pi)}\sigma} \exp(-\frac{(x-\mu)^2}{2\sigma^2}) \tag{2.1}$$

Turhan et. al. assumed that source code attributes are correlated and inspected the multivariate approach which takes into account the relations between attributes while determining their distribution. A multivariate normal distribution considers x as a d-dimensional vector normally distributed by $N(\vec{\mu}, \vec{\Sigma})$ which results in a probability density function, $p(\vec{x})$ as given in Equation 2.2 [26]:

$$p(x) = \frac{1}{\sqrt{(2\Pi)^{d/2}\Sigma^{d/2}}} \exp\left(-\frac{1}{2}(\vec{x}-\vec{\mu})^T\Sigma^{-1}(\vec{x}-\vec{\mu})\right) \tag{2.2}$$

Results of the experiments have shown that combining features while determining the distribution can provide better performance since it extracts information from the attributes rather than performing feature subset selection. Authors stated that using additional information about the relations of the attributes should be further inspected

along with different distribution assumptions for attributes.

Menzies et. al. have also performed some dataset operations before applying Naïve Bayes classification and obtained better prediction performances with the help of data mining technique they applied [2]. They analyzed the dataset they use for performing defect prediction and observed that attributes have an exponential distribution. They performed log-transformation for the attribute values which reduced the exponential distribution and provided an improvement in the overall performance of the model. Since they obtained the best results on Naïve Bayes classifier with log-filtering applied, they claim that Naïve Bayes can be a better predictor to exploit the association of source code metrics and that log-filtering helps reveal this association better [2].

Current studies in the literature show that existing defect prediction methodologies suffer from "the Ceiling Effect" [6]. Ceiling effect is defined as "some inherent upper bound on the amount of information offered by, say, static code features when identifying modules which contain faults" [6]. It states that the predictions of the defect predictions techniques, no matter which one, is bounded with some limitation based on the data used in predictions. It is proposed that in order to overcome this limitation, some improvements on the data must be accomplished in order to improve the information content of the data. Koru inspected existence of ceiling effect in his study [31] and applied some data mining technique to extract additional information from NASA dataset. Prediction performance of the Naïve Bayes classifier turned out to increase after applying the data mining technique. This result supports the existence of ceiling effect and motivates us in performing additional experiments on testing ceiling effect.

2.2.2. Defect Prediction with Decision Tree Classifier

Decision tree is a widely used machine learning technique for defect prediction purposes [14, 15, 26, 30, 31, 37, 45, 55]. Main factor that increases use of decision trees is that they can provide human-friendly outputs that can be easily interpreted and

explained to managers or developers [14, 26, 31, 37, 55].

Koru and Liu have used decision trees to perform defect prediction in their study [15]. The authors analyzed the NASA datasets and found out that these datasets contain many "small-sized" modules [15] and claimed that having many "small-sized" modules limited their model's ability to learn. Thus they decided to perform defect prediction on subsets of original dataset where each subset contains modules of similar sizes. Results of their experiments have shown that defect prediction model had better performance values in terms of f-measure criteria for subsets containing larger modules and worse performance values for subsets containing smaller modules. Authors state that larger modules will have a better chance to show variation in their measurements which increase the model's ability to learn from data samples [15].

Challagulla et. al. implemented many different machine learning based defect prediction techniques in their study [37] such as decision tree, Naïve Bayes, neural network, nearest neighbor, 1-Rule, and several regression models. They evaluated the models on 4 different NASA datasets and compared the performances of models in terms of Mean Absolute Error (MAE) criteria. Results of the study have shown that even though decision tree model was not successful in predicting the error counts of defective modules, it was the second best approach to predict defects for most of the datasets.

2.3. Naïve Bayes Classifier

Naïve Bayes Classifier is a classifier based on applying the Bayesian probability theory. Bayesian probability theory states that probability of some hypotheses being true when certain conditions are observed in an environment depends on following criteria [61, 62]:

- The probability of observing the same conditions when that hypothesis was known to be true in the past
- The probability of that hypothesis being true (regardless of conditions)

- The probabilities of every distinct conditions in the past (regardless of hypothesis)

In the context of defect prediction, conditions (x) are the source code feature values of structures for which prediction will be performed. A hypothesis in defect prediction context is defectiveness (D) of a source code structure, namely a module or a source file. Thus general Bayesian probability theory formulation can be specialized for defect prediction as in following equation:

$$P(D \mid x) = \frac{P(x \mid D)\, P(D)}{P(x)} \tag{2.3}$$

Naïve Bayes Classifier (NBC), Naïve Bayes in short, is a supervised learning algorithm which is used for data mining purposes and machine learning applications [1]. The main principle that Naïve Bayes depends on is the ”conditional independence of features” principle. According to this principle, all the conditions that are assumed to affect the hypothesis are considered to be unrelated from each other. Without this assumption, implementing a Bayesian probability theory is more challenging since gathering the data needed to train the model becomes challenging. Conditional independence assumption lets the classifier to work on different features as if they are not correlated to any other features, thus makes the model more manageable and easy to use.

General Naïve Bayes algorithms also make a further assumption. The features that affect the hypothesis are assumed to have equal weights on the probability of the hypothesis to be true. A specific version of Naïve Bayes Classifier assigns weights to features using several heuristics and there exist studies in the literature which has proven feature-weight assignment to give promising results [7].

Bayes theorem states that the posterior distribution of a sample is proportional to the prior distribution and the likelihood of the given sample [1]. Formally:

$$P(C_i \mid x) = \frac{P(x \mid C_i)\, P(C_i)}{P(x)} \tag{2.4}$$

$P(x)$, is called the evidence which is calculated from the equation:

$$P(x) = \sum_i P(x \,|\, C_i)P(C_i) \tag{2.5}$$

Thus, the value of the evidence does not depend on any specific C_i, but rather it is a normalizing constant for all classes. Eliminating the evidence from the first equation, we obtain the main equation for Naïve Bayes Classifier:

$$P(C_i \,|\, x) = P(x \,|\, C_i)P(C_i) \tag{2.6}$$

In this equation, $P(C_i)$ is the prior probability of a class, which reveals the probability of finding a data sample which belongs to class C_i. Assuming that there are c classes, namely $C_1, C_2, .., C_n$; $P(C_i)$ can be formulated as follows:

$$P(x) = \frac{\sum_{x=1}^{c} (1 \,|\, C_x = C_i)}{c} \tag{2.7}$$

$P(x|C_i)$ is known as the likelihood function, which calculates the probability of finding a data sample x when we know that it belongs to a specific class C_i. Likelihood of a data sample is calculated by finding the frequency of that data sample in the dataset. This operation is performed for each class exclusively, that is the data samples belonging to class C_i are used to calculate $P(x|C_i)$.

Finally, $P(C_i|x)$ is the probability of a given sample's being belonged to a specific class C_i, namely the posterior probability of the class C_i for the data sample x. Classification of a data sample is straight forward after calculating posterior probabilities of each class for that specific data sample. It is achieved by simply finding the class having the highest posterior probability, and assigning the data sample to that class.

2.4. Decision Tree Classifier

Trees are data structures which are composed of nodes containing some data samples and edges which connect the nodes representing the relations between different data samples [31, 33]. Trees can be defined as special forms of graphs such that two different nodes are connected with only one path. A tree can be used to represent some data in a structural way where data samples matching similar patterns are either placed in the same node, or placed down in the hierarchical structure of some node.

Definition of a decision tree is given by Alpaydın as follows [1]: "A decision tree is a hierarchical model for supervised learning whereby the local region is identified in a sequence of recursive splits in a smaller number of steps". As stated in the definition, decision tree learning is a supervised learning which divides the data into smaller groups by performing a recursive procedure. Resulting decision tree can be used as a supporting tool in decision making purposes since it provides necessary information on how similar data samples can be grouped and also provides brief information about past data having similar characteristics with existing conditions. In order to increase usability of a decision tree and make it possible to be used in a decision making process, tree can also be converted into a set of rules which represent the tree in a clearer way. Ability to represent the tree as a set of rules increases usage areas of decision trees even though their performances are sometimes worse than some other machine learning techniques.

Constructing a simple decision tree is a recursive procedure where at each step a test is performed in order to determine whether the tree should grow further or not. Impurity measure is calculated in order to decide whether a split is necessary [1, 45, 50, 51, 53]. Impurity measure calculates how good a split will be according to its ability of classification among samples. If a split can not provide a good distinction between data samples from different classes, then a low impurity measure result is obtained and the algorithm stops growing the tree. On the other hand, if the algorithm finds a criterion which provides a good split between data samples from different classes, then impurity measure will eventually have a high value and the algorithm will continue

splitting the data using that criterion. If multiple criteria are found with high impurity measures, then the algorithms chooses the best criterion by comparing their impurity measures. Calculations performed during construction of a decision tree, including impurity calculations are explained in detail in the next paragraph.

Assume that for a specific node m in a decision tree, N_m is the number of data samples reaching node m. If there exist i different classes, number of instances belonging to each class can be represented as N_m^i such that $N_m = \sum_i N_m^i$. Thus the probability of a new instance reaching node m belonging to a specific class i can be written as follows:

$$p_m^i = \frac{N_m^i}{N_m} \tag{2.8}$$

Probability values for all classes provide the information about how good the split is. If p_m^i are close to each other, then deciding which class to assign to a specific data sample is not clear since each class has similar probabilities. Thus impurity measure must be high in order to force the algorithm to continue on splitting data into smaller and more pure datasets. Entropy is a function which fits this purpose, thus can be used as an impurity measure:

$$E_m = -\sum_{i=1}^{K} p_m^i \log_2 p_m^i \tag{2.9}$$

A special case for the Equation 2.9 is $0 \times log\, 0$ which is considered to be 0, since in this case no data samples are of class C_i and we want impurity to be small in this case. According to result of Equation 2.9, the algorithm decides whether node m is pure enough or not. In defect prediction techniques, since we have only two classes (defective or non-defective), we can rewrite the E_m equation as follows:

$$E_m = -p_m^0 \log_2 p_m^0 - p_m^1 \log_2 p_m^1 \tag{2.10}$$

The E_m value is compared to a predefined threshold value, E_{max}, which must be

optimized for a specific domain. If E_m value is smaller than the determined threshold value, then algorithm places a leaf node and keeps the p^i_m values in the leaf node for future referencing. When a new data sample reaches a leaf node, p^i_m values are used to determine which class it will be assigned. If E_m value is greater than the threshold value, then algorithm tries to find the best split which can improve purity of the data in sub trees. An iterative process is applied for this purpose and for each possible split positions; the impurity after the split is calculated. After calculating split impurities for each possible split, the criterion providing minimum impurity is chosen to be next splitting criterion.

When calculating the best split position, let N^i_{mj} of data samples reaching node m are of class i and will be assigned to node j after the split such that $N^i_m = \sum_j N^i_{mj}$ and $N^j_m = \sum_i N^i_{mj}$. Thus we can calculate probability of a data sample reaching to the node j as follows:

$$p^i_{mj} = \frac{N^i_{mj}}{N^i_m} \tag{2.11}$$

A similar calculation that we perform to calculate E_m value is performed in order to find the impurity after the split at node m is performed using the inspected criteria.

$$E^t_m = -\sum_j \left(p^0_m \log_2 p^0_m + p^1_m \log_2 p^1_m\right) \tag{2.12}$$

As explained before, the algorithm calculates Equation 2.12 for every possible split position of every criterion in order to find the best splitting position and the criterion that provides minimum E_m value at node m for creating sub trees. When data sample values are numeric, as it is in defect prediction, arithmetic means of each consecutive data sample value is considered as a split position (assuming data sample values are ordered). Assume that the best splitting criterion for node m is c and the best split position for criterion c is v. Decision tree algorithm compares values of criterion c of each N_m data sample with v and assigns the data samples that have smaller v values to m_0 node, and data samples that have greater v values to m_1 node where m_0 and m_1

are direct sub nodes of node m.

2.5. Software Code Metrics

A software metric is defined as "a quantitative measure of the degree to which a system, component, or process possesses for a given attribute." by Halstead [9]. Software metrics can be used for many different purposes, some of which are product release management, software quality analysis, and process quality and efficiency analysis.

Software code metrics, as the name implies, are the types of metrics that are directly extracted from the source code of the software. Software code metrics are also called static code attributes since the metrics do not change in time unless the source code itself changes or the tool that was used to extract the metrics do not change [52]. Extracting the metrics is a complex task which requires a tool support to be handled correctly and efficiently. Tool support is also necessary to make the extracted metrics objective, thus more reliable for the researchers [26, 34].

There are several different software metrics which serve different types of purposes. Source lines of code (SLOC) metrics measure the size of the software program in different size measures like blank LOC, commented LOC, executable LOC, etc. These metrics represent the effort needed to develop the software.

Cyclomatic complexity metrics measure the complexity of the software program. Thomas McCabe used graph-theory to calculate the number of linearly-independent paths through a program module [8]. This complexity measure is important since it is measures of how many different conditions can the software end up with, and each of these conditions must be tested to ensure that the software is defect free. Thus McCabe's metrics can be assumed to represent the effort needed to test the software.

Halstead also provided some complexity metrics in his study [9]. He represented the length of the software as the total number of operators and operands used. He also defined the "vocabulary" term which indicates the number of unique operators

and operands. These metrics are accepted to represent the complexity of a software program and is used widely in the literature [10, 11, 12, 13, 23, 44].

Source code metrics can be extracted on different levels of source code structures. Two widely used source code structures for defect prediction purposes are modules and source files. A module is the smallest unit of source code that is responsible for a specific operation. Modules are called "functions" or "methods" in some programming languages. One or more modules are grouped into a higher level structure, namely a source file.

Previous studies in the literature proven that module level metrics are useful, easy-to-use and also reliable for defect prediction purposes [2]. In the reference study, the author provided a literature survey in the background section of the paper which revealed that software code metrics are widely used in the literature and according to validation and verification textbooks, software code metrics are worthy of manual inspections. The author also shows that static code attributes are useful and can provide promising results on defect prediction domain. He also compares the effort necessary to collect the data required for defect prediction with the effort necessary to test the system, and claims that collecting source code metrics is far easier since they can be collected automatically and evaluated with a defect prediction model.

3. PROBLEM STATEMENT

Testing phase is one of the major phases of software development cycle since it requires nearly half of the effort of entire development [5]. Defect prediction tools have been useful in guiding software product managers to plan their testing phase [40]. There have been various machine learning techniques to build predictors in the literature [2, 3, 7, 15, 16, 26, 27, 28, 37, 38, 39].

Most of the defect prediction techniques in the literature perform defect prediction on module level source code metrics. Menzies et. al. implemented a Naïve Bayes with log-filtering and obtained high pd rates (up to 72%) and low pf rates (down to 25%) [2]. In a recent study Hall used decision trees to assign weights to certain attributes and remove any unnecessary attributes to improve performance of the Naïve Bayes Classifier and have obtained similar results for Naïve Bayes Classifier. Even though such good results are obtained for defect prediction using module level source code metrics, one should note that collecting defect information on module level is a difficult and costly task. In order to collect metrics on lower levels of source code such as modules, companies have to align their software development processes to enable this task, train their employees to correctly follow steps required to collect module level metrics and find appropriate tools that support this task. Such complex issues are not only impractical for most of the companies, but also technically unavailable at the time of this study. Even though large software companies which have mature processes evidenced by various quality certificates (ISO, CMMI.. etc), may have difficulty in adopting their existing processes to regularly collect data. Moreover, construction of a metrics program requires skilled and costly employees. On the other hand, all well known bug (defect) tracking software like Jira, Bugzilla, or TrackStudio let users relate the defect information to certain files that are affected by the defect, which in turn requires additional effort to be spent by development team in order to relate the defects with modules. As a result, there exists limited number of module level metrics in the public data repositories [64, 65].

Considering the difficulties explained in the previous paragraph, our first research question can be formed as: *"Can a defect prediction algorithm be implemented using source file level defect information to provide similar or better performance than a defect predictor using module level defect information?"*. The answer of this question will be a guide to determine the effect of a change in granularity of collected defect information on the performance of the defect predictor which will be accomplished by comparing defect predictors using less granular and high granular structures.

Another aim of this study is to inspect the "ceiling effect for defect predictors" as proposed by Menzies et. al [6]. Authors claimed that existing machine learning techniques can not be improved further since the datasets used have a limited information content which prevents reaching higher prediction performances. In order to inspect this behavior, authors applied micro-sampling technique to reduce the number of samples in the dataset down to 50 samples, and it is observed that datasets of that size can even provide as high performances as larger datasets. The authors concluded that improvements in defect prediction algorithms require information content of the datasets to be increased, rather than trying to provide better predictors.

Second research question of this study would then be addressed as: *"Can the information content of the data be increased with the help of data mining techniques to provide better prediction performances?"*. In order to increase information content of the data, some data operators would be applied as proposed by Koru [31]. If the performance of the defect predictor increases after applying data mining techniques, then the existence of a ceiling effect will eventually be supported in the context of granularity.

4. PROPOSED MODEL

In order to find an answer to the research questions introduced in the previous chapter, we have constructed two defect prediction models in this research. We have addressed the problem of defect prediction as a two way classification (machine learning problem) to investigate the effect of granularity in metric data collection. This chapter will explain the steps followed to implement machine learning techniques.

Naïve Bayes classifier (NBC) is proven to outperform other defect prediction models in the literature [2] and decision tree classifier (DTC) is a widely used machine learning technique for defect prediction purposes [30, 31, 32]. Even though DTC does not provide best results in this area, it is used widely because of the fact that its outputs can be used to draw "human-friendly" conclusions and perform some decision making [14, 26, 31, 55].

In this chapter, detailed information about all dataset operations and implemented models (See Figure 4.1) will be presented. Section 4.1 will explain how Naïve Bayes classifier is implemented and will provide information on log-filtering which is a special operation applied on data as proposed by Menzies. Similarly section 4.2 will provide details of implementation of decision tree classifier proposed in the study. Section 4.3 will provide information on the dataset processor system that is implemented as a part of this thesis study, and that plays an important role in obtaining source code metrics on highly granular source code structures. Finally, section 4.4 will provide information on what outputs is retrieved from the proposed models, and how can they be evaluated to measure performance of the model.

4.1. Naïve Bayes Classifier

A formulation for the Naïve Bayes algorithm was given on Section 2.3. Equation 2.4 can be used to implement a Naïve Bayes algorithm if the data values are discrete values. However defect prediction datasets consist of extracted source code metrics

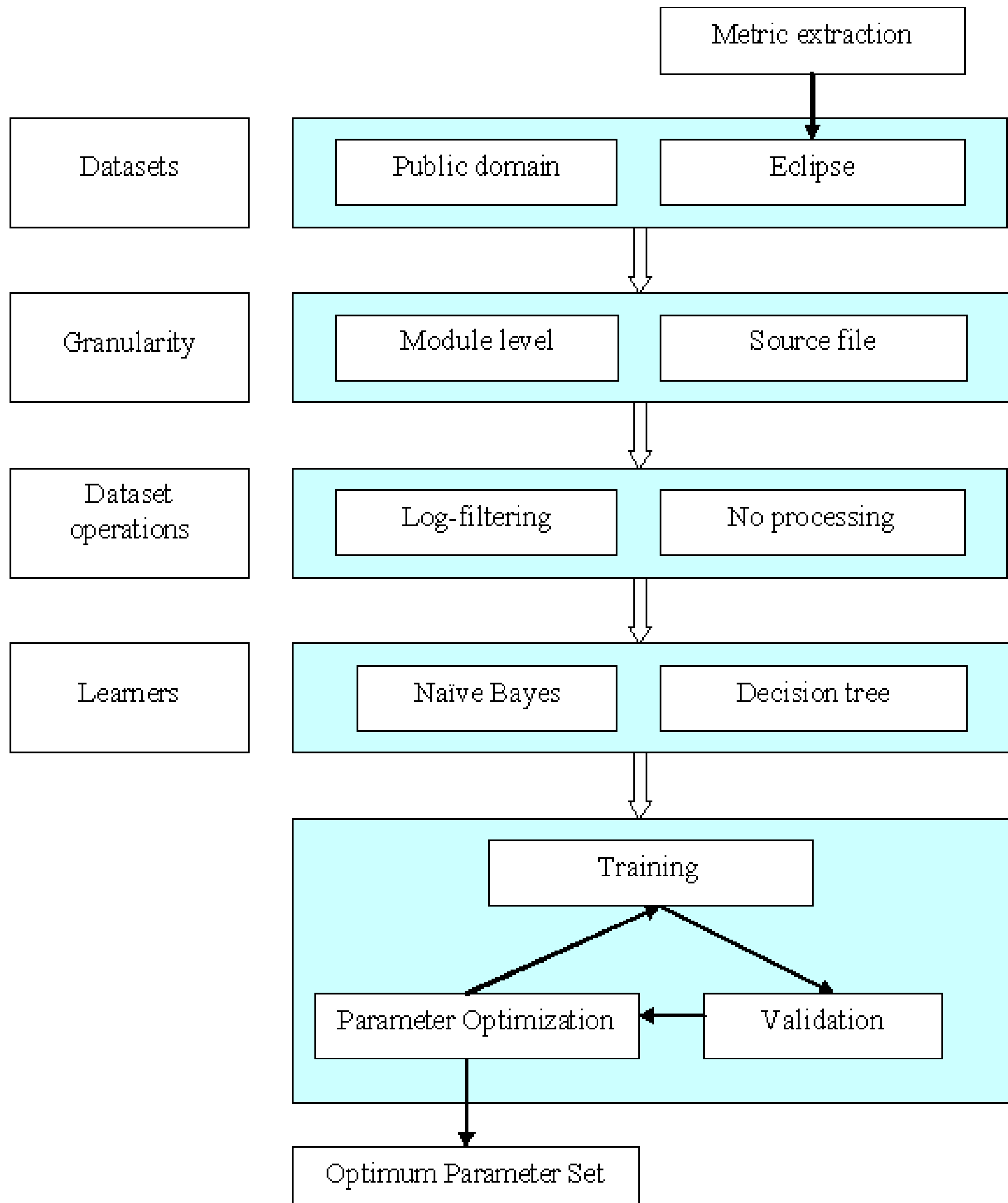

Figure 4.1. Overview of the proposed model

which means all the data values in the dataset are numeric, rather than discrete.

When data samples are discrete values, the prior probability $P(x|C_i)$ of a data sample is calculated by finding the frequency of that specific data sample among all data samples. However when the data samples are continuous values, rather than discrete values, we need to represent the data distribution as polynomials for each of the classes, and calculate the posterior probabilities using that polynomial function. For this purpose, multivariate normal distribution is used to fit the polynomial to the data samples in Naïve Bayes Classifiers.

In general, instead of using Equation 2.4 directly, logarithms are used for computational convenience. Introducing the normal distribution formulation and taking logarithms of both sides of Equation 2.4 gives us following formulation which is also used in this study:

$$g_i(x) = -\frac{1}{2}\sum_{j=1}^{d}\left(\frac{x_j^t - \mu_{ij}}{\sigma_j}\right)^2 + log(P(C_i)) \tag{4.1}$$

μ_{ij} is the mean of jth attribute of data samples belonging to C_i. σ_j is the standard deviation of data samples of jth attribute. Even though there exist i different classes, only a single standard deviation value σ_j is seen in Equation 4.1. σ_j is called the weighted standard deviation. Using a single standard deviation means that the data values of each class are assumed to be similarly spread around the mean of that corresponding class. Thus in the proposed model, instead of using a separate standard deviation for each class, the standard deviations of the classes are used to calculate the weighted standard deviation σ_j. In the proposed model, weighted standard deviation σ_j is calculated by using the following equation:

$$\sigma_j = \sum_i (\sigma_{ij} P(C_i)) \tag{4.2}$$

Training of the proposed model consists of finding the values of the parameters of $g_i(x)$ function, which are μ_{ij}, σ_j and $P(C_i)$. Original dataset that is given as input to the

model is split into two distinct datasets, training and validation datasets. Training dataset is used to determine the model parameters, and data samples from the validation set are classified using the learned parameters. Predicted labels of the model are compared to the real values and prediction performance of the model is found.

Main steps of the proposed model are visualized in Figure 4.2 as follows:

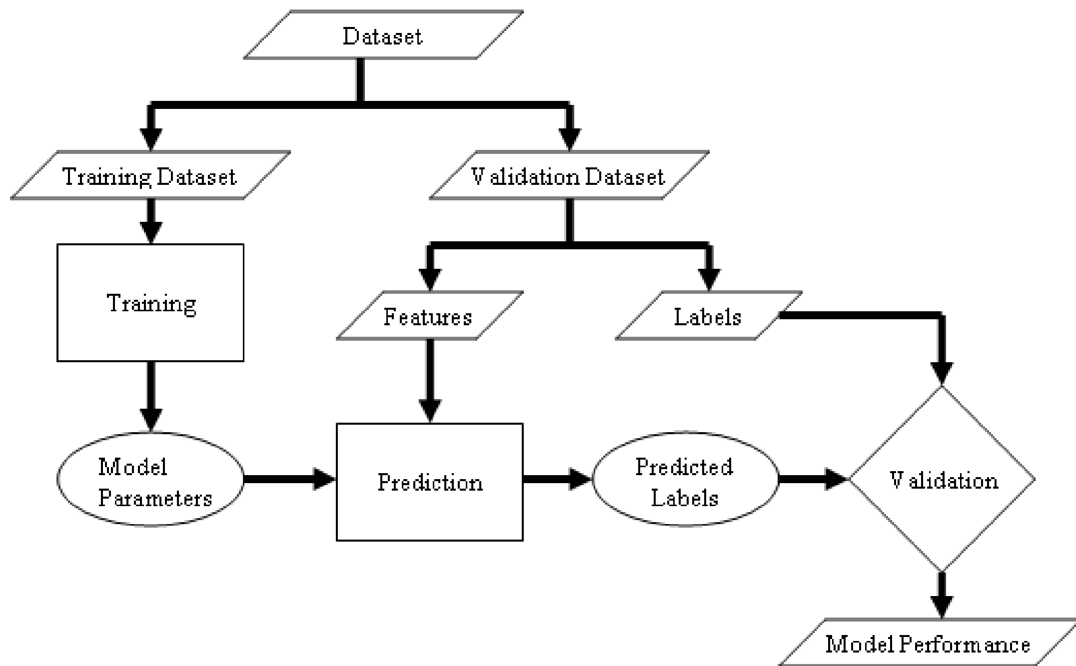

Figure 4.2. Flowchart of the proposed model

In defect prediction, classification means predicting a data sample as either ”defective” or ”non-defective”. Thus in a defect prediction problem, there exist two classes, C_0 being the class of non-defective items and C_1 being the class of defective items.

Classification of the data samples is straight-forward after the model parameters are calculated. For a given data sample (a module or a source file) with its attributes as the data values, the proposed model evaluates Equation 4.1 for both C_0 and C_1, and finds values of $g_0(x)$ and $g_1(x)$ respectively. Then the model assigns the data sample to the class with the greater $g_i(x)$ value.

4.1.1. Log-filtering

Menzies et. al. showed in their study that the distribution of values of features for NASA dataset are exponential [2]. This behavior can be easily visualized by sorting any features value in ascending order and plotting the values on an x-y graph as in Figure 4.3:

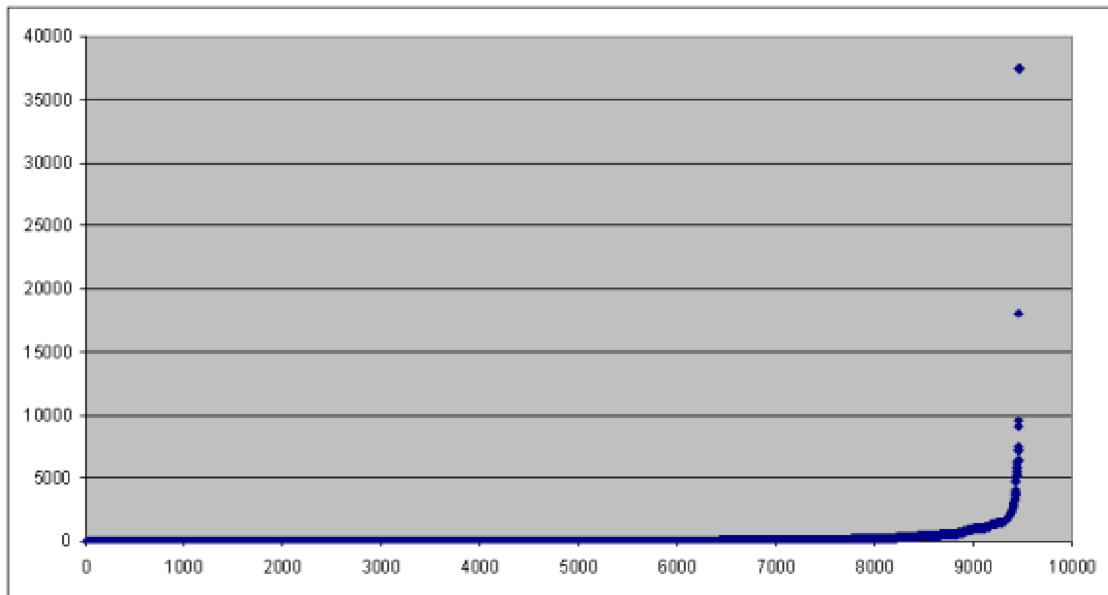

Figure 4.3. Data distribution before log-filter

It is also shown that removing this exponential distribution by applying a log-filter improves the predictor performance. Applying a log-filter simply means that all numeric values in the dataset are replaced with their natural logarithm values. When applying log-filter, it should be noted that some data values are 0 in the original dataset. Thus before applying the log-filter, all data values that are equal to 0 must be changed to a very low pivot value whose log value can be calculated. In the reference study, this pivot value is chosen to be 0.000001 which is negligible compared to the data values of range [0-40000] (see Figure 4.3). Thus in this study, the same pivot value is replaced with all 0 valued data attributes before log-filter is applied [2].

After applying the log-filter to the dataset, it can be seen from the following figure that the data values become spread across the y-axes. This change in the data makes the data more meaningful and lets the Naïve Bayes Classifier reason about the

data easily.

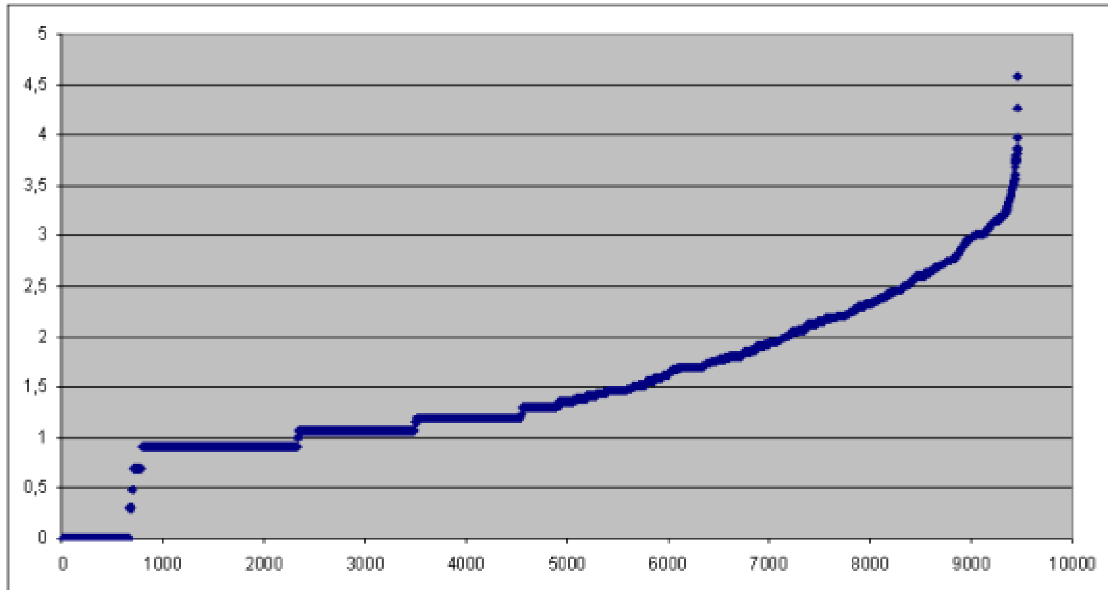

Figure 4.4. Data distribution after log-filter

As seen in Figure 4.4, the data points are now spread over the y-axes. In Figure 4.3, it can be seen that nearly 80% of the data points are close to the x-axes while in Figure 4.4 more than 90% of the data points are spread around the y-axes. This change in the behavior increases the ability of the Naïve Bayes algorithm to train [2].

4.2. Decision Tree Classifier

A general decision tree classifier is explained in section 2.4. This subsection will provide information on specific operations and modifications performed on the general DTC in order to perform defect prediction on source file level source code metrics. Decision trees are capable of handling both numerical data values and discrete data values at the same time. However in defect prediction only numerical data is used for training and validating the model. Thus the model is optimized so that it can handle numerical data better and effectively.

One of the improvements that are implemented in the proposed model is reducing the test criteria when trying to find the best split. In the general case, as explained earlier, a decision tree algorithm tries to find the best attribute which provides a good

split at a certain node. In order to find the best splitting attribute, it inspects each of the attributes and performs an iterative procedure for each attribute. An initial splitting position is selected for a specific attribute, and the impurity after splitting data according to that initial position is calculated. Then the algorithm iteratively increases the splitting position by a predefined amount, Δs, and calculates impurity after split at each iterate. This approach guarantees a constant amount of iterations to be performed at each iterate, however it does not guarantee to find the best splitting position. Thus the proposed model tries to find the best splitting position by trying every possible splitting position while keeping the test criteria as low as possible. For this purpose, the data values are sorted in increasing order and mean values of each adjacent data samples are used for splitting the data. Also while sorting the data, data samples having the same value are treated as a single value which further decreased the search space. Assuming that a node contains n data samples, but only m of them are unique; then the algorithm saves $m - n$ impurity calculations which increases the performance of the model.

Proposed model is also implemented such that the model takes the E_{max} value as a parameter which lets modifying the model according to different dataset easily. Value of E_{max} plays an important role in determining the size of the tree which in turn affects the performance of the model dramatically. In defect prediction, datasets containing high number of defective data samples and very low rate of non-defective data samples the E_{max} value must be kept as low as possible since even at the root of the tree the impurity values will be high due to the structure of the dataset. On the other hand, E_{max} value must be optimized around a high value for datasets which contain similar rates of samples from different classes.

The pseudocode of a general decision tree model is provided in [1], modified version of the proposed decision tree model can be given as follows:

Algorithm GenerateTree(X)

1: **if** $Entropy(X) < E_{max}$ **then** {Stop growing the tree}
2:   Create a leaf node and save ratios of classes at node
3:   Return
4: **end if**
5: $[i, v] \leftarrow$ SplitAttribute (X)
6: $X_L = \{\forall x_i \in X \mid x_i < v\}$
7: $X_R = \{\forall x_i \in X \mid x_i \geq v\}$
8: GenerateTree(X_L)
9: GenerateTree(X_R)
10: Return X

Figure 4.5. Decision tree model - generate tree procedure

Algorithm SplitAttribute(X)

1: MinEnt $\leftarrow$ MAX
2: **for** $i = 1$ to d **do** {For all attributes}
3:   $X \leftarrow$ Sort(X, i) { sort X according to i^{th} attribute, filter unique values }
4: **end for**
5: **for** $x_i \in X$ **do** {For all split positions}
6:   Split X into X_L and X_R
7:   $e \leftarrow$ SplitEntropy(X_L, X_R)
8:   **if** $e <$ MinEnt **then** {Mark minimum entropy position}
9:     MinEnt $\leftarrow e$;
10:     bestAttribute $\leftarrow i$
11:   **end if**
12:   Return bestAttribute
13: **end for**

Figure 4.6. Decision tree model - split attribute procedure

4.3. Dataset Processor

In this study, 5 datasets (CM1, MC1, PC1, PC2, and PC3) from the NASA MDP repository and 3 datasets (AR3, AR4, and AR5) from SoftLab data repository are used. Additionally, software code metrics for 11 sub-projects of Eclipse project is extracted on module level and these datasets are evaluated with the proposed model. All 19 datasets contain defect information and metrics extracted on module level. However this study aims at performing defect prediction on highly granular structures such as source files, so that a dataset processor is implemented to extract source file level defect information and source code metrics from module level defect information and source code metrics, respectively.

A source file may contain one or more modules, thus the metrics that belong to the modules of a source file should be consolidated to obtain source file level metrics. In order to perform this task, first it is necessary to construct a hierarchical representation of the source code and determine which module belongs to which source file. Datasets from NASA and SoftLab repositories contain this information with different representations. Proposed dataset processor takes a NASA dataset or a SoftLab dataset as input, processes it to construct source code hierarchy, and provides an output dataset which contains source file level source code metrics and defect information.

NASA datasets contain several excel files, each providing different types of information about the dataset. Files with "_product_module_metrics" suffix contain module level source code metrics and defect information. Files with "_product_hierarchy.csv" suffix contain the hierarchy information of source code as a mapping of each module to the source file it belongs to. These two input files are used to consolidate the metrics from module level to source file level for NASA datasets.

SoftLab data repository contains one excel file for each dataset which presents both module level and source file level source code metrics as separate excel sheets. Module level source code metrics sheet also contains the hierarchy information such that the source file name of a module is stored in a column of the sheet. In order to

find the source file of a module, a mapping between source file names and source file level source code metrics is created and the mapping is used to consolidate metrics from module level to source file level. When consolidating metrics to source file level, original source code metrics for the source files are also stored and used with the proposed defect prediction models.

The output of the dataset processor is a single excel file for each dataset, which contains both the module level and source file level metrics of the corresponding dataset. Module level information are provided in a sheet named "Module Based Hierarchical" and source file level information are provided in a sheet named "SourceFile Based" in the output file.

General flow of the dataset processor can be visualized in Figure 4.7 as follows:

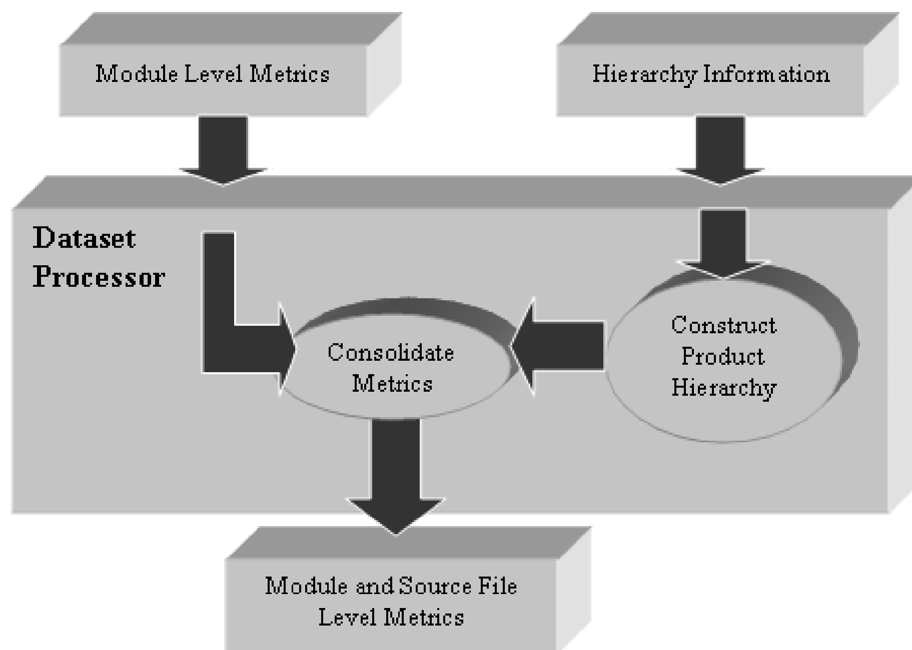

Figure 4.7. Flowchart of dataset processor

When consolidating the metrics from module level to source file level, four different operators are applied: *min*, *max*, *sum* and average (*avg*). These operators are applied for each feature of the modules belonging to the same source file, and results are assigned to be the source file's corresponding feature. Assuming that,

A_{mi} represents the ith feature value of mth module,

$m \in S_j$ means that module m is a module within the source file S_j,

S_{ji}^{op} represents the ith feature value of jth source file when operator *op* is applied, where *op* can be one of the four operators, namely *min*, *max*, *sum* or *avg*. Under these assumptions *min*, *max*, *sum* and *avg* operators can be formulized as follows:

$$S_{ji}^{min} = \min\left(A_{mi} | \forall m \in S_j\right) \tag{4.3}$$

$$S_{ji}^{max} = \max\left(A_{mi} | \forall m \in S_j\right) \tag{4.4}$$

$$S_{ji}^{sum} = \sum_{m}\left(A_{mi} | \forall m \in S_j\right) \tag{4.5}$$

$$S_{ji}^{avg} = \frac{\sum_{m}\left(A_{mi} | \forall m \in S_j\right)}{\sum_{m}\left(1 | \forall m \in S_j\right)} \tag{4.6}$$

Defectiveness of the modules is determined by looking at the *errorcount* attribute for NASA datasets and *defectcount* attribute for SoftLab datasets. Modules which have 1 or more errors (defects) are assumed to be defective, and the ones with error count as 0 are assumed to be non-defective. When consolidating the metrics from module level to source file level, error count values are also consolidated with the sum operator (see Equation 4.4). Resulting value is the error count for the corresponding source file and defectiveness of a source file is also determined using the error count value of that source file such that source files having error value equal to 0 are assumed to be non-defective; and they are assumed to be defective otherwise. This means that, if a source file has at least one defective module, its error count will be greater than 0; thus it is assumed to be defective.

4.4. Model Output

The proposed models are trained and validated after several experiments which are explained in detail in chapter 5. After the experiments, an optimum parameter set is determined for each of the datasets. Optimum set of parameters can then be used to predict defectiveness of new datasets.

For Naïve Bayes Classifier the output of the proposed model is the optimum set of parameters which are:

$$\mu_{ij} \mid \forall a_j \in A \;\wedge\; i = \{0, 1\} \tag{4.7}$$

$$\sigma_j \mid \forall a_j \in A \tag{4.8}$$

$$P(C_i) \mid i = \{0, 1\} \tag{4.9}$$

For decision tree classifier, the model provides the structure of the resulting decision tree. Each node in the tree is either a leaf node or a decision node. Leaf nodes keep the ratios of the classes that data samples at that node belong to. Decision nodes keep the value of the attribute which provided the best split for that particular node and also keeps the best splitting value for that attribute.

When model parameters are obtained, both models can be used to predict defectiveness of newly implemented source code structures, namely modules or source files. Predictions of the models can then be used to compare with actual defectiveness of the source code structures to measure the performance of the model.

5. EXPERIMENTAL DESIGN

In order to validate the proposed model and evaluate its performance, several experiments are performed. This chapter will provide detailed information about the designs and steps of the experiments. Section 5.1 will explain the designs of performed experiments including datasets used and steps to ensure validity of the experiments. Section 5.2 will explain 3 major types of experiments that are performed. Section 5.3 will conclude this chapter by explaining some important points that we consider as threats against validity of the experiments.

5.1. Experimental Setup

Experiments that are performed throughout this study are designed in order to provide valid and reliable results. This section will identify all the steps taken to ensure validity of the experiments.

5.1.1. Datasets

In this study 19 datasets from different data repositories are evaluated. 5 public datasets are selected from the NASA data repository which contains SoftLab data repository with 5 different datasets and 3 of these datasets are selected for evaluation in this research. Remaining 11 datasets are extracted from source code of an open source project and evaluated for the first time for defect prediction purposes in the literature. Both NASA and SoftLab data repositories are publicly available and can be downloaded from [65].

NASA data repository contains 11 datasets from several domains and implemented in several programming languages including procedural languages like C and object-oriented languages like Java and C++. MC2 and PC5 datasets from NASA data repository have different formats than the other datasets, so they are not included in this study. PC4 dataset is also not included in the study, since it contains only 1 defec-

tive data sample and training and validating the model with only one single defective data sample is inapplicable. Similarly, KC4 and MW1 datasets are also not applicable for evaluation since both datasets contain only 1 source file. KC3 dataset is also not evaluated since it contains only 5 source files (2 non-defective, 3 defective). Since K-fold cross validation (explained later in this section) is applied, the model could not be trained and validated using only 2 non-defective data samples. Remaining datasets, namely CM1, PC1, PC2, PC3 and MC1 are evaluated using both machine learning techniques proposed in this study and results are reported for these datasets. For a detailed report on the NASA Datasets, refer to Appendix A.

Some basic information about NASA datasets both on module level and source file level is presented in the following table:

Table 5.1. Overview of NASA datasets

	Module level			Source file level		
Name	# of features	# of modules	Defect rate (%)	# of features	# of source files	Defect rate (%)
CM1	38	505	9,50	152	21	43,86
MC1	37	9466	0,71	148	57	45,61
PC1	38	1107	6,86	152	29	58,62
PC2	38	5589	0,41	152	11	81,82
PC3	38	1563	10,24	152	29	58,62

One difference to be noted here among the datasets is that, MC1 contains one less feature for module level metrics (thus 4 less features for source file level metrics), which is DECISION_DENSITY. Existence of another metric named DECISION_COUNT makes it possible to omit the DECISION_DENSITY feature since it would not affect the performance of the model significantly.

SoftLab data repository is created by the SoftLab members at Boğaziçi University in cooperation with a white goods manufacturer company in the industry. At the time of this thesis study the data repository contained a total of 5 datasets, all from the

same domain (Embedded Systems Domain) and all implemented with C programming language. In this research 3 datasets from SoftLab data repository, namely AR3, AR4 and AR5 are used in order to be able to make comparisons with the existing study in the literature using the same datasets [6].

Some basic information about SoftLab datasets both on module level and source file level is presented in the following table:

Table 5.2. Overview of SoftLab datasets

	Module level			Source file level		
Name	# of features	# of modules	Defect rate (%)	# of features	# of source files	Defect rate (%)
Ar3	29	63	12,69	145	8	37,50
Ar4	29	107	18,69	145	15	73,33
Ar5	29	36	22,22	145	7	42,86

Table 5.2 shows that module level source code metrics are composed of 29 features, but source file level source code metrics are composed of 145 features. The ratio of number of features for a source file to number of features for a module is 5 for SoftLab datasets (it is 4 for NASA datasets, see Table 5.1). Additional features for SoftLab datasets are present in the dataset itself, which is source file level metrics extracted directly from the source files. These metrics are not subtracted from the final dataset since they will also provide additional information and might help the model capture source file structures better.

Eclipse is an open source project aiming at developing a universal toolset for development environment and is protected under Eclipse Public License (EPL) [67]. Source code of eclipse project is available either as bundled to the project executable which can be downloaded from [68] or over concurrent versioning system (CVS) access [69]. Eclipse source code is analyzed with a metric extraction tool named Predictive 3 [70], and both module level and source file level metrics are extracted. Predictive 3 assesses the risk level of the source code and provides a defect prediction for each

module according to its risk level. Thus resulting dataset consists of both module level data and also predicted defect information for any source code.

Some basic information about Eclipse datasets both on module level and source file level is presented in the following table:

Table 5.3. Overview of eclipse datasets

	Module level			Source file level		
Name	# of features	# of modules	Defect rate (%)	# of features	# of source files	Defect rate (%)
jdi	25	1393	42	114	133	18
jdt_apt	25	1247	43	114	148	24
jdt_common	25	797	45	114	57	20
jdt_core	25	3947	135	114	222	35
jdt_internal	25	43938	2519	114	3244	978
jdt_ui	25	2040	63	114	152	30
jface	25	7822	211	114	496	110
pde	25	18011	622	114	1649	370
swt	25	4359	349	114	302	100
team	25	11510	321	114	962	208
ui	25	26054	977	114	2349	522

5.1.2. Stratified K-fold Cross Validation

When performing the experiments, stratified K-fold cross validation is applied in order to make sure that, valid and reliable results will be obtained. General strategy of K-fold cross validation can be summarized as follows: Firstly, the dataset is divided into k folds each having equal number of data samples. Then, one of the folds is kept as the validation fold, and remaining $k-1$ folds are marked as training folds. The model is trained by using the data samples from training folds, and validation fold is only used for evaluating the models performance after training is complete. This training-validation cycle is repeated n times, by selecting a different fold as validation fold at each cycle and training the model with the remaining $k-1$ folds in order to

make sure that each fold is used for validation exactly once.

Datasets used in this study have very different characteristics, especially from size perspective. Some datasets contain tens of thousands of modules and thousands of source files, while some others contain only tens of modules and up to ten source files in total. Using an identical dataset splitting strategy for all datasets would result in erroneous experimental setups. Assuming that a fixed value for the number of folds to be created is set in designing experiments (e.g. $k = 10$), smaller datasets would be split into small-sized folds and larger datasets would be split into large-sized folds. This turns out to be a problem especially for SoftLab datasets, where folds would contain only 1 or 2 data samples which would limit the learning capability of the model from the data. Thus the value of k must be optimized for all datasets, in order to let the machine learning algorithm to have optimum number of data samples in both training and validation folds. The proposed model is implemented to let the users to select the number of folds to be created before running the defect prediction algorithm. By this feature, the proposed model is capable of performing 2-fold cross validation for small datasets (as in AR3 and AR5 datasets which contain only 8 and 7 source files respectively), and it can also perform 10-fold cross validation which is a de-facto standard for validating larger datasets.

When applying K-fold cross validation, the ratio of defective and non-defective data samples in the original dataset is preserved as much as possible for all k folds. This process is important when a dataset contains many defective data samples and only few non-defective data samples, or vice versa. If such a dataset is being evaluated, there is a chance that all defective (or non-defective) data samples may be placed into the same fold which will affect the performance of the predictor negatively. In order to ensure that such a case will never occur, we try to preserve the ratios of classes in resulting folds, which classifies our validation strategy as stratified K-fold cross validation.

Preserving ratios of defective and non-defective samples also helps building up folds that are better subsets of the original dataset. At the very least it is ensured that the ratio of defective and non-defective samples are equal to the ratios from original

dataset, which provides one major advantage: the model will be able to learn both defective and non-defective data samples so it will be able to learn behaviors of both classes. Thus resulting model will be a better predictor for future predictions assuming that the original dataset is a qualified representation of the real world.

In order to ensure that validation set will contain both defective and non-defective data samples, dataset is first split into two sub-datasets, one containing only defective samples and the other containing only non-defective samples. After splitting is done, the samples in both sub-datasets are shuffled to randomize their orders. k, being the number of bins to be created, first $(k-1)/k$ of defective samples sub-dataset and first $(k-1)/k$ of non-defective samples sub-dataset are selected and merged to construct the $k-1$ training bins. Remaining samples in the defective and non-defective samples sub-datasets are merged to construct the validation bin. Finally the orders in the training and validation bins are randomized in order to ensure that a random dataset is created. Even though the order of samples in the dataset is not important for Naïve Bayes Classifier, this randomization is implemented to let the dataset splitter usable for other classifiers as well.

However applying stratified K-fold cross validation by itself is prone to failures depending on the initial selection of folds [2, 15, 24, 31]. For example if the validation fold is not a good sample of the remainder of the dataset, then the model's performance will probably be lower than it might be. In order to decrease the negative effect of selection of badly structured folds on the final results, K-fold cross validation is repeated n times. Orders of the dataset items are randomized before each repetition in order to decrease the probability of working on the same folds. In order to increase the probability of working with every possible set of folds, the value of n needs to be increased, however the algorithm will surely suffer from long execution times. On the other hand, choosing small values for n will probably cause the overall performance of the model to change significantly depending on the validation fold selected. In order to provide a basis for comparison, the value of n is chosen to be 10 as in the reference study [6].

As explained in subsection 4.1.1, the distributions of the data samples are exponential for most of the attributes in NASA datasets. Exponential distribution means that most of the data samples are close to 0 and only few of them are spread around y-axes (see Figure 4.2). Thus a log-filter is applied to eliminate this exponential distribution which is proven to increase the performance of the classifier [2] since it leads to more spread distribution of data samples around the y-axes (see Figure 4.3).

5.1.3. Statistical Testing: T-Test

A *statistical test* (or statistical hypothesis test) is a mechanism to determine whether a hypothesis should be accepted or rejected. According to the results of statistical test, a hypothesis can either be rejected based on the existing evidence, or it can be concluded that there is not enough evidence to reject the hypothesis and it is accepted. Generally, instead of accepting a hypothesis as it is, reverse of the hypothesis is examined and if it can be rejected with existing evidence then original hypothesis is said to be accepted. Reverse of the hypothesis is called "*null hypothesis*".

T-test is a kind of statistical test which assesses whether the means of two groups are statistically different from each other [36]. This kind of statistical test is useful in determining whether two independent groups of data are related to each other by finding the degree of their relations. In defect prediction studies (and many other domains), t-test is used to assess whether results of two defect prediction models are significantly different than each other. This is useful when a researcher wants to check whether results of a newly developed model are different enough than those existing in the literature to claim that the proposed model can provide distinct behavior. It should be noted that t-test does not evaluate the performance of the model; it rather examines the results of the study from the significance point of view.

Inspection of significance with t-test starts with defining a null hypothesis, which may either be directional or non-directional. A directional null hypothesis aims to find a relation between the results in a predetermined direction such that the mean of one group of results is significantly greater than mean of the other group or vice

versa. However a non-directional null-hypothesis only aims at finding a significant difference between results, no matter in which direction (which mean is greater than the other). In this thesis study, non-directional null-hypothesis is inspected since our aim is to determine whether the results are significantly different than each other and the performance of the model will be evaluated using the criteria defined in Chapter 6.

After defining the null-hypothesis, means and standard deviations of two groups of data are calculated. Assuming that first group of data (group A) contains N_A elements and is distributed normally with (μ_A, σ_A^2) and second group of data (group B) contains N_B elements and is distributed normally with (μ_B, σ_B^2); variance of the entire source population (group A and group B) is calculated using the following equation:

$$\sigma_p^2 = \frac{\sigma_A + \sigma_B}{(N_A - 1) + (N_B - 1)} \tag{5.1}$$

Variance of the source population is used to calculate standard deviation of the source population with the following equation:

$$\sigma_p = \sqrt{\frac{\sigma_p^2}{N_A} + \frac{\sigma_p^2}{N_B}} \tag{5.2}$$

The t-value for the source population can then be calculated using Equation 5.3 from the means of groups and the standard deviation obtained from Equation 5.2:

$$t = \frac{\mu_A - \mu_B}{\sigma_p} \tag{5.3}$$

The resulting t-value is used to determine whether results in the groups are statistically significant with the help of t-table. T-table provides t-values for specific source population sizes and for specific significance levels. Significance level (or risk level) is defined as the probability of making a decision to reject the null hypothesis when the null hypothesis is actually true. Significance level is represented with an alpha parameter which is taken to be 0.05 as a rule of thumb in many studies [6, 7, 18, 19], thus the same level of significance is used in this thesis study.

5.2. Experiments

In this section, three main experiments that are performed to validate the models are presented. In subsection 5.2.1, experiments for validating the models and comparing the results with the existing studies in the literature are presented. Subsection 5.2.2 explains some data related issues that were presented in [56], and further explains the experiments that are performed in this study to overcome those problems.

5.2.1. Experiments for Model Validation

In the first phase of experiments, main aim is to inspect whether the proposed model is valid and can be used as an alternative to the existing models on defect prediction. In order to ensure that performed experiments can be used to compare the results with the results of the existing studies, the experimental setup defined in section 5.1 is used.

Menzies et. al. has shown that the Naïve Bayes algorithm give the most useful results among the other defect prediction models [2]. The results of proposed Naïve Bayes Classifier are compared to the results in the reference study since it already provides the best results for defect prediction in the literature so far. In order to be able to make comparisons, Naïve Bayes Classifier is trained and validated with the same dataset and same features.

Koru et. al. implemented a Decision Tree Classifier using Weka's J48 learner in order to predict the defects on class level for NASA datasets [15]. J48 is an implementation of a special form of decision tree learning algorithm. The proposed model will also be compared to the reference study in terms of pd, pf, balance and f-measure criteria and results will be reported in chapter 6.

5.2.2. Experiments for Cost-Benefit Analysis

In order to evaluate the model in terms of its usefulness an additional set of experiments is performed. Results of the experiments are used to perform cost-benefit analysis which compares the model's benefits with its costs, and provides an understandable interpretation of the model's results for non-technical staff.

In order to measure the cost of the model, the "size" of the source code that needs to be tested must be measured. NASA dataset contains 6 features for measuring the size of the source code in terms of SLOC measures. These features are:

- # of lines of codes (LOC_TOTAL)
- # of blank lines (LOC_BLANK)
- # of lines with both code and comment (LOC_CODE_AND_COMMENT)
- # of lines of comments (LOC_COMMENTS)
- # of lines of executable statements (LOC_EXECUTABLE)
- Total # of lines (NUMBER_OF_LINES)

However, in a recent workshop [56] the quality of these metrics is discussed and many researchers concluded that they are not reliable enough to be used in defect prediction. Koru [66] has shown that MC1 dataset contains nearly half of its SLOC measures as 0, which raises a question about the quality of the dataset. The researcher who extracted the source code metrics explained this problem and emphasized that this is an issue related to the SLOC measures only, and that other metrics are not corrupted [66]. It is also addressed that this problem existed in only MC1, PC2, PC3, PC4, and PC5 datasets and that SLOC measures of these datasets should not be used for defect prediction purposes.

Reliable SLOC measures are necessary to measure the cost of the proposed model. Thus, cost-benefit analysis could not be performed for the projects listed above. Depending on the emphasis of the researcher, CM1 and PC1 datasets contain reliable SLOC measures and these datasets are used for performing cost-benefit

analysis throughout this research. Furthermore, the datasets from SoftLab data repository and Eclipse data repository are also available for the purposes of cost-benefit analysis, and they will be evaluated in the experiments.

5.2.3. Experiments for Multi-level Prediction

Another set of experiments are designed in order to evaluate the performance of multi-level prediction. Multi-level prediction can be defined as predicting the defects on source file level and on module level consecutively. Firstly, a source file level prediction is performed to predict defective source files in the system, and a module level prediction for the modules of the defectively predicted source files is performed to narrow down the code coverage necessary to be tested.

Main aim of performing multi-level prediction is to take advantage of both prediction approaches. Source file level prediction will hopefully narrow initial search space to a smaller set of source files, and module level prediction will decrease the total lines of codes needed to be tested in the testing phase.

5.3. Threats to Validity

One problem that was encountered during the study was addressed after the conclusion from a recent workshop was published, as it is explained in 5.2.2. According to the researchers, three of the datasets that were used in this study contained problems with some static code features, thus it was not reliable to use them for defect prediction purposes. For this reason, when performing cost-benefit analysis, these datasets became unavailable for evaluation.

6. RESULTS

In this section and its subsections, the criteria that are considered when comparing the model to the existing models and the results in terms of those criteria are presented.

6.1. Performance Criteria

The main criteria for comparing this study with the studies in the literature are the probability of detection (pd), probability of false alarm (pf) and balance (balance) criteria. As shown in Figure 4.1, the model is validated by taking two input sets, which are prediction labels and validation labels. Validation labels are the classes of data samples from the validation bin, and prediction labels are the classes that are predicted by the model to the same samples from the validation bin. Thus, by using the predicted and actual label values, a confusion matrix is created as follows:

Table 6.1. Confusion matrix

		Predicted	
		Defective	Non-defective
Real	Defective	tp	fn
	Non-defective	fp	tn

The terms in the table can be explained as follows:

True Positive (tp) : a defective module is classified as defective.
False Positive (fp) : a non-defective module is classified as defective.
False Negative (fn) : a defective module is classified as non-defective.
True Negative (tn) : a non-defective module is classified as non-defective.

These definitions can be used to calculate the pd and pf values as given in the following equations:

$$pd = \frac{tp}{tp + fn} \tag{6.1}$$

$$pf = \frac{fp}{fp + tn} \tag{6.2}$$

Probability of detection (pd) value measures the rate of finding the defective data samples. Probability of false alarm (pf) value, in turn, measures the rate of the models' incorrect classifications for non-defective data samples. An ideal classifier would classify the samples so that pd value would be 1 and the pf value would be 0. This case means that the system correctly classified all the defective items and did not classify any non-defective item as defective. However, such a case is not always possible to reach in real life. Thus, we can drive a third criterion, the balance criterion, using the pd and pf values with the following equation:

$$balance = 1 - \frac{\sqrt{(0 - pf)^2 + (1 - pd)^2}}{\sqrt{2}} \tag{6.3}$$

In general, having higher pd rates, lower pf rates and high balance value is expected. A predictor resulting in high pd and low pf rates will have a high balance value, as desired. However, according to the domain of interest, it may be acceptable to have high pf rates, i.e. embedded system domain. Oral et. al. explained in their study [41], that deployment process for software from embedded system is more difficult than deployment of software from other domains, which make detection of defects more important. Furthermore, embedded system software are generally implemented in low level programming languages, such as C, which limit the debugging and tracing abilities during testing phase. This limitation not only increases the testing efforts required to ensure quality of such software but also makes it harder to detect the defects by manual inspection since it would require exhaustive inspection of entire code. Thus it can be concluded that if high pd rates are assured by the defect predictor embedded system

domain software, then high pf rates can be acceptable.

Another measure that is used to compare model performances is *f-measure* as used by Koru in his study [31]. F-measure is used in comparing performances of information retrieval and statistical classification methods in terms of effectiveness of the method. In order to calculate f-measure, *precision* and *recall* values of the model must be calculated. Precision, from defect prediction perspective, is defined as the ratio of the number of data samples that are classified as defective to the number of all defectively predicted data samples. Having high precision values means that modules that are predicted to be defective are more likely to be defective than non-defective, thus higher precision values are expected. Recall is defined as the ratio of correctly classified defective data samples among all correct classifications. Equations for precision and recall are provided as follows [15]:

$$precision = \frac{tp}{tp + fp} \tag{6.4}$$

$$recall = \frac{tp}{tp + fn} \tag{6.5}$$

F-measure value can be calculated using precision and recall values from the following Equation [15]:

$$f - measure = \frac{2 * precision * recall}{precision + recall} \tag{6.6}$$

Rather than evaluating the performance in terms of precision and recall, f-measure is evaluated since it takes into account both recall and precision. Having high precision and high recall values means that the model can classify defective data samples correctly, which is an expected behavior. High precision and recall values will eventually lead to high f-measure values, thus it should be noted that when evaluating the performance higher f-measures will represent better prediction performance.

In addition to the performance analysis on *pd*, *pf*, *balance*, and *f-measure* criteria defined above, cost-benefit analysis of the proposed models is also performed. Cost-benefit analysis is mainly used in monetary systems in order to compare the value of a study with its expected benefits [35]. Decision makers may choose to assent negative effects of taking a decision if they think its benefits will be of much higher importance for their interests. Enabling this additional information is a major advantage of cost-benefit analysis for decision makers.

Arisholm and Briand performed cost-benefit analysis of defect prediction of fault prone software systems in their study [19]. In the reference study, the authors proposed a logistic regression model to detect fault proneness of a software product using class level data extracted from a Telecom Software implemented in Java programming language [20]. They collected class level source code metrics using two different source code analyzers implemented in Java programming language, XRadar and JHawk. Extracted metrics include structural information on classes like coupling between classes, class characteristics like type and size of the class, as well as change history for faults.

In a previous study, Briand and Wuest addressed an important issue relating the performance of defect predictors [21] such that there was little evidence that a defect predictor can have direct economical effect for a company. Arisholm and Briand inspected this statement and found out that common performance evaluation strategies like confusion matrix do not directly relate the cost-effectiveness of a model [19]. The authors reached to a similar conclusion from the results of experiments that they performed. Their results revealed that their model can predict more than 70% of the defects in less than 30% of classes, which seem to be a promising result. However, future analysis of the results has shown that those 30% of all classes contain 50% of all source code. Thus they claim that using size of defectively predicted source code for assessing the performance of predictor is a better measure since it reflects verification cost of the prediction more precisely.

Tosun also performed cost-benefit analysis of defect predictors in her study [18]. In the reference study, three different defect prediction techniques are used to perform

defect prediction and an ensemble is implemented which combines results of the three models to provide the final prediction such that a module is predicted as defective only if majority of algorithms predict that module as defective. Performance evaluation of the model is performed by comparing with pure Naïve Bayes Classifier. Authors also performed cost-benefit analysis for the results they obtained by computing the decreases in the verification effort. They compared the size of source code that would be inspected by random selection of classes, with the size of source code that the ensemble marked for inspection to determine the change in verification effort and used the term "gained efficiency" to reflect this change rate. They evaluated the model on three datasets from SoftLab data repository, namely AR3, AR4 and AR5 and obtained better pd values for each dataset. However when the model's costs is taken into account it is seen that gained efficiency for AR4 dataset is better for Naïve Bayes Classifier than for the ensemble of classifiers.

The ability to make observations like described in the previous paragraph lets decision makers to choose which classifier fits better to the domain of interest. Consider the case of AR4 dataset explained above: If the domain of interest requires detection of defects with a high probability and verification costs is not the main consideration, then a project manager might want to use ensemble of classifiers rather than Naïve Bayes Classifier. However if the verification cost is more important than defect detection rate, than a project manager might choose pure Naïve Bayes Classifier in order to decrease the time/effort spent on verification of source code. Without cost-benefit analysis, this kind of information would not be available to the project managers.

Figure 6.1 represents the plot of gained efficiency and probability of detection (pd) which might help explain cost-benefit analysis easier. Plot area can be divided into three types of regions namely desired region, inefficient region and beware of region as shown in the plot. Locations of intersection lines that separate these regions can not be generalized since importance of prediction performance and verification effort can change from domain to domain as explained in previous paragraph. Once the intersection lines are determined for a specific domain, cost-benefit analysis plot enables graphical interpretation of the results. X-axes of the plot represent the pd

values ranging from 0 to 1 and y-axes of the plot represent gained efficiency values ranging from 0 to 1.

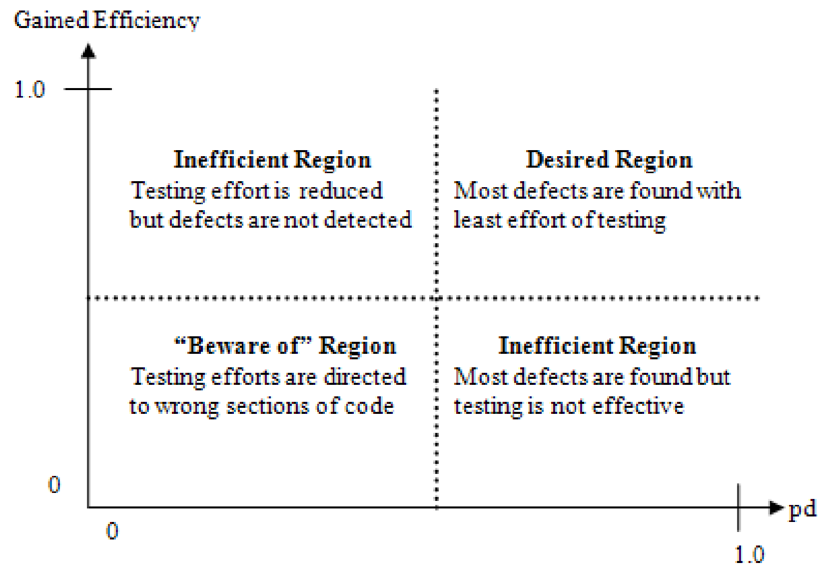

Figure 6.1. Cost-benefit Analysis Plot

Upper-right region of the plot can be defined as "desired region" since results falling in this region means the defect prediction model provided high gain in verification costs as well as high pd values, which means that defects in the system are addressed with high probability while keeping the defectively predicted source code size small. Such a defect prediction model can be helpful in decreasing the verification efforts necessary to test the software while increasing the quality of the resulting product by making it less erroneous.

Upper-left and lower-right regions of the plot are defined as inefficient regions, since results falling in these regions lack from either pd or gained efficiency criteria. According to the inefficient region of plot that a result falls, corrective actions might be planned by the project managers while planning their verification or testing phase. For example consider a case where the defect prediction model provided 40% pd rate and 85% gained efficiency rate, then a project manager might think that 40% is a low pd rate and will not be suitable for the domain of the project. Thus he can decide to widen the code coverage to be verified which will in turn decrease the verification effort decrease provided by the defect prediction model.

Finally, lower-left region of the plot can be defined as "beware of" region since results falling in this regions lacks from both gained efficiency and pd criteria, which means that the defect prediction model addressed much of the source code to be verified but the addressed source code contains less of the defects. In this case, using results of the defect prediction model will eventually lead the testing team to focus on wrong sections of code and will most likely waste more resources like time and man-power than it will save.

In this research, results for cost-benefit analysis will be provided in the format used in [18, 19]. In the reference studies, the authors defined several measurements to address cost-benefit of the system. Measurements that are used and their explanations are given in Table 6.2.

Table 6.2. Measurements used in cost-benefit analysis

Measurement	Abbr	Explanation
totalLOC	TL	Total LOC of project
neededLOC	NL	Number of LOC required for random selection of classes
inspectedLOC	IL	Number of LOC inspected by the model
changeLOC	CL	Decrease in total LOC to be inspected
Gained efficiency	GE	Decrease in verification effort to predict defective modules

Values from Table 6.2 will be calculated for both module level prediction model and source file level prediction model and according to their probability of defect detection and gained efficiency values, a conclusion about which predictor is better will be drawn. In order to make the tables simpler, abbreviations for each measurement will be used in tables.

TotalLOC and inspectedLOC measurements are calculated directly from the source code metrics or from the output of the model. Remaining measurements need to be calculated from other measurements and equations that are used to calculate those values are provided below.

$$neededLOC = totalLOC * pd \tag{6.7}$$

$$changeLOC = \frac{totalLOC - inspectedLOC}{totalLOC} \quad (6.8)$$

$$Gainedefficiency = \frac{neededLOC - inspectedLOC}{neededLOC} \quad (6.9)$$

6.2. Performance

The criteria used to measure the performance are already defined in previous section. In this section, the results of the proposed model when evaluated with the tests defined in chapter 5 will be presented. For each data repository, performance results of both models and cost-benefit analysis of the results will be provided consecutively.

6.2.1. Model Validation

First step to evaluate the performance of the predictor is to compare the proposed model's performance with the results of existing models in the literature. As explained in chapter 4, defect prediction on high levels of source code structures is proposed in this thesis study, thus the proposed model will be compared to existing models in the literature which perform module level defect prediction model [2, 3, 7, 26, 30, 31]. Performance evaluations of classifiers for different levels of granularity are compared in terms of pd, pf, and balance criteria which are also used for model validation in the given reference studies. F-measure criteria will then be used to compare different classifiers' performances.

The results presented in this subsection are extracted from experiments using K-fold cross validation. For module level prediction the value of k is taken as 10, in order to have similar results of those from the reference study. However the same value could not be used for source file level prediction for some datasets since the datasets did not contain enough number of defective and non-defective source files to be divided into 10 folds. Thus when designing experiments with source files, the value of k is taken to

be 2 for Nasa and SoftLab datasets, which lets construction of random training and validation datasets (in terms of ratios of defective and non-defective data samples).

As explained in section 5.2, in order to validate the Naïve Bayes Classifier against the existing studies in the literature, the problems with the dataset is omitted and SLOC metrics are also used when training and validating the model. Thus, all 5 datasets from NASA repository, namely CM1, MC1, PC1, PC2 and PC3, are evaluated at this step.

All comparisons between module level prediction and source file level prediction will be provided in the same format but since each data repository contains datasets having different characteristics, their results will be presented separately. In order to provide a clear understanding, first some information regarding the table structure is provided. Rows of the table contain results for a single dataset (product) and columns of the table contain results for different levels of predictions. Evaluations of predictors will be performed on pd, pf and balance criteria and minimum, maximum and average values of each criteria are also provided for each product as min, max and avg rows, respectively. The numbers written in green represent better prediction values achieved by the proposed algorithm and the numbers in red represent worse prediction values achieved by the proposed algorithm.

According to Table 6.3, source file level prediction has a positive impact on probability of defect detection and a negative effect on probability of false alarms. On the overall, it is seen that the proposed model can provide better balance values for 2 of the datasets and worse results on the remaining 3 datasets. T-test results verify that all pd values and pf values from Table 6.3 are statistically significant. On the other hand, t-test results show that balance results are not statistically significant except for CM1 dataset. Thus it can be stated that source file level prediction can provide significantly better pd values with the cost of worse pf values, but on the overall the balance values will be similar for both of the models. Obtaining similar results on balance criteria lets a project manager to determine which criteria (either pd or pf) is more important for the domain of interest and decide which model to use for defect prediction.

Table 6.3. Module level vs source file level NB results (NASA with SLOC metrics)

		Module Level Naïve Bayes			Source File Level Naïve Bayes		
		pd	pf	bal	pd	pf	bal
CM1	min	0.20	0.18	0.37	0.20	0.17	0.37
	max	1.00	0.51	0.86	1.00	0.80	0.76
	avg	0.73	0.34	0.67	0.57	0.38	0.56
MC1	min	0.29	0.15	0.48	0.85	0.06	0.67
	max	1.00	0.20	0.90	1.00	0.44	0.93
	avg	0.81	0.18	0.80	**0.94**	0.23	**0.83**
PC1	min	0.25	0.21	0.44	0.56	0.17	0.40
	max	1.00	0.39	0.81	1.00	0.83	0.79
	avg	0.66	0.30	0.66	**0.82**	0.44	0.63
PC2	min	0.00	0.12	0.28	0.25	0.00	0.12
	max	1.00	0.21	0.92	1.00	1.00	1.00
	avg	0.73	0.16	0.74	**0.96**	0.35	**0.74**
PC3	min	0.50	0.24	0.58	0.88	0.17	0.40
	max	1.00	0.42	0.82	1.00	0.83	0.88
	avg	0.78	0.31	0.72	**0.94**	0.42	0.70

In order to have a better understanding of the results, cost-benefit analysis results are provided in Table 6.4:

Table 6.4. Cost-benefit analysis of NB on NASA datasets

	CM1		PC1	
	Module Level	S. File Level	Module Level	S. File Level
TL	16903	16903	25922	25922
pd	0.73	0.57	0.66	0.82
pf	0.34	0.38	0.30	0.44
balance	0.67	0.56	0.67	0.63
NL	12283	9635	17206	21242
IL	1303	5205	1849	11986
CL (%)	0.92	0.69	0.92	0.54
GE (%)	0.89	0.46	0.89	0.44

Cost-benefit analysis results show that module level prediction can provide an improvement of 89% on efficiency which is a relatively high rate compared to source file level prediction's 46% improvement. Considering that source file level prediction provided worse pd, pf and balance values it is concluded that module level prediction has clearly a better performance than source file level prediction for CM1 dataset.

Even though source file level prediction can detect defects in PC1 dataset with significantly better performance, cost-benefit analysis results show that it can not decrease the effort required for testing the software as much as module level prediction. Module level prediction can decrease the effort down to 11% while source file level prediction can decrease the effort only down to 56% which causes source code predictor to examine over 10K lines of codes more than module level predictor.

As explained in subsection 5.2.2, SLOC measurements of some projects within NASA dataset are not accurate, thus it is not safe to use these values for defect prediction purposes. Even though these metrics are error prone, they were used in the previous experiments since main purpose was to validate the model with existing models. In order to inspect the effect of erroneous SLOC measurements to the proposed model, a second experiment is performed. In the second experiment, the same datasets are used for testing the model, but their SLOC metrics are eliminated.

Table 6.5. Module level vs source file level NB results (NASA without SLOC metrics)

		Module Level Naïve Bayes			Source File Level Naïve Bayes		
		pd	pf	bal	pd	pf	bal
CM1	min	0.20	0.16	0.33	0.00	0.00	0.25
	max	1.00	0.58	0.86	1.00	1.00	0.80
	avg	0.72	0.34	0.66	0.57	0.45	0.51
MC1	min	0.43	0.13	0.58	0.85	0.00	0.67
	max	1.00	0.20	0.89	1.00	0.47	0.93
	avg	0.72	0.17	0.76	**0.95**	0.24	**0.82**
PC1	min	0.13	0.21	0.34	0.44	0.17	0.29
	max	1.00	0.42	0.84	1.00	1.00	0.80
	avg	0.65	0.31	0.64	**0.83**	0.45	**0.64**
PC2	min	0.00	0.13	0.28	0.50	0.00	0.29
	max	1.00	0.22	0.91	1.00	1.00	1.00
	avg	0.73	0.16	0.75	**0.95**	0.35	0.72
PC3	min	0.44	0.26	0.54	0.88	0.17	0.29
	max	0.94	0.45	0.79	1.00	1.00	1.00
	avg	0.73	0.33	0.69	**0.94**	0.48	0.65

Comparing results from Table 6.3 and Table 6.5, it is seen that performances of both predictors did not change significantly for both of the predictors and even in some cases it is observed that the prediction performance decreases when SLOC metrics are

removed (CM1 and PC3). These results show that problematic SLOC measurements did not affect the performance of the model much in the first place and it is not possible to claim that they have a negative impact on the performance of the predictor.

Results of the second experiment must also be evaluated in conjunction with t-test results. T-test results show us that, as in previous experiment, all pd and pf results are statistically significant at level. Thus we can conclude that source file level prediction provides improvements in defect prediction probability but will also suffer from increasing false alarms for NASA datasets. T-test results also reveal that balance values are statistically significant for only MC1 and PC3 datasets. We can see from Table 6.5 that source file level prediction provided better balance result for MC1 dataset and worse balance result for PC3 dataset, which again makes it impossible to draw a strict conclusion on the quality of either defect prediction models when balance criteria is considered.

Naïve Bayes Classifier is also evaluated using SoftLab repository datasets for both module level and source file level predictions and results are presented in the following table.

Table 6.6. Module level vs source file level NB results (SoftLab)

		Module Level Naïve Bayes			Source File Level Naïve Bayes		
		pd	pf	bal	pd	pf	Bal
AR3	min	0.50	0.14	0.57	0.00	0.00	0.00
	max	1.00	0.37	0.82	1.00	1.00	0.76
	avg	0.75	0.29	0.71	0.45	0.59	0.35
AR4	min	0.50	0.18	0.60	0.20	0.00	0.29
	max	0.90	0.35	0.81	1.00	1.00	1.00
	avg	0.72	0.27	0.71	**0.82**	0.40	0.60
AR5	min	0.75	0.00	0.70	0.00	0.00	0.00
	max	1.00	0.43	0.90	1.00	1.00	1.00
	avg	0.88	0.21	0.80	0.75	0.48	0.55

The results of Naïve Bayes Classifier on source file level turned out to perform worse than on module level for SoftLab repository datasets. Not only the probability of false alarm rates are worse for source file level prediction, as it was the case for NASA

repository datasets, but also probability of detection and balance values are always worse than or equal to the module level prediction. These results are also tested by t-test and it turned out that all pd, pf and balance values are statistically significant.

Cross-referencing the results with structure of the datasets (Table 5.2), it is observed that source file level prediction's performance decreases substantially for those datasets which are relatively small in size. AR3 contains only 8 source files, 3 of them being defective and AR5 contains only 7 source files, 3 of them being defective. The model's defect detection rates decrease substantially for these two datasets, from 0.75 down to 0.45 for AR3 and from 0.88 down to 0.75 for AR5 dataset. On the other hand, the defect detection performance for AR4 increases from 0.72 up to 0.82, as it would be expected after observing results for NASA datasets. Considering that K-fold cross validation is performed 10 times and the results being statistically examined, we claim that number of source files is an important parameter that affects the performance of Naïve Bayes Classifier.

Before proceeding to Eclipse data repository results, cost-benefit analysis for SoftLab datasets is provided.

Table 6.7. Cost-benefit analysis of NB on SoftLab datasets

	AR3		AR4		AR5	
	ML	SFL	ML	SFL	ML	SFL
TL	5624	5624	9196	9196	2732	2732
pd	0.75	0.45	0.72	0.82	0.88	0.75
pf	0.29	0.59	0.27	0.40	0.21	0.48
balance	0.71	0.35	0.71	0.60	0.80	0.55
NL	4218	2531	6576	7511	2391	2049
IL	2130	1790	3227	3994	1042	924
CL (%)	0.62	**0.68**	0.65	0.57	0.62	**0.66**
GE (%)	0.50	0.29	0.51	0.47	0.56	**0.56**

Cost-benefit analysis results show that source file level prediction can decrease the number of lines of code to be inspected (CL row values of the table), however these results do not mean that it is a better alternative since prediction performances

of the model also decreases significantly. For AR3 dataset, source file level prediction addresses 1790 lines of code as defective, but inspecting that much of source code would lead to 45% of all defects. On the other hand, module level prediction addresses 2130 lines of code as defective and 75% of all defects will be covered during the testing. So it is seen that module level prediction can improve the prediction performance by 30% while increasing the code coverage by only 16%.

A similar analysis for AR4 dataset shows that source file level prediction will lead to detection of 82% of all defects within 3994 lines of code inspection area, while module level prediction will lead to 72% of defects within 3227 lines of code. In this case, there is a trade-off between inspection size and defect detection rate. If domain of interest requires immediate completion of verification phase, then module level prediction will be useful since it addresses 19% less source code as defective. But if domain of interest requires high confidence of final product being less defective, then source file level prediction will be a better alternative since it can address additional 10% of defects by addressing additional 700 lines of code as defective.

Results presented in 6.8 are consistent with results from NASA datasets in 6.3. Source file level prediction provides slightly better defect detection probability rates for most of the datasets, half of them being statistically insignificant. On the other hand, source file level prediction provides worse false alarm rates than module level prediction on all datasets and all of these values are statistically significantly. As a result, most of the balance results are significantly worse for source file level prediction than for module level prediction. Nevertheless, it is seen that the decrease in the rate of balance values are not as high as decrease rates for SoftLab dataset results. Thus we can claim that this is a supporting factor on our hypothesis which states that in order to have better performance on balance criteria, larger datasets are required.

Cost-benefit analysis for eclipse data repository (see Table 6.9) show that source file level prediction provides worse gained efficiency values for all of the datasets (worse by 2% - 4%). Recall from Table 6.8 that defect prediction probabilities are better for

Table 6.8. Module level vs source file level NB results (eclipse)

		Module Level Naïve Bayes			Source File Level Naïve Bayes		
		pd	pf	bal	pd	pf	Bal
jdi	min	0.75	0.06	0.79	1.00	0.00	0.61
	max	1.00	0.19	0.96	1.00	0.55	1.00
	avg	0.98	0.13	0.90	**1.00**	0.22	0.84
jdt_apt	min	0.75	0.10	0.78	0.50	0.00	0.38
	max	1.00	0.27	0.93	1.00	0.88	1.00
	avg	0.98	0.16	0.88	0.96	0.23	0.82
jdt_common	min	0.75	0.05	0.78	0.50	0.00	0.29
	max	1.00	0.31	0.96	1.00	1.00	1.00
	avg	0.98	0.17	0.87	0.95	0.32	0.75
jdt_core	min	0.93	0.00	0.92	1.00	0.00	0.84
	max	1.00	0.12	0.96	1.00	0.22	1.00
	avg	0.99	0.08	0.94	**1.00**	0.09	**0.94**
jdt_internal	min	0.94	0.11	0.90	0.93	0.14	0.81
	max	0.99	0.14	0.92	1.00	0.27	0.90
	avg	0.97	0.12	0.91	**0.97**	0.21	0.85
jdt_ui	min	0.67	0.07	0.75	0.67	0.00	0.71
	max	1.00	0.18	0.95	1.00	0.42	1.00
	avg	0.97	0.12	0.90	**0.97**	0.14	0.88
jface	min	0.91	0.07	0.89	0.82	0.11	0.65
	max	1.00	0.14	0.94	1.00	0.50	0.93
	avg	0.99	0.11	0.92	0.98	0.26	0.81
pde	min	0.87	0.11	0.87	0.84	0.16	0.74
	max	1.00	0.16	0.91	1.00	0.36	0.89
	avg	0.96	0.14	0.90	**0.98**	0.25	0.82
swt	min	0.97	0.10	0.88	1.00	0.00	0.68
	max	1.00	0.16	0.93	1.00	0.45	1.00
	avg	0.99	0.14	0.90	**1.00**	0.19	0.86
team	min	0.88	0.11	0.87	0.90	0.17	0.71
	max	1.00	0.15	0.92	1.00	0.41	0.88
	avg	0.97	0.13	0.90	**0.99**	0.29	0.80
ui	min	0.91	0.11	0.88	0.91	0.14	0.78
	max	1.00	0.15	0.92	1.00	0.31	0.90
	avg	0.96	0.13	0.90	**0.97**	0.22	0.84

Table 6.9. Cost-benefit analysis of NB on eclipse datasets

	jdi		jdt_apt		jdt_common		jdt_core	
	ML	SFL	ML	SFL	ML	SFL	ML	SFL
TL	14196	14196	13858	13858	9687	9687	37521	37521
pd	0.98	1.00	0.98	0.96	0.98	0.95	0.99	1.00
pf	0.13	0.22	0.16	0.23	0.17	0.32	0.08	0.09
balance	0.90	0.84	0.88	0.82	0.87	0.75	0.94	0.94
NL	13853	14196	13557	13350	9472	9203	37253	37521
IL	784	1256	771	1093	584	884	1829	2611
CL (%)	0.94	0.91	0.94	0.92	0.94	0.91	0.95	0.93
GE (%)	0.94	0.91	0.94	0.92	0.94	0.90	0.95	0.93

	jdt_internal		jdt_ui		jface		pde	
	ML	SFL	ML	SFL	ML	SFL	ML	SFL
TL	584727	584727	21423	21423	79860	79860	193351	193351
Pd	0.97	0.97	0.97	0.97	0.99	0.98	0.96	0.98
Pf	0.12	0.21	0.12	0.14	0.11	0.26	0.14	0.25
balance	0.91	0.85	0.90	0.88	0.92	0.81	0.90	0.82
NL	565879	568933	20733	20709	78674	78336	185076	188596
IL	33684	51670	981	1541	3740	6660	9591	15625
CL (%)	0.94	0.91	0.95	0.93	0.95	0.92	0.95	0.92
GE (%)	0.94	0.91	0.95	0.93	0.95	0.91	0.95	0.92

	swt		team		ui	
	ML	SFL	ML	SFL	ML	SFL
TL	68585	68585	115282	115282	306513	306513
pd	0.99	1.00	0.97	0.99	0.96	0.97
pf	0.14	0.19	0.13	0.29	0.13	0.22
balance	0.90	0.86	0.90	0.80	0.90	0.84
NL	68191	68595	111697	113624	294272	298120
IL	4458	6521	5498	9316	15795	25195
CL (%)	0.94	0.90	0.95	0.92	0.95	0.92
GE (%)	0.93	0.90	0.95	0.92	0.95	0.92

source file level prediction, but according to cost-benefit analysis amount of source code that needs to be validated is smaller when module level prediction is used. Thus there is a trade-off between detection rate and verification size and according to domain of interest either of the models can be useful.

In order to demonstrate how to analyze the results of cost-benefit analysis, let us consider jdi and ui datasets. Probability of detection for jdi dataset on source file level is significantly better than on module level, but amount of code that source file level predictor addresses as defective is also higher than module level predictor. Thus a project manager will have to make a decision between detecting additional 2% defects or additional 500 lines of code inspection. Considering that 500 lines of code will probably not require so much testing effort then the choice will be beneath source file level predictor. For the ui dataset, source file level prediction provides significantly better pd value (at $\alpha = 0.05$) but this time the additional amount of source code to be validated is 10K (increased by 66%). Decision to be made at this point will be, is detecting 1% of defects worth validating additional 10K lines of source code. If domain of interest is not one that requires zero defect product to be resulted, then apparently the choice will be beneath module level predictor.

6.2.2. Decision Tree Classifier

Experiments with decision tree classifier are performed to inspect the effect of performing defect prediction on different granularity levels for a second type of machine learning algorithm. As in case of Naïve Bayes classifier experiments, K-fold cross validation is performed for both module and source file level experiments where for module level prediction and for source file level prediction.

For decision tree classifier, E_{max} criterion also needs to be optimized as explained in Section 2.4. The value of E_{max} is used when the algorithm decides whether a node is pure enough or not. So E_{max} directly affects the size of the tree and must be optimized. For this purpose, the experiments are repeated for several E_{max} values in the range [0.0002, 0.2] and the value providing the best results are presented in this subsection.

It is observed that optimum value of E_{max} is related to the size of the dataset and the percentage of defective samples in it. E_{max} values used in experiments are presented in the Table 6.10:

Table 6.10. Optimum set of E_{max} values

Datasets	Module Level	Source File Level
CM1	0.15	0.05
MC1	0.05	0.01
PC1	0.15	0.05
PC2	0.01	0.001
PC3	0.10	0.01
AR3	0.05	0.005
AR4	0.05	0.01
AR5	0.10	0.01
Eclipse	0.01	0.001

A common E_{max} value for eclipse data repository is provided based on the observations made. Eclipse datasets are large in size and their defectiveness characteristics are similar to each other, thus their error threshold values turned out to be similar. Values provided in Table 6.10 are obtained after performing several experiments and the optimum value is presented.

Recall that previous section showed that performance of defect prediction for module level and source file level was not affected significantly when SLOC metrics were removed from NASA datasets. Thus when evaluating decision tree model, those metrics are included in the dataset in order to be able to perform cost-benefit analysis as well. Results of experiments with the E_{max} values in Table 6.10 will be provided in separate tables for each data repository.

Table 6.11 presents an overview of results for decision tree classifier for both module level and source file level predictions. Performing defect prediction on source file level provides better pd values for all of the projects and according to t-test results the difference between pd values are significant at $\alpha = 0.05$ level. But, like in our previous experiments, source file level prediction provided significantly worse pf values for most

Table 6.11. Module level vs source file level DT results (NASA)

		Module Level Decision Tree			Source File Level Decision Tree		
		pd	pf	bal	pd	pf	Bal
CM1	min	0.00	0.02	0.29	0.00	0.00	0.00
	max	0.75	0.15	0.82	1.00	1.00	1.00
	avg	0.24	0.08	0.46	**0.44**	0.31	**0.49**
MC1	min	0.21	0.00	0.44	0.50	0.00	0.65
	max	0.75	0.01	0.88	1.00	0.50	1.00
	avg	0.52	0.00	0.66	**0.92**	0.14	**0.85**
PC1	min	0.19	0.02	0.42	0.22	0.00	0.35
	max	0.75	0.09	0.82	1.00	0.83	0.75
	avg	0.40	0.05	0.58	**0.67**	0.52	0.55
PC2	min	0.00	0.00	0.29	0.75	0.00	0.82
	max	0.67	0.01	0.76	1.00	0.00	1.00
	avg	0.10	0.00	0.37	**0.81**	**0.00**	**0.87**
PC3	min	0.13	0.03	0.38	0.50	0.00	0.41
	max	0.63	0.14	0.73	1.00	0.83	0.86
	avg	0.35	0.07	0.54	**0.80**	0.36	**0.68**

of the datasets. Table 6.10 also shows that source file level prediction provides better balance rates for most of the datasets and according to t-test results only two of these results (CM1 and PC1) are not statistically significant. Thus we can conclude that source file level prediction might be a better alternative on NASA datasets especially for cases where false alarm rate is not a main concern.

Table 6.12. Cost-benefit analysis of DT on NASA datasets

	CM1		PC1	
	Module Level	S. File Level	Module Level	S. File Level
TL	16903	16903	25922	25922
pd	0.24	0.44	0.36	0.67
pf	0.08	0.31	0.04	0.52
balance	0.46	0.49	0.54	0.54
NL	3981	7438	9235	17264
IL	655	1770	422	9630
CL (%)	0.96	0.90	0.98	0.63
GE (%)	0.84	0.76	0.95	0.44

Cost benefit analysis results show that even though source file level prediction provides high defect prediction probability, it suffers from gained efficiency criteria. Main reason for having low gain in terms of verification size is the high rates of false alarms of source file level predictor. Source file level predictor will lead to detection

of 20% more defects by addressing additional 1K lines of codes but not all of that additional source code will be defective. Thus a project manager will have to choose whether detecting additional 20% defects is worth tripling the verification size. This behavior is even more crucial for PC1 dataset, because source file level prediction increases the defect prediction probability from 36% up to 67% but it will requires verification of 40 times the code that would be validated if module level predictor was used.

Table 6.13. Module level vs source file level DT results (SoftLab)

		Module Level Decision Tree			Source File Level Decision Tree		
		pd	pf	bal	pd	pf	Bal
AR3	min	0.25	0.00	0.46	0.00	0.00	0.21
	max	0.75	0.19	0.82	1.00	1.00	0.65
	avg	0.45	0.06	0.62	0.43	0.37	0.38
AR4	min	0.10	0.05	0.36	0.83	0.00	0.65
	max	0.80	0.21	0.83	1.00	0.50	1.00
	avg	0.45	0.12	0.60	**0.99**	0.25	**0.82**
AR5	min	0.00	0.00	0.29	0.00	0.00	0.29
	max	1.00	0.29	0.90	1.00	0.50	1.00
	avg	0.46	0.13	0.60	**0.63**	**0.05**	**0.70**

Table 6.13 shows the results for module level and source file level predictions using decision tree classifier. The proposed model provides worse values for all criteria on AR3 dataset, and pf and balance results are significant. Results on AR5 dataset shows that the model's performance is better on all criteria but t-test results reveal that they are not significantly better than module level prediction results. Using this information from the results and information about the datasets from Table 5.2, we can see that decision tree classifier requires larger datasets to work on, in order to provide significantly better values. Results for AR4 support this claim since significantly better defect detection and balance rates are achieved on this dataset. They are also similar to results of NASA datasets in terms of resulting in worse pf rates and NASA datasets are relatively large compared to SoftLab datasets.

Results of NASA and SoftLab data repositories reveal that decision tree classifier can predict the defect with significantly high performance and also can provide

improved balance values if the dataset contains enough samples to let decision tree to learn the patterns in it. Decision tree classifiers can build effective tree structures that correctly identifies defective and non-defective data samples, but when the dataset size is small, as it is in AR3 (8 source files, 3 defective) and AR5 (7 source files, 3 defective), the performance of the model becomes instable and performance may not be improved significantly.

Table 6.14. Cost-benefit analysis of DT on SoftLab datasets

	AR3		AR4		AR5	
	ML	SFL	ML	SFL	ML	SFL
TL	5624	5624	9196	9196	2732	2732
pd	0.45	0.43	0.45	0.99	0.46	0.63
pf	0.06	0.37	0.12	0.25	0.13	0.05
balance	0.60	0.38	0.60	0.82	0.60	0.70
NL	2531	2391	4139	9120	1264	1708
IL	1048	1581	1889	4443	631	843
CL (%)	0.81	0.72	0.79	0.52	0.77	0.69
GE (%)	0.59	0.34	0.54	0.51	0.50	**0.51**

According to cost-benefit analysis results, source file level tends to increase verification size by 2600 LOC and 200 LOC for AR4 and AR5 datasets respectively. With the contrary to this increase, source file level prediction provides detection of additional 44% of defects for AR4 dataset and 17% of defects for AR5 dataset. Thus it seems reasonable to use source file level prediction for these two dataset, since the increases in prediction rates are significant compared to increase in verification size. However it is seen that the pd rate decreases slightly for AR3 dataset and also verification size increases by 500 LOC which means that source file level does not serve well for the defect prediction purposes on AR3 dataset.

Table 6.15 provides very interesting results for performing defect prediction by decision tree classifier on Eclipse dataset. Source file level prediction results show that they are worse according to all criteria for most of the datasets. Only exceptions for this statement appear in pd result of jdt_common dataset which is not a significant result according to t-test. Except for pd value of pde dataset and pd and balance values

Table 6.15. Module level vs source file level DT results (eclipse)

		Module Level Naïve Bayes			Source File Level Naïve Bayes		
		pd	pf	bal	pd	pf	Bal
jdi	min	0.56	0.00	0.69	0.00	0.00	0.28
	max	1.00	0.01	1.00	1.00	0.18	1.00
	avg	0.97	0.00	0.98	0.74	0.03	0.80
jdt_apt	min	0.44	0.00	0.61	0.00	0.00	0.01
	max	1.00	0.01	1.00	1.00	0.25	1.00
	avg	0.88	0.00	0.91	0.82	0.03	0.84
jdt_common	min	0.78	0.00	0.84	0.50	0.00	0.65
	max	1.00	0.01	1.00	1.00	0.33	1.00
	avg	0.94	0.00	0.96	**0.95**	0.01	0.96
jdt_core	min	0.85	0.00	0.90	0.33	0.00	0.53
	max	1.00	0.01	1.00	1.00	0.11	1.00
	avg	0.97	0.00	0.98	0.90	0.02	0.92
jdt_internal	min	0.99	0.00	0.99	0.92	0.00	0.94
	max	1.00	0.00	1.00	1.00	0.03	1.00
	avg	1.00	0.00	1.00	0.97	0.01	0.98
jdt_ui	min	0.77	0.00	0.84	0.00	0.00	0.29
	max	1.00	0.01	1.00	1.00	0.17	1.00
	avg	0.94	0.00	0.96	0.84	0.03	0.87
jface	min	0.86	0.00	0.90	0.45	0.00	0.61
	max	1.00	0.00	1.00	1.00	0.08	1.00
	avg	0.98	0.00	0.99	0.90	0.02	0.92
pde	min	0.97	0.00	0.99	0.89	0.00	0.92
	max	1.00	0.00	1.00	1.00	0.03	1.00
	avg	0.98	0.00	1.00	0.97	0.01	0.98
swt	min	0.96	0.00	0.97	0.60	0.00	0.71
	max	1.00	0.01	1.00	1.00	0.15	1.00
	avg	0.99	0.00	0.99	0.91	0.04	0.92
team	min	0.94	0.00	0.95	0.81	0.00	0.86
	max	1.00	0.00	1.00	1.00	0.05	1.00
	avg	0.96	0.00	0.97	0.97	0.01	0.97
ui	min	0.98	0.00	0.99	0.89	0.00	0.92
	max	1.00	0.00	1.00	1.00	0.03	1.00
	avg	0.99	0.00	1.00	0.95	0.01	0.96

of team dataset, all other results are statistically significant even though the means are both very high. Significantly worse results for source file level prediction do not mean that source file level prediction is not a good option. Because, it can be observed that the results are very high for both of the defect predictors and not just for module level prediction (e.g. pde dataset). One possible reason for having such rates of detections is summarized in the following paragraph.

One reason that might have caused such high levels of prediction performances can be related to the extraction method of datasets. As explained in sub-section 5.1.1, Eclipse source code metrics are extracted directly from source code of Eclipse project with the help of a tool named Predictive 3. Defect data related to modules are also extracted with the help of the same tool, not from any bug tracking software which means that the defect data is a "prediction" of actual defects. Thus the algorithm that predictive 3 uses to predict the defects in the source code is important since if the tool uses decision tree classifier to predict the defects either, then it might explain high rates of defect detection. Predictive 3 assigns possibilities of being defective for each module and according to those possibilities it addresses the modules as low risk, normal risk and high risk modules. Afterwards it predicts the defectiveness of a module according to its risk factor. Thus it would be possible for our model to capture similar behaviors in the source code and assign similar rates for the modules which would lead similar defect prediction behaviors. In order to prove or disprove this hypothesis, a web research on the tool is performed but since the tool is currently out-of-shell product, necessary information could not be obtained. Naïve Bayes results from Table 6.8 support this claim in that the false alarm rates are not as low as decision tree results and also balance values are not as high as decision tree results. But in order to prove or disprove this claim, additional models must be implemented to work on eclipse datasets.

Cost-benefit results from Table 6.16 show us that both defect prediction models provide similar gained efficiency values (3% difference at most). Also analyzing the table for "Inspected LOC (IL)" row, it can be seen that both models address similar

Table 6.16. Cost-benefit analysis of DT on eclipse datasets

	jdi		jdt_apt		jdt_common		jdt_core	
	ML	SFL	ML	SFL	ML	SFL	ML	SFL
TL	14196	14196	13858	13858	9687	9687	37521	37521
pd	0.97	0.74	0.88	0.82	0.94	0.95	0.97	0.90
pf	0.00	0.03	0.00	0.03	0.00	0.00	0.00	0.02
balance	0.98	0.80	0.91	0.85	0.96	0.96	0.98	0.92
NL	13723	10506	12178	11318	9106	9203	36521	33644
IL	662	871	563	761	543	751	2113	2198
CL (%)	0.95	0.94	0.96	0.95	0.94	0.92	0.94	**0.94**
GE (%)	0.95	0.92	0.96	0.93	0.94	0.92	0.94	0.93

	jdt_internal		jdt_ui		jface		pde	
	ML	SFL	ML	SFL	ML	SFL	ML	SFL
TL	584727	584727	21423	21423	79860	79860	193351	193351
Pd	1.00	0.97	0.94	0.74	0.98	0.90	1.00	0.97
Pf	0.00	0.01	0.00	0.03	0.00	0.02	0.00	0.01
balance	1.00	0.98	0.96	0.87	0.99	0.92	1.00	0.98
NL	583567	568819	20132	17996	78337	71874	192639	188126
IL	41334	44080	894	1211	3079	4650	8117	11084
CL (%)	0.93	0.92	0.96	**0.96**	0.96	0.94	0.96	0.94
GE (%)	0.93	0.92	0.96	0.93	0.96	0.94	0.96	0.94

	swt		team		ui	
	ML	SFL	ML	SFL	ML	SFL
TL	68585	68585	115282	115282	306513	306513
pd	0.99	0.91	0.98	0.97	0.99	0.95
pf	0.00	0.04	0.00	0.01	0.00	0.01
balance	0.99	0.92	0.99	0.97	1.00	0.96
NL	67720	62416	113171	111576	304599	292060
IL	5787	5946	4229	5917	14740	18717
CL (%)	0.92	0.91	0.96	0.95	0.95	0.94
GE (%)	0.91	0.90	0.96	0.95	0.95	0.94

amount of source code as defective for most of the projects. Each of these datasets has similar structures and characteristics in that, they are sub-projects of the same project, implemented with an object-oriented language and developed using the same processes. Having that information and the results in Table 6.16, it is not possible to draw a definite conclusion about which defect prediction model is better under what circumstances. However, it is observed that balance values for source file level prediction did not worsen dramatically and this is a supporting factor for our hypothesis on the number of source file's affecting the balance results of the predictor. This will be further examined in the conclusions section.

6.2.3. Multi-level Prediction

Multi-level defect prediction is defined as performing source file level prediction and module level prediction consecutively. Previous experiments have shown that source file level prediction can provide better pd values but it suffers from higher verification sizes. In order to inspect whether a model that takes advantage of both approaches can be developed, we offer performing both prediction models consecutively.

Experiments on multi-level prediction are performed using the same parameters as defined for module level and source file level predictions. Results of the experiments are compared to the results of source file level prediction in order to inspect whether the performance of the model can be improved by applying module level prediction after source file level prediction.

Table 6.17. Source file level vs multi-level NB results (NASA)

		Source File Level Naïve Bayes			Multi-Level Naïve Bayes		
		pd	pf	bal	pd	pf	bal
CM1	min	0.20	0.17	0.37	0.00	0.00	0.24
	max	1.00	0.80	0.76	1.00	0.75	1.00
	avg	0.57	0.38	0.56	0.32	**0.12**	0.50
PC1	min	0.56	0.17	0.40	0.00	0.00	0.29
	max	1.00	0.83	0.79	0.83	0.22	0.88
	avg	0.82	0.44	0.63	0.33	**0.07**	0.53

Table 6.18. Cost-benefit analysis of NB on NASA datasets

	CM1		PC1	
	Source File Level	Multi-Level	Source File Level	Multi-Level
TL	16903	16903	25922	25922
Pd	0.57	0.32	0.82	0.33
Pf	0.38	0.12	0.44	0.07
balance	0.56	0.50	0.63	0.53
NL	9635	5391	21242	8644
IL	5205	411	11986	941
CL (%)	0.69	**0.98**	0.54	**0.96**
GE (%)	0.46	**0.92**	0.44	**0.89**

Results for NASA datasets show that multi-level Naïve Bayes prediction provides significantly worse defect detection rates. Despite significant decrease in probability of false alarm rates for multi-level prediction, balance values are still significantly worse than module level prediction. On the other hand, cost-benefit analysis results show that multi-level prediction can provide more efficient results on both of the projects. For CM1 dataset, results show that multi-level prediction can predict 25% less defects (57% vs. 32%) by addressing 92% less amount of source code as defective (5205 vs. 411) which also means detecting 32% of all defects by addressing only 411 lines of source code out of 16903 lines of source code. Thus it can be concluded that multi-level prediction can provide higher improvement for the code coverage to be verified than decrease in prediction performance, which makes multi-level prediction a better alternative for CM1 dataset.

Source file level Naïve Bayes can predict 82% of defects for PC1 dataset by addressing 50% of entire source code as defective according to Table 6.18. On the other hand, multi-level Naïve Bayes can narrow down the defectively predicted source code coverage down to 4% with the cost of reduced probability of detection. Multi-level prediction can predict 33% of all defects within 941 lines of codes. Thus it can be concluded that even though the prediction performance of multi-level prediction is not high enough to be trusted on planning the verification phase, it can still be helpful in a quick verification of product, since it can predict each one of three defect by addressing a relatively small amount of source code.

Table 6.19. Source file level vs multi-level NB results (Eclipse)

		Source File Level Naïve Bayes			Multi-Level Naïve Bayes		
		pd	pf	bal	pd	pf	Bal
jdi	min	1.00	0.00	0.61	0.91	0.07	0.82
	max	1.00	0.55	1.00	1.00	0.25	0.94
	avg	1.00	0.22	0.84	0.99	**0.15**	**0.89**
jdt_apt	min	0.50	0.00	0.38	0.75	0.10	0.75
	max	1.00	0.88	1.00	1.00	0.33	0.93
	avg	0.96	0.23	0.82	**0.97**	**0.19**	**0.86**
jdt_common	min	0.50	0.00	0.29	0.00	0.10	0.79
	max	1.00	1.00	1.00	1.00	0.30	0.93
	avg	0.95	0.32	0.75	**0.97**	**0.19**	**0.86**
jdt_core	min	1.00	0.00	0.84	0.95	0.07	0.83
	max	1.00	0.22	1.00	1.00	0.23	0.95
	avg	1.00	0.09	0.94	0.99	0.13	0.91
jdt_internal	min	0.93	0.14	0.81	0.96	0.13	0.88
	max	1.00	0.27	0.90	0.99	0.17	0.91
	avg	0.97	0.21	0.85	**0.97**	**0.14**	**0.90**
jdt_ui	min	0.67	0.00	0.71	0.00	0.07	0.83
	max	1.00	0.42	1.00	1.00	0.24	0.94
	avg	0.97	0.14	0.88	**0.97**	0.15	**0.89**
jface	min	0.82	0.11	0.65	0.91	0.10	0.88
	max	1.00	0.50	0.93	1.00	0.17	0.93
	avg	0.98	0.26	0.81	**0.98**	**0.13**	**0.91**
pde	min	0.84	0.16	0.74	0.92	0.14	0.86
	max	1.00	0.36	0.89	1.00	0.19	1.90
	avg	0.98	0.25	0.82	0.97	**0.16**	**0.88**
swt	min	1.00	0.00	0.68	0.97	0.10	0.87
	max	1.00	0.45	1.00	1.00	0.19	0.93
	avg	1.00	0.19	0.86	0.99	**0.14**	**0.90**
team	min	0.90	0.17	0.71	0.92	0.12	0.87
	max	1.00	0.41	0.88	1.00	0.18	0.91
	avg	0.99	0.29	0.80	0.97	**0.15**	**0.89**
ui	min	0.91	0.14	0.78	0.93	0.14	0.87
	max	1.00	0.31	0.90	1.00	0.18	0.90
	avg	0.97	0.22	0.84	**0.97**	**0.16**	**0.89**

According to Table 6.19, performing module level Naïve Bayes after source file level Naïve Bayes provides significant improvements on the performance of the predictor. Defect detection probabilities are similar for both models, neither better values nor worse values are statistically significant than each other. However considering probability of false alarm and balance criteria shows that multi-level prediction has clearly a better prediction performance. T-test results reveal that both pf and balance values are statistically significant (the only exception is jdt_ui dataset). Thus it can be concluded that performance of source file level Naïve Bayes can be improved in terms of performance criteria by applying module level Naïve Bayes on the results of source file level prediction. In order to inspect the effect of this process, cost-benefit analysis will also be performed.

Even though results presented in Table 6.19 has shown that multi-level prediction can provide better performance values, cost-benefit analysis shows that using multi-level prediction will result in increase in verification efforts. Considering that probability of defect detection rates are similar for both models, increase in inspected lines of code values for multi-level prediction model means that it will cause more source code to be analyzed to detect the same amount of defects. However decrease in false alarm rates shows that the model does not predict high volumes of source code by addressing most of modules as defective. Thus the only logical explanation for the increase in inspected loc is that source file level prediction can predict smaller source files with higher accuracy and can miss few of defects in larger source files which results in higher prediction performance but smaller source code coverage. Thus it can be concluded that multi-level prediction will be a better model for eclipse datasets, and that if source file level predictor will be used then further inspection might be necessary for larger source files.

Table 6.21 shows decision tree results for source file level and multi-level prediction. It is seen that multi-level prediction has a lower performance on both pd and pf criteria and t-test results verify that these differences are statistically significant.

Table 6.20. Cost-benefit analysis of NB on eclipse datasets

	jdi		jdt_apt		jdt_common		jdt_core	
	SFL	MuL	SFL	MuL	SFL	MuL	SFL	MuL
TL	14196	14196	13858	13858	9687	9687	37521	37521
pd	1.00	0.99	0.96	0.97	0.95	0.98	1.00	0.99
pf	0.22	0.15	0.23	0.19	0.32	0.19	0.09	0.13
balance	0.84	0.89	0.82	0.86	0.75	0.86	0.94	0.91
NL	14196	14061	13350	13459	9203	9447	37521	37207
IL	1256	1794	1093	1573	884	1386	2611	4090
CL (%)	0.91	0.87	0.92	0.89	0.91	0.86	0.93	0.89
GE (%)	0.91	0.87	0.92	0.88	0.90	0.85	0.93	0.89

	jdt_internal		jdt_ui		jface		pde	
	SFL	MuL	SFL	MuL	SFL	MuL	SFL	MuL
TL	584727	584727	21423	21423	79860	79860	193351	193351
pd	0.97	0.97	0.97	0.97	0.98	0.98	0.98	0.97
pf	0.21	0.14	0.14	0.15	0.26	0.13	0.25	0.16
balance	0.85	0.90	0.88	0.89	0.81	0.91	0.82	0.88
NL	568933	569545	20709	20762	78336	78531	188596	186662
IL	51670	80284	1541	2117	6660	7890	15625	21510
CL (%)	0.91	0.86	0.93	0.90	0.92	0.90	0.92	0.89
GE (%)	0.91	0.86	0.93	0.90	0.91	0.90	0.92	0.88

	swt		team		ui	
	SFL	MuL	SFL	MuL	SFL	MuL
TL	68585	68585	115282	115282	306513	306513
pd	1.00	0.99	0.99	0.97	0.97	0.97
pf	0.19	0.14	0.29	0.15	0.22	0.16
balance	0.86	0.90	0.80	0.89	0.84	0.89
NL	68595	68192	113624	112110	298120	297002
IL	6521	10806	9316	12143	25195	36739
CL (%)	0.90	0.84	0.92	0.89	0.92	0.88
GE (%)	0.90	0.84	0.92	0.89	0.92	0.88

Table 6.21. Source file level vs multi-level DT results (NASA)

		Source File Level Decision Tree			Multi-Level Decision Tree		
		pd	pf	bal	pd	pf	Bal
CM1	min	0.00	0.00	0.00	0.00	0.00	0.12
	max	1.00	1.00	1.00	1.00	0.75	1.00
	avg	0.44	0.31	0.49	0.33	0.12	**0.51**
PC1	min	0.22	0.00	0.35	0.00	0.00	0.29
	max	1.00	0.83	0.75	0.78	0.28	0.84
	avg	0.67	0.52	0.54	0.32	0.08	0.51

Even though multi-level prediction provides better balance value for CM1 dataset, this value is not significantly better. Thus it is concluded that multi-level decision tree will clearly provide worse performance on defect prediction point of view.

Table 6.22. Cost-benefit analysis of DT on NASA datasets

	CM1		PC1	
	Source File Level	Multi-Level	Source File Level	Multi-Level
TL	16903	16903	25922	25922
pd	0.44	0.33	0.67	0.32
pf	0.31	0.12	0.52	0.08
balance	0.49	0.51	0.54	0.51
NL	7438	5642	17264	8323
IL	1770	481	9630	1030
CL (%)	0.90	**0.97**	0.63	**0.96**
GE (%)	0.76	**0.91**	0.44	**0.88**

Cost benefit analysis of decision tree for source file level and multi-level models show that multi-level prediction can provide more efficient results in terms of verification effort point of view. CM1 results reveal that multi-level prediction can address 33% of defects by addressing only 481 lines of code which is 9% of entire source code. On the other hand, source file level prediction can predict 44% of defects by addressing 1770 lines of codes which is 24% of entire source code. Thus the ratio of number of defects covered to the effort to be spent on verification of that coverage is higher for multi-level prediction which means it can be used for a quick verification and testing phase.

PC1 results also show that multi-level prediction can provide significant improvement in inspected lines of code criteria with the cost of decreased pd values. However it is seen that multi-level prediction can decrease total lines of code to be verified from 9630 down to 1030 which decreases the verification effort by 89% and the probability of prediction decreases by only 50% (from 67% down to 32%). Thus it can be concluded that multi-level decision tree classifier can be useful in order to plan a quick and efficient testing phase, but it should be noted that the probability of detection will be low and further analysis will be necessary to detect all defects.

Table 6.23. Source file level vs multi-level DT results (eclipse)

		Source File Level Decision Tree			Multi-Level Decision Tree		
		pd	pf	bal	pd	pf	Bal
jdi	min	0.00	0.00	0.28	0.33	0.00	0.53
	max	1.00	0.18	1.00	1.00	0.05	1.00
	avg	0.74	0.03	0.80	**0.85**	**0.01**	**0.89**
jdt_apt	min	0.00	0.00	0.01	0.00	0.00	0.29
	max	1.00	0.25	1.00	1.00	0.33	1.00
	avg	0.82	0.03	0.84	0.77	**0.02**	0.83
jdt_common	min	0.50	0.00	0.65	0.00	0.00	0.29
	max	1.00	0.33	1.00	1.00	0.23	1.00
	avg	0.95	0.01	0.96	0.79	**0.01**	0.85
jdt_core	min	0.33	0.00	0.53	0.20	0.00	0.43
	max	1.00	0.11	1.00	1.00	0.10	1.00
	avg	0.90	0.02	0.92	0.85	**0.01**	0.89
jdt_internal	min	0.92	0.00	0.94	0.93	0.00	0.95
	max	1.00	0.03	1.00	1.00	0.01	1.00
	avg	0.97	0.01	0.98	**0.99**	**0.00**	**0.99**
jdt_ui	min	0.00	0.00	0.29	0.00	0.00	0.29
	max	1.00	0.17	1.00	1.00	0.18	1.00
	avg	0.84	0.03	0.87	0.82	**0.01**	0.86
jface	min	0.45	0.00	0.61	0.33	0.00	0.53
	max	1.00	0.08	1.00	1.00	0.04	1.00
	avg	0.90	0.02	0.92	0.83	**0.01**	0.88
pde	min	0.89	0.00	0.92	0.74	0.00	0.81
	max	1.00	0.03	1.00	1.00	0.02	1.00
	avg	0.97	0.01	0.98	0.94	**0.01**	0.95
swt	min	0.60	0.00	0.71	0.58	0.00	0.71
	max	1.00	0.15	1.00	1.00	0.05	1.00
	avg	0.91	0.04	0.92	**0.91**	**0.01**	**0.93**
team	min	0.81	0.00	0.86	0.55	0.00	0.68
	max	1.00	0.05	1.00	1.00	0.04	1.00
	avg	0.97	0.01	0.97	0.90	**0.01**	0.93
ui	min	0.89	0.00	0.92	0.77	0.00	0.00
	max	1.00	0.03	1.00	1.00	0.02	1.00
	avg	0.95	0.01	0.96	**0.97**	**0.00**	**0.98**

Decision tree results for eclipse dataset show that multi-level prediction provides better probability of detection values for 4 of the datasets and worse results for remaining 7 datasets. Even though probability of false alarm rate values are better for all datasets, balance values are still in parallel with pd values (4 better values and 7 worse values). Since all datasets are from the same domain and have similar structures, it is hard to draw a strict conclusion on the results for this experiment and define what affected the outcome of it. Cost-benefit analysis may provide some useful information on which model is better than the other.

Table 6.24. Cost-benefit analysis of DT on eclipse datasets

	jdi		jdt_apt		jdt_common		jdt_core	
	SFL	MuL	SFL	MuL	SFL	MuL	SFL	MuL
TL	14196	14196	13858	13858	9687	9687	37521	37521
pd	0.74	0.85	0.82	0.77	0.95	0.79	0.90	0.85
pf	0.03	0.01	0.03	0.02	0.00	0.02	0.02	0.01
balance	0.80	0.89	0.85	0.83	0.96	0.85	0.92	0.89
NL	10506	12107	11318	10638	9203	7679	33644	31968
IL	871	728	761	248	751	254	2198	988
CL (%)	0.94	**0.95**	0.95	**0.98**	0.92	**0.97**	0.94	**0.97**
GE (%)	0.92	**0.94**	0.93	**0.98**	0.92	**0.97**	0.93	**0.97**

	jdt_internal		jdt_ui		jface		pde	
	SFL	MuL	SFL	MuL	SFL	MuL	SFL	MuL
TL	584727	584727	21423	21423	79860	79860	193351	193351
Pd	0.97	0.99	0.74	0.80	0.90	0.83	0.97	0.94
Pf	0.01	0.00	0.03	0.01	0.02	0.01	0.01	0.01
balance	0.98	0.99	0.87	0.86	0.92	0.88	0.98	0.95
NL	568819	578047	17996	17224	71874	66643	188126	181213
IL	44080	20503	1211	376	4650	1473	11084	3996
CL (%)	0.92	**0.96**	0.96	**0.98**	0.94	**0.98**	0.94	**0.98**
GE (%)	0.92	**0.96**	0.93	**0.98**	0.94	**0.98**	0.94	**0.98**

	swt		team		ui	
	SFL	MuL	SFL	MuL	SFL	MuL
TL	68585	68585	115282	115282	306513	306513
pd	0.91	0.91	0.97	0.90	0.95	0.97
pf	0.04	0.01	0.01	0.01	0.01	0.00
balance	0.92	0.93	0.97	0.93	0.96	0.98
NL	62416	62336	111576	103855	292060	297607
IL	5946	2822	5917	2068	18717	7238
CL (%)	0.91	**0.96**	0.95	**0.98**	0.94	**0.98**
GE (%)	0.90	**0.95**	0.95	**0.98**	0.94	**0.98**

Cost-benefit analysis of source file level and multi-level prediction models show that multi-level decision tree classifier has a consistently better performance in terms of gained efficiency criteria. For jdi, jdt_internal, jdt_ui, swt and ui datasets, multi-level prediction have better or similar pd performance and even better gained efficiency rates, which means that multi-level prediction can both increase the prediction performance and decrease the verification costs. Thus it is reasonable to choose multi-level prediction for these datasets. For the remaining datasets, multi-level dataset still provides better improvements in terms of gained efficiency criteria than decrease in probability of detection criteria. Consider jdt_common dataset where the decrease in pd value is the highest. Multi-level prediction can predict 79% of all defects within 254 lines of codes in average where source file level prediction can predict additional 16% defects by addressing 497 lines of codes. Thus it can be concluded that even though the probability of detection (or balance values) decrease for multi-level prediction, it can still provide important information that will help in planning testing phase efficiently.

6.3. Discussion of the Results

In order to evaluate the proposed approach, namely source file defect prediction, two different machine learning algorithms in our proposed model are implemented and several experiments are performed. Results of the experiments were neither discouraging nor very promising. It is observed that the proposed approach can improve the performance of predictor especially in terms of prediction performance, but an increase in one aspect generally caused a decrease in another aspect which makes the proposed approach an alternative to existing models, rather than a replacement to them.

Naïve Bayes classifier results have shown that performing source file level prediction on small datasets (containing only tens of source files) produces low probability of detection rates and higher probability of false alarm rates. As the number of source files decrease, the ability of the model to learn defective and non-defective samples diminish and the model fails to provide better performance values. More qualified results are obtained from eclipse dataset experiments, which resulted in increased defect detection rates for most of the projects. However, source file level prediction model

suffered from high false alarm rates which decreased the balance value of the model as well as resulting in increased inspected lines of code values.

Decision tree classifier provided better when performed on source file level for NASA datasets. However results for module level decision tree classifier turned out to be unexpectedly low, especially for PC2 dataset. In order to find the cause of this low probability of detection rates, datasets are inspected against many aspects but none of the inspected aspects turned out to be directly related to performance of the model (See Appendix C). Thus we decided to inspect the only known problem about the datasets, which is the problematic line of code measures. It is observed in Table 6.11 that source file level prediction outperformed module level prediction for all datasets; however the rate of improvement is much more for MC1, PC2 and PC3 datasets, which are known to have erroneous line of code measurements. The improvement in the model's performance is limited for CM1 and PC1 datasets and these datasets have accurate line of code measurements. There seemed to be a relation between the problem with line of code metrics and the decrease in the performance of the model, thus additional experiments are performed on NASA datasets after removing line of code metrics. Results of the additional experiments have shown that the performance of the model decreased by 2% for CM1 dataset, 1% for PC1 dataset and increased by 4% for MC1 dataset and by 1% for PC2 and PC3 datasets. Even though changes in the model performances are not statistically significant, we think that it can be considered as a factor that affects the model performance.

Results for decision tree classifier on eclipse datasets have shown that source file level prediction fails on both improving the performance of the predictor and also decreasing the verification effort. Considering that decrease of probability of detection and balance values are statistically significant, it can be considered that module level prediction outperforms decision tree classifier on these datasets. Eclipse datasets are relatively large in size compared to both NASA and SoftLab datasets. Taking into account that source file level prediction have provided equal performance to module level prediction on jdt_common dataset, which is the smallest dataset in eclipse repository; we can conclude that source file level prediction can be a good defect prediction model

for small datasets but as the size of dataset grows the performance of source file level prediction will tend to be worse than module level prediction.

Our last set of experiments revealed that performing module level prediction after source file level prediction can improve the model's performance either from detection rates perspective or net effect perspective in terms of verification effort or even in both perspectives. We found out in our previous experiments that source file level prediction tends to increase the probability of detection with the cost of increased false alarm rates. Performing module level prediction after source file level prediction turned out to decrease the false alarm rates without decreasing detection rates, especially for bigger datasets. Resulting balance values are higher than source file level prediction balance values as expected. Trade-off for increased balance values appears in the form of increased verification efforts for most of the projects. However considering that verification effort gain rates does not decrease substantially (6% at most) and they are still as high as 90% for most projects, it can be concluded that increase in verification effort can be acceptable in order to detect more defects.

7. CONCLUSIONS AND FUTURE WORK

In this chapter, the conclusions drawn from the thesis study and topics which we think to be open for improvement is addressed.

7.1. Conclusions

In this research, we tried to examine the effect of performing defect prediction on different granularity of source code structures. We implemented different machine learning models to predict defects on module level and source file levels and evaluated the models in both performance criteria and also performed cost-benefit analysis.

Results of this research have revealed that there is a trade-off between the prediction performance of a model and its verification effort gain when the model is used to predict defects using different granularity levels. For most of the projects, whenever the prediction performance of the model increased, we observed a decrease in the verification effort gain and inversely, an increase in verification effort gain resulted in a decrease in prediction performance. A project manager who wants to perform defect prediction needs to consider this relation and must choose which criterion is more important for the projects needs. If the project has mission-critical requirements and needs to be tested thoroughly, then the manager would choose the prediction model providing higher prediction performance.

Comparisons of source file level and module level prediction results have shown that performing defect prediction on higher levels of source code structures have different impacts on smaller datasets and larger datasets. Performing defect prediction on higher levels of structures like source files tends to provide better balance values on NASA and SoftLab datasets than on Eclipse datasets. Even though defect prediction probability rates on source file level prediction is better than module level prediction, since source file level prediction has significantly worse false alarm rates; the overall performance of the model turns out to be worse. Thus we can conclude that when

performing source file level prediction on highly granular structures, a project manager should keep in mind that model will provide relatively high false alarms and hence the total size of source code to be verified will increase.

Considering the problem with source file level prediction explained in previous paragraph, we decided to perform both levels of predictions consecutively. Proposed multi-level prediction model performed defect prediction firstly on less granular structures like source files and high granular structures like modules. Re-evaluating the outcome of source file level prediction with module level prediction resulted in improved prediction and verification gain ratios for most of the projects, especially when Naïve Bayes classifier is used as the machine learning algorithm. Considering that multi-level prediction provided good results, we hope that this approach will be further investigated by other researchers.

7.2. Contributions

This research performed defect prediction on a higher level of software hierarchy, namely source file level and evaluated its performance in terms of performance of the prediction and also provided cost-benefit analysis of the predictor. Similar studies in the literature performed source file level prediction either by using direct attributes for source files, or did not evaluate the performance of the model in terms of its costs and benefits. Thus this study can be accepted as the first study to evaluate source file level defect prediction in both performance and cost-benefit analysis points of view. From this point of view, results of this study can be seen as a basis for comparison for future studies in this area.

This research is the first study in the literature to perform "multi-level defect prediction". When analyzing the effect of granularity on defect prediction performance, we decided to perform prediction on both granularity levels consecutively, which is not performed by any other researchers to the best of our knowledge.

Another contribution of this research is evaluating the performance of two dif-

ferent machine learning classifiers on source file level prediction for SoftLab datasets. SoftLab dataset repository is a relatively new and developing data repository which needed to be evaluated with different machine learning applications and this thesis study evaluated the datasets using Naïve Bayes Classifier and Decision Tree Classifier.

We stated one major challenge of learning based models as collecting data to train and validate the model. During this research we analyzed an open source software product, Eclipse, and extracted source code metrics from its source code. Additionally we used a commercial off-the-shelf product that performs defect prediction to relate defect information with the source code metrics. Both source code metrics and predicted defect information will be available for public use in future researches.

One major contribution of this research is development of a tool which is capable of performing module level and source file level defect prediction using both Naïve Bayes and Decision Tree classifiers. The tool is developed in Java programming language and reached to a size of 10K lines of codes in total which shows a remarkable effort being spent on the development. The tool supports the following operations:

- Data mining module level datasets to create source file level datasets
- Performing Naïve Bayes and Decision Tree classification on both module and source file levels
- Visual representation of results of experiments in both a table format and box plots graphs
- Ability to change model parameters such as n value for n-folding and value for DT at runtime, and to optimize these parameters for experiments
- Providing excel format outputs of results of experiments, as well as graphical objects of box plots for future reference
- Performing statistical tests (t-test) on results of experiments

7.3. Future Work

In section 5.3 we addressed a threat to validity of our experiments for NASA dataset and explained that in order to remove that threat we used 14 additional datasets from 2 different data repositories. However a future work can also be addressed in order to overcome this threat by predicting the SLOC metrics, which are addressed to be faulty, from the values of other correct metrics of the same data sample and SLOC metrics of other correct data samples. If the missing values can be filled up, then MC1, PC2 and PC3 datasets will also be available for evaluating the proposed model. Regarding the fact that NBC provided better performances on these datasets on source file level, achieving this goal would probably provide supporting results for our study.

Implementation of multi-level defect predictors was a contribution of this study, and it turned out that multi-level defect prediction can be helpful especially to improve the performance of source file level Naïve Bayes classifier. A further model based on applying multi-level prediction can be developed which uses both levels of prediction in parallel, not consecutively. Module level defect prediction and source file level prediction can be applied on the same dataset, and results of one of the models can be used to justify results of the other model. Finding a good method for justification of results will be critical in such a method, since it will be the key factor to increase detection probability and decrease false alarm rates.

APPENDIX A: SOFTWARE CODE METRICS

Detailed information on software code metric features that are used in this study is given in this chapter. Further information on these features and their explanations can be found in [2, 4, 8, 9, 63].

SLOC metrics measure the source code's length as lines of codes and they are extracted directly from the source code of the software product. Other source code metrics also measure the source code's structure in many different perspectives. Details of these features can be found in Table A.1. These metrics are also known as Base Metrics since they are not derived from any other metrics.

Other software code metrics are known as composite metrics since they are derived from the base metrics. McCabe's metrics and Halstead's metrics are these types of metrics which are presented in Table A.2.

Table A.1. Base source code metrics

	Base Metrics	Explanation
SLOC	Lines of Comment	Source lines of code that are purely comments
	Lines of Code and Comment	Lines that contain both code and comment.
	Blank Lines	Lines with only white space or no text content.
	Lines of Code	As its name indicates, total number of lines in a module.
	Executable of Lines of Code	Source lines of code that contain only code and white space.
Others	Branch Count	Number of branches in a given module.
	Call Pairs	Number of calls to other functions in a module.
	Condition Count	Number of conditionals in a given module.
	Decision Count	Number of decision points in a given module.
	Edge Count	Number of edges found in a given module.
	Modified Condition Count	Every condition shown to independently affect a decision outcome.
	Multiple Condition Count	Number of multiple conditions that exist within a module.
	Node Count	Number of nodes found in a given module.
	Normalized Cyc. Comp.	*Norm V(g) = V(g) / lines of code*
	Operators	Total number of operators found in a module.
	Operands	Total number of operands found in a module.
	Unique Operators	Number of unique operators found in a module.
	Unique Operands	Number of unique operands found in a module.

Table A.2. Composite source code metrics

	Composite Metrics	Calculation
Halstead	Halstead Vocabulary (n)	*n = number of unique operands + number of unique operators*
	Halstead Length (N)	*N =operands + operators*
	Halstead Volume (V)	*V = N*log(n)*
	Halstead Level (L)	*L = V*/V*
	Halstead Difficulty (D)	*D = 1/L*
	Halstead Programming Effort (E)	*E = V / L*
	Halstead Error Estimate (B)	*B = V / S**
	Halstead Programming Time (T)	*T = E / 18*
McCabe	Cyclomatic Complexity - V(g)	*V(g) = edge count - node count + 2*num. unconnected parts in g*
	Cyclomatic Density - Vd(g)	*V(g) / executable lines of code*
	Decision Density - Dd(g)	*condition count / decision count*

Table A.3. Object-oriented source code metrics

Metric	Explanation
# of Files	The total number of files in a project
# of Classes	The total number of classes in a project
# NTLC	The total number of top level classes in a project
LOC	The total lines of code found in a project
Preprocess	The total number of import statements in a project
Blank	The total number of blank lines (no comment, no code) in a project
Comments	The total number of comments (freestanding or inline comments)
CP	Comment Percentage = Comments / (LOC-Blank)
Exec. Stmts	The total number of executable statements in a project
NOM	The total number of methods in a class definition
WMC	Weighted Methods per Class = sum of cyclomatic complexities in a class
CBO	Coupling Between Objects = number of other classes whose methods or instance variables are used by the methods of this class
RFC	Response for a Class = number of methods in the class + number of methods called by each of these methods, where each called method is counted once

APPENDIX B: DETAILED ANALYSIS OF DATASETS

Table B.1. Detailed dataset analysis - module level (part 1)

Project	# of attr.	# of Modules			Lines of Code			Defects per Module		
		total	def.	non.def	total	def.	non.def	min	max	avg
CM1	38	505	48	457	16903	3277	13626	1	5	1,46
MC1	38	9466	68	9398	66583	3367	63216	1	3	1,16
PC1	38	1107	76	1031	25922	4459	21463	1	9	1,83
PC2	38	5589	23	5566	26863	1057	25806	1	2	1,13
PC3	38	1563	160	1403	36473	6026	30447	1	9	1,62
AR3	29	63	8	55	5624	2170	3454	1	14	4,25
AR4	29	107	20	87	9196	3818	5378	1	2	1,05
AR5	29	36	8	28	2732	1468	1264	1	4	2,13
jdi	25	1393	42	1351	14196	3249	10947	1	4	1,21
jdt_apt	25	1247	43	1204	13858	2861	10997	1	3	1,07
jdt_common	25	797	45	752	9687	2746	6941	1	2	1,09
jdt_internal	25	3947	135	3812	37521	10571	26950	1	5	1,23
jdt_ui	25	43938	2519	41419	584727	206925	377802	1	23	1,27
jface	25	2040	63	1977	21423	4359	17064	1	4	1,11
pde	25	7822	211	7611	79860	15580	64280	1	4	1,15
swt	25	18010	622	17388	193351	40608	152743	1	9	1,10
team	25	4359	349	4010	68585	29054	39531	1	7	1,28
ui	25	11510	321	11189	115282	21218	94064	1	5	1,08

Table B.2. Detailed dataset analysis - source file level (part 1)

Project	# of attr.	# of Source Files			Lines of Code			Defects / Source File		
		total	def.	nondef	total	def.	nondef	min	max	avg
CM1	152	20	9	11	16903	9979	6924	1	17	7,78
MC1	148	57	26	31	37510	35621	1889	1	36	21,23
PC1	152	29	17	12	25922	16335	9587	1	31	8,18
PC2	152	10	8	2	26863	24190	2673	1	10	3,25
PC3	152	29	17	12	36473	20333	16140	1	109	15,24
AR3	145	8	3	5	5624	3319	2305	1	32	11,33
AR4	145	15	11	4	9196	8217	979	1	4	1,91
AR5	145	7	3	4	2732	2037	695	2	12	5,67
jdi	108	133	18	115	14196	9038	5158	1	17	2,83
jdt_apt	108	113	24	89	13858	8100	5758	1	5	1,92
jdt_common	108	57	20	37	9687	7673	2014	1	11	2,45
jdt_internal	108	222	35	187	37521	22326	15195	1	46	4,74
jdt_ui	108	3244	978	2266	584727	442410	142317	1	114	3,27
jface	108	152	30	122	21423	12472	8951	1	11	2,33
pde	108	496	110	386	79860	46236	33624	1	11	2,21
swt	108	1649	370	1279	193351	111236	82115	1	32	1,85
team	108	302	100	202	68585	60387	8198	1	51	4,48
ui	108	962	208	754	115282	58326	56956	1	15	1,67

Table B.3. Detailed dataset analysis - module level (part 2)

Project	Defective Module Size			Nondefective Module Size		
	min	max	avg	min	max	avg
CM1	7	456	68,27	2	503	29,82
MC1	2	639	49,51	0	392	6,73
PC1	3	602	58,67	0	253	20,82
PC2	2	316	45,96	0	663	4,64
PC3	2	165	37,66	0	817	21,70
AR3	18	670	271,25	3	285	62,80
AR4	27	907	190,90	6	530	61,82
AR5	49	477	183,50	5	151	45,14
jdi	39	302	77,36	1	59	8,10
jdt_apt	41	186	66,53	1	75	9,13
jdt_common	38	149	61,02	2	44	9,23
jdt_internal	38	365	78,30	3	68	7,07
jdt_ui	38	1674	82,15	1	97	9,12
jface	38	306	69,19	1	53	8,63
pde	38	272	73,84	1	86	8,45
swt	38	686	65,29	1	77	8,78
team	38	519	83,25	1	89	9,86
ui	38	404	66,10	1	66	8,41

Table B.4. Detailed dataset analysis - source file level (part 2)

Project	Defective Source File Size			Nondefective Source File Size		
	min	max	avg	min	max	avg
CM1	90	4392	1.108,78	64	2337	629,45
MC1	16	1725	1.370,04	0	559	60,94
PC1	96	2648	960,88	7	2962	798,92
PC2	1749	8351	3.023,75	746	1927	1.336,50
PC3	281	4281	1.196,06	6	4796	1.345,00
AR3	307	2375	1.106,33	160	966	461,00
AR4	318	1653	747,00	9	716	244,75
AR5	351	1162	679,00	82	261	173,75
jdi	58	2540	502,11	3	391	44,85
jdt_apt	64	799	337,50	2	411	64,70
jdt_common	62	1534	383,65	3	178	54,43
jdt_internal	86	4873	637,89	3	933	81,26
jdt_ui	41	10166	452,36	1	1607	62,81
jface	57	1565	415,73	3	440	73,37
pde	60	3181	420,33	2	1213	87,11
swt	38	2596	300,64	2	528	64,20
team	59	5354	603,87	1	277	40,58
ui	53	1740	280,41	2	998	75,54

APPENDIX C: BOX PLOTS OF NAÏVE BAYES EXPERIMENTS

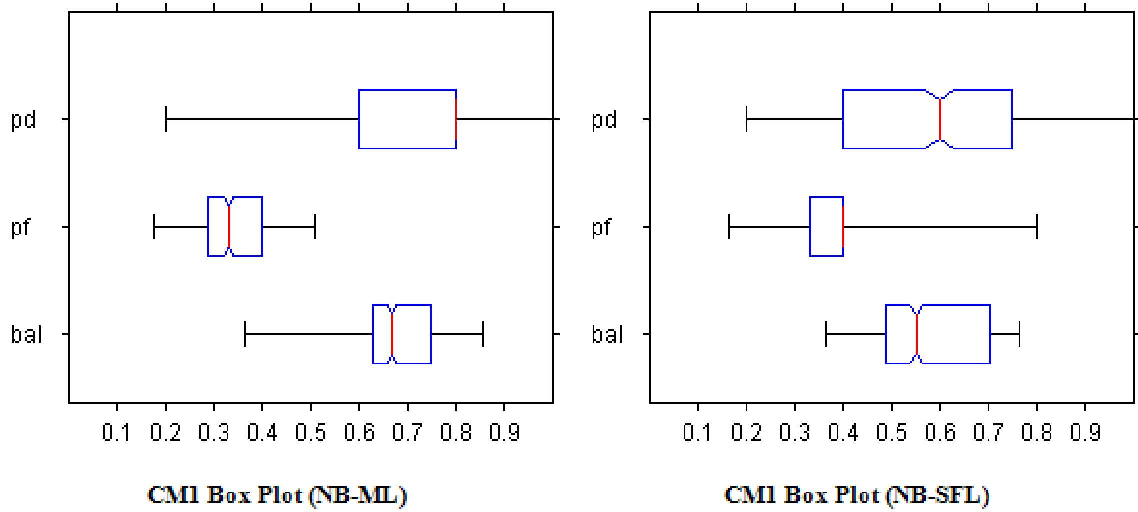

Figure C.1. Comparison of Naïve Bayes box plots for CM1 dataset

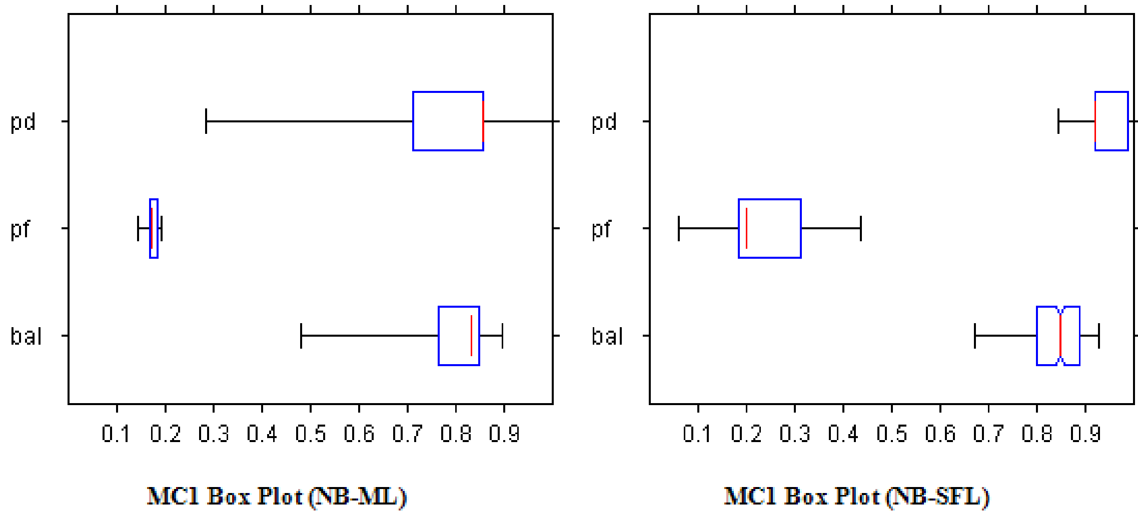

Figure C.2. Comparison of Naïve Bayes box plots for MC1 dataset

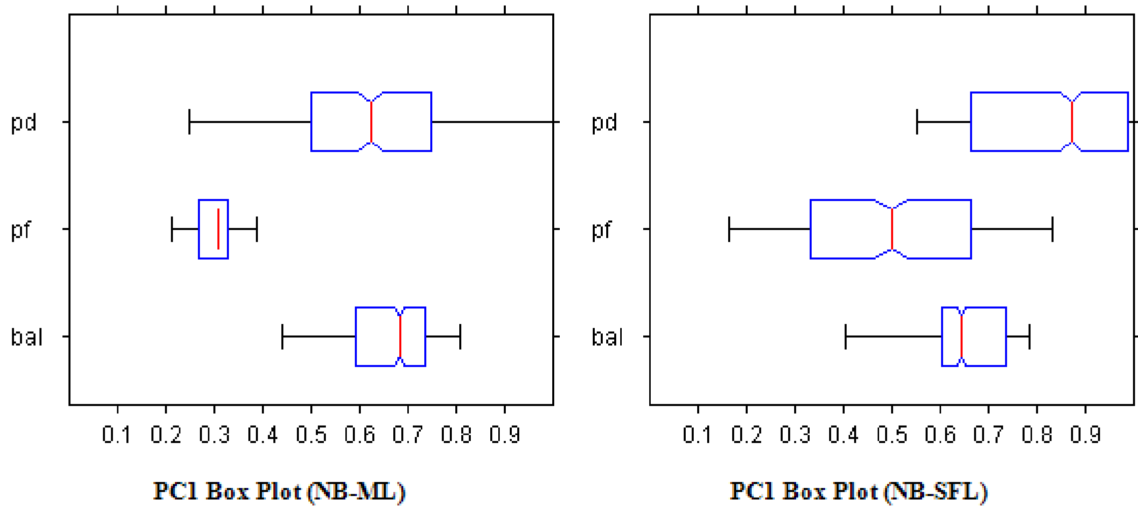

Figure C.3. Comparison of Naïve Bayes box plots for PC1 dataset

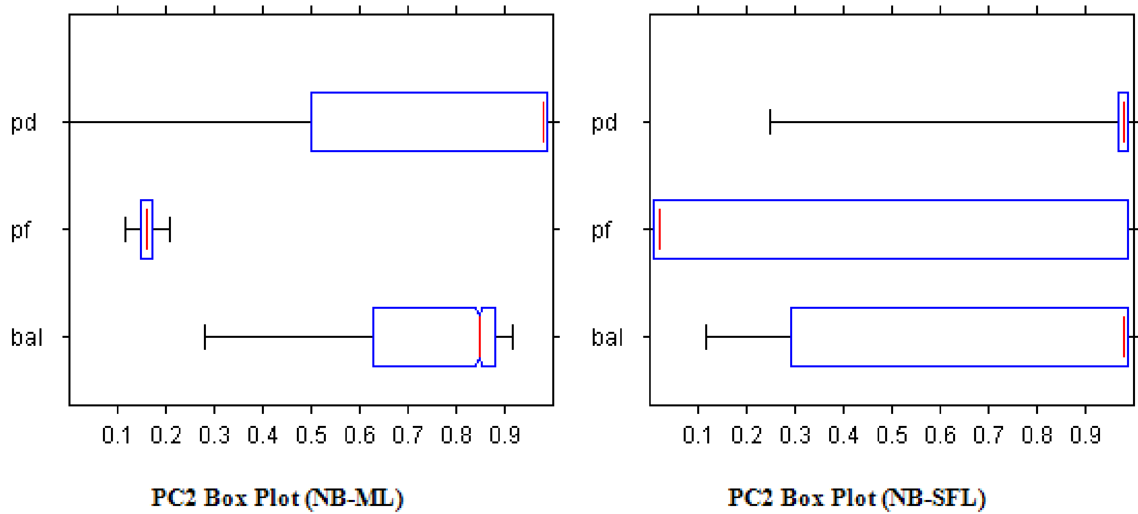

Figure C.4. Comparison of Naïve Bayes box plots for PC2 dataset

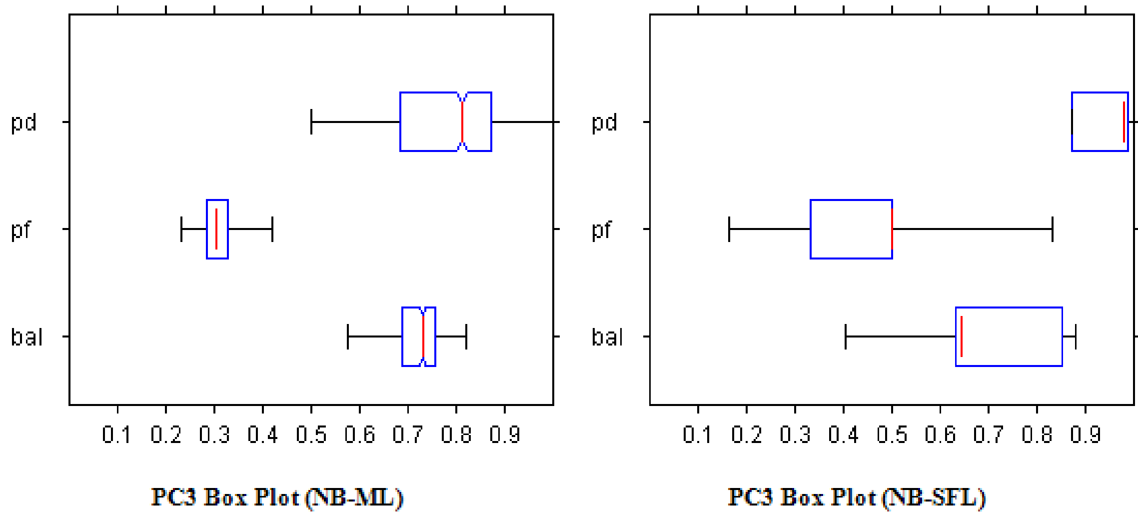

Figure C.5. Comparison of Naïve Bayes box plots for PC3 dataset

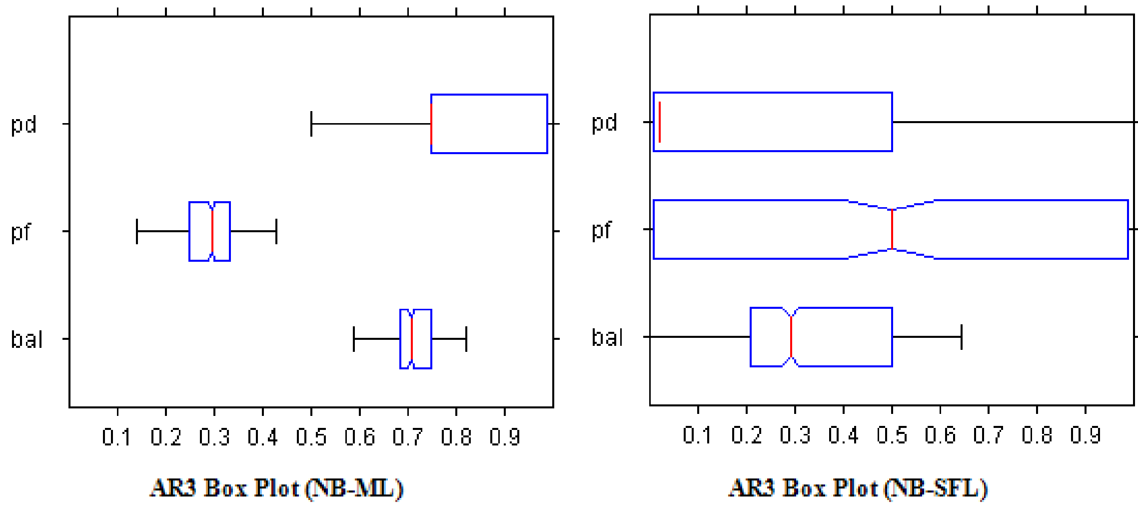

Figure C.6. Comparison of Naïve Bayes box plots for AR3 dataset

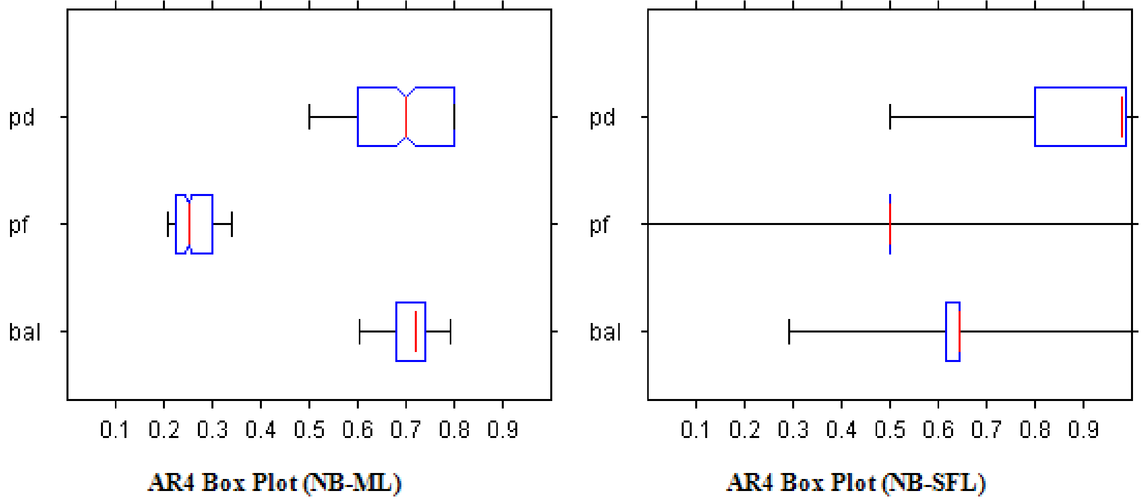

Figure C.7. Comparison of Naïve Bayes box plots for AR4 dataset

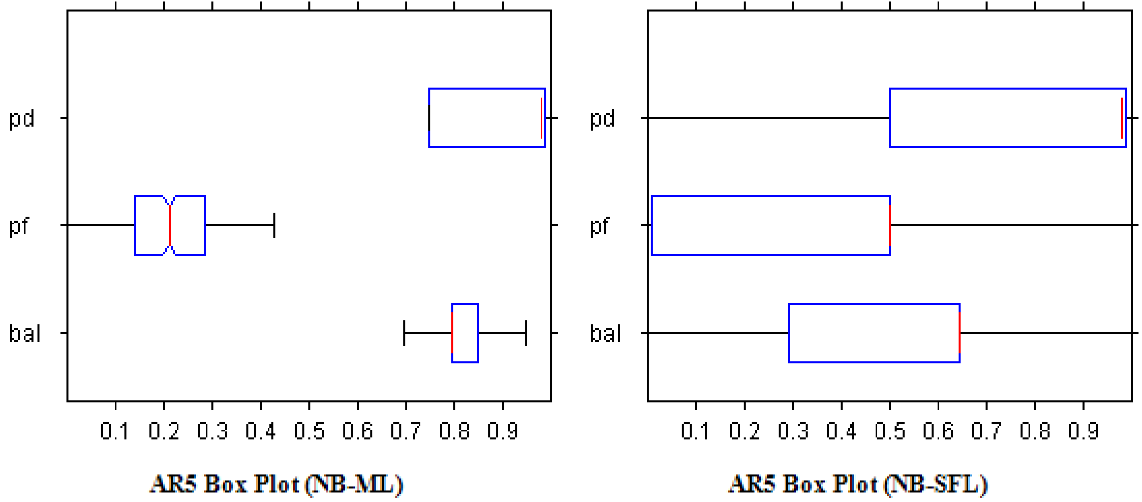

Figure C.8. Comparison of Naïve Bayes box plots for AR5 dataset

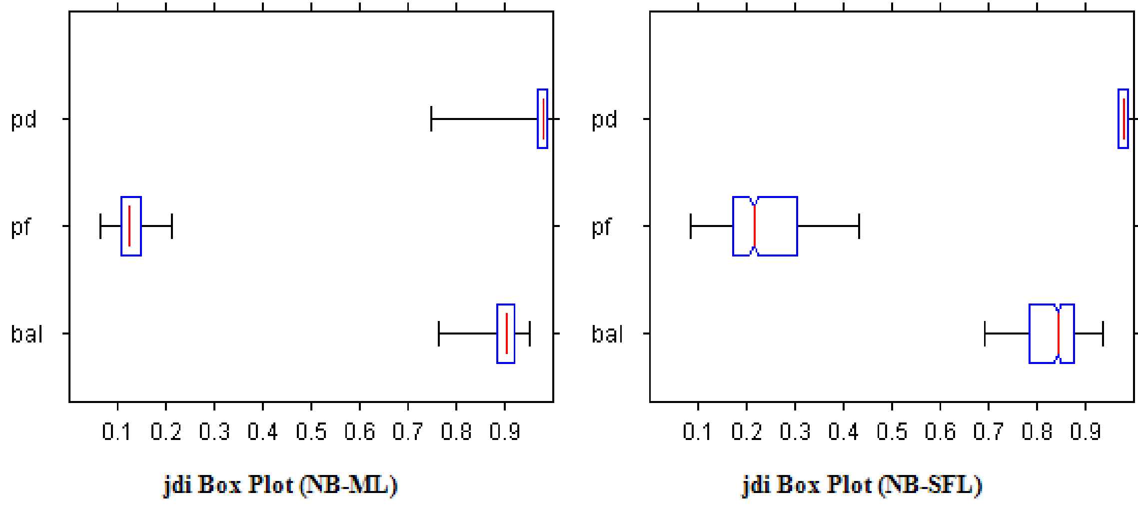

Figure C.9. Comparison of Naïve Bayes box plots for jdi dataset

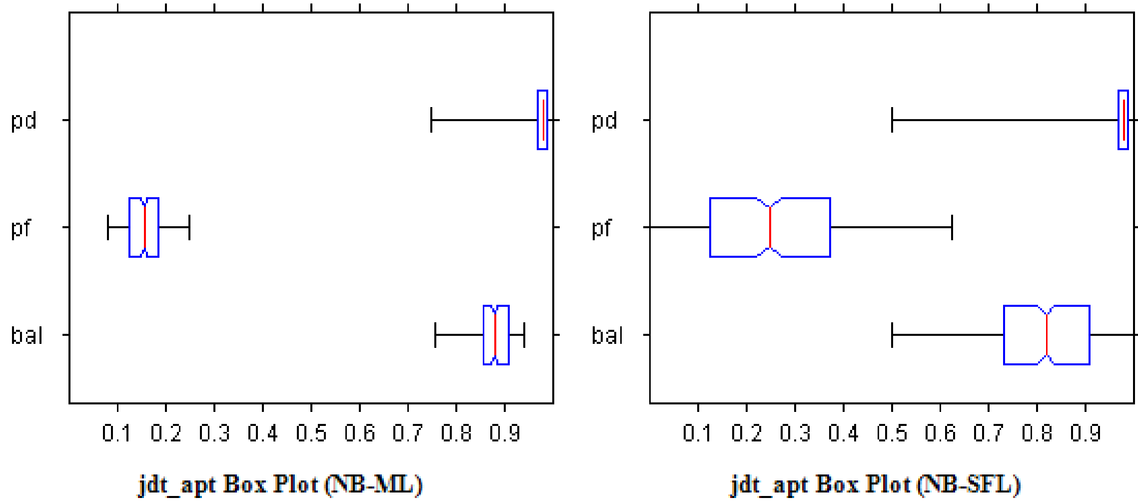

Figure C.10. Comparison of Naïve Bayes box plots for jdt_apt dataset

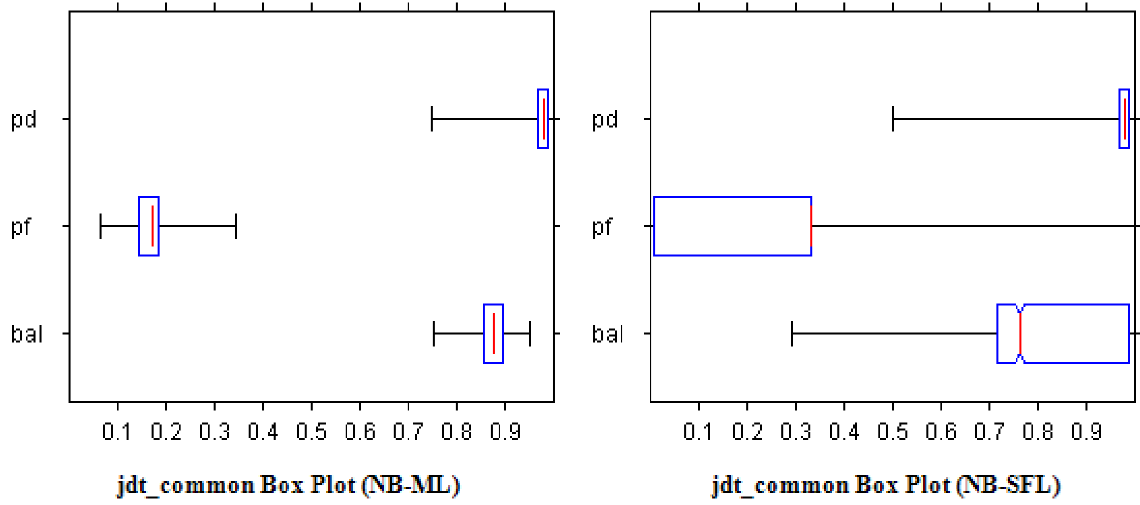

Figure C.11. Comparison of Naïve Bayes box plots for jdt_common dataset

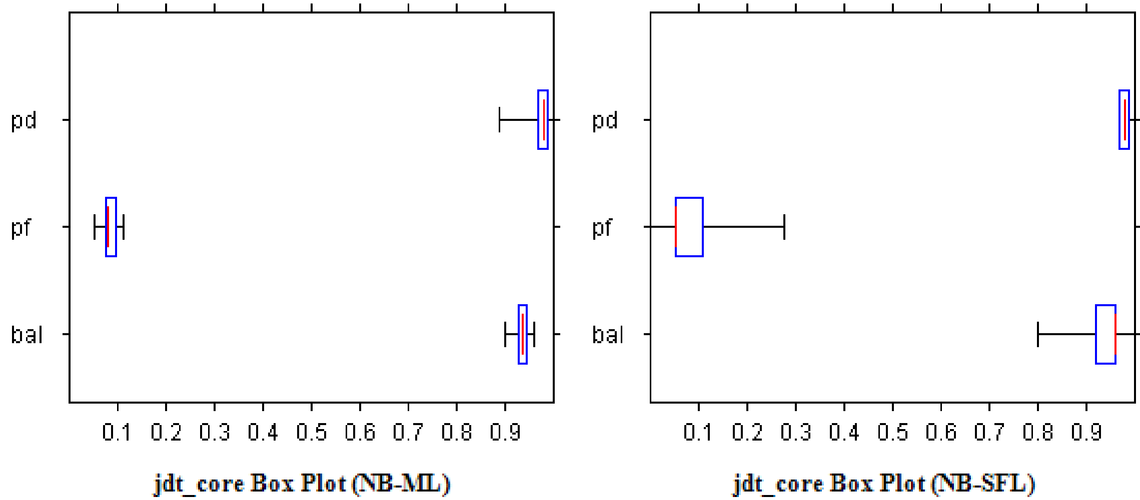

Figure C.12. Comparison of Naïve Bayes box plots for jdt_core dataset

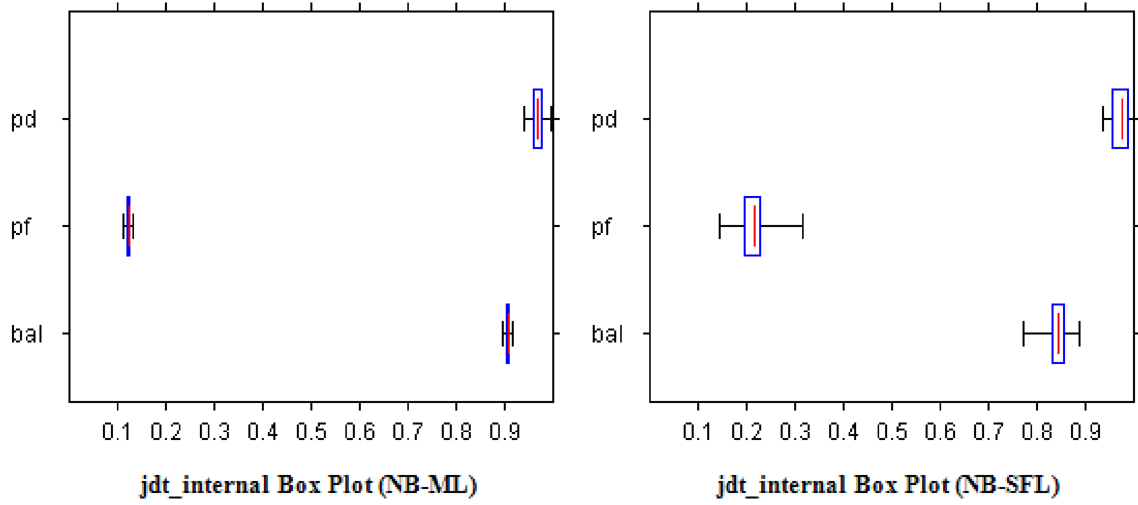

Figure C.13. Comparison of Naïve Bayes box plots for jdt_internal dataset

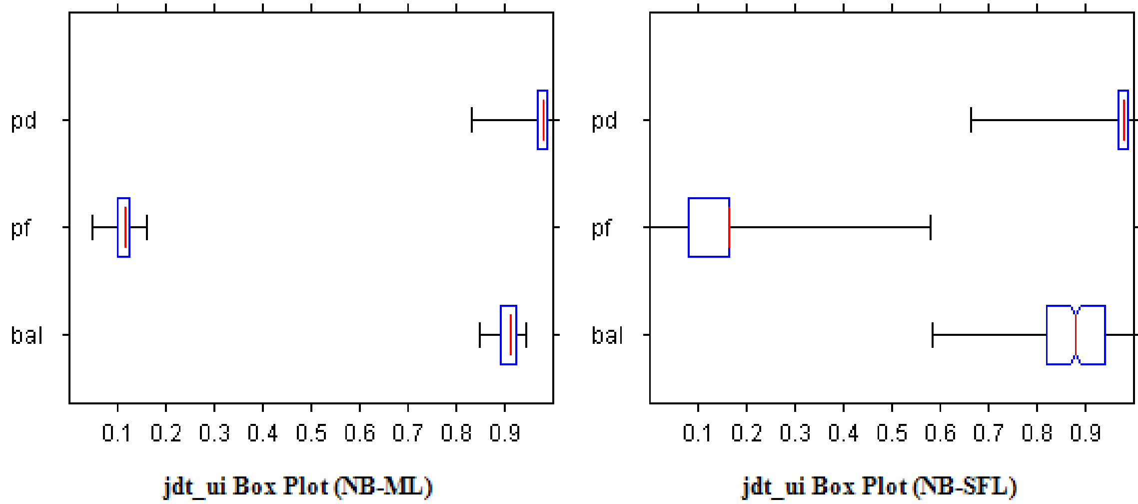

Figure C.14. Comparison of Naïve Bayes box plots for jdt_ui dataset

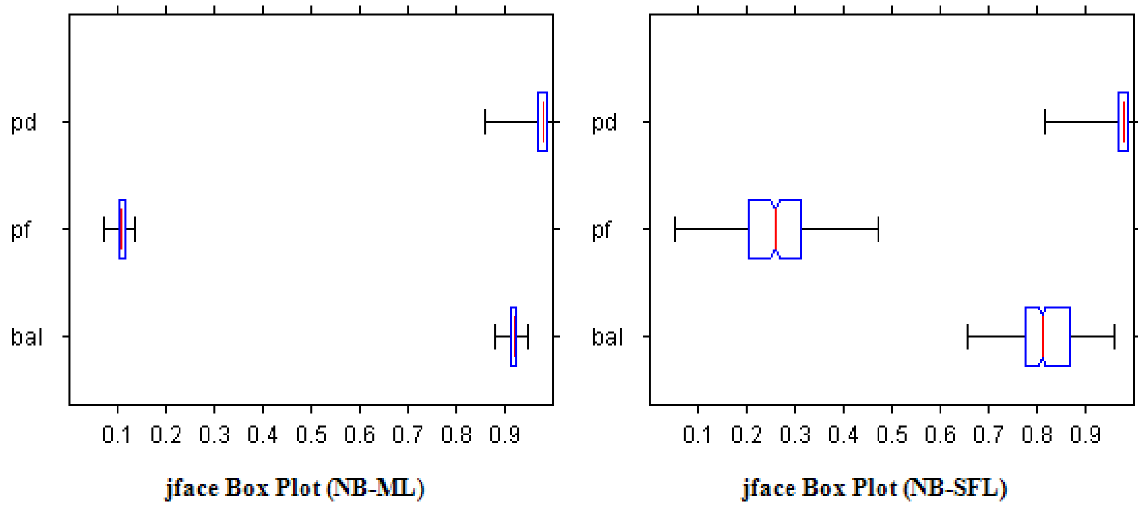

Figure C.15. Comparison of Naïve Bayes box plots for jface dataset

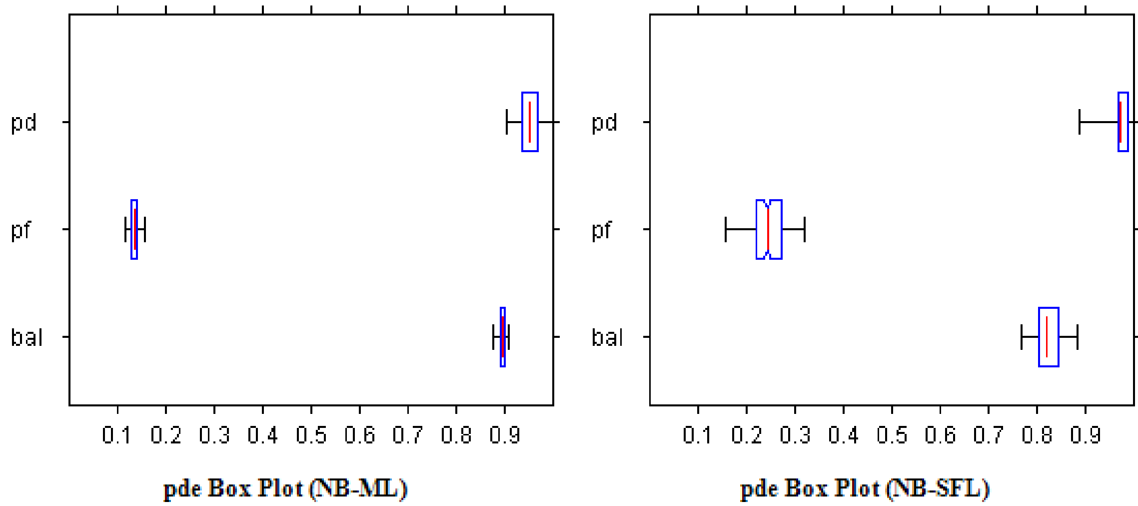

Figure C.16. Comparison of Naïve Bayes box plots for pde dataset

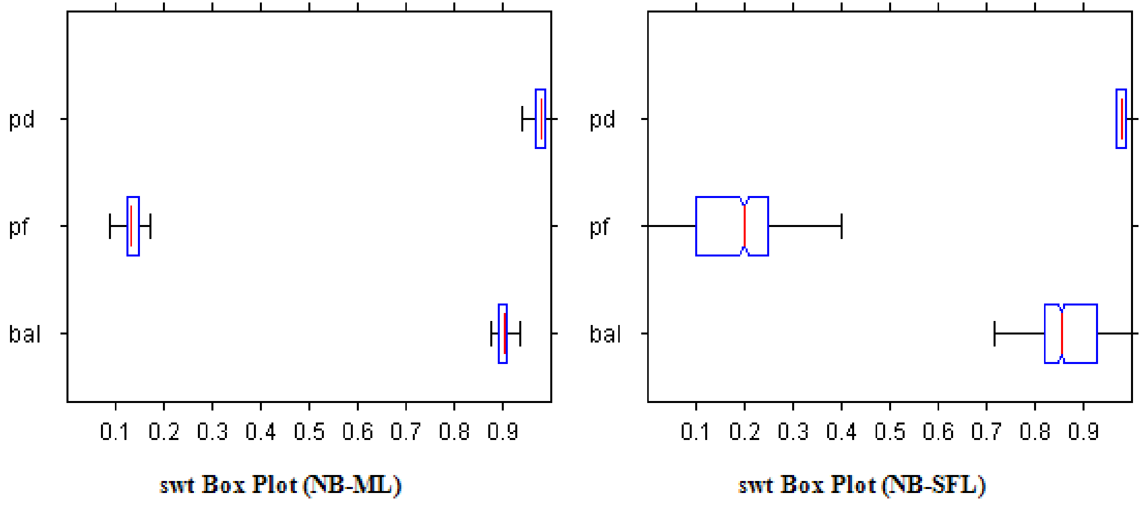

Figure C.17. Comparison of Naïve Bayes box plots for swt dataset

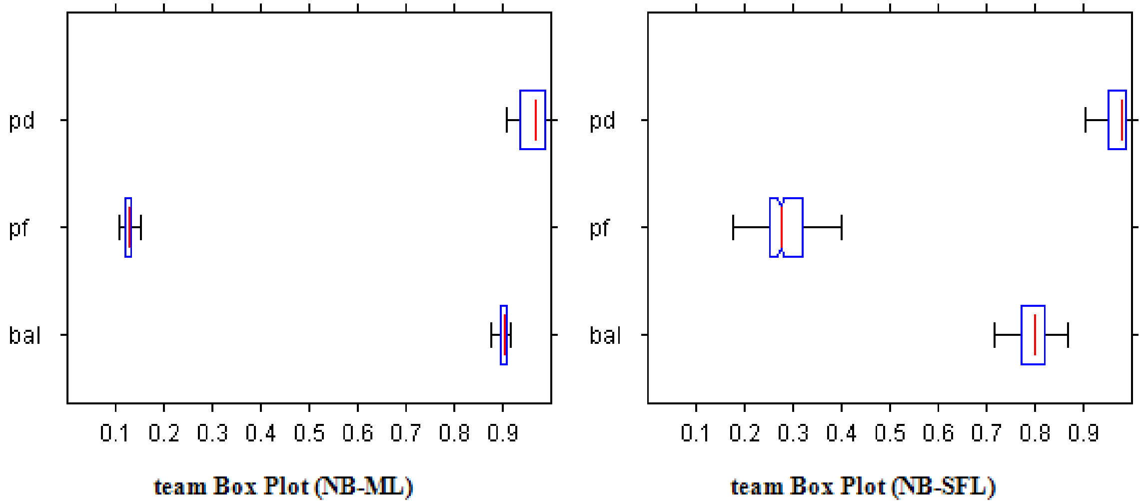

Figure C.18. Comparison of Naïve Bayes box plots for team dataset

APPENDIX D: BOX PLOTS OF DECISION TREE EXPERIMENTS

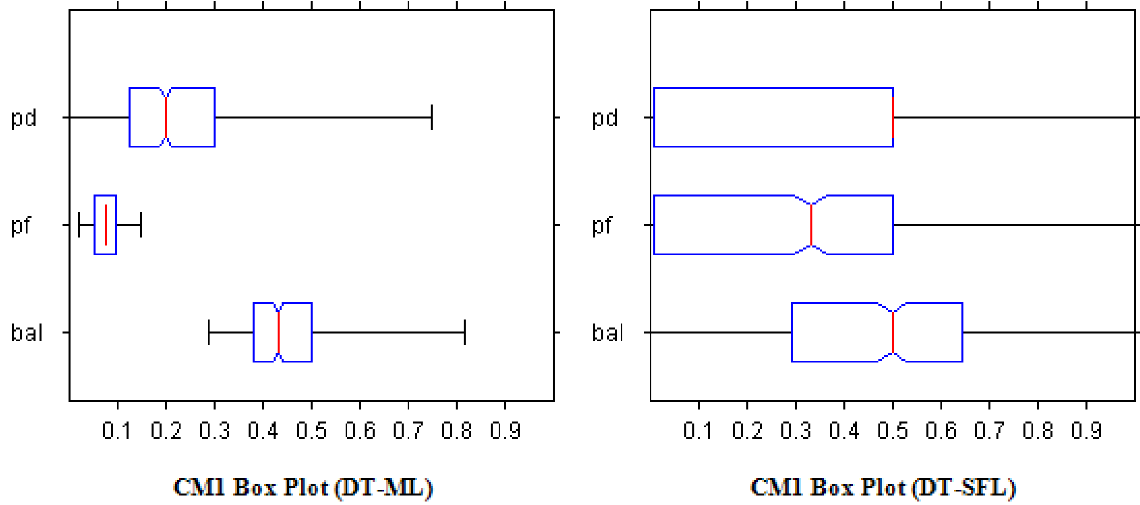

Figure D.1. Comparison of Decision Tree box plots for CM1 dataset

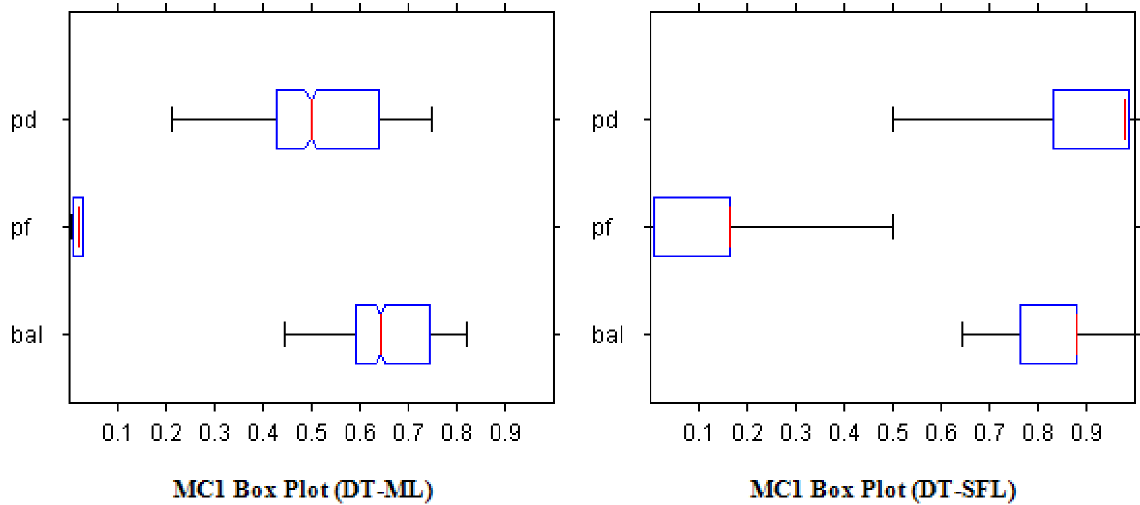

Figure D.2. Comparison of Decision Tree box plots for MC1 dataset

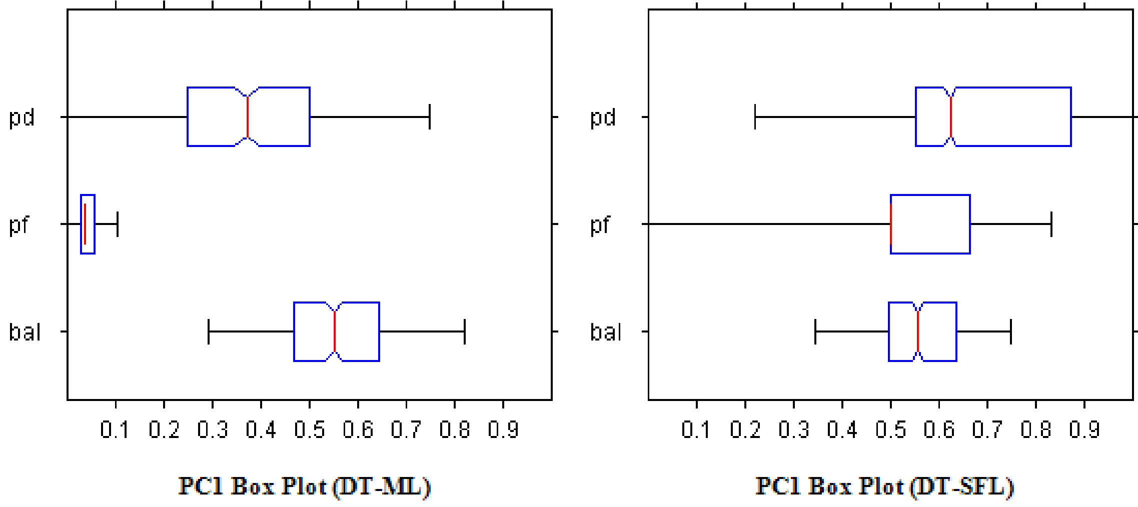

Figure D.3. Comparison of Decision Tree box plots for PC1 dataset

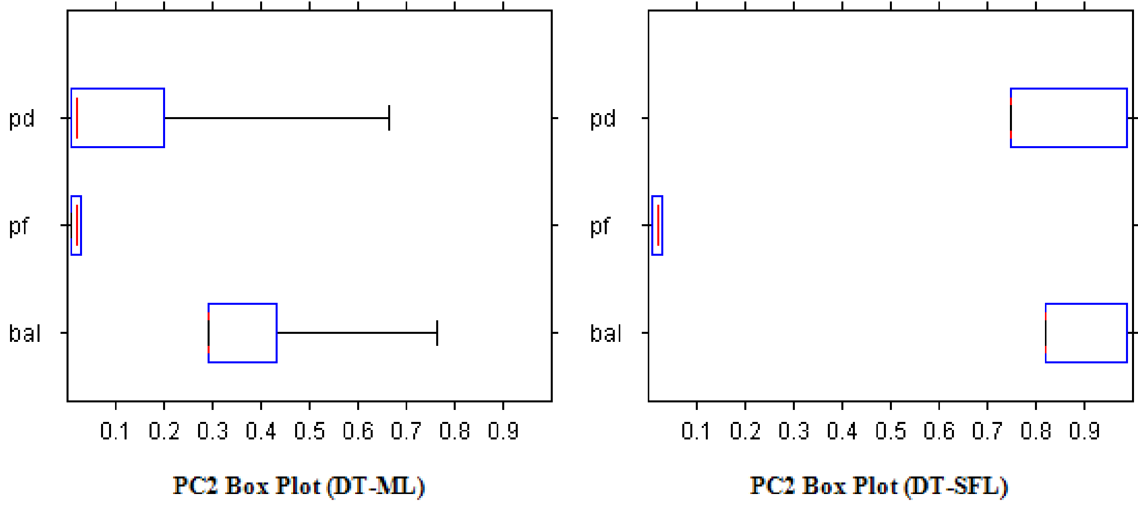

Figure D.4. Comparison of Decision Tree box plots for PC2 dataset

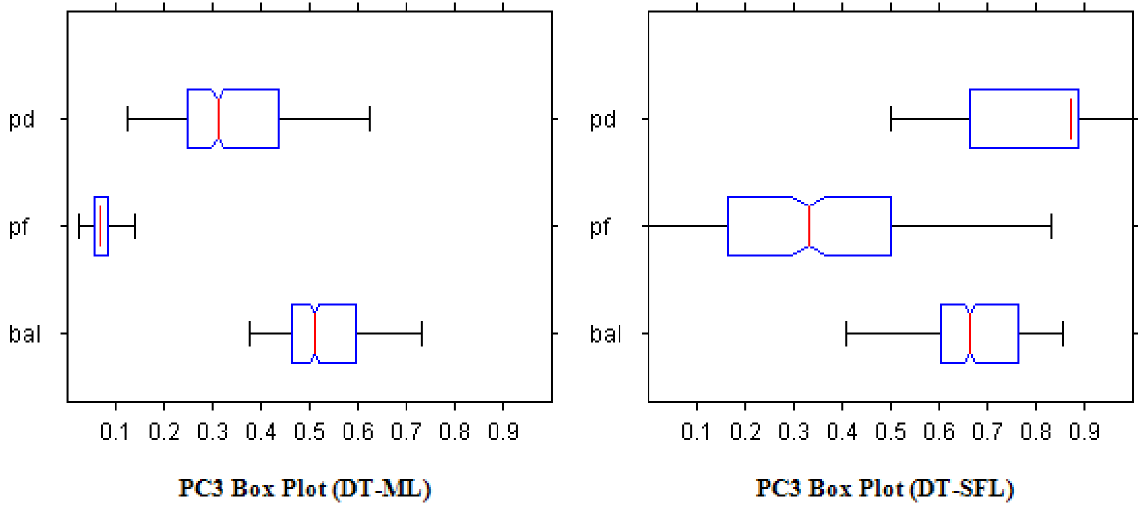

Figure D.5. Comparison of Decision Tree box plots for PC3 dataset

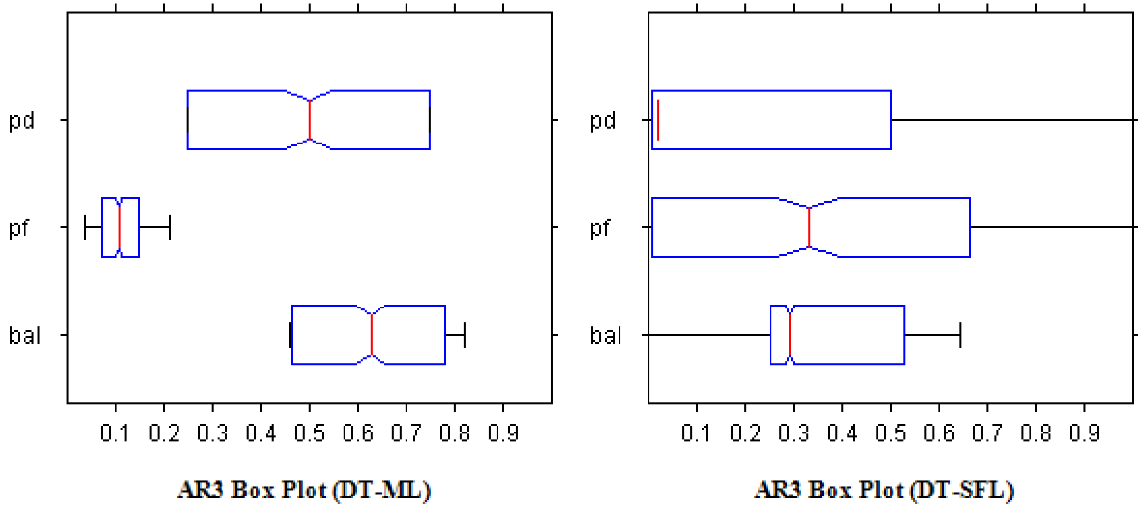

Figure D.6. Comparison of Decision Tree box plots for AR3 dataset

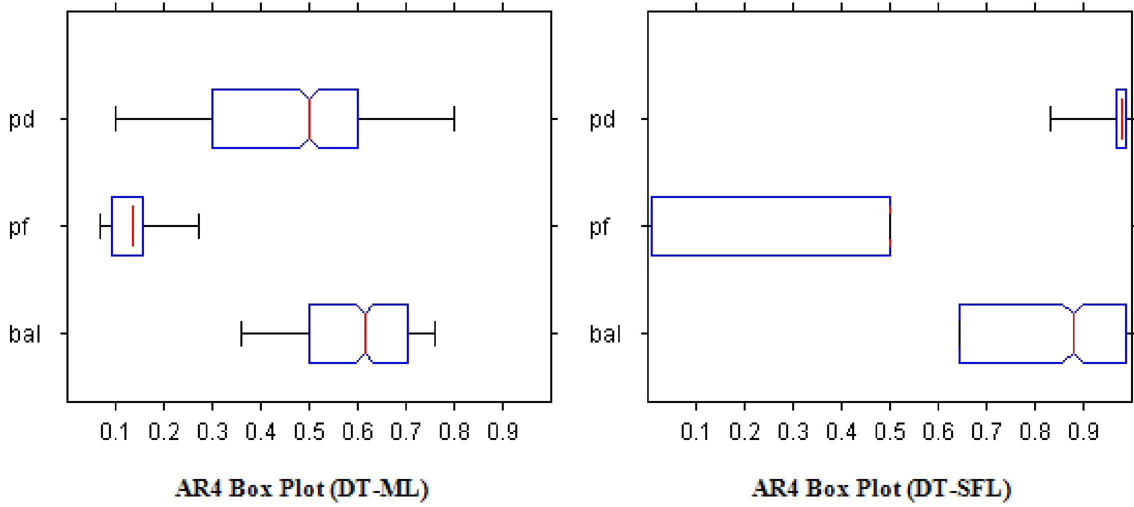

Figure D.7. Comparison of Decision Tree box plots for AR4 dataset

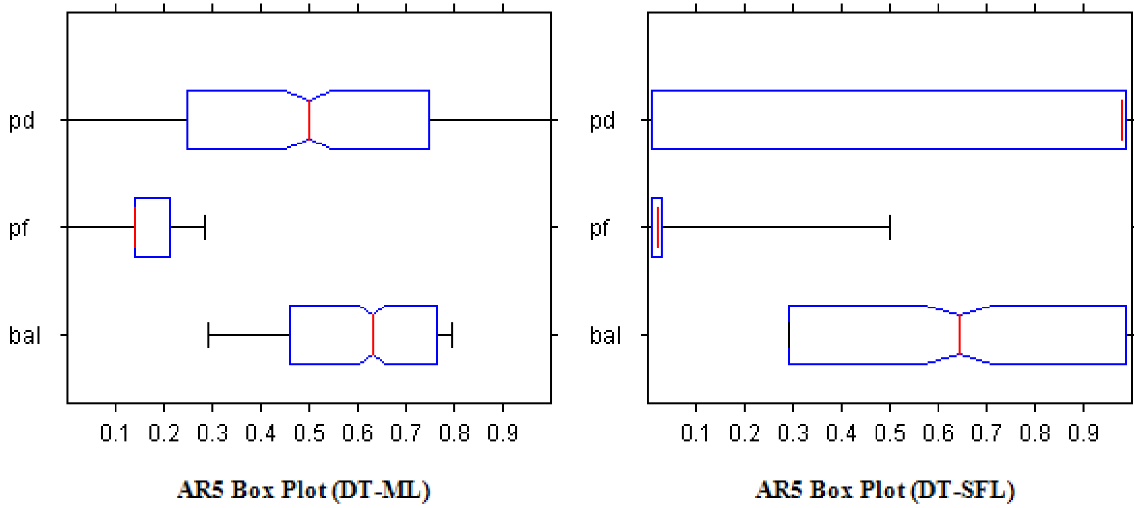

Figure D.8. Comparison of Decision Tree box plots for AR5 dataset

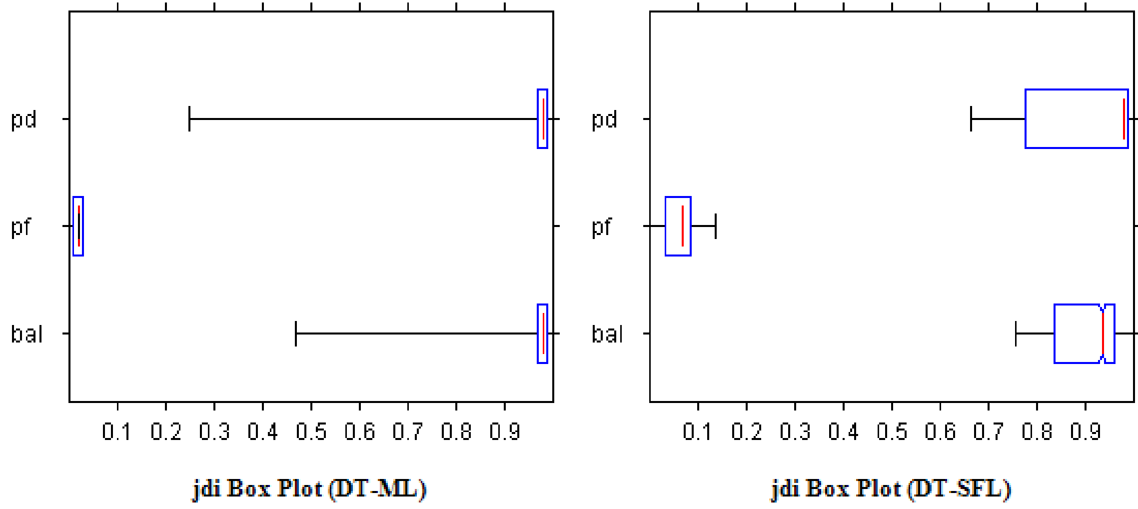

Figure D.9. Comparison of Decision Tree box plots for jdi dataset

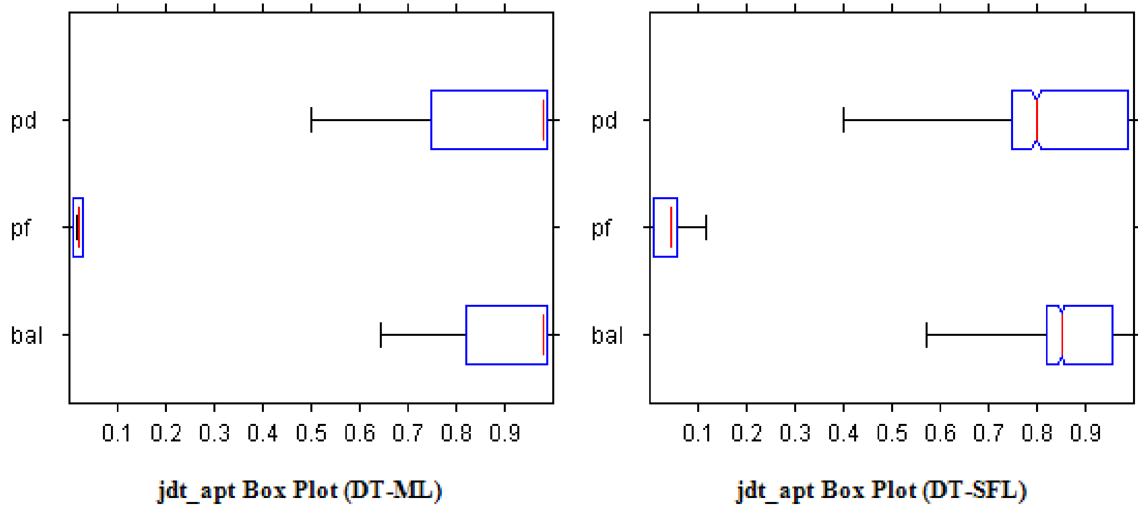

Figure D.10. Comparison of Decision Tree box plots for jdt_apt dataset

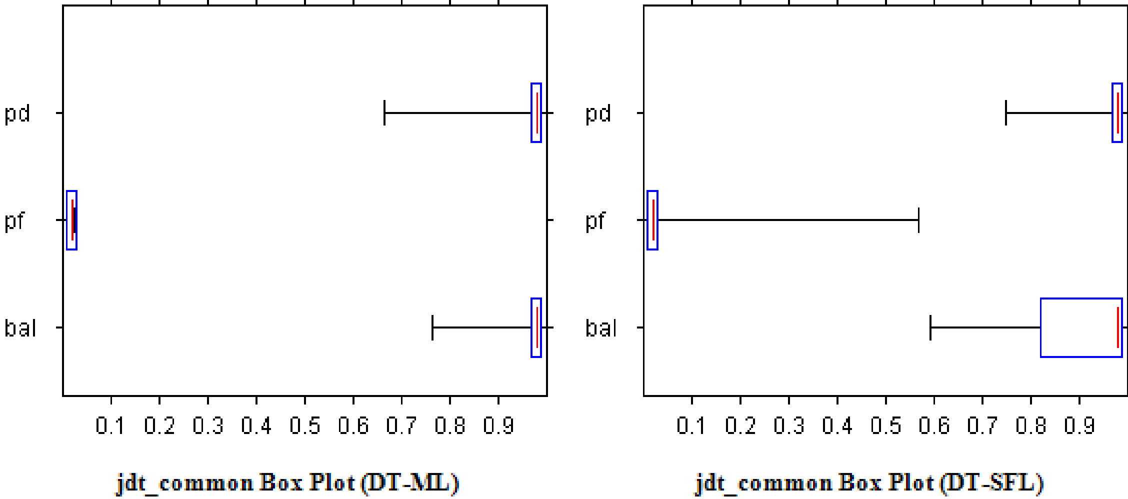

Figure D.11. Comparison of Decision Tree box plots for jdt_common dataset

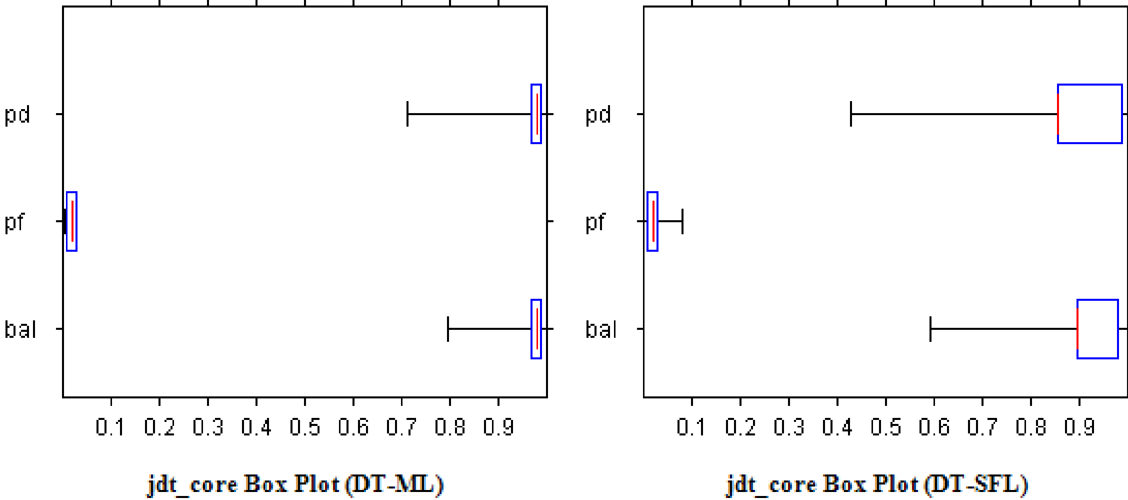

Figure D.12. Comparison of Decision Tree box plots for jdt_core dataset

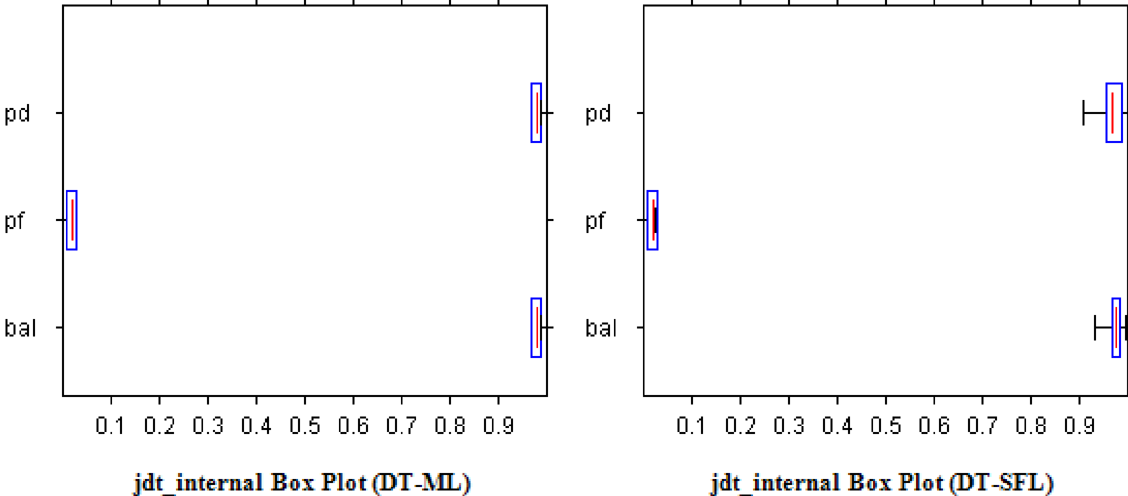

Figure D.13. Comparison of Decision Tree box plots for jdt_internal dataset

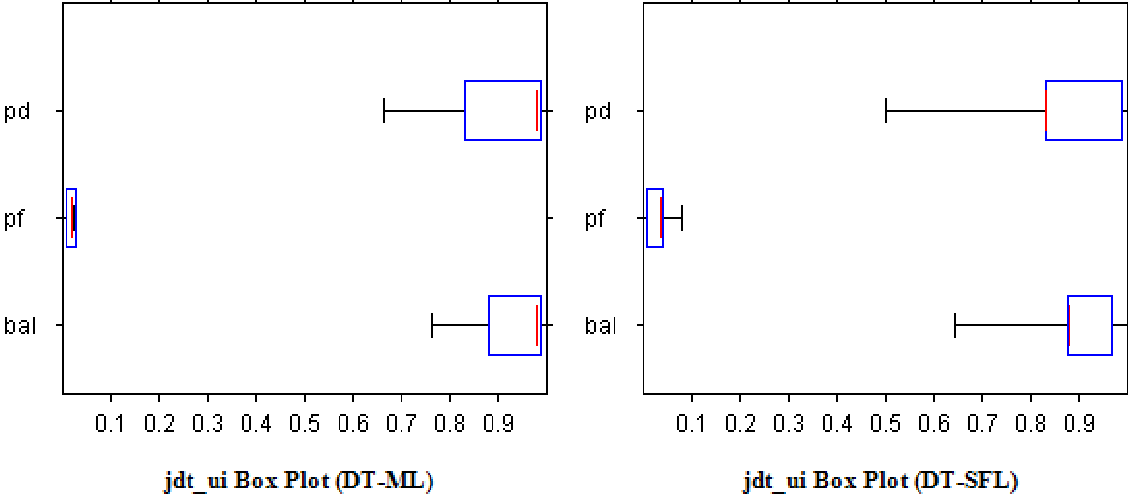

Figure D.14. Comparison of Decision Tree box plots for jdt_ui dataset

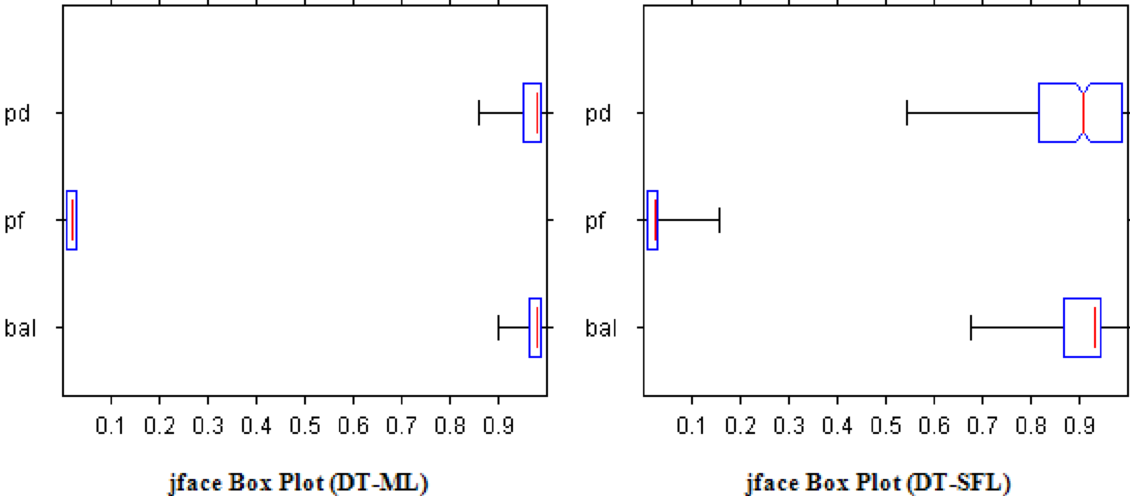

Figure D.15. Comparison of Decision Tree box plots for jface dataset

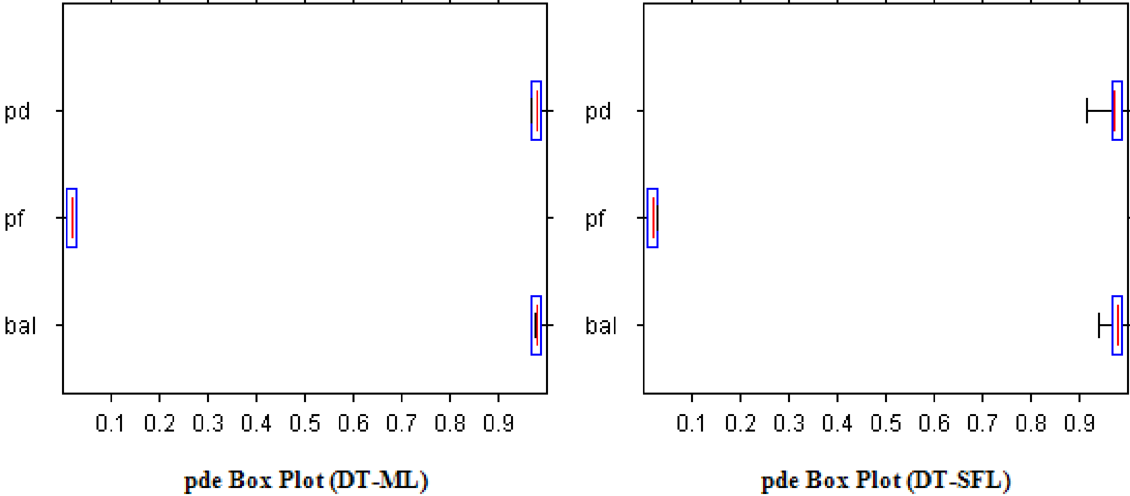

Figure D.16. Comparison of Decision Tree box plots for pde dataset

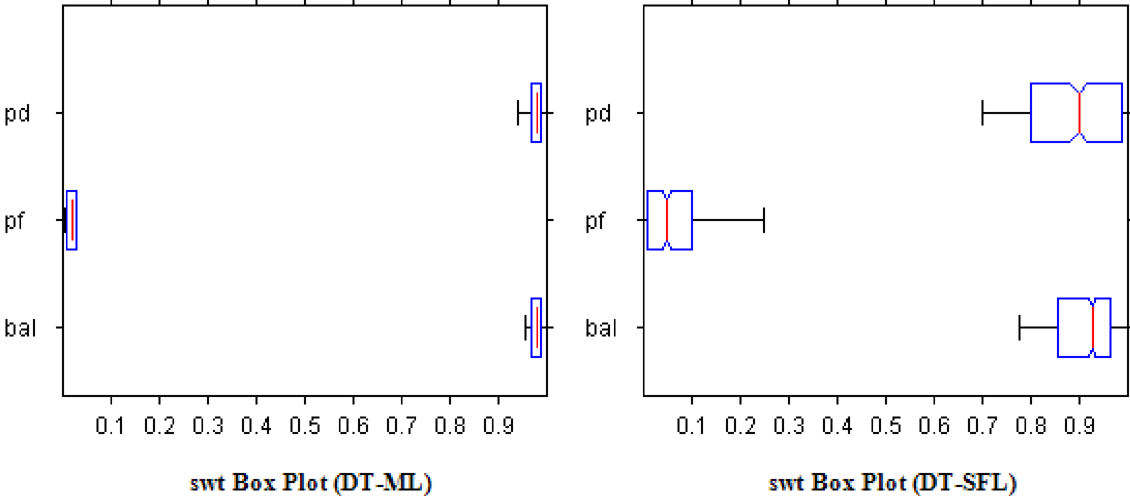

Figure D.17. Comparison of Decision Tree box plots for swt dataset

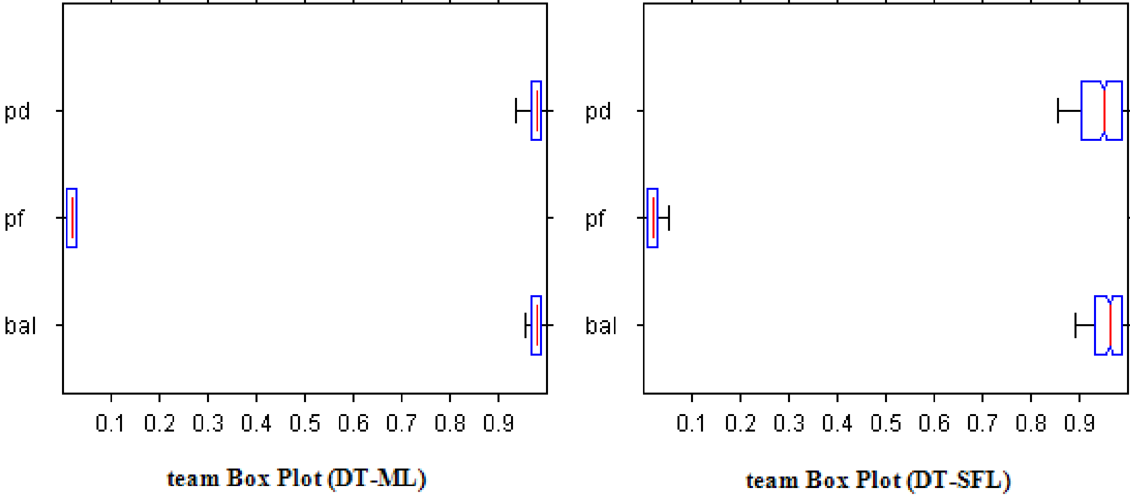

Figure D.18. Comparison of Decision Tree box plots for team dataset

REFERENCES

1. Alpaydın, E., *Introduction to Machine Learning*, The MIT Press, Massachusetts, 2004.

2. Menzies, T., Greenwald, J. and Frank, A., "Data Mining Static Code Attributes to Learn Defect Predictors", *IEEE Transactions on Software Engineering*, Vol. 33, No.1, pp. 2-13, 2007.

3. Menzies, T., Turhan, B., Bener, A. B. and Distefano, J., *Cross- vs Within-company Defect Prediction Studies*, Technical Report, Computer Science, West Virginia University, 2007.

4. Boetticher, G., Menzies, T. and Ostrand, T., *PROMISE Repository of Empirical Software Engineering Data, West Virginia University, Department of Computer Science*, http://promisedata.org/repository, 2007.

5. Harold, M. J., "Testing: a Roadmap", *Conference on the Future of Software Engineering*, pp. 61-72, ACM Press, New York, 2000.

6. Menzies, T., Turhan, B., Bener, A. B., Gay, G., Cukic, B. and Jiang, Y., "Implications of Ceiling Effects in Defect Predictors", *Proceedings of PROMISE Workshop in ICSE 2008*, Leipzig, Germany, 2008.

7. Turhan, B., and Bener, A. B., "Software Defect Prediction: Heuristics for Weighted Naïve Bayes", *ICSOFT 2007*, pp. 22-25, Barcelona, Spain, 2007.

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

9. Halstead, M. H., *Elements of Software Science*, Elsevier, New York, 1977.

10. Kan, S. H., *Metrics and Models in Software Quality*, Addison-Wesley, New York,

2005.

11. Jones, C., “Software Metrics: Good, Bad, and Missing”, Computer, Vol. 27, No. 9, pp. 98-100, 1994.

12. Oman, P., *HP-MAS: A Tool for Software Maintainability*, Software Engineering No. #91-08-TR, Moscow, 1991.

13. Szulewski, P., *Automating Software Design Metrics*, No. RADC-TR-84-27, Rome Air Development Center, Rome, 1984.

14. Fenton, N. E., “A Critique of Software Defect Prediction Models”, *IEEE Transaction on Software Engineering*, Vol.25, No. 5, 1999.

15. Koru, A. G. and Liu, H., “Building effective defect-prediction models in practice Software”, IEEE Transactions on Software Engineering, Vol. 22, Issue 6, pp. 23 - 29, November-December 2005.

16. Munson J. and Khoshgoftaar, T. M., “The Detection of Fault-Prone Programs”, *IEEE Transactions on Software Engineering*, Vol. 18, No. 5, pp. 423-433, 1992.

17. Koru, A. G. and Tian, J., “An Empirical Comparison and Characterization of High Defect and High Complexity Modules”, *The Journal of Systems and Software*, Vol. 67, issue 3, pp. 153-163, 2003.

18. Tosun, A., Turhan, B. and Bener, A. B., “Ensemble of Software Defect Predictors: A Case Study”, 2nd *International Symposium on Empirical Software Engineering and Measurement*, 2008.

19. Arisholm, E. and Briand, L. C., “Predicting Fault-prone Components in a Java Legacy System”, *Proceedings of ACM/IEEE International Symposium on Empirical Software Engineering (ISESE)*, Rio de Janeiro, Brazil, 2006.

20. Arisholm, E., Briand, L. and Fuglerud, M., “Data Mining Techniques for Build-

ing Fault-proneness Models in Telecom Java Software", *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE 2007)*, pp. 215-224, Sweden, 2007.

21. Briand, L. C. and Wuest, J., "Empirical Studies of Quality Models in Object-Oriented Systems", *Advances in Computers*, Vol. 59, pp. 97-166, 2002.

22. Albrecht, A. J., "Measuring Application Development Productivity", *Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium*, October 1979, pp. 83-92, Monterey, California, IBM Corporation, 1979.

23. Menzies, T., Stefano, J.D., Ammar, K., McGill, K., Callis, P., Chapman, R. and Davis, J., "When Can We Test Less?", *9th International Symposium on Software Metrics*, pp. 98-99, 2003.

24. Fisher, D., Xu, L. and Zard, N., "Ordering Effects in Clustering", *Proceedings Ninth International Conference on Machine Learning*, United Kingdom, 1992.

25. Cain, J. W. and McCrindle, R. J., "An Investigation into the Effects of Code Coupling on Team Dynamics and Productivity", *Proceedings of the* 26th *Annual International Computer Software and Applications Conference*, pp. 907, Washington, DC, USA, 2002.

26. Turhan, B. and Bener, A. B., "A Multivariate Analysis of Static Code Attributes for Defect Prediction", 7th *International Conference on Quality Software*, 11-12 October 2007, pp. 231-237, Los Alamitos, CA, USA, 2007.

27. Turhan, B. and Bener, A. B. , *Data Sampling for Cross Company Defect Predictors*, Technical Report, Computer Engineering, Bogazici University, 2008.

28. Munson, J. and Khoshgoftaar, T. M., "Regression Modelling of Software Quality: Empirical Investigation", *Journal of Electronic Materials*, Vol. 19, No. 6, pp. 106 - 114, 1990.

29. Pressman, R., *Software Engineering: A Practitioner's Approach*, Sixth Edition, International, p. 746. McGraw-Hill Education, 2005.

30. Ceylan, E., Kutlubay, F. O. and A. Bener, "Software Defect Identification Using Machine Learning Techniques", *Software Engineering and Advanced Applications*, pp. 240 - 247, August 2006.

31. Koru, A.G. and Liu, H., "An Investigation of the Effect of Module Size on Defect Prediction Using Static Measures", *ACM SIGSOFT Software Engineering Notes*, Vol. 30, No. 4, pp. 1-5, July 2007.

32. Menzies, T., Stefano, J. S., Cunanan, C. and Chapman, R., *Mining Repositories to Assist in Project Planning and Resource Allocation*, International Workshop on Mining Software Repositories, 2004.

33. Banfield, R.E., Hall, L.O., Bowyer, K.W. and Kegelmeyer, W.P., "A Comparison of Decision Tree Ensemble Creation Techniques", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 29, No. 1, pp. 173-180, January 2007.

34. Kutlubay, O., Turhan, B. and Bener, A.B., "A Two-Step Model for Defect Density Estimation", *Software Engineering and Advanced Applications*, pp. 263-270, 28-31 August 2007.

35. Holder, A., *Cost-Benefit Analysis of Monetary and Financial Statistics*, Quarterly Bulletin Technical Report, Bank of England, Summer 2006.

36. Lee, N.Y. and Litecky, N.Y., "An empirical study of software reuse with special attention to Ada", *IEEE Transactions on Software Engineering*, Vol. 23, No. 9, pp. 537 - 549, September 1997.

37. Challagulla, V. U. B., Bastani, F.B., I-Ling, Y. and Paul, R.A., "Empirical Assessment of Machine Learning Based Software Defect Prediction Techniques", *Proceedings of the* 10th *IEEE International Workshop on Object-Oriented Real-Time*

Dependable Systems, pp. 263-270, 2-4 February 2005.

38. Emam, K.E., Melo, W. and Machado, J.C., “The Prediction of Faulty Classes Using Object-Oriented Design Metrics”, *Journal of Systems and Software*, Vol. 56, pp. 63-75, 2001.

39. Bibi, S., Tsoumakas, G., Stamelos, I. and Vlahvas, I., ”Software Defect Prediction Using Regression via Classification”, *IEEE International Conference on Computer Systems and Applications*, pp. 330-336, 8 March 2006.

40. Luo Li, P., Herbsleb, J., Shaw, M. and Robinson, B., “Experiences and Results from Initiating Field Defect Prediction and Product Test Prioritization Efforts at ABB Inc.”, *International Conference on Software Engineering*, pp. 413-422, 2006.

41. Oral, A. D. and Bener, A. B., “Defect Prediction for Embedded Software”, *ISCIS 2007*, 9-11 November 2007, Ankara, Turkey, 2007.

42. Lessmann, S., Baesens, B., Mues, C. and Pietsch, S., “Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings”, *IEEE Transactions on Software Engineering*, Vol. 34, No. 4, pp. 485-496, July/August 2008.

43. Hall, M., “A Decision Tree-based Attribute Weighting Filter for Naïve Bayes”, *Knowledge-Based Systems*, Vol. 20, Issue 2, pp. 120-126, March 2007.

44. Menzies, T., Di Stefano, J. S., Chapman, M. and McGill, K., “Metrics That Matter”, 27th Annual NASA Goddard Software Engineering Workshop, pp. 51, 2002.

45. Khoshgoftaar, T. M. and Allen, E.B., “Predicting Fault-Prone Software Modules in Embedded Systems with Classification Trees”, 4th *IEEE International Symposium on High-Assurance Systems Engineering*, pp. 105, 1999.

46. Henry, S. and Kafura, D., “The Evaluation of Software System’s Structure Using Quantitative Software Metrics”, *Software Practice and Experience*, Vol. 14, No. 6,

pp. 561-573, 1984.

47. Chang, C.P., Lv, J.L. and Chu, C.P., “A Defect Estimation Approach for Sequential Inspection Using a Modified Capture-Recapture Model”, 29th Annual International Computer Software and Applications Conference, Vol. 1, pp. 41-46, 2005.

48. Menzies, T. and Gukic, B., “When to Test Less (Software Testing)”, *IEEE Transactions on Software Engineering*, Vol. 17, No. 5, pp. 107-112, September/October 2000.

49. Michael, J.B., Bossuyt, B.J., Snyder, B.B., “Metrics for Measuring the Effectiveness of Software-testing Tools”, 13th *International Symposium on Software Reliability Engineering*, pp. 117-128, 12-15 November 2002.

50. Benbrahim, H. and Bensaid, A., “A Comparative Study of Pruned Decision Trees and Fuzzy Decision Trees”, *International Conference of the North American Fuzzy Information Processing Society*, pp. 227-231, July 2000.

51. Quinlan, J. R., “Introduction of Decision Trees”, *Machine Learning*, Vol. 1, pp. 81-106, 1986.

52. Jiang, Y., Cukic, B. and Menzies, T., “Fault Prediction Using Early Lifecycle Data”, 18th *IEEE International Symposium on Software Reliability*, pp. 237-246, 2007.

53. Xiao-Bai, L., Sweigart, J. R., Teng, J. T. C., Donohue, J. M., Thombs, L. A. and Wang, S. M., “Multivariate Decision Trees Using Linear Discriminants and Tabu Search”, *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 33, No. 2, pp. 194-205, March 2003.

54. Turhan, B., “Software Defect Prediction Modeling”, *IDoESE 2007*, Madrid, Spain, September 19, 2007.

55. Breiman, L., Friedman, J. H., Olshen, R.A. and Stone, C. J., *Classification and*

Regression Trees, Wadsworth & Brooks, 1984.

56. 4th International Workshop on Predictor Models in Software Engineering (an ICSE 08 Workshop), Leipzig, Germany, May 2008.

57. R. Rice, *Software Testing Glossary*, Rice consulting, 1999.

58. RTI International, *The Economic Impacts of Inadequate Infrastructure for Software Testing*, http://www.nist.gov/director/prog-ofc/report02-3.pdf, 2002.

59. Larry Dignan, *One in Three IT Projects Fail, Management ok with it*, http://blogs.zdnet.com/BTL/?p=7298, 2007.

60. Andy McCue, *One in Three Offshore Projects Fail*, http://www.silicon.com/research/specialreports/offshoring /0,3800003026,39125059,00.htm, October 2004.

61. Los Alamos Center, *What is Bayesian Inference?*, http://drambuie.lanl.gov/b̃ayes/tutorial.htm, August 2008.

62. Wikipedia, *Bayes' Theorem*, http://en.wikipedia.org/wiki/Bayes%27s_theorem, August 2008.

63. NASA, *SATC Code Metric Historical Database*, http://satc.gsfc.nasa.gov/metrics/codemetrics/index.html, August 2008.

64. NASA, *NASA Data Repository*, http://mdp.ivv.nasa.gov/repository.html, August 2008.

65. PROMISE, *Promise Data Repository*, http://promisedata.org, July 2008.

66. Gunes Koru, *Problems with MC1 Dataset*, http://promisedata.org/?p=30, August 2008.

67. Eclipse, *Eclipse Software Development Tool*, http://www.eclipse.org/, August 2008.

68. Eclipse, *Eclipse Source Code*, http://www.eclipse.org/downloads/, August 2008.

69. Eclipse Wikipedia, *How to Use CVS*, http://wiki.eclipse.org/index.php/CVS_Howto, August 2008.

70. Predictive, *Predictive Software Metrics Tool*, http://www.ismwv.com, November 2007.