Prediction and Probability Distribution of Defects in Software Systems

THESIS

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

By

MUTHUKUMARAN K.

ID. No. 2011PHXF0415H

Under the Supervision of

Prof. N L Bhanu Murthy

Co-Supervision of

Dr. Aruna Malapati

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

2017

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

CERTIFICATE

This is to certify that the thesis entitled **"Prediction and Probability Distribution of Defects in Software Systems"** and submitted by **Muthukumaran K.** ID.No. **2011PHXF0415H** for the award of Ph.D. degree of the institute embodies original work done by him under my supervision.

DR. N L BHANU MURTHY
Associate Professor,
Department of Computer Science &
Information Systems,
BITS Pilani, Hyderabad Campus
Hyderabad, Telangana - 500078.

Date:

DR. ARUNA MALAPATI
Assistant Professor,
Department of Computer Science &
Information Systems,
BITS Pilani, Hyderabad Campus
Hyderabad, Telangana - 500078.

Date:

Acknowledgement

I would like to thank all the amazing people who contributed in some way to the work described in this thesis. First and foremost, I would like to express my profound gratitude and sincere thanks to my Supervisor Prof. N L Bhanu Murthy, whose expertise, understanding and patience, added great value at every stage of this thesis work.

I would also like to thank my Co-Supervisor Dr Aruna Malapati for her generous advice and encouragement. I am thankful to my DAC members Prof. R Gururaj and Prof. Tathagata Ray, for their encouragement, insightful comments and hard questions. I am thankful to Prof. K Srinivasa Raju for his insightful questions and suggestions.

I express my gratitude towards Prof. G Sundar (Director and Senior Professor of BITS Pilani Hyderabad campus) and Prof. M B Srinivas (Dean, Administration). I would also like to express my gratitude to Prof. Vidya Rajesh (Associate Dean, Academic Research Division), for her constant support during my Ph.D course.

I would like to thank the co-authors of my papers, Abhinav Choudhury, G Karthik Reddy, Prateek Talishetti, Akhila Rallapalli, Swapnil Shukla, T Radhakrishnan, Srinivas Suri, Amrita Dasgupta, Abhidnya Shirode, Pranav Ramarao, Siddharth Dash, C K Shriram and Janani. I am thankful to fellow PhD scholars Jagan Mohan Reddy, Rakesh Prasanna, Neha Singh and Sai Kiranmai for their continuous support and encouragement.

I express my gratitude to my parents - Kasinathan Manickam and Selvi Kasinathan, my sister - Mala Kasinathan and my fiancée - Vanita Jaitly for their constant love, support and unfailing guidance. I owe them everything.

Abstract

Defect prediction models help software project teams to identify defect prone source files of software systems. Software project teams can prioritize and put up rigorous Quality Assurance (QA) activities on these predicted defect prone files to minimize post-release defects so that better-quality software can be delivered. Defect prediction models are built by making use of historical data of the same project or similar projects to predict defects in the current version. The source code, change and complexity metrics of files or classes are generally used as features to build these models.

Effective defect prediction approaches will assist the quality assurance team to perform resource intensive, time consuming and high cost involved assurance activities on comparatively fewer segments of the system. In this thesis, approaches or methodologies have been proposed to enhance prediction accuracies of defect prediction models.

If the generic model for bug distribution across projects and domains is known, quality assurance activities like testing and review of future releases can be planned efficiently. Hence finding the right probability distribution that models defects in software systems is an interesting problem and is attempted to be solved in this thesis.

The list of major contributions of this thesis is presented below.

- The application of feature selection techniques is shown to improve prediction accuracies of defect prediction models and many other related questions are answered.

- Proposed two novel change metrics as features to build defect models and presented prediction accuracies of these models.

- Effort-aware defect prediction models are built based on the effort spent in performing specific quality assurance activities like code review, testing and showed that these models are efficient.

- By relaxing conditional independence assumptions on features, Bayesian Network structures are built and the performance of these augmented Bayesian Network classifiers have been explored in this thesis.

- By considering misclassification costs, defect prediction models have been built using cost-sensitive boosting Neural Network methods and their effectiveness has been compared with competent classifiers.

- Defect Prediction Problem is formulated as multi-objective optimization problem with contrasting objective functions and benefits / efficiency of these models as compared to single objective defect prediction models is discussed in this thesis.

- The intuitive idea for probability distributions like Weibull, Bounded Generalized Pareto, Double Pareto, Log-normal and Yule-Simon distributions to model defects is discussed and these probability distributions are evaluated for their fitness by making use of information theoretic approaches to model selection.

Contents

List of Figures

List of Figures

List of Tables

List of Abbreviations

Term	Definition
AdaBoost	Adaptive Boosting
AIC	Akaike Information Criterion
AUC	Area Under the Receiver Operating Characteristics Curve
BDE	Bayesian Dirichlet Equivalent
BIC	Bayesian Information Criterion
BOW	Bag-of-Words
C3	Conceptual Cohesion of Classes
CC	Cyclomatic Complexity
CD	Critical Difference
CDF	Cumulative Distribution Function
CE	Cost Effectiveness
CK	Chidamber-Kemerer
CSBNN	Cost Sensitive Boosting of Neural Networks
CSBNN-TM	Cost Sensitive Boosting of Neural Networks using Threshold Moving
CSBNN-WU1	Cost Sensitive Boosting of Neural Networks with Weight Updation-I
CSBNN-WU2	Cost Sensitive Boosting of Neural Networks with Weight Updation-II
DT	Decision Tree
FAN	Forest Augmented Naïve Bayes
FN	False Negatives
FP	False Positives
GPD	Generalized Pareto Distribution
GS	Grow-Shrink Algorithm
HC	Hill Climbing
KNN	K-Nearest Neighbors
LDD	Linearly Decayed Entropy
LOC	Lines of Code
LR	Logistic Regression
LSE	Least Square Estimate
LTR	Learning-To-Rank
MDL	Minimum Description Length
MDP	Metrics Data repository
MLE	Maximum Likelihood Estimate
MMHC	Max-Min Hill Climbing algorithm
MMPC	Max-Min Parents and Children
MOGA	Multi-Objective Genetic Algorithm

Term	Definition
MOLR	Multi-objective Logistic Regression
MOM	Module Order Model
MoM	Method of Moments
MOO	Multi-objective optimization
NASA	National Aeronautics and Space Administration
NB	Naïve Bayes
NBT	Naïve Bayes Tree
NECM	Normalized Expected Cost of Misclassification
NN	Neural Network
NSGA-II	Non-dominated Sorting Genetic Algorithm II
OO	Object-oriented
P	Precision
PCA	Principal Component Analysis
PDF	Probability Density Function
POA	Percentage of Area
QA	Quality Assurance
R	Recall
RD	Relative Differences
RF	Random Forest
RMSE	Root Mean Square Error
RSMAX2	Two-stage Restricted Maximization
SFAN	Selected Forest Augmented Naïve Bayes
SFAND	Selected Forest Augmented Naïve Bayes Discarding
SOLR	Single-objective Genetic Algorithm
STAN	Selective Tree Augmented Naïve Bayes
STAND	Selective Tree Augmented Naïve Bayes with Discarding
SVM	Support Vector Machine
TABU	Tabu Search
TAN	Tree Augmented Naïve Bayes
TN	True Negatives
TP	True Positives
WCHU	Weighted Churn

1 Introduction

The development of today's large and complex software systems is a challenging task particularly with constraints on resources such as manpower and time. The software quality assurance activities encompass all the phases of software development life cycle and consume major fraction of the total cost of the project. Software defects* detected during post-release phase are proved to be very costly as compared to defects uncovered in pre-release phases like coding, testing etc. According to a study, fixing a software bug after deployment is 100 times costlier than fixing it during development [2]. Various methods have been proposed to identify defects in software systems. Static code analysis is one such popular method used to identify the defects. Although code review helps the Quality Assurance (QA) team to identify bugs, it demands considerable amount of time and expertise for the effective review process. Interestingly, the software bugs or faults tend to cluster i.e. they are not evenly distributed across all the modules as per previous studies [3]. The generic pareto principle is observed to be true for defects in software systems as well and the principle in this context means that 20% of files have 80% of bugs [4]. The defect prediction model predicts defect prone files using learned models built from the historical data.

Defect Prediction models are built with training data and they are evaluated for performance on the testing data. Defect prediction models are built using three prediction techniques that differ based on the source of training data and testing data. The different prediction techniques are: 1) Cross Validation prediction: In this approach, training and testing data are taken from the same version of a project. The defect prediction models are built by taking 70% of data as the training data and tested on remaining 30% of the

*We use "bugs", "faults" and "software defects" interchangeably.

data, or by using 10-fold cross validation technique; 2) Cross Version prediction: In this approach, the data of the previous version of a software project is taken as the training data to build prediction model and it is tested on the data of current version of the same project; 3) Cross Project prediction: The prediction models are built by taking data from different projects as the training data and tested on the data of the project under study.

The classification and regression based machine learning algorithms are widely used to build defect prediction models. The classification models output whether a given file is buggy or not [5] [6] whereas regression techniques determine the number of bugs in a file [7]. There are many classification algorithms that are used in developing defect prediction models and the list includes Naïve Bayes Classifier, Logistic Regression, Decision Tree and ensemble classifiers like Random Forest algorithm. In recent past, defect prediction problem is also seen as an optimization problem and search-based techniques are applied to find near-optimal or "good-enough" solutions.

A wide range of software metrics have been developed as defect predictors over the years. The most widely used metrics for defect prediction models are source code metrics like CK metrics [8], object oriented metrics (Fan In, Fan Out, Number of Attributes, Number of Attributes Inherited, Number of Lines of Code, Number of Methods, Number of Methods Inherited, Number of Private Attributes, Number of Private Methods, Number of Public Attributes, Number of Public Methods), McCabe and the Halstead metrics [9] [10] and change metrics like code churn metrics etc. [11] or a combination of source code and change metrics [12] [6] [13].

Defect Prediction has generated wide spread interest in academic research. The interesting research problem is to develop ***effective*** defect prediction approaches so that the quality assurance team performs resource intensive, time consuming and high cost involved assurance activities on ***fewer*** segments of the system as against to the entire system. The main objectives of this thesis are listed below.

- Study the impact of feature selection on defect prediction models.

- Develop defect prediction models using various machine learning techniques and study the effectiveness of these models.

- Formulate defect prediction problem as a multi objective optimization problems with different objective functions and discuss the advantages of these models.

- Find out the probability distribution that software defects and metrics follow.

1.1 Thesis Overview

We present the overview of thesis in this section. There are four major chapters and they are described as follows.

CHAPTER 2 Several change metrics and source code metrics have been introduced and proved to be effective features in building defect prediction models. Defect prediction models are built using the individual metrics as well as combination of these metrics. Feature selection is a widely used preprocessing technique to enhance accuracy of prediction model and it achieves the same by selecting most appropriate features amongst all available features. We investigate whether the prediction accuracy of defect prediction models is improved by applying feature selection techniques. We explore if there is one algorithm amongst ten popular feature selection algorithms that consistently fares better than others across sixteen bench marked open source projects in Section 2.2. We investigate whether the same set of features be selected as the best features for all similar projects.

In Section 2.3, we investigate whether the changes that take place in source files over period of time contribute to defects. It is intuitive to think that the distribution of changes (commits) to a file within the timeline correlates to post-release defects. We propose couple of code change metrics around the distribution of changes and build defect prediction model with these new metrics.

CHAPTER 3 Naïve Bayes classifier has been widely used for building defect prediction models [14] [6]. Naïve Bayes classifier assumes conditional independence of all features while building learning models. As most of the features used in defect prediction models are not independent, we find it interesting to study whether prediction accuracies can be improved if conditional independence assumption is relaxed. In Section 3.2, we use three classes of algorithms to build Bayesian Networks namely score-based, constraint-based and hybrid class of algorithms. We propose an approach to augment these Bayesian Network structures. We have compared performance of the respective Bayesian Network classifiers before augmentation and after augmentation with popular classifiers like Random Forest, Naïve Bayes and Logistic Regression.

In Section 3.3, we build cost sensitive defect prediction models. In the recent past, quite a few studies have dealt with cost sensitive learning of software defect prediction models. And very few compare the performance of their cost sensitive defect prediction models with traditional machine learning approaches. As there is no comprehensive comparative study of cost sensitive defect prediction models, we attempt to conduct the comparative study to figure out the effectiveness of proposed Cost Sensitive Neural Networks for Software Defect Prediction. We use Normalized Expected Cost of Misclassification (NECM) [15] as the performance measure in our comparative study.

In Section 3.4, we perform cost-effectiveness analysis of defect prediction models. Thilo Mende et al. [16] proposed a strategy to include the notion of effort in defect prediction models. They propose to rank files with respect to their effort per bug and used McCabe's cyclomatic complexity as the surrogate measure for effort. In this work we argue that the measure of effort should not be a generic measure such as cyclomatic complexity but instead it should be one that is specific to the kind of activity involved in the quality assurance process. We identify two most popular quality assurance activities namely code review and unit testing. We use lines of code to measure the effort involved in code reviews and the number of test cases to measure the effort in case of unit testing. We compare the cost-effectiveness of our specific effort based models to generic effort based models.

CHAPTER 4 Cross version defect prediction is building a prediction model from the previous version of a software project to predict defects in the current version. This is more practical than the other two ways of building models, i.e., cross project prediction model and cross validation prediction models, as previous version of same software project will have similar parameter distribution among files. We formulate cross version defect prediction problem as a multi-objective optimization problem with two objective functions: (a) maximizing recall by minimizing misclassification cost and (b) maximizing recall by minimizing cost of QA activities on defect prone files. The two multi-objective defect prediction models are compared with four traditional machine learning algorithms, namely logistic regression, Naïve Bayes, decision tree and random forest.

CHAPTER 5 Object oriented metrics suite and past defects are extensively used to build defect prediction models with various machine learning approaches. Finding out the distributions followed by the past defects is a starting point to explore the stochastic process that the development and maintenance of software systems follow. Software defects that measure the quality of software systems provide means to the managers and developers to track and achieve the goals of software development. Though the primary goal of software development process is to implement the functional and non-functional requirements effectively and efficiently, the entire process has an apparent randomness though the resulting software system metrics have statistical regularities. If the generic model for bug distribution across projects and domains is known, quality assurance activities like testing and review of future releases can be planned efficiently. We study the distribution followed by defects and some popular software metrics.

CHAPTER 6 Conclusions and future work of this thesis is discussed in this chapter.

2 Feature Selection and Novel Change Metrics

2.1 Introduction

Feature selection is the process of selecting a subset of relevant features for building effective learning models. Feature selection has been an active field of research for the past three decades. It is extensively used in text categorization and gene selection problems where there are enormous numbers of features [17] [18]. The enormity of datasets usually results in scalability and performance issues while applying learning algorithms. Feature selection algorithms aim to resolve these issues by eliminating irrelevant or redundant features. However, they are also used to enhance the predictive accuracies and interpretability of models.

Applying a feature selection technique to find the appropriate feature subset prior to building defect prediction models is found to be useful [6]. A wide range of feature selection algorithms have been developed and more number of datasets are made available publicly over the years. In Section 2.2, we have studied the impact of various feature selection techniques on defect prediction models on publicly available datasets. We investigate whether a subset of all features under consideration improves prediction accuracies of learning model. We have also extended our study to find whether the relevant features of the projects that improve the prediction accuracy are consistent across similar projects.

Defect prediction models are built using static code attributes, change metrics and past defect data. Change metrics are found to be better predictors of defect prone files [19] [11].

In Section 2.3, we have proposed couple of novel change metrics. We have extracted the proposed metrics from Eclipse project hosted on Github and built prediction models to prove their significance.

2.2 Feature Selection

Over the years, defect prediction models have been built by considering all metrics under the study as features. There has been limited study in the direction of figuring out whether a subset of these metrics might improve the predictive accuracies of learning models as compared to model with all features. Krishnan et al. in [20] try to assess if there exists consistency amongst best features of change metrics across different versions of eclipse based projects. Shivaji et al. [21] apply feature selection techniques using Naïve Bayes and Support Vector Machine (SVM) classifiers to determine if there is an improvement in prediction accuracy of the models. They consider distinct lexemes in the churned source code as features along with source code complexity measures and features from change metadata.

In this work, we investigate whether a subset of all features under consideration improves prediction accuracies of learning model and deal with many other related interesting questions. There are feature selection algorithms in literature to find out optimal feature subset and we implement ten popular feature selection algorithms for our study. We have considered three machine learning algorithms as underlying algorithms to build defect prediction models. The three underlying machine learning algorithms that we have considered for this study are Naïve Bayes Classifier, Logistic Regression and Random Forest Classifier. We have made use of non-parametric measure, AUC (Area under Curve) of ROC curve, to compute the prediction accuracies of the defect prediction models. There are comparative studies in literature that show that change metrics are better predictors than source code metrics [22]. We consider combination of change metrics and source code metrics to check if the best feature subset has metrics only from change metrics or not.

We put forth four Research Questions in this study and experiments were conducted on

16 projects by implementing 10 feature selection algorithms with 3 underlying machine learning algorithms.

RQ1: Will prediction accuracy of defect prediction models be improved by implementing feature selection techniques?

RQ2: Amongst the existing feature subset selection algorithms, will there be algorithm(s) that offer best feature subset consistently across the projects?

RQ3: Will the feature selection algorithm that offers the best feature subset be independent of the underlying machine learning algorithms?

RQ4: Will the metrics in best feature subset be consistent across projects?

2.2.1 Related Work

Numerous software metrics have been introduced as potential predictors to detect bugs accurately. They can be broadly classified as source code metrics and change metrics. There is good amount of research that has been carried out using these metrics separately as well as combined. Some of the source code metrics include McCabe and Halstead metrics [10] [9], Chidamber-Kemerer (CK) metrics suite [8], the Conceptual Cohesion of Classes (C3) measure [23] and various file, component based metrics [24] [4]. Fenton and Neil [25] theorized that the most widely used static code features include Lines Of Code (LOC) based measures, Halstead metrics and McCabe complexity metrics.

Lessman et al. proposed a framework to compare different defect prediction models and applied it in a large-scale empirical comparison of 22 classifiers over 10 public domain data sets from the NASA Metrics Data repository (MDP) and PROMISE repository [5]. Classifiers are compared based on the area under the receiver operating characteristics curve (AUC). Further, hypothesis testing methods suggested by Demšar are used to validate the statistical significance of performance differences among different classification models. The results confirm the effectiveness of ensemble classifier Random Forest. However, no

significant performance differences could be detected among the top 17 classifiers. Our study follows this framework in comparing the performance of feature selection algorithms. Lessman et al. also suggest using pre-processing techniques such as feature selection and discretization of attributes to improve the performance of classifiers.

Yue Jiang et al. investigate the effectiveness of four pre-processing techniques in predicting defects by making use of nine datasets from NASA Metrics Data Programs (MDP) and ten classification algorithms [26]. The four pre-processing methods include the original data (i.e., without any transformation), log transformed data, discretized data, and discretization of the log transformed data. The results indicate that the impact of data transformations on prediction models built with distinct learning algorithms differs. Random forest algorithm, for example, performs better with original and log transformed data set whereas Boosting and Naïve Bayes perform significantly better with discretized data. It is concluded that no general benefit can be expected from data transformations. Also it is recommended to use transformation techniques specific to classification algorithms.

The outstanding results by exploring feature selection techniques in other fields, like gene selection and text categorization, inspire us to undertake this study. In bio-informatics, Huiqing Liu et al. [17] show that feature selection improves the classification accuracy significantly in their comparative study on feature selection and classification methods. Yang et al. in [18] compare feature selection algorithms in statistical learning of text categorization. They focus on aggressive dimensionality reduction using five different feature selection algorithms.

Ambros et al. present an extensive comparison of the explanative and predictive power of well-known defect prediction approaches over publicly available datasets [27]. Bug prediction models were built using source code metrics, change metrics, entropy of changes, churn of source code metrics, entropy of source code metrics and single metrics such as bug-fixes with and without feature reduction techniques like Principal Component Analysis (PCA). The results indicate that approaches based on churn and entropy of source code metrics have good, stable explanative and predictive power, better than all the other applied approaches. Also defect prediction approaches based on a single metric were observed to be unstable over the case studies.

Moser et al. presented a comparative analysis of the predictive power of two different sets of metrics for defect prediction. The two different sets of metrics comprised of 18 change metrics and 31 source code metrics. Cost-sensitive classification models were built for three releases of Eclipse. The results are very promising and change metrics clearly outperform predictors based on static code attributes for the Eclipse project [13]. Moser et al. analyzed the reliability of a subset of the above mentioned 18 change metrics for defect prediction and demonstrated that 3 out of 18 change metrics contain most information about software defects across three releases of Eclipse project [22]. The results also indicate that prediction accuracy is not affected much by using a subset of 3 metrics. It is worthwhile to note that their work is not towards identifying common predictors across projects but to find common predictors across different versions of the same project.

Krishnan et al. find that change metrics are consistently good and incrementally better predictors across the evolving products in Eclipse. Their work tries to explore whether the set of good predictors change over time for one product and whether the set differs among products [20]. The data sets for their experiments are sequential releases of various products in a product line. They conclude that a small subset of these change metrics is fairly stable and consistent across products and releases. And we consider change and source code metrics in this work to corroborate this point.

At the same time Menzies et al. argue that static code attributes or source code metrics also have significant role in prediction of faults and identify the best predictors among source code metrics [6]. They try to define a baseline experiment by choosing publicly available datasets for the study. Their results indicate that the static code attributes with log filtering pre-processor on the numeric data outperform models without any pre-processing. This work demonstrates the superior performance of models using preprocessing techniques.

Most of the researchers in this field concentrate their efforts to predict whether a source file or binary is defect prone or not. But Shivaji et al. predict whether a change request is defect prone or not based on history of change requests [21]. They consider distinct lexims in the churned source code, which are quite huge in number, as features and extract them from churned source code by bag-of-words approach (BOW) [28]. They also consider other

Table 2.1: CK & OO Metrics

CK Metrics	
WMC	Weighted Methods per Class
DIT	Depth of Inheritance Tree
RFC	Response For Class
NOC	Number Of Children
CBO	Coupling Between Objects
LCOM	Lack of Cohesion in Methods
OO Metrics	
Fan-In	Number of classes that reference the class
Fan-Out	Number of classes referenced by the class
NOA	Number of attributes
NOPA	Number of public attributes
NOPRA	Number of private attributes
NOAI	Number of attributes inherited
LOC	Number of lines of code
NOM	Number of methods
NOPM	Number of public methods
NOPRM	Number of private methods
NOMI	Number of methods inherited

features from change metadata, source code complexity metrics. They applied feature selection algorithm on all these features to build defect prediction model for change requests and show that feature selection makes a huge improvement in prediction accuracy.

2.2.2 Datasets

We conducted our experiments on 16 projects and metrics of all these projects are described in this section. Bug prediction models are built with features as metrics in the project. Eleven of these projects are NASA projects and the rest are Eclipse based projects.

- NASA Datasets

  The data used in this study comprises of the datasets in NASA MDP repository [29]. These include MC1, MC2, MW1, CM1, JM1, KC1, KC3, PC1, PC3, PC4 and PC5. Besides LOC counts, the NASA MDP data sets include several Halstead attributes as well as McCabe complexity measures. Halstead attributes estimate reading complexity by counting operators and operands in a module, whereas the McCabe

complexity measures are derived from a module's flow graph. The description of these code metrics and the origin of the MDP data sets are detailed in [10] and [9]. The metrics of all these eleven projects are detailed in Table 2.3. All metrics are not available for each project and we represent the availability of the metrics for a given project by a matrix and the same is represented in Table 2.3.

Table 2.2: Change Metrics

Metrics	**Definition**
REVISIONS	Number of revisions of a file
REFACTORINGS	Number of times a file has been Refactored
BUGFIXES	Number of times a file was involved in bug-fixing
AUTHORS	Number of distinct authors that checked a file into the repository
LOC_ADDED	Sum over all revisions of the lines of code added to a file
MAX_LOC_ADDED	Maximum number of LOC added for all revisions
AVE_ LOC_ADDED	Average lines of code added per revision
LOC_DELETED	Sum over all revisions of the lines of code deleted from a file
MAX_ LOC_DELETED	Maximum number of lines of code deleted for all revisions
AVE_ LOC_DELETED	Average lines of code deleted per Revision
CODECHURN	Sum of (added lines of code − deleted lines of code) over all revisions
MAX_ CODECHURN	Maximum CODECHURN for all Revisions
AVE_ CODECHURN	Average CODECHURN per revision
AGE	Age of a file in weeks (counting backwards from a specific release)
WEIGHTED_AGE	$Age(i)$ is the number of weeks starting from the release date for revision i and $LOC\_ADDED(i)$ is the number of lines of code added at revision i.

- Other Datasets

We also consider other publicly available datasets of five Java based open-source software systems. These consist of both source code and change metrics. The five systems include Eclipse JDT Core, Eclipse PDE UI, Equinox Framework, Mylyn and Apache Lucene. CK Metrics [8] and object oriented metrics have been conside-

Table 2.3: Source Code Metrics for NASA projects

Category	Metrics	CM1	KC1	KC3	MC1	MC2	MW1	JM1	PC1	PC3	PC4	PC5
LOC based metrics	LOC_Total	X	X	X	X	X	X	X	X	X	X	X
	LOC_Blank	X	X	X	X	X	X	X	X	X	X	X
	LOC_Executable	X	X	X	X	X	X	X	X	X	X	X
	LOC_Comments	X	X	X	X	X	X	X	X	X	X	X
	LOC_Code_and_Comments	X	X	X	X	X	X	X	X	X	X	X
	Number_of_lines	X		X	X	X	X		X	X	X	X
	Percent_Comments	X		X	X	X	X		X	X	X	X
Halstead Metrics	Content	X	X	X	X	X	X	X	X	X	X	X
	Difficulty	X	X	X	X	X	X	X	X	X	X	X
	Programming_Effort	X	X	X	X	X	X	X	X	X	X	X
	Error_Estimate	X	X	X	X	X	X	X	X	X	X	X
	Length	X	X	X	X	X	X	X	X	X	X	X
	Level	X	X	X	X	X	X	X	X	X	X	X
	Programming_Time	X	X	X	X	X	X	X	X	X	X	X
	Volume	X	X	X	X		X	X	X	X	X	X
	Num_Operands	X	X	X	X	X	X	X	X	X	X	X
	Num_Operators	X	X	X	X	X	X	X	X	X	X	X
	Num_Unique_Operands	X	X	X	X	X	X	X	X	X	X	X
	Num_Unique_Operators	X	X	X	X	X	X	X	X	X	X	X
McCabe Metrics	Cyclomatic_Complexity	X	X	X	X	X	X	X	X	X	X	X
	Cyclomatic_Density	X		X	X	X	X		X	X	X	X
	Decision_Density	X		X		X	X		X	X	X	
	Design_complexity	X	X	X	X	X	X	X	X	X	X	X
	Design_Density	X		X	X	X	X		X	X	X	X
	Essential_Complexity	X	X	X	X	X	X	X	X	X	X	X
	Essential_Density	X		X	X	X	X		X	X	X	X
	Global_Data_Complexity			X	X	X						X
	Global_Data_Density			X	X	X						X
	Normalized_Cyclomatic_Complexity	X		X	X	X	X		X	X	X	X
	Maintenance Severity	X		X	X		X		X	X	X	X
Miscella-neous	Branch_Count	X	X	X	X	X	X	X	X	X	X	X
	Call_Pairs	X		X	X	X	X		X	X	X	X
	Condition_Count	X		X	X	X	X		X	X	X	X
	Decision_Count	X		X	X	X	X		X	X	X	X
	Edge_Count	X		X	X	X	X		X	X	X	X
	Node_Count	X		X	X	X	X		X	X	X	X
	Parameter_count	X		X	X	X	X		X	X	X	X
	Multiple_Condition_Count	X		X	X	X	X		X	X	X	X
	Modified_Condition_Count	X		X	X	X	X		X	X	X	X

red under source code metrics category and Table 2.1 describes details about these metrics. We have used some of the change metrics that are used by Moser et al. for our experiments [11] [30]. All these metrics are described in Table 2.2.

2.2.3 Feature Selection Algorithms

Feature selection is primarily the task of choosing a subset of features that enhances prediction accuracy of learning models and decreases their execution time. It is also an effective means to identify relevant features for dimensionality reduction. We have imple-

mented ten feature selection algorithms to study research questions RQ1 through RQ4. All these algorithms have been implemented in MATLAB by Arizona State University*. We have made use of these packages for our proposed work.

Feature selection algorithms can be classified into two kinds of algorithms namely Greedy based feature selection and Rank based feature selection algorithms. The brief description of these algorithms is discussed below.

- **Rank based feature selection algorithms**: Every feature of learning problem inherently contains some information about the target variable and this has been clearly explored by statistical measures like Information Gain, Gini Index etc. Feature ranking algorithms initially sort all attributes with respect to a statistical measure. Models are built with top 1 feature, top 2 features, . . , top n features. We divide the data randomly into two parts, namely training data and testing data containing 70% and 30% of total data respectively. Models are built with training data and the performance measure like AUC will be calculated by validating the model against testing data. We conduct this experiment for five times by randomly selecting training data, testing data and average of five runs is considered as the AUC for the model. The features in the model, that achieves the maximum AUC value, will be considered as best feature subset. Figure 2.1 depicts graph for Mylyn dataset using Naïve Bayes classifier. Y axis represents AUC values of the models and X axis depicts the models having feature subset containing top 1 feature, top 2 features, . . , top n features. The model with highest accuracy is the best one and those features constitute the best subset. The maximum AUC value is 0.774 and the corresponding features comprise the best subset.

  We use Information Gain, Relief, CFS, Chi-squared, Gini Index, Kruskal Wallis and Fisher Score measures for our study [31] [32].

- **Greedy based feature selection algorithms**: There are two greedy algorithms namely Greedy Forward Selection Algorithm and Greedy Backward Selection Algorithm to find out the optimal feature subset.

*http://featureselection.asu.edu

Figure 2.1: AUC vs # of Feature – Naïve Bayes Classifier – Mylyn Project

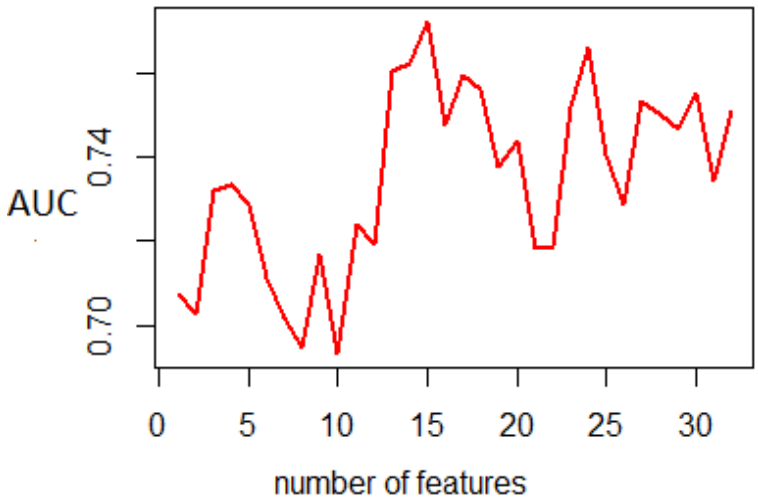
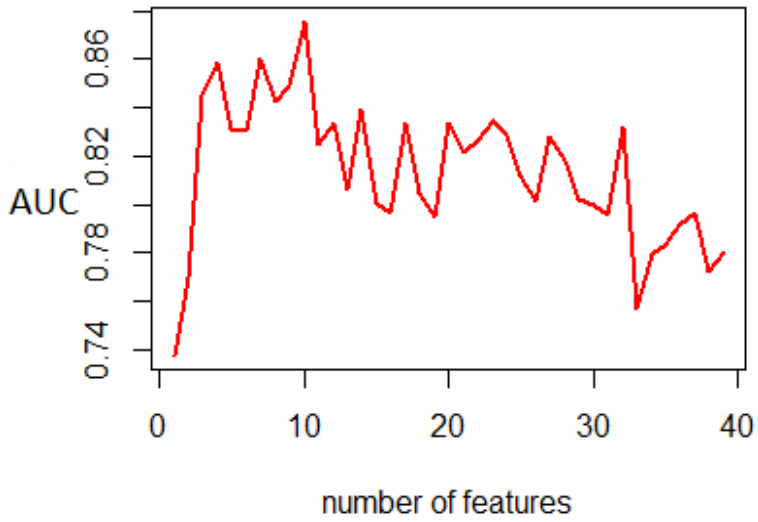


Figure 2.2: AUC vs # of Feature – Greedy Forward Selection Algorithm – KC3 Project



Greedy Forward Selection Algorithm: Prediction models are built by considering one feature at a time. The model with best prediction accuracy (in this case, AUC) is chosen. Subsequently, the selected feature is combined with the remaining features one at a time. The model with best prediction accuracy is chosen again. This process is further repeated until a model comprising all features is built. The feature subset with maximum prediction accuracy is chosen as the best feature subset. The prediction accuracy of model will be calculated as average of AUCs of five runs as explained previously in ranking selection algorithms. Figure 2.2 depicts the execution of forward greedy selection algorithm for KC3 dataset comprising of 39 features. Y axis represents the AUC value of each model and X axis represents the model

with corresponding number of features where in Random Forest Classifier is the underlying machine learning algorithm. The algorithm as stated above builds models until all the features are included. The model with maximum AUC value is chosen. In this particular case, the maximum value of AUC is 0.876 which occurs when the number of features selected is 10.

Greedy Backward Selection Algorithm: This algorithm is similar to greedy forward selection except that it begins by considering all the features as best subset. First, a model is built by considering all the features. Next, models are built by removing one feature at a time from all the features. The model with best prediction accuracy is chosen (in this case, AUC). This is repeated until all the features are exhausted. Out of all the models built, the model with best prediction accuracy is chosen and the corresponding features comprise the best feature subset.

- We also make use of the following feature selection algorithm in our study
  BlogReg Feature Selection Algorithm: This algorithm is an embedded feature selection algorithm presented in [33] to eliminate the regularization parameter λ from Sparse Logistic Regression SLogReg, which is proposed in [34]. BLogReg uses a Bayesian approach to remove λ by maximizing the marginalized likelihood of the sparse logistic regression model.

2.2.4 Experimental Methodology

We have conducted experiments on the datasets presented in Section 2.2.2 using the aforementioned feature selection algorithms detailed in Section 2.2.3. We describe machine learning algorithms used in our study and measures that are used to assess the performance of the prediction models in this Section.

2.2.4.1 Learning Algorithms

We implement machine learning algorithms to find the best feature subset as discussed in Section 2.2.3. There are numerous machine learning algorithms amongst which we have chosen the following three algorithms.

1. Naïve Bayes Classifier (NB)

2. Logistic Regression (LR)

3. Random Forest (RF)

The reasons for the selection of these algorithms are discussed below. Naïve Bayes model is extensively used in defect prediction and it is easy to interpret as well as computationally efficient model as pointed by Menzies et al. [6]. Random Forest is an effective, widely used ensemble classifier for software defect predictions [32]. It is fast to train and is robust toward parameter settings. Also it is consistent and demonstrates substantial improvement over individual tree classifiers [35].

Logistic regression is an equally common technique used especially in building defect prediction models [36]. Ease in interpretation of its coefficients, wide availability of easily used and reliable software to perform the computations and the ability to estimate the probability of an outcome are a few reasons behind its extensive usage.

2.2.4.2 Evaluation Criterion

The aim of defect prediction model is to determine if a module is defect prone or not. And the confusion matrix is generally calculated indicating number of True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN). Using these indicators, measures such as precision, accuracy and recall are calculated. However, these measures are parametric and are valid only for a certain operating point. An operating point is a

value or threshold based on which we classify the samples as buggy or not. For example, in a binary model implementing logistic regression if the operating point is 0.5, all samples with probability greater than 0.5 are classified as positive and others as negative. It is also argued that even though such error-based metrics have undoubted practical value, they are conceptually inappropriate for empirical comparisons of the performance of classification algorithms [5].

A ROC (Receiver Operating Characteristic) curve is a plot with false positive rate (false alarm) on the X axis and the true positive rate on Y axis. (0, 1) is the ideal point as there would be no misclassifications. A straight line from (0,0) to (1,1) is a completely random classifier and offers no information. Any curve above this straight line represents a good classifier (better than random). The advantages of the ROC analysis are its robustness toward imbalanced class distributions and to varying and asymmetric misclassification costs [30]. Due its robustness and being non-parametric measure, we have made use of AUC as prediction accuracy for our models. Out of the several measures associated with ROC analysis, area under the curve (AUC) is one of the most popular measures. It can take values ranging from zero to one, and an AUC value of one represents a perfectly accurate classifier. An AUC value greater than 0.5 demonstrates an effective classifier which performs better than a random classifier. Statistically, it measures the probability that a classifier ranks a randomly chosen fault prone module higher than a randomly chosen non-fault prone module [37].

2.2.4.3 Ranking Prediction Models

With number of new machine learning algorithms being presented, methods to compare their performance have become quite popular. Many of the studies make use of statistical inference to compare various algorithms.

With regard to the current practices employed in comparing performance of algorithms, Demšar reviewed theoretically and empirically several suitable tests. He recommends a set of simple and robust non-parametric tests for statistical comparisons of classifiers: the Wilcoxon signed ranks test for comparison of two classifiers and the Friedman test

with the corresponding post-hoc tests for comparison of more classifiers over multiple data sets. Results of the Friedman test are called CD (critical difference) diagrams [38]. Friedman's test is based on ranked performances rather than actual performance estimates and is therefore less susceptible to outliers. All classifiers are ranked according to their performance in ascending order for each data set and the mean rank of a classifier i, AR_i is computed across all data sets. The corresponding test is calculated as

$$\chi_F^2 = \frac{12K}{L(L+1)} \left[\sum_{i=1}^{L} AR_i^2 - \frac{L(L+1)^2}{4} \right] \tag{2.1}$$

$$AR_i = \frac{1}{K} \sum_{j=1}^{K} r_j^i \tag{2.2}$$

with K representing the overall number of data sets, L the number of classifiers, and r_j^i the rank of classifier i on data set j. The null hypothesis for this test is that all algorithms perform alike. If the null hypothesis is satisfied, the test statistic is distributed according to the Chi-Square distribution with $L - 1$ degrees of freedom. If the null hypothesis is rejected, Demšar recommends the test of Nemenyi for comparing the performance of all classifiers with each other. The performance of two classifiers is significantly different if the corresponding average ranks differ by at least the critical difference

$$CD = q_{a;\infty;L} \sqrt{\frac{L(L+1)}{12K}} \tag{2.3}$$

The value $q_{a;\infty;L}$ is based on the Studentized range statistic which can be looked up in standard statistical tables. We adopt the above mentioned methodology to answer our problems RQ1 through RQ4.

2.2.5 Results and Discussion

The primary focus of this work is to study the impact of feature selection on defect prediction. As discussed in Section 2.2.3, we have considered ten feature selection algorithms to study the research questions RQ1 through RQ4. For each of these ten algorithms, we build defect prediction models with the features as selected by the algorithm. Thus for each of the project and underlying learning algorithm, we build ten defect prediction models and one more defect prediction model that considers all features. As per Section 2.2.4.2, we will make use of Area Under ROC Curve (AUC) as prediction accuracy to determine the merit of each model. The project data is randomly partitioned where in 2/3rd of data is for training the prediction model and remaining 1/3rd of data is used for testing the model. And this is repeated for five times and average of prediction accuracies is taken as the final value of the prediction accuracy for the project. We will explore Demšar tests and simple statistical measures to answer research questions laid down in Section 2.2.

Table 2.4: Prediction Accuracy (AUC) of all projects with Naïve Bayes Classifier as underlying algorithm

Project	Relief	Info Gain	Forward Greedy	Backward Greedy	CFS	Chi-Sq	Blog Reg	Gini Index	Kruskal-Wallis	Fisher Score	All Features
JDT	0.818	**0.828**	0.826	0.818	0.715	0.816	0.724	0.801	0.791	0.805	0.769
PDE	0.736	0.776	**0.798**	0.768	0.728	0.767	0.726	0.768	0.752	0.759	0.728
Equinox	0.845	0.865	0.881	**0.886**	0.829	0.848	0.818	0.848	0.861	0.852	0.85
Lucene	0.836	0.834	**0.849**	0.831	0.754	0.802	0.778	0.807	0.81	0.807	0.767
Mylene	0.784	0.772	**0.818**	0.805	0.759	0.77	0.755	0.77	0.769	0.771	0.723
CM1	0.794	0.802	**0.826**	0.785	0.766	0.771	0.708	0.785	0.757	0.773	0.698
JM1	0.697	0.685	**0.7**	0.699	0.654	0.691	0.651	0.688	0.693	0.69	0.679
KC1	0.8	0.804	0.822	**0.826**	0.788	0.824	0.776	0.822	0.822	0.823	0.786
KC3	0.75	0.843	**0.85**	0.826	0.797	0.784	0.738	0.807	0.768	0.795	0.694
MC1	0.924	0.934	**0.947**	0.942	0.893	0.931	0.831	0.937	0.929	0.935	0.921
MC2	0.784	0.811	**0.859**	0.843	0.699	0.775	0.707	0.798	0.778	0.773	0.725
MW1	0.852	0.843	**0.883**	0.871	0.797	0.806	0.719	0.807	0.786	0.785	0.802
PC1	0.833	0.823	**0.843**	0.839	0.772	0.809	0.737	0.832	0.816	0.817	0.79
PC3	0.829	0.801	**0.841**	0.834	0.801	0.822	0.825	0.825	0.822	0.822	0.761
PC4	0.848	0.853	0.888	**0.896**	0.788	0.852	0.87	0.853	0.854	0.853	0.806
PC5	0.95	0.956	**0.965**	0.963	0.95	0.951	0.936	0.95	0.947	0.962	0.948

We have conducted experiments on data sets of 16 projects by implementing 10 feature selection algorithms with 3 underlying machine learning algorithms. We have tabulated the results of 16 projects with three different underlying machine learning algorithm as (Naïve Bayes Classifier, Logistic Regression and Random forest) shown in Table 2.4, Table 2.5 and Table 2.6. For example, if prediction models are built with all features, features selected by

forward greedy algorithm and backward greedy algorithm wherein the underlying machine learning algorithm is Naïve Bayes Classifier, AUC values for PC3 project are 0.761, 0.841 and 0.834 respectively (refer to Table 2.4).

Table 2.5: Prediction Accuracy (AUC) of all projects with Logistic Regression as underlying algorithm

Project	Relief	Info gain	Forward Greedy	Backward Greedy	CFS	Chi-Sq	Blog Reg	Gini Index	Kruskal-Wallis	Fisher Score	All Features
JDT	0.86	0.831	0.852	**0.863**	0.753	0.831	0.78	0.839	0.837	0.837	0.826
PDE	0.771	0.756	**0.791**	0.775	0.74	0.757	0.764	0.773	0.769	0.782	0.731
Equinox	0.86	0.847	**0.874**	0.871	0.829	0.847	0.831	0.845	0.845	0.871	0.757
Lucene	0.83	0.809	0.867	**0.868**	0.725	0.809	0.733	0.809	0.809	0.81	0.756
Mylene	0.824	0.819	0.834	**0.84**	0.784	0.819	0.783	0.817	0.815	0.821	0.794
CM1	0.857	0.811	**0.857**	0.846	0.765	0.811	0.771	0.836	0.788	0.819	0.756
JM1	0.714	0.717	**0.72**	0.719	0.706	0.717	0.707	0.716	0.718	0.718	0.708
KC1	0.814	0.808	0.822	**0.823**	0.789	0.806	0.789	0.812	0.811	0.82	0.79
KC3	0.787	0.765	0.859	**0.86**	0.803	0.765	0.812	0.77	0.768	0.793	0.686
MC1	0.925	0.945	0.952	**0.958**	0.905	0.938	0.832	0.918	0.913	0.928	0.866
MC2	0.739	0.701	0.808	**0.833**	0.741	0.726	0.743	0.754	0.776	0.732	0.69
MW1	0.797	0.808	**0.867**	0.858	0.806	0.808	0.752	0.831	0.796	0.798	0.75
PC1	0.875	0.856	0.884	**0.886**	0.793	0.857	0.805	0.867	0.854	0.869	0.767
PC3	0.827	0.842	**0.867**	0.86	0.816	0.842	0.825	0.842	0.842	0.842	0.85
PC4	0.903	0.906	0.913	**0.919**	0.85	0.906	0.881	0.913	0.885	0.913	0.906
PC5	0.958	0.956	**0.965**	**0.965**	0.958	0.956	0.952	0.956	0.955	0.957	0.945

Table 2.6: Prediction Accuracy (AUC) of all projects with Random Forst as underlying algorithm

Project	Relief	Info gain	Forward Greedy	Backward Greedy	CFS	Chi-Sq	Blog Reg	Gini Index	Kruskal-Wallis	Fisher Score	All Features
JDT	0.879	0.881	**0.894**	0.894	0.817	0.872	0.794	0.874	0.879	0.882	0.843
PDE	0.799	0.796	**0.818**	0.806	0.77	0.796	0.773	0.787	0.789	0.811	0.779
Equinox	0.872	0.883	0.889	**0.897**	0.866	0.875	0.826	0.877	0.886	0.881	0.855
Lucene	0.835	0.838	**0.877**	0.866	0.792	0.841	0.754	0.837	0.867	0.818	0.817
Mylene	0.842	0.841	0.849	0.848	0.823	0.832	0.805	0.837	**0.854**	0.838	0.828
CM1	0.813	0.786	**0.855**	0.845	0.761	0.78	0.705	0.76	0.766	0.767	0.767
JM1	0.761	0.763	**0.765**	0.762	0.735	0.753	0.745	0.759	0.755	0.761	0.754
KC1	0.829	0.847	**0.854**	0.848	0.795	0.844	0.74	0.834	0.848	0.837	0.827
KC3	0.798	0.797	**0.876**	0.852	0.758	0.788	0.81	0.8	0.816	0.787	0.781
MC1	0.967	0.948	0.947	**0.973**	0.785	0.962	0.673	0.96	0.964	0.961	0.942
MC2	0.819	0.793	**0.832**	0.828	0.731	0.757	0.73	0.758	0.744	0.745	0.719
MW1	0.78	0.792	**0.852**	0.846	0.672	0.829	0.745	0.782	0.829	0.766	0.739
PC1	0.869	0.895	**0.904**	0.899	0.823	0.884	0.816	0.878	0.877	0.866	0.851
PC3	0.864	0.863	0.868	**0.874**	0.816	0.87	0.716	0.855	0.845	0.854	0.827
PC4	0.948	0.948	0.956	**0.96**	0.841	0.944	0.91	0.947	0.948	0.945	0.942
PC5	**0.982**	0.978	0.981	0.981	0.947	0.979	0.956	0.981	0.98	0.977	0.975

The averages of AUCs of NASA and Eclipse projects for each of feature selection algorithm are tabulated in Table 2.7. For example, if prediction models are built with all features, features as selected by forward greedy algorithm and backward greedy algorithm wherein the underlying machine learning algorithm is Naïve Bayes Classifier, the average of AUCs for NASA projects are 0.783, 0.857 and 0.848 (refer to Table 2.7).

For a given project, the AUCs of models built with features that are outputs of feature selection algorithms are going to be distinct. For each project, the best AUC may be attained by a model built using features obtained by implementing a certain feature selection algorithm. The averages of best AUCs of NASA projects (best refer to the best feature selection algorithm for that project) is calculated and recorded in Table 2.7 as the value of 'Best Algo'. For example, this value is 0.858 (refer to Table 2.4) if the underlying machine learning algorithm is Naïve Bayes Classifier. In addition to average AUC for each model, this table also represents the percentage of Relative Differences (RD%) in AUCs of model implementing a particular feature selection technique and model comprising all features. In other words, it indicates the performance of each feature selection algorithm compared to model comprising all features.

Table 2.7: Comparison of average AUCs for NASA and Eclipse Projects

Projects	Eclipse						NASA					
Algorithms	NB		LR		RF		NB		LR		RF	
Statistical Measure	AUC	RD†%	AUC	RD%	AUC	RD%	AUC	RD%	AUC	RD%	AUC	RD%
All features	0.767	**NA**	0.773	**NA**	0.824	**NA**	0.783	**NA**	0.79	**NA**	0.829	**NA**
Best Algo	0.836	8.996	0.847	9.573	0.868	5.34	0.858	9.579	0.869	10	0.884	6.634
Forward Greedy	0.834	8.735	0.844	9.185	0.865	4.976	0.857	9.451	0.865	9.494	0.881	6.273
Backward Greedy	0.821	7.04	0.844	9.185	0.862	4.612	0.848	8.301	0.866	9.62	0.879	6.031
Relief	0.804	4.824	0.829	7.245	0.846	2.67	0.824	5.236	0.836	5.823	0.857	3.378
Information Gain	0.815	6.258	0.812	5.045	0.848	2.913	0.832	6.258	0.829	4.937	0.855	3.136
CFS	0.757	-1.304	0.766	-0.906	0.814	-1.214	0.791	1.022	0.812	2.785	0.788	-4.946
Chi-Squared	0.801	4.433	0.812	5.045	0.843	2.306	0.82	4.725	0.83	5.063	0.854	3.016
BLogReg	0.76	-0.913	0.778	0.647	0.79	-4.126	0.773	-1.277	0.806	2.025	0.777	-6.273
GiniIndex	0.799	4.172	0.817	5.692	0.843	2.306	0.828	5.747	0.838	6.076	0.847	2.171
Kruskal-Wallis	0.796	3.781	0.815	5.433	0.855	3.762	0.816	4.215	0.828	4.81	0.852	2.774
Fisher score	0.799	4.172	0.824	6.598	0.846	2.67	0.821	4.853	0.835	5.696	0.842	1.568

For each underlying machine learning algorithm, we have performed Friedman's test to check the null hypothesis whether there is significant difference in the prediction accuracies of 11 models (ten models resulting from 10 feature selection algorithms and another model built by considering all features) on 16 projects. It is found that null hypothesis is rejected and hence the Nemenyi test, that compares the performance of all models with each other, is conducted with an alpha value of 0.05. The Nemenyi test plots the algorithms against their mean ranks, with the ranks on X axis and algorithms on Y axis. The length of line segment of each classifier represents its corresponding critical difference. Any algorithm which starts to the right of the line segment is outperformed significantly. The results of these tests are shown using Nemenyi diagrams in Figure 2.3, Figure 2.5a and Figure 2.5b.

Figure 2.3: Nemenyi Diagram for Naïve Bayes

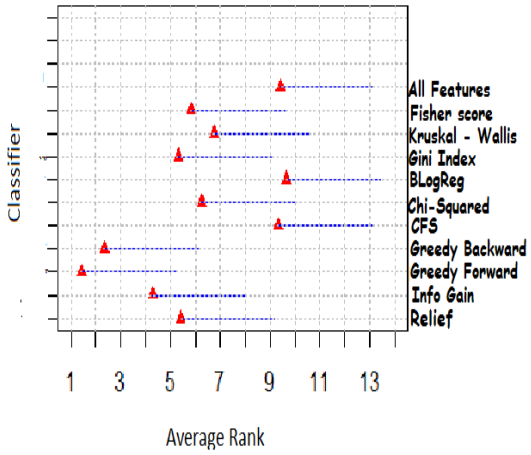
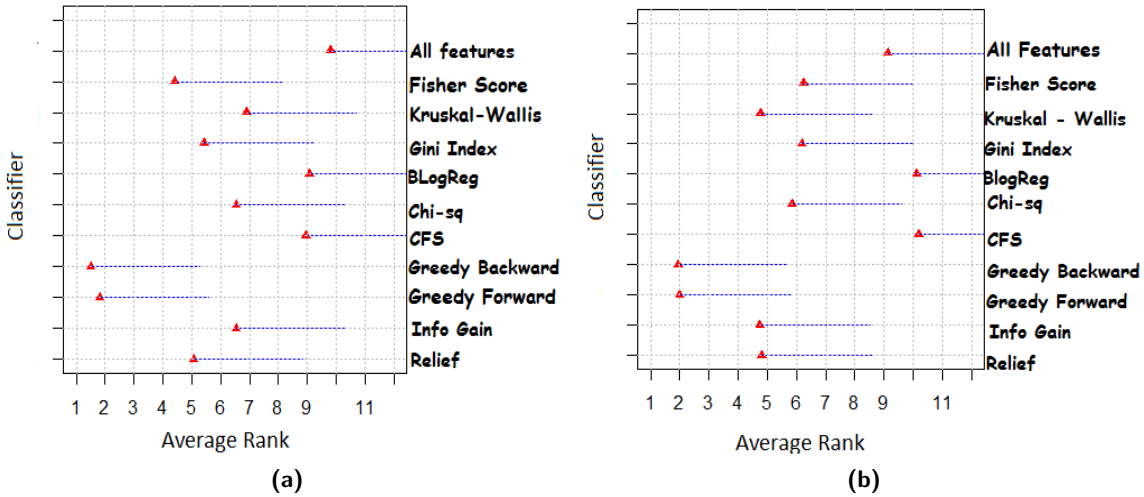


Figure 2.4: (a)Nemenyi Diagram for Logistic Regression, (b) Nemenyi Diagram for Random Forest



RQ1: Will prediction accuracy of defect prediction models be improved by implementing feature selection techniques?

Our experimental results are in tune with the intuitive idea that feature selection techniques improve predictive accuracy of defect prediction models. For KC3 project, AUC values for the prediction models built with all features and features obtained by applying the greedy forward selection algorithm are 0.694 and 0.850 (refer to Table 2.4), there is an increase of 15% in prediction accuracy. And there is an increase of 7.4% on Average AUC for all of NASA projects when greedy forward feature selection algorithm is implemented (refer to Table 2.7). However, from Table 2.7 we observe that few feature selection techniques namely CFS and BLogReg negatively impact the performance of prediction accuracies in some cases. For example, the prediction accuracy is shrunk by 6% when the

underlying machine learning algorithm is Random Forest; BLogReg is the feature selection technique and the experiments are conducted on NASA data sets.

Nemenyi tests also reveal that there is significant improvement in AUCs if forward greedy, backward greedy, gini index, information gain and relief based feature selection algorithms are implemented to select features optimally (refer to Figure 2.3). It is to be noted that all the above observations are from the experiments in which the underlying learning algorithm to build prediction model is Naïve Bayes classifier. Similar results are observed when the underlying learning algorithm is logistic regression and random forest (refer to Table 2.5, Table 2.6, Table 2.7, Figure 2.5a and Figure 2.5b). Hence we recommend performing best feature subset selection as part of pre-processing activities to build defect prediction model to enhance performance accuracies.

RQ2: Amongst the existing feature subset selection algorithms, will there be algorithm(s) that offer best feature subset consistently across the projects?

This question is whether there exists any feature selection algorithm that consistently outputs best feature subset across projects. We observed that prediction models built using the features obtained by implementing forward greedy feature selection algorithm are having maximum AUC values in 13 projects out of 16 projects. And the second consistent algorithm happens to be the backward greedy feature selection algorithm (refer to Table 2.4, Table 2.5, Table 2.6 and Table 2.7). For all of NASA projects, the average AUC value for the prediction model built by features obtained by implementing forward greedy and backward greedy algorithms are 0.857 and 0.848 whereas average AUC for prediction model built with all features is 0.783. And in fact these two algorithms top the list (refer to Table 2.7). And also the average AUC value for the models built by features obtained by implementing the forward greedy feature selection algorithm is almost the same as average of best AUC of NASA projects (best refer to the best feature selection algorithm for that project).

The Nemenyi's tests also confirm these observations and declare that forward greedy and backward greedy algorithms are consistently giving the bests features across projects. It is to be noted that all the above observations are from the experiments in which the underlying learning algorithm to build prediction model is Naïve Bayes. Similar results are observed when the underlying learning algorithm is logistic regression and random

forest (refer to Table 2.5, Table 2.6, Table 2.7, Figure 2.5a and Figure 2.5b). Hence we strongly recommend implementing forward greedy or backward greedy feature selection algorithms as compared to any other feature selection algorithms.

RQ3: Will the feature selection algorithm that offers the best feature subset be independent of the underlying machine learning algorithms?

We have observed that for NASA projects, forward greedy feature selection algorithm is found to be the best feature selection algorithm if the underlying machine leaning algorithm is Naïve Bayes algorithm. And the question here is that will it continue to be the best feature selection algorithm if the underlying machine algorithm is logistic regression or random forest. After conducting experiments with underlying learning algorithms as logistic regression and random forest, forward greedy feature selection algorithm continues to the best feature selection algorithm. Hence we conclude that the best feature selection algorithm is independent of the underlying algorithm.

RQ4: Will the metrics in best feature subset be consistent across projects?

The fourth question is whether the metrics in the best feature subset will be consistent across the similar projects, say NASA projects or not. Equivalently the features, which are found in the best feature subset for one of the NASA projects, say CM1, are there in the best feature subset of other NASA projects as well. We found that this is not totally correct as there are very few metrics that are common amongst the best features subsets of NASA projects and similar observations are found in eclipse based projects also.

2.2.6 Threats to Validity

Like any other empirical experiment, the conclusions of this study may be biased on the data used in obtaining them. Sampling bias is a valid threat to this study. However, the appropriateness and representativeness of NASA MDP repository has been appreciated by researchers in this area and they have made use of some of its data sets for their experiments (e.g. [6] [39] [40]). And hence we are confident that the obtained results on these data sets are relevant and valuable for the defect prediction community. The class imbalance is a common threat to most of the studies on defect prediction problem and this work is no exception.

2.2.7 Contributions

An extensive empirical study on the impact of feature selection algorithms on defect prediction models has been carried out and it is established that prediction accuracies can be enhanced significantly by applying feature selection techniques during pre-processing steps. Amongst many feature selection algorithms, greedy based algorithms stand out and perform much better than the other competent algorithms. It is also found out that the impact of feature selection algorithm is independent of the underlying machine learning algorithms. It is re-emphasized that there may not be any common metrics amongst optimal feature subsets of similar projects considered defect prediction.

2.3 Novel Change Metrics

Software development teams continue to deliver software releases in evolutionary fashion with almost always changing requirements. Changes are inevitable in constantly changing business environment. Some of the recent studies in defect prediction focused on using different change metrics as defect predictors. Change metrics clearly outperform code complexity and other source code metrics in defect prediction [13].

Hence, we have investigated whether the changes that take place in source files over period of time contribute to defect prone source files. We have built defect prediction models with existing and proposed change metrics mined from open source softwares hosted on Github. With the proposed metrics, we expect significant improvement in the performance of prediction models.

2.3.1 Related Work

Kim et al. proposed FixCache model that uses past bugs to search the vicinity of future bugs with the assumption that faults do not occur in isolation, but rather in bursts of several related faults [41]. Zimmermann et al. mapped defects from the bug database

of Eclipse to source code locations and built prediction models that showed that the combination of complexity metrics can predict defects, suggesting that the more complex the code, more bugs it has [42]. Studies done by Hassan shows that change metrics are as good as or even better predictor of faults than prior faults and he concludes that the more complex changes to a file, higher the chance that file contains faults [19]. Nagappan et al. found that change burst metrics yield excellent predictive capability in projects with high-quality changes. They claim that precision and recall exceed 90% for Windows Vista: the highest predictive power ever observed [11]. Nagappan and Ball took code churn measures to build defect prediction model and found out that relative code churn measures are excellent predictors of defect density in large industrial software systems [43]. Moser et al. shows that change metrics clearly outperform predictors based on static code attributes for the Eclipse project [13] [27]. They confirm the observations made by other researchers, as change data, and more in general process related metrics, contain more discriminatory and meaningful information about the defect distribution in software than the source code itself. They suggest that while most of the past research effort has been invested in code metrics based approaches and only produced mixed results, there remains much more to be explored in the area of how the software process impacts the generation of defects during the software development life cycle.

2.3.2 Change Metrics

Unlike object oriented metrics and CK metrics, change metrics do not concern themselves with the contents of a source file alone, but rather with the details of changes made to the source file over time. Change metrics are gathered from the software configuration management repository of a particular project. One advantage of using change metrics over source metrics is that they are independent of the programming language. Change metrics are extracted for the time period between two consecutive major releases of software and are calculated on a per-file basis. We refer to the time-period between two consecutive releases as 'timeline'.

In our work, we propose two novel change metrics namely Entropy of changes, Mean period of change and build prediction models with these two novel metrics along with ten widely

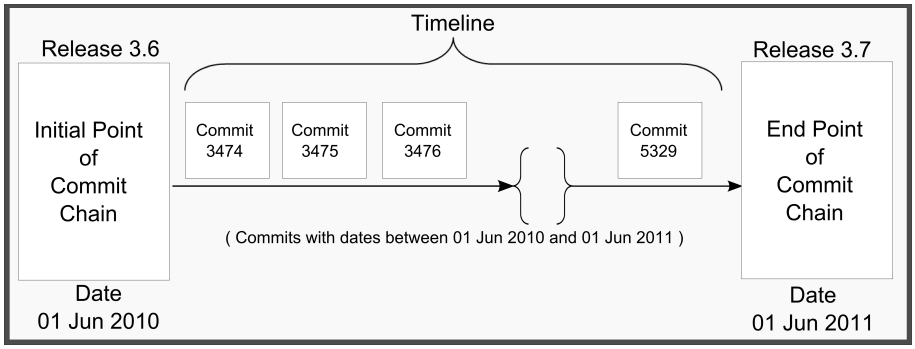


Figure 2.5: Data Extraction

used change metrics. We have considered all major releases of Eclipse from 3.0 to 3.5, and calculated these metrics for the time period between any two consecutive releases. We explain the metrics below, along with the rationale for considering them in the feature set used for defect prediction.

2.3.3 Novel Change Metrics

Entropy of Code Change It is intuitive to think that the distribution of changes (commits) to a file within the timeline might correlate to post-release defects. If we divide the timeline into N equal periods, a 'flat' distribution would have nearly equal number of changes in each of the N periods. On the other hand, changes might be concentrated to one or more regions resulting in peaks and lows in the change distribution graph. We use Shannon's definition of Entropy to estimate this change distribution. If C_i is the number of commits in the ith period that affect the file and C_{total} is the total number of commits within timeline that affect the file, then we define Entropy of code change for the file as,

$$-\sum_{i=1}^{N}\left(\frac{C_i}{C_{total}}\right)\times\log\left(\frac{C_i}{C_{total}}\right) \tag{2.4}$$

All major Eclipse releases are usually spaced one year apart. Hence, we have considered N as 12. It is to be observed that the way we defined Entropy is fundamentally different from Hassan's Entropy [19].

Mean Period of Change (MPC) We divide the timeline into 12 equal periods as in case

of ENTROPY. If all changes (commits) for a file are concentrated in one particular period, say period i, then we expect that different values of this period of concentration i should affect post-release defects in different ways. A low value for i implies that the file was last changed in the early periods of the timeline and has been free of changes for the remaining periods. So it can be considered relatively stable, as compared to a file for which period of concentration is localized closer to the date of release. Mean Period of Change is an attempt to estimate this center of concentration of changes.

$$MPC = \sum_{i=1}^{N} i * \left(\frac{C_i}{C_{total}} \right) \tag{2.5}$$

2.3.4 Other Change Metrics

Commits The number of commits within the timeline that have modified a file.

Rationale: More a file is changed, more the probability of a defect.

Add The total number of lines added to the file due to all the commits within the timeline.

Rationale: More the amount of change, more the probability of a defect.

Delete The total number of lines deleted from the file due to all the commits within the timeline.

Rationale: More the amount of change, more the probability of a defect.

Authors The number of unique authors involved in commits that have modified this file within the timeline.

Rationale: More the number of different authors involved in modifications to the file, greater the chances of a defect.

Commits60 The number of times the file was changed (the no. of commits to the file) in the last 60 days of the later release. That is, if D is the date of the later release (release 3.7 in figure(2.5)) and $d = D - (60 days)$, then COMMITS60 is the number of commits within dates d and D that modify this file.

Rationale: It is assumed that if a file is changed more just before the release, greater the possibility of post-release defects.

Last Commit The number of days before the later release date when the file was last changed in a commit.

Rationale: A file is expected to be more prone to post-release defects if the last change is closer to the release date.

In Development Bugs The number of bug-fix commits affecting the file within the timeline (the development period).

Rationale: We expect a file to be more defect prone if it has been frequently fixed for bugs.

Maximum Burst Nagappan et al. define a change burst as a "sequence of consecutive changes" to a file. Two parameters are used to detect change bursts, gap size and burst size. Gap size determines the minimum time gap between two changes (commits) to a file. For a file, commits with time gap less than the gap size will belong to the same change burst. The burst size determines the minimum no. of changes (commits) in a change burst. If the no. of commits in a change burst is less than the burst size, the change burst will not be considered [11]. MAXBURST represents the maximum change burst for a file, and is the maximum number of commits in any change burst for the file, i.e. $max|B|/B\epsilon bursts(file)$. We used gap size as 3 and max burst as 3 in our experiments.

Maximum Change Set Both Moser et al. [22] and Krishnan et al. [20] consider two change metrics, MAXCHANGESET and AVGCHANGESET (described next) that relate to the number of files changed along with the current file. For every file, the MAXCHANGESET is the maximum number of files changed along with this file in any commit within the timeline. If C is the subset of commits within the timeline that affect this file, then MAXCHANGESET is

$$\max_{c \epsilon C} n(c) \tag{2.6}$$

where $n(c)$ is the number of files modified by commit c, other than the current file.

Average Change Set For every file, the AVGCHANGESET is the average of the number of files changed together with this file, calculated over all commits affecting this file. If C is the subset of commits within timeline that affect this file and $|C|$ is the size of this subset, the AVGCHANGESET is

$$\frac{1}{|C|}\sum_{i=1}^{N} n\left(c_i\right) \tag{2.7}$$

where $n\left(c_i\right)$ is number of files modified by commit, other than the current file.

2.3.5 Feature Extraction

Github is a web-based code hosting platform that uses the Git revision control system developed by Linus Torvalds. Some popular projects hosted in Github are rails, jquery, node, html5-boilerplate, homebrew and diaspora. In our research, we have focused on extracting data from open-source software projects hosted on Github because of the growing popularity of the platform and also because of the easy availability of the Github API in different programming languages. To access the Github API, we have found the open-source Python library PyGithub[‡] to be extremely helpful. In revision control terminology, a commit represents a change to the code repository/database. In Github, every commit is uniquely represented by a 40 character hex-string, generated from the commit data using SHA-1 hash algorithm.

Extracting six change metrics We describe the procedure followed in data extraction of first six metrics here (explained in section 2.3.4). We assume we are gathering change metrics between release 3.2 (release date D1) and release 3.3 (release date D2) of some open-source repository on Github.

- All commits made to the repository are scanned sequentially and commits with

[‡]https://github.com/jacquev6/PyGithub

date between D1 and D2 are filtered out. The SHA strings of these commits are listed to a file.

- For each commit SHA in the file, the following details of the commit are retrieved:
  - Author, the person responsible for the change.
  - Date and Time, when the change was updated to the repository.
  - Number of files modified as a part of the change.
  - Names of the modified files.
  - Number of lines added to each modified file
  - Number of lines deleted from each modified file.

  The above details for a particular commit are output to a file in an easily parsable format, with the commit SHA as the filename, to guarantee uniqueness.

- All Java files from the two releases are listed separately, and only files common to both releases are considered.

- The commit data files generated in previous steps are processed to extract the first 6 metrics for each Java file common to both releases:

At this point, we have the name of each Java file along with the 6 generated metrics for each file. Files that have not been changed at all during the timeline will have zero values for all 6 metrics. Currently we do not consider such files in our study, because we focus only on how changes to a file affect the post-release bugs. We filter out such files.

Extracting number of in-development bug fixes In-development bugs refer to the number of bug fix commits made to a file during the period between the two releases. To recognize a commit as a bug fix, we parse the commit message string for the words 'Bug', 'Fix', 'fixed' [44]. If any of the words are matched in the commit message, the bug fix count for that file is incremented by one.

Calculating the code change entropy We divide the timeline into N periods and calculate the start and end dates for each period. For each file, we loop through the list of commits (considering only commits affecting the particular file) and distribute them in the N periods according to their commit dates. So, for every file, we know C_i, the number of commits to the file in period i. Also, C_{total} is the number of commits out of all the commits in the timeline that modify the file under consideration. Entropy for the file is then calculated according to the definition in Section 2.3.3.

Calculating Mean Period of Change As in case of entropy above, the C_i values are calculated for each file ($i\epsilon[1 \dots 12]$) and then MPC (Mean Period of Change) is calculated as per definition in Section 2.3.3.

Calculating MAXBURST MAXBURST is calculated as per definition in Section 2.3.4

Calculating MAXCHANGESET and AVGCHANGESET For every file, three variables are maintained: Max – Holds the MAXCHANGESET value. Initialized to zero. Sum – Holds the sum of the number of files changed along with this file by commits. Initialized to zero. $Count$ – Holds the number of commits in the timeline that affect this file. Initialized to zero. For every commit in the timeline:

- For every file that the commit modifies, increment the file's $Count$ by 1
- N is the total number of files modifies by the commit. For every file modified by the commit, add $(N-1)$ to the file's Sum. Also for every file that it modifies, replace max by $(N-1)$, if max is less than $(N-1)$

The final *max* value of a file is the MAXCHANGESET value of the file. The AVG-CHANGESET value for the file is the *Sum* divided by the *Count*.

Calculating post release bugs Bug reports are traced backwards from bug fixes by identifying the bug number from the commit message and then looking up the report date for that particular bug in a bug repository. We make an important assumption here: "Any bugs reported (for the projects mentioned in this work) are fixed within six months from the report date" With this assumption, all bugs reported within six months post-release will have been fixed within one year post-release. So we need to consider all bug fix commits within one year from the later release date. We maintain a list of files along with the no. of post-release bug reports, and initialize the no. of reports to zero for every file. For every bug fix commit within 1 year post-release:(1) The commit message is analyzed to strip out the bug ID number for the bug that was fixed in the commit. (2) The bug repository for our project[§] is then consulted and the document for the particular bug ID is parsed to extract the report date.(3) The report date is checked to see if it lies between the release date of the later release and within 6 months from that. (4) If it does, the bug report counts for each file modified by the commit is incremented by 1.

At this point, we have, for each file, the number of post-release defects (bugs) which is a numeric. Since we would like the output class to be binary nominal (YES or NO), all zero values for post-release defects are replaced by a 'NO' indicating the absence of post-release defects and all non-zero values are replaced by a 'YES' indicating the presence of post-release defects for that file.

2.3.6 Experimental Methodology

For our experiments, we have used the core Java Development Tools[¶] repository of Eclipse from Github. Metrics were extracted for all releases starting from Eclipse 3.0 to Eclipse 3.5. The Eclipse archives[‖] provides the release dates:

[§]http://bugs.eclipse.org
[¶]http://eclipse.jdt.core
[‖]http://archive.eclipse.org/eclipse/downloads

Table 2.8: Eclipse Data Set

Release Version	Release Date
3.0	25-Jun-04
3.1	27-Jun-05
3.2	29-Jun-06
3.3	25-Jun-07
3.4	17-Jun-08
3.5	11-Jun-09

The reader should notice that the period between 2 consecutive releases from the above table is nearly 1 year. Keeping this in mind, we have taken every 2 consecutive releases (3.0 to 3.1, 3.1 to 3.2 and so on) and generated change metrics for the time-period.

2.3.6.1 Training Classifiers

A study of previous defect prediction research shows that the most popular classification algorithms used in defect prediction are Naïve Bayes (NB), Decision Trees and Logistic regression. Menzies et al. [6] used NB successfully for defect prediction using static code attributes and concluded that it performs better than Decision Trees. Similarly, Moser et al. [13] did a comparative analysis of the performance of NB, Logistic regression and Decision Trees while using change metrics to predict defects. Zimmerman et al. [42] used static code metrics for the Eclipse project and logistic regression models for classification. For the sake of comparison, we have built prediction models using 10-fold cross-validation method for all three classification models separately. We have also considered the NB tree algorithm from Ron Kohavi's work [45], which is a hybrid of the NB and Decision Trees algorithms and supposedly has better predictive capabilities than the component algorithms alone. As per our knowledge, this algorithm has not been used in defect prediction models.

[‖]Relative Difference

2.3.6.2 10-Fold Cross Validations

Cross-validation is a general model validation technique for measuring the predictive ability, usually expressed in terms of accuracy or F-measure of a classification model.

We have used the 10-fold cross-validation technique that works by partitioning the training data set into 10 nearly equal subsets. Taking one of the subsets as testing/validation set, the classifier is trained on the remaining 9 subsets and the number of True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN) are calculated. This process is repeated by taking each of the 10 subsets as testing/validation set once. The values for each of the 10 subsets are then combined to get the mean precision, recall and F-measure.

Precision(P): Precision measures the fractions of instances that the classifier predicts as belonging to a particular class actually belong to that class. It is defined as,

$$\left(\frac{n_p}{n} * \frac{TP}{TP+FP}\right) + \left(\frac{n_n}{n} * \frac{TN}{TN+FN}\right) \tag{2.8}$$

where n_p = number of instances of positive class in the test set, n_n = number of instances of negative class in the test set, n = total number of instances in the test set.

The precision for the positive output class is:

$$P_{(pos)} = \left(\frac{TP}{TP+FP}\right) \tag{2.9}$$

and that for the negative output class is:

$$P_{(neg)} = \left(\frac{TN}{TN+FN}\right) \tag{2.10}$$

So the final precision of the classification is a weighted average of the precision for the individual output classes.

Recall(R): Recall measures the fractions of instances that actually belong to a class are

predicted by the classifier as belonging to that class. It is defined as,

$$\left(\frac{n_p}{n} * \frac{TP}{TP+FN}\right) + \left(\frac{n_n}{n} * \frac{TN}{TN+FP}\right) \tag{2.11}$$

The recall for the positive output class is:

$$R_{(pos)} = \frac{TP}{TP+FN} \tag{2.12}$$

The recall for the negative output class is:

$$R_{(neg)} = \frac{TN}{TN+FP} \tag{2.13}$$

F-measure(F): F-measure combines Precision and Recall into a single value while assigning different weights to each. In the most general sense, F-measure is given as,

$$F(\alpha) = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} \tag{2.14}$$

where P = precision R = recall α = factor that determines the weights assigned to P and R; $0 \leq \alpha \leq 1$ When $\alpha = 0.5$, F-measure is just the harmonic mean of P and R.

$$F = \frac{2PR}{P+R} \tag{2.15}$$

If $\alpha > 0.5$, the F-measure is precision-oriented and when $\alpha < 0.5$, the F-measure is recall-oriented. Keeping in accordance with Precision and Recall calculations, we calculate F-measure separately for each output classes.

$$F_{(pos)} = \frac{1}{\alpha\frac{1}{P_{(pos)}} + (1-\alpha)\frac{1}{R_{(pos)}}} \tag{2.16}$$

and

$$F_{(neg)} = \frac{1}{\alpha\frac{1}{P_{(neg)}} + (1-\alpha)\frac{1}{R_{(neg)}}} \tag{2.17}$$

The final F-measure calculated for the classification is a weighted average of the two:

$$\frac{n_p}{n} * F_{(pos)} + \frac{n_n}{n} * F_{(neg)} \tag{2.18}$$

We have done several experiments with α taking different values but the F-measure values revolve around the same value as that of F-measure with α as 0.5. Hence we conducted all our experiments with α as 0.5.

Table 2.9: Moser's (Recall) Results

	Algorithms			
Metrics	NB	DT	LR	DT with CSA
Change Metrics	24%	60%	28%	80%
Source Code Metrics	40%	38%	27%	65%
Change + Code Metrics	36%	58%	38%	74%

2.3.7 Results and Discussion

As explained in Section 2.3.5, we obtained all code change metrics for the timeline between two consecutive releases. We have built defect models using 10-fold cross validation on the data acquired between different releases. We calculated the precision, recall and F-Measure according to the definitions given in section 2.3.6.2. We have used Gaussian NB, CART Decision Tree, Logistic Regression and NB Tree algorithms to build defect prediction models for all considered versions of Eclipse JDT project. NB Tree algorithm has not been explored for defect prediction models though many researchers have used first three algorithms frequently. The results of different algorithms with our change metrics set for versions 3.0 through version 3.5 are depicted in Table 2.10. The consistency of results across five release by all four algorithms shows that our metrics set consisting of novel

Table 2.10: Bug Prediction Results with change metrics on five Eclipse JDT Releases

	Algorithms											
	Gaussian Naïve Bayes			CART Decision Tree			Logistic Regression			Naïve Bayes Tree		
	P	R	F	P	R	F	P	R	F	P	R	F
Eclipse 3.0 - 3.1	73.31	68.448	66.074	71.019	71.034	71.023	73.3	73.3	73.2	76.4	76.4	76.3
Eclipse 3.1 - 3.2	77.445	78.168	76.788	70.621	71.016	70.8	76.3	77.2	75.7	75.4	76.4	75
Eclipse 3.2 - 3.3	72.726	76.149	73.387	70.105	70.678	70.38	73.2	77.2	f71.9	74.6	77.9	73.6
Eclipse 3.3 - 3.4	73.532	74.1	71.001	70.466	70.863	70.641	72.5	73.7	71.8	75.3	76.1	75.3
Eclipse 3.4 - 3.5	72.132	74.177	72.448	70.461	69.873	70.146	71.2	73.9	70.8	73.4	75.4	72.9
Average	73.829	74.2084	71.94	70.5344	70.6928	70.598	73.3	75.06	72.68	75	76.4	74.62
S D	2.0934	3.62765	3.9098	0.33045	0.48014	0.3446	1.875	1.9655	1.8913	1.11	0.91	1.363

metrics like Entropy, MPC and other prominent metrics is one of the ideal set to build defect prediction models. The recall average of 76.44% is competitive in contemporary defect predictions models. NB Tree algorithm is better than all other three algorithms individually for each release. The precision, recall, F-measure averages of NB Tree are 75.02, 76.44, 74.62 and they are better than all other three algorithms. The standard deviation is around 1 and hence, we can say that results are very consistent across the releases. Hence, we recommend this algorithm to be used more often in tandem with other algorithms for all comparison studies.

Nagappan et al., claim recall of 92% on windows vista using change burst metrics [11]. It is worthwhile to note that these predictions differ from our predictions as described below: (1) The granularity for their prediction is binaries rather than file. A binary may contain on an average of 40 files and if a binary is predicted to be buggy then all files in it are to be reviewed / acted upon. If granularity is file level as in our case then the only predicted files need to be reviewed and this is much more cost effective than earlier model. The defect prediction models with granularity such as binary or package level may have good recall, precision values but they may not be desirable with respect to cost effectiveness. (2) Windows Vista is developed under controlled environment where in processes are very much streamlined and are adhered to the core by all Microsoft developers. But whereas Eclipse being an open source project, the change process may be not as controlled as it is in Microsoft projects and also there are fundamental differences between the two developments / change processes.

Nagappan et al. conducted experiments on Eclipse 2.0 with same set of metrics and their results (Recall - 51% and Precision - 67%) are in no way comparable to the experiments they conducted on Windows Vista (Recall - 92% and Precision - 91%) [11]. Hence we believe that our metrics set is worth considering and our results on Eclipse Releases are competitive. For us to compare the effectiveness of our model with other models, we have experimented on Eclipse JDT Release 2.0 to Release 2.1 as well. We compare all the models by making use of the measure, Recall, as it is common across all works and significant measure for our research problem. Moser et al. performed experiments on Eclipse 2.1 Release with 31 source code metrics and 18 change metrics. They have built three models with - (1) source code metrics alone (2) change metrics alone (3) both source code and

change metrics. They have used NB Classifier, Logistics Regression (LR), Decision Tree (DT) and also improvised their results using cost-sensitive analysis on Decision Tree (CT with CSA) [13] to build defect prediction models. Their results are depicted in Table 2.9.

Schroter et al. considered the impact of importing packages / classes in files on defect prediction and they performed experiments on Eclipse 2.1 using Support Vector Machines (SVM) [46]. Two defect prediction models have been built using SVM - the imported classes inside a file as feature in the first model and imported packages inside a file as feature in the second model. Recall of 16.9% is observed in first model and 9.69% is observed in second model.

Zimmerman et al. studied the impact of forty odd source code complexity metrics on defect prediction using Eclipse 2.1 version data. Recall of 16% is observed when prediction model is built using Eclipse 2.1 version data and tested again on the same data [42]. We have gathered all our change metrics in the timeline between Eclipse JDT Release 2.0 and Release 2.1 and post release bug data for six months after the release date of Release 2.1. We have experimented with all four algorithms, Gaussian NB, CART Decision Tree (DT), Logistic Regression (LR) and NB Tree. The results are displayed in Table 2.11.

Table 2.11: Recall of Eclipse JDT 2.1 Release

	Algorithms			
Metrics	NB	DT	LR	NBT
12 Code,Change Metrics (mix of novel and existing metrics) as proposed in this work	83.7	85	90.4	86.8

Our change metrics lead to good results with the Recall of 90.4% using LR and the other three algorithms have performed exceedingly well as compared to any other model.

2.3.8 Threats to Validity

For comparing the effectiveness of our model with other models we have conducted experiments on Eclipse JDT 2.1 Release where as other prediction models can be built using

different or all Packages of Eclipse 2.1 Release. Hence, results of experiments conducted on all packages of Eclipse 2.1 Release might not be the same. Though we confirmed the correctness of our datasets with repeated experiments there could still be some issues. For example, if developer made many overlapping edits to a file in single check-out/check-in then we could have wrongly captured related metrics. There are chances that we could have missed some bug data due to lack of proper documentation which is quite natural in open source platforms. Another limitation of our work is, we have done the experiments only in different versions of Eclipse project and not with any other project. Hence the similar metrics for other projects may not yield same consistent results.

2.3.9 Contributions

A definitive procedure has been put forth to get popular code change metrics for any project that has been hosted on Github. We have proposed couple of new change metrics namely, Entropy of Code Change and Mean Period of Change, to figure out whether commits are uniformly distributed over the timeline between the two consecutive releases. The defect prediction models are built with these novel metrics along with some existing prominent change metrics by making use of four different algorithms. Models yield consistent and competent results for five consecutive releases (Release 3.0 through Release 3.5) of Eclipse JDT project. NB Tree Algorithm, which has not been explored in this problem domain found to be the better performing algorithm. We compared our results on Eclipse 2.1 Release with the results of other existing defect prediction models on the same version and discovered that our model is doing exceedingly well.

3 Defect Prediction Models

3.1 Introduction

With rapidly growing size and complexity of software systems, the development and maintenance of huge code bases have become extremely difficult and time consuming task. A huge amount of resources are spent in testing and review processes of the software to ensure defect free releases. Researchers are trying to establish methods to identify the software source files that are more vulnerable to defects. It helps verification and validation team to allocate or manage their critical resources well. Hence the research on defect prediction models has become significant. Since last decade, there has been a lot of emphasis on building defect prediction models by implementing machine learning algorithms on historical data. We study different machine learning approaches to build defect prediction models and evaluate their effectiveness.

A comparative study is carried out on four source code metric sets, namely, CK, OO, CK + OO and LOC. The experiments are conducted on five open source projects, namely, Eclipse JDT, Eclipse PDE, Lucene, Mylyn and Equinox. The salient features of this work is outlined below.

We develop models using Naïve Bayes Classifier for all four sets of data - CK, OO, CK + OO and LOC. It is observed that there is not much difference between the recall of all these four models. For model with CK + OO metrics, the recall is found to be 0.82 whereas the percentage of bugs caught is 46.37% by performing quality assurance activities on 10.78% of files. We would like to make a significant observation from these experiments that the

recall appears to be very high in all scenarios but the actual number of bugs found is not encouraging. Moreover, if project team would like to catch more bugs, say 80% of total bugs, by spending more money on associated quality assurance activities then these models and evaluation methodologies will limit them to do so.

Khoshgoftaar et al. proposed software quality model, module-order model that is generally used to predict the rank-order of modules or files according to a quality factor [47]. We take this quality factor to be number of defects in our case study. We apply stepwise multiple linear regressions and order files as per their predicted value of bugs. We build module-order models using four source code metric sets namely CK, OO, CK + OO and LOC for each of the five open source projects and thus project team has the flexibility of picking up the right model depending on the cost borne by the project team towards quality assurance activities.

In Section 3.2, we build Bayesian Network structures for defect prediction using different classes of algorithms namely score-based, constraint-based and hybrid algorithms. We propose an approach to augment these Bayesian Network structures with class node. Bayesian Network classifiers along with Random Forests, Logistic Regression and Naïve Bayes classifiers are then evaluated using AUC and H-measure.

In Section 3.3, we study defect prediction models built using three cost-sensitive boosting Neural Network methods, namely, CSBNN-TM, CSBNN-WU1 and CSBNN-WU2. We have compared the performance of these cost sensitive Neural Networks with the traditional machine learning algorithms like Logistic Regression, Naïve Bayes, Random Forest, Bayesian Network, Neural Networks, k-Nearest Neighbors and Decision Tree. Performance of the resultant models is evaluated using cost centric measure - Normalized Expected Cost of Misclassification (NECM).

In Section 3.4, we propose testing and code review based effort-aware defect prediction models. Idea is to promote effort-aware models which consider the effort required to perform the specific quality assurance activity. We empirically prove the supremacy of the proposed models over the existing effort-aware models.

3.2 Defect Prediction Using Augmented Bayesian Networks

Despite the assumption of conditional independence between the features, Naïve Bayes classifier is found to be one of the consistently better performing techniques in defect prediction [14]. It motivated researchers to check whether prediction accuracies can be improved if conditional independence assumption is relaxed. This is also supported by the intuition that all the features/attributes used for defect prediction are not conditionally independent. For example, lines of code (LOC) and number of operands of a source file can't be taken as independent. Different Bayesian networks have been studied by Dejaegar et al. [48] and they have shown that performance of some of the classifiers is as good as Naïve Bayes and Random Forests classifiers. They have constructed Bayesian Network structure by implementing model learning algorithms like Tree Augmented Naïve Bayes (TAN), Forest Augmented Naïve Bayes (FAN), Selective Tree Augmented Naïve Bayes (STAN), Selective Tree Augmented Naïve Bayes with Discarding (STAND), Selected Forest Augmented Naïve Bayes (SFAN), Selected Forest Augmented Naïve Bayes Discarding (SFAND), K2 and MMHC. They have found out that the performance of TAN is significantly better than MMHC, STAN-SB though there is no significant difference between TAN and others variants of FAN, STAND, SFAN, SFAND and STAN. However Random Forests is found to be better than TAN though the difference is not statistically significant. Hence in our study, we have considered TAN as one of the algorithms amongst all other algorithm for building Bayesian Networks.

There are three classes of algorithms namely score-based, constraint-based and hybrid algorithms to build Bayesian Networks. We have built Bayesian networks using two algorithms from each of three classes. We have built Bayesian Networks structures using Hill Climbing, TABU from score-based algorithms; TAN, Grow-Shrink from constraint-based algorithms and MMHC, RSMAX2 from Hybrid class of algorithms. We have compared performance of the respective Bayesian Network classifiers with popular classifiers like Random Forest, Naïve Bayes and Logistic Regression.

We have conducted experiments on 15 different datasets. The results of these classifiers are presented in terms of both AUC and H-measures.

3.2.1 Related Work

Studies in defect prediction is either focused on establishing software metric(s) that effectively captures defect characteristics or algorithm(s) which learns from past data and improves prediction accuracy consistently. As a consequence, various defect prediction approaches and their stability in performance on numerous datasets have been reported and Bayesian Network has always been one of the best performing classification method [5]. Wang et al. [49] has shown that Multi-variant Gauss Naïve Bayes version of Naïve Bayes classifiers performs better than Naïve Bayes in defect prediction. Fenton et al. used Bayesian Networks to predict software defects and reliability. According to their study, as BN improves the prediction accuracy significantly, if organization collects metrics data, there is a compelling argument for them in using BNs for defect prediction [50]. Okutan and Yıldız used Bayesian networks to determine the probabilistic influential relationships among software metrics and defect proneness [51]. Despite it's simple assumption of conditional independence, Naïve Bayes is found to be the better performing technique. General Bayesian Networks haven't been explored much in the field of Software defect Prediction. This motivated us to consider a few other ways of constructing Bayesian Networks for Software fault prediction.

Dejaeger et al. [48] have studied a few other Naïve Bayes classifiers including TAN, variants of TAN, MMHC and K2 Algorithms. They have shown that these networks perform better than Naïve Bayes and have compared the results to Random Forests, and conclude that some of them perform as good as Random Forests. There are many ways to construct the Bayesian network. Naïve Bayes is just one such Bayesian Networks. TAN [52] is a constraint-based bayesian network. There are other augmented networks [53] like Selective TAN (STAN), Selective TAN with Discarding (STAND), Forest Augmented Network (FAN)and variants of FAN. Another version of TAN called SP-TAN [54] has been considered by Keogh. Later Webb has shown that aggregating one-dependence estimators have better performance than SP-TAN [55]. Sahami [56] shows how to build a limited dependence Bayesian network. Jing et al. introduced boosted bayesian network classifier, a framework which combines both discriminative data-weighting and generative training of intermediate models, which require less training time but gives better or comparable results in comparison with other models [57]. Cheng and Greiner have shown that augmen-

ted general Bayesian Networks constructed on constraint-based methods perform better than Naïve Bayes classifier [58].

3.2.2 Background

Bayesian Networks can be defined as a tuple <S,θ>, where S represents network structure with the nodes and their directed edges. A directed edge from one node (node A) to another node (node B) implies that node B is dependent on node A. Thus, the structure S represents dependence of nodes in the Bayesian Network. θ represents conditional Probability Tables of all the nodes given their direct parents in the graph. Building Bayesian Network classifiers involves 3 main steps - 1)Learning the Bayesian Network structure, 2)Estimating the parameters and 3)Using the Bayesian Network structure, find the probability of the class (Yes/No) given the testing instance, and classifying the instance accordingly. There are three broad categories of algorithms for learning the structure of Bayesian Networks. We will not be discussing these algorithms in detail but attempt to give an overview of these algorithms. We recommend interested readers to refer to the original papers of these algorithms.

3.2.2.1 Score-based Algorithms

These algorithms learn the structure of a Bayesian networks by maximizing/minimizing a certain score metric and consider the structure learning problem as a model selection problem, which means they select the best hypothesis from a competing set of hypothesis. We have used Hill Climbing and TABU search strategies [59] [60] [61] in our study.

- Hill Climbing Algorithm

  Hill Climbing search is a greedy search algorithm. Search will start from an empty network and if there is a background knowledge about dependency of features, it can be used to seed the initial network. Then it starts with estimating parameters

of the local pdfs, given a Bayesian network structure. Typically this is a maximum-likelihood estimation of the probability entries from the data set. Algorithm attempts every possible single-edge addition, removal, or reversal, which increases score of the network, and iterates. The process stops when there is no single-edge change that increases the score. There is no guarantee that this algorithm will settle at a global maximum, note that, Hill Climbing algorithm can get struck in local minima [61]. We have built three Hill Climbing classifiers, each with one of the scores - K2 [62], BDE [63] and AIC.

- TABU Algorithm

  TABU search is a slight modification to Hill Climbing search. As Hill Climbing search can get struck in a local minima, TABU search addresses this issue by maintaining a TABU list (list of previously visited states). TABU search will not allow any addition or removal of the edge which makes the network go to the state that is already in TABU list. It should be noted that, in the package we have used (bnlearn in R), the default setting for the TABU list size is 10. That means TABU list keeps track of the 10 previously visited states. We have built three TABU classifiers, each with one of the scores - K2 [62], BDE [63] and AIC.

3.2.2.2 Constraint-based Algorithms

Constraint based algorithms learn the structure of Bayesian Network by performing conditional independence (statistical) tests between the attributes.

- TAN Algorithm

  Tree Augmented Network [52] is based on Chow Liu algorithm. Let X denote all the attributes and c denote the classification attribute. A complete graph is learnt between all attributes in X-{c} using Mutual Independence tests where an edge represents dependency between the two connected nodes. A maximum spanning tree is built from this complete graph, and directions of the tree are set as per

Chow-Liu algorithm so that every node has exactly one parent. This spanning tree is augmented with Naïve Bayes classifier and with classification node as root node.

- Grow-Shrink Algorithm

  Recovery of local structure around each node is greatly facilitated by knowledge of the node's Markov blankets. GS (grow-shrink) algorithm, recovers Markov blanket of X based on pairwise independence tests. The process consists of two phases, namely, a growing phase and a shrinking phase. It starts from an empty set S, and the growing phase adds variables to S as long as they are dependent with X, given the current contents of S. In this process however, there might be some variables that were added to S, really outside the blanket. In shrinking phase, such variables are identified and removed. [61].

3.2.2.3 Hybrid Algorithms

Hybrid algorithms make use of both constraint-based approaches and score-based approaches. We have considered two Hybrid algorithms.

- MMHC Algorithm

  Max-Min Hill Climbing algorithm (MMHC) is for learning the structure of a Bayesian network. The algorithm first identifies the parents and children set of each variable using MMPC algorithm, and then performs a greedy Hill Climbing search in the space of Bayesian networks. The search begins with an empty graph. The edge addition, removal, or direction reversal that leads to maximum increase in score is taken and the search will continue in a similar fashion recursively. The major difference from standard greedy search is that the search is constrained to only consider adding an edge if it was discovered by MMPC in the first phase [64].

- RSMAX2 Algorithm

RSMAX2 [65] is a generalized version of MMHC. MMHC is restricted to MMPC for constraint-based learning and Hill Climbing for score-based learning. On the other hand, RSMAX2 is more general in the sense that we can input both constraint-based method and score-based method. Note that, by default, Grow-Shrink is the constraint-based method and Hill Climb is the default score-based method used in RSMAX2. bnlearn package in R implements BIC score as the default score for RSMAX2 algorithm. All these structure learning algorithms are available in bnlearn in R [65].

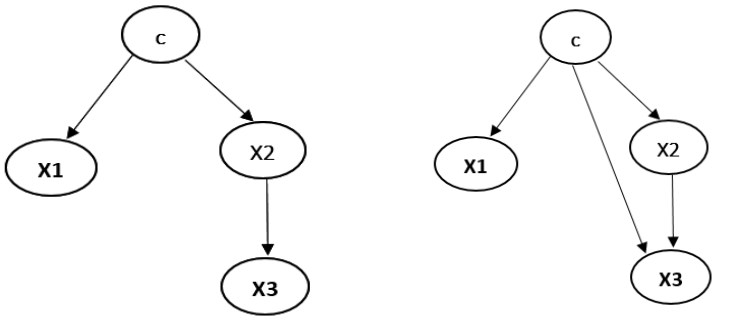
3.2.2.4 Augmentation with class node

Let X denote set of attributes of the dataset and c denote the classification node; we define augmentation with class node as follows: Let G be the Bayesian Network structure learnt from the dataset D having attributes X and classification node c by implementing any of the algorithms mentioned above; there need not be an edge between class node and each node in X. We augment Bayesian Network structure G with class node c by adding edges between class node and all other nodes in X regardless of the structure G having edges between c and attributes in X. A simple Bayesian Network is illustrated in Figure 3.2a. As it does not have an arc from c to every other node, it's not augmented. Bayesian Network in Figure 3.2b is augmented as it has an arc from c to every other node. For our experiments we have augmented Bayesian Network structures learnt using HC, TABU, GS, MMHC and RSMAX2.

3.2.3 Experimental Setup

The data used in this study comprises of the datasets in NASA MDP repository [66] and these include JM1, JM2, KC1, KC2, MW1, PC1, PC2, PC3, PC4, MC1 and MC2 projects. We have used metrics of these projects to build defect prediction models. We have also considered Eclipse 2.0, Eclipse 2.1 and Eclipse 3.0 projects [42]. The metrics considered for building defect prediction models are FOUT avg, MLOC avg, NBD avg, PAR avg, VG avg, NOF avg, NOM avg, NSF avg, NSM avg, ACD, NOI, NOT, TLOC

Figure 3.1: Augmentation with Naïve Bayes



(a) Bayesian Network

(b) Augmented Bayesian Network

and post release bugs. Each Dataset is discretized using the method described by Fayyad and Irani [67] followed by randomizing the dataset (Shuffling the dataset row wise). It is then split into $2/3^{rd}$ Training set and $1/3^{rd}$ Test set. Each classifier is built on training set and has been tested on test set. First the structure is learned, and then Bayesian structures are propagated for inference. This process is repeated 10 times for each dataset and for each classifier. i.e each classifier is evaluated on a single dataset 10 times and the results presented (AUC and H-Measure) are average of the 10 runs. Logistic regression and Random Forests have been run on the continuous data as opposed to other Bayesian classifiers where Bayesian classifiers have been run on discretized data.

AUC is one of the most used classifier evaluation measures with values ranging from 0 to 1, 1 being the best classifier and anything less than 0.5 is worse than a random classifier. If AUC value is 0.5, the classifier is just as good as random guessing.

Hand developed a performance measure based on expected minimum misclassification loss, whereby misclassification costs are not exactly known but follow a probability distribution [68]. Assume that misclassifying a faulty file or class as not fault-prone has misclassification cost c_0, whereas a fault-free file or class classified as fault-prone costs c_1. H-Measure can be calculated for classifier by assuming that the ratio of these two costs follows beta distribution with parameters α and β. We refer to the original paper by Hand [68] for detailed discussion of this measure. H-Measure is a normalized measure based on expected minimum misclassification loss, ranging from zero for a random classifier to one for a

perfect classifier. Hand and Anagnostopoulos [69] suggested better parameter values for beta distribution as (2,2) when the misclassification costs of the domain are not known in prior. We follow the recommendation by Hand Anagnostopoulos for our problem as misclassification costs for defect prediction models are not known.

In our study, we have computed H-Measure plots at different cost ratios ranging from 1/500 to 500 (in intervals of 20. That is 1/500, 1/480....1, 20,...500). We have made use of H-Measure available in hmeasure package in R.

We have computed the results in terms of AUC and H-Measure. The classifiers are then subjected to Friedman Test to check whether the null hypothesis is rejected. Null hypothesis for this test is that all classifiers perform alike. If the null hypothesis is rejected, as Demšar suggests, we apply Nemenyi post-hoc test to find out the best performing classifiers [70]. The performance of two classifiers is significantly different if the corresponding average ranks differ at least by the critical difference. Nemenyi plots at 5% significance level have been plotted for both AUC and H-Measure where

$$CD_{5\%} = q_{5\%,k=15} * \sqrt{n * (n+1)/(12 * k)} \tag{3.1}$$

where n represents number of classifiers and k, the number of datasets. In this study n and k are 13 and 15 respectively. Figure 3.4 shows evaluation of classifiers based on H-Measures at different cost ratio distributions. Results presented in Figure 3.4 are also presented after the datasets are discretized and averaging the results after running each classifier on each dataset for 10 times. Logistic regression and random forests have been learned on continuous data as in case of AUC and H-Measure.

3.2.4 Results and Discussion

The results obtained from various Bayesian Network classifiers and the standard classification algorithms like Random Forests and Logistic Regression are presented both in terms of AUC and H-Measure. Further, all classifiers have been evaluated on H-Measure at different cost ratio distributions as well. Figure 3.4 shows the mean rank of classifiers

Table 3.1: Performance of classifiers based on AUC Measure before Augmentation

Dataset	JM1	KC1	MC1	MC2	PC1	PC2	PC3	PC4	E2.0	E2.1	E3.0	CM1	KC2	MW1	KC3
Naïve Bayes	0.677	0.807	0.817	0.725	0.868	0.874	0.820	0.847	0.790	0.733	0.763	0.784	0.865	0.851	0.839
TAN	0.668	0.765	0.791	0.666	0.866	0.877	0.805	0.896	0.791	0.749	0.757	0.739	0.756	0.765	0.743
HC:K2	0.670	0.792	0.803	0.563	0.839	0.828	0.799	0.884	0.781	0.736	0.761	0.713	0.837	0.725	.799
HC:BDE	0.665	0.790	0.818	0.579	0.849	0.847	0.796	0.904	0.781	0.735	0.760	0.737	0.847	0.658	.822
HC:AIC	0.672	0.791	0.821	0.573	0.837	0.839	0.808	0.894	0.785	0.738	0.766	0.719	0.840	0.581	.851
TABU:K2	0.67	0.789	0.803	0.572	0.837	0.808	0.799	0.881	0.781	0.736	0.761	0.709	0.833	0.718	.799
TABU:BDE	0.665	0.789	0.811	0.572	0.846	0.846	0.795	0.901	0.781	0.735	0.760	0.737	0.843	0.664	.816
TABU:AIC	0.672	0.791	0.811	0.575	0.837	0.839	0.805	0.894	0.784	0.738	0.766	0.717	0.837	0.589	0.811
GS	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.492	0.5	0.5	0.5	0.5	0.5	0.523	0.5
MMHC	0.583	0.477	0.791	0.5	0.755	0.5	0.795	0.896	0.781	0.735	0.760	0.547	0.563	0.461	0.583
RSMAX2	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.492	0.5	0.5	0.5	0.5	0.5	0.523	0.5
Random Forests	0.696	0.836	0.843	0.712	0.864	0.814	0.826	0.944	0.822	0.749	0.778	7 0.743	0.832	0.767	0.821
Logistic Regression	0.672	0.792	0.691	0.595	0.796	0.760	0.808	0.914	0.784	0.730	0.748	0.754	0.800	0.669	0.766

at different cost ratios. We have conducted experiments with Bayesian Network classifiers and Augmented Bayesian Network classifiers.

In Table 3.1, we have presented AUC values of different Bayesian Network classifiers. The Grow-Shrink and RSMAX2 algorithms yield low AUC values. They are just as good as random guessing. To know if there is a significant difference between performances of algorithms, we have applied Friedman test and null hypothesis is strongly rejected with a p-value $< 2.2^* 10^{-16}$. The rejection of null hypothesis suggests that there is significant difference between performances of algorithms. So, we proceeded with Nemenyi post-hoc test. The classifiers performance is shown using Nemenyi plots. Though Random Forests is coming out as one of the best performing techniques with respect to AUC in Nemenyi Post-hoc test, Figure 3.3a shows that classifiers, Hill Climbing with AIC Score, Hill Climbing with BDE Score, TAN and Naïve Bayes structures perform as good as Random Forests. In fact, Naïve Bayes has a mean rank of 3.067 and Random Forests has a mean rank of 2.53 which shows that Naïve Bayes is one of the best performing classifiers.

As MMHC, RSMAX2 and Grow-Shrink have AUC Values of 0.5, we have augmented them with class node. We have also augmented Bayesian Network structures built using HC and TABU algorithms. Table 3.2 shows the values after augmentation with Naïve Bayes. Significant improvement can be seen in case of MMHC, Grow-Shrink and RSMAX2 classifiers. Friedman test showed a strong rejection of Null Hypothesis with a p value of 2.426^*10^{-07}. Figure 3.3b shows Nemenyi plots for Bayesian classifiers after augmentation.

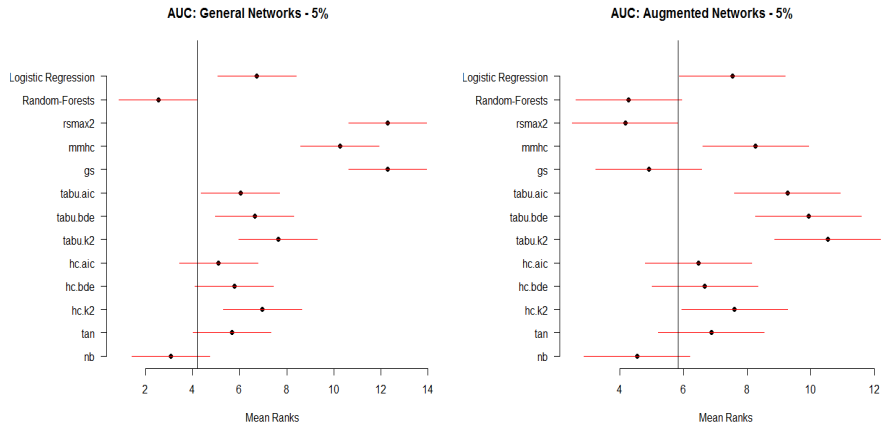
Table 3.2: Performance of classifiers based on AUC Measure after Augmentation

Dataset	JM1	KC1	MC1	MC2	PC1	PC2	PC3	PC4	E2.0	E2.1	E3.0	CM1	KC2	MW1	KC3
Naïve Bayes	0.677	0.807	0.817	0.725	0.868	0.874	0.820	0.847	0.790	0.733	0.763	0.784	0.865	0.851	0.839
TAN	0.668	0.765	0.791	0.666	0.866	0.877	0.805	0.896	0.791	0.749	0.757	0.739	0.756	0.765	0.743
HC:K2	0.653	0.776	0.849	0.757	0.888	0.869	0.765	0.896	0.783	0.728	0.723	0.710	0.797	0.596	0.714
HC:BDE	0.653	0.766	0.868	0.740	0.887	0.865	0.759	0.907	0.789	0.733	0.747	0.726	0.797	0.544	0.746
HC:AIC	0.658	0.769	0.859	0.700	0.889	0.867	0.769	0.905	0.791	0.730	0.736	0.731	0.815	0.589	0.748
TABU:K2	0.650	0.759	0.814	0.731	0.887	0.855	0.743	0.887	0.771	0.722	0.711	0.683	0.733	0.739	0.638
TABU:BDE	0.651	0.748	0.863	0.730	0.882	0.860	0.742	0.896	0.783	0.728	0.737	0.696	0.728	0.656	0.659
TABU:AIC	0.654	0.750	0.849	0.671	0.889	0.843	0.744	0.896	0.785	0.724	0.726	0.702	0.761	0.725	0.699
GS	0.671	0.801	0.820	0.740	0.882	0.885	0.795	0.854	0.786	0.737	0.762	0.784	0.853	0.783	0.854
MMHC	0.652	0.750	0.863	0.699	0.884	0.865	0.755	0.899	0.782	0.731	0.742	0.704	0.826	0.708	0.688
RSMAX2	0.671	0.803	0.822	0.740	0.884	0.886	0.791	0.855	0.786	0.739	0.762	0.785	0.853	0.785	0.855
Random Forests	0.696	0.836	0.843	0.712	0.864	0.814	0.826	0.944	0.822	0.749	0.778	0.743	0.832	0.767	0.821
Logistic Regression	0.672	0.792	0.691	0.595	0.796	0.760	0.808	0.914	0.784	0.730	0.748	0.754	0.800	0.669	0.766

Although Augmented RSMAX2 and Random Forests statistically perform as good as one another, we can see that RSMAX2 has a lower mean rank(4.167) as against(4.2667) of Random Forests, which shows that augmented RSMAX2 can be strongly recommended along with Random Forests when one considers performance of classifiers based on AUC metric. Naïve bayes, Grow-Shrink (after augmentation), TAN and Hill Climbing algorithms perform as good as the best performing classifier. This shows that Augmentation has significantly improved the predictive performance of Bayesian classifiers.

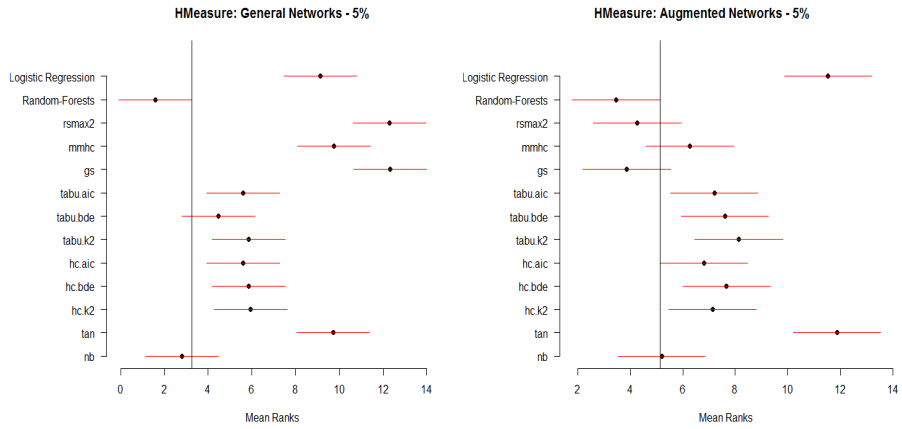
Table 3.3 shows H-Measure of Bayesian Network classifiers. Friedman test showed strong rejection of NULL hypothesis with a p-value less than $2.2^* \ 10^{-16}$. Grow-Shrink and RSMAX2 have H-Measure values of 0 in some cases, confirming the fact that their AUC values are 0.5. Figure 3.4a shows Nemenyi plot for H-Measure. Similar to AUC results,

Figure 3.2: Performance of classifiers: AUC



(a) AUC Measure before augmentation **(b)** AUC Measure after augmentation

Figure 3.3: Performance of classifiers: H-measure



(a) H-measure before augmentation **(b)** H-measure after augmentation

Random Forests is the best performing classifier though it is not significantly better than Naïve Bayes and TABU structure with BDE score techniques. With H-Measure, TABU structure with BDE score comes out as one of the best performing classifiers, which was not a best performing classifier with respect to AUC values. In contrast, Hill Climbing with AIC Score, Hill Climbing with BDE Score and TAN, which are among top performing techniques with respect to AUC are not figured in top performing techniques with respect to H-Measure.

Based on the significant improvement in AUC Measures after augmentation, we have computed the H-Measure values of the Bayesian classifiers as well after augmentation. Table 3.4 shows the H-Measure values after augmentation. We can see improved results in both RSMAX2 and GrowShrink classifiers. Friedman test is rejected with a p-value of $6.14*10^{-12}$. Figure (b) shows Nemenyi plot of classifiers at 5% significance level. Similar to our previous results Random forests appears to be best performing technique but it is not significantly better than RSMAX2, Grow-Shrink, MMHC and Naïve Bayes. Once again our results show that augmentation has increased predictive performance of Bayesian classifiers.

Figure 3.4 shows the evaluation of classifiers at different cost ratio distributions. Let c_1 denote the cost of misclassifying a buggy file as non buggy file and c_0 denote the cost of misclassifying a non buggy file as buggy. Let c denote the ratio of their misclassification

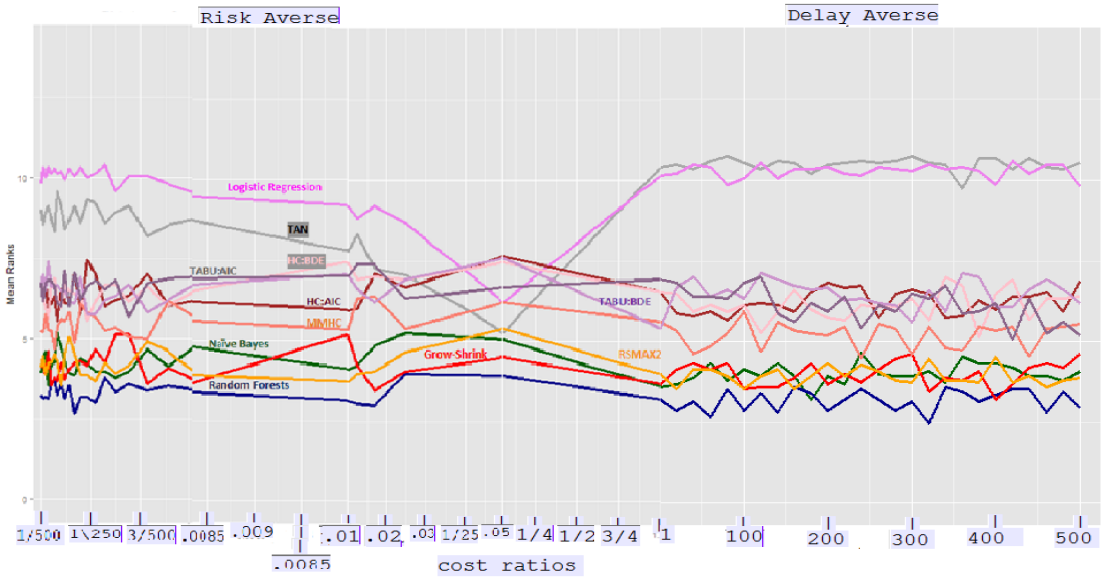
Table 3.3: Performance of classifiers based on H-Measure Beta(2,2) before Augmentation

Dataset	JM1	KC1	MC1	MC2	PC1	PC2	PC3	PC4	E2.0	E2.1	E3.0	CM1	KC2	MW1	KC3
Naïve Bayes	0.135	0.313	0.387	0.263	0.442	0.658	0.294	0.421	0.286	0.209	0.241	0.258	0.489	0.474	0.491
TAN	0.092	0.182	0.111	0.297	0.22	0.033	0.189	0.395	0.191	0.123	0.157	0.103	0.332	0.283	0.202
HC:K2	0.12	0.286	0.297	0.181	0.388	0.372	0.303	0.527	0.266	0.195	0.221	0.174	0.427	0.36	0.333
HC:BDE	0.118	0.267	0.379	0.266	0.41	0.473	0.29	0.552	0.256	0.175	0.226	0.191	0.402	0.325	0.327
HC:AIC	0.116	0.286	0.405	0.181	0.367	0.382	0.303	0.526	0.278	0.186	0.237	0.191	0.459	0.306	0.351
TABU:K2	0.117	0.281	0.341	0.272	0.349	0.422	0.309	0.51	0.25	0.189	0.232	0.171	0.385	0.397	0.39
TABU:BDE	0.121	0.27	0.345	0.25	0.383	0.41	0.334	0.533	0.268	0.19	0.238	0.209	0.441	0.331	0.408
TABU:AIC	0.119	0.272	0.358	0.22	0.344	0.396	0.322	0.541	0.265	0.199	0.235	0.188	0.416	0.324	0.352
GS	0.002	0	0	0.08	0	0	0	0	0	0	0	0	0	0.068	0
MMHC	0.073	0.062	0.246	0.001	0.25	0	0.281	0.497	0.273	0.187	0.222	0.036	0.104	0.009	0.03
RSMAX2	0	0	0.014	0	0	0.078	0	0	0	0	0	0	0.047	0	0
Random Forests	0.147	0.355	0.608	0.31	0.44	0.439	0.357	0.631	0.338	0.212	0.261	0.293	0.446	0.34	0.47
Logistic Regression	0.123	0.239	0.045	0.228	0.251	0.043	0.227	0.421	0.201	0.11	0.149	0.166	0.352	0.125	0.227

Table 3.4: Performance of classifiers based on H-Measure Beta(2,2) after Augmentation

Dataset	JM1	KC1	MC1	MC2	PC1	PC2	PC3	PC4	E2.0	E2.1	E3.0	CM1	KC2	MW1	KC3
Naïve Bayes	0.131	0.313	0.344	0.398	0.459	0.577	0.306	0.409	0.286	0.187	0.235	0.266	0.49	0.481	0.453
TAN	0.098	0.178	0.106	0.34	0.23	0.048	0.18	0.418	0.182	0.116	0.157	0.114	0.311	0.298	0.189
HC:K2	0.108	0.265	0.466	0.284	0.471	0.588	0.237	0.51	0.283	0.186	0.184	0.271	0.349	0.363	0.305
HC:BDE	0.107	0.247	0.466	0.299	0.455	0.564	0.225	0.508	0.301	0.196	0.21	0.226	0.325	0.34	0.304
HC:AIC	0.111	0.245	0.433	0.296	0.49	0.52	0.249	0.531	0.292	0.199	0.202	0.231	0.35	0.366	0.318
TABU:K2	0.101	0.254	0.427	0.329	0.474	0.569	0.259	0.53	0.276	0.199	0.191	0.209	0.316	0.326	0.241
TABU:BDE	0.099	0.253	0.376	0.255	0.454	0.508	0.257	0.553	0.299	0.212	0.195	0.221	0.393	0.399	0.289
TABU:AIC	0.105	0.259	0.466	0.303	0.481	0.495	0.279	0.539	0.287	0.2	0.197	0.23	0.349	0.309	0.275
GS	0.125	0.32	0.365	0.332	0.467	0.647	0.315	0.42	0.303	0.202	0.244	0.268	0.472	0.514	0.444
MMHC	0.113	0.246	0.379	0.275	0.447	0.578	0.284	0.539	0.275	0.207	0.216	0.243	0.408	0.438	0.367
RSMAX2	0.131	0.315	0.427	0.328	0.429	0.619	0.292	0.422	0.293	0.204	0.239	0.308	0.487	0.48	0.47
Random Forests	0.148	0.373	0.547	0.298	0.437	0.457	0.357	0.645	0.343	0.218	0.265	0.252	0.427	0.407	0.473
Logistic Regression	0.11	0.229	0.048	0.211	0.188	0.081	0.224	0.443	0.204	0.113	0.147	0.199	0.374	0.152	0.202

Figure 3.4: H-measure at different cost distributions



cost, $c = c_0/c_1$. X-axis shown in Figure 3.4 represents the misclassification ratio c. When Cost ratio is < 1, cost of misclassifying a buggy file as non buggy file becomes more severe than cost of misclassifying a non-buggy file as buggy file. This situation is called risk averse situation. On the other hand, when cost ratio is > 1, cost of misclassifying a non-buggy file as buggy file becomes more severe than misclassifying a buggy file as non-buggy file. This situation is called delay averse situation.

In case of risk averse situation, Random Forests stand out as the best performing classifier among all classifiers; augmented RSMAX2, Grow-Shrink and Naïve Bayes being the next best performing techniques. Although Random Forests ranked lower than any other classifier most of the times, there are places at which other Bayesian classifiers have ranked equal to Random forests and sometimes have ranked lower than Random Forests. At a cost ratio distribution of .0023, classifier built using RSMAX2 has a mean rank of 3.53 against a mean rank of 3.60 of Random Forests. At a cost ratio of .0027, classifier built on Grow-Shrink has a mean rank of 3.53 as against the mean rank of Random Forests with 3.60 . Thus at certain cost ratios, these classifiers have a mean rank even lower than Random Forests. From the Figure 3.4 based on Nemenyi plots, we can also infer that the classifiers GS, RSMAX2 and Naïve Bayes perform as good as Random Forests.

It is observed that augmented structures learned using constraint-based methods (Grow-shrink, MMHC, RSMAX2, MMHC) outperform other augmented classifiers. From Figure 3.4, we can see that classifier built using RSMAX2, Naïve Bayes always perform as good as Random Forests in case of Risk Averse situations. In case of delay averse scenario, we show that the classifiers built using RSMAX2, Grow-Shrink and Naïve Bayes structures once again perform as good as Random Forests and this is shown using Demsar framework. And thus we show that RSMAX2, Grow-Shrink classifiers perform as good as Random Forests classifier based on our experiments with different evaluation measures like AUC and H-Measure.

3.2.5 Contributions

We have attempted to relax conditional independence assumption amongst features and build Bayesian Network structures using ten variants of six algorithms namely TAN, HC, TABU, Grow-Shrink, MMHC and RSMAX2. And the corresponding Bayesain Network classifiers are compared with best performing traditional classifiers, Random Forest, Logistic Regression and Naïve Bayes classifiers. The evaluation methods used are AUC and H-Measure. We have conducted experiments on 15 datasets and 13 classifiers. Random Forest classifier is found significantly better than Bayesian Network classifiers when evaluated using AUC or H-Measure. However, by our proposed approach on augmenting Bayesian Network structure, it is observed that RSMAX2 and Grow-Shrink are consistently better, and there is no significant difference between these classifiers and Random Forest. We have compared classifiers using different cost ratio distributions leading to Risk Averse and Delay Averse scenarios. In both the scenarios, RSMAX2 and Grow-Shrink classifiers are performing very close to Random Forests and are found better than Naïve Bayes classifier.

3.3 Cost Sensitive Neural Networks for Defect Prediction

To carry out software defect prediction, various software metrics defined and extracted from source code repositories. There are some extensive studies to figure out efficient classification algorithms and the best possible feature subset to build effective defect prediction [6] [5] [71] [72]. However, the aspect of misclassification costs has not been considered while developing prediction models and comparing their performances.

There are two types of misclassification errors. Type I error is misclassifying defect free file to be defect prone file. Here defect prone file is meant to be the file which is predicted to be defective by the prediction model. And Type II error is misclassifying defective file to be non-defect prone file.

The defect prone file will undergo quality assurance activities like expert code review,

rigorous testing etc. In general, cost of Type I error is the amount spent by project team towards quality assurance activities of the defect prone file, and the cost of Type II error is effort spent by project team to fix the defect that has been uncovered in post-release phase. As per study [73], the cost of Type II error is much more as compared to Type I error. Hence it makes sense to build cost sensitive defect prediction models.

In the recent past, quite a few studies have dealt with cost sensitive learning of software defect prediction models [74] [13] [75] [76] [77] [78] [79]. However, most of them compare the performances of the software defect prediction models using the cost sensitive approaches proposed by them. And very few compare the performance of their cost sensitive defect prediction models with traditional machine learning approaches [13]. Zheng et al. performed comparative study amongst cost sensitive defect prediction models proposed by them but not with the cost sensitive software defect prediction models proposed in previous studies [79]. As there is no comprehensive comparative study of cost sensitive defect prediction models, we attempt to conduct the comparative study in this work.

We have considered Logistic Regression, Naïve Bayes and J48 Decision Tree [13], Cost Sensitive Boosting of Neural Networks using Threshold Moving(CSBNN-TM), Cost Sensitive Boosting of Neural Networks with Weight Updation-I(CSBNNWU1), Cost Sensitive Boosting of Neural Networks with Weight Updation-II(CSBNN-WU2) [79] and four other popular machine learning classifiers, namely, Random Forest, Bayesian Network, K-Nearest Neighbors and Neural Networks. Excepting k-NN, most of the classifiers have shown to give considerably good performance [32] [80] [81] [82] [83]. In our study, excepting CSBNN-TM, CSBNN-WU1 and CSBNN-WU2, all the other classifiers are made cost sensitive using the approach suggested by Moser et al. [13].

Majority of the previous studies used performance measures like Accuracy, Recall, AUC and False-Positive Rate to compare the models built using various learning algorithms. Amongst several defect prediction models, the model with minimum misclassification cost is preferred, and hence we use Normalized Expected Cost of Misclassification (NECM) [15] as the performance measure in our comparative study.

3.3.1 Related Work

Software defect prediction is largely seen as a classification problem. Considerable amount of work has been done to identify classifiers and metrics which predict software defects accurately. Random Forest is considered to be one of the best classifier for defect prediction. It is consistently accurate and also efficient in large datasets. Guo et al. reports that defect detection rate is up to 87% when the model is built using Random Forest method [32]. Gyimothy et el. analyzed object-oriented metrics and their relation with fault-proneness. They have observed that CBO(Coupling Between Object classes) is the best metric in predicting the fault-proneness of classes [84]. Menzies et al. recommended Naïve Bayes classifier for building defect prediction model. They report 71% defect detection rate and 25% false alarm rate [6]. Few other studies used Tree based methods [85] [86] [32], J48 [87], Support Vector Machines (SVM) [88] and Fuzzy classification [89] to address this problem.

A major concern related to software defect prediction is the class imbalance problem. A study by Boehm [90] has shown that approximately 80% of the software defects are present in 20% of the source files. Dick et al. addressed class imbalance problem in two levels, namely, data level and algorithm level approaches. At data level they handled it by implementing oversampling, under-sampling and also by combining both the approaches. Oversampling, increases the examples constituting the minority class and under-sampling, reduces the number of examples constituting the majority class to obtain a more balanced dataset [91] [92]. At algorithm level, class imbalance problem is addressed by implementing cost-sensitive learning algorithms [75] [74].

Moser et al. have compared three machine learning algorithms, J48 Decision Tree, Naïve Bayes and Logistic Regression with and without taking cost sensitive analysis into consideration [13]. The study was carried out on the Eclipse datasets provided by Zimmerman et al. [42]. The cost sensitive analysis gave $> 75\%$ accuracy, $> 80\%$ recall and $< 30\%$ false positive rate. It outperformed the performance of classifiers without cost sensitive analysis. The study also observes that process metrics are more efficient defect predictors, and J48 Decision reported as the best performing classifier compared to Naïve Bayes and Logistic regression. Studies have also shown that due to a non-linear and complicated

relationship between the source code metrics and defects, Neural Networks outperform the traditional machine learning algorithms [82].

To solve the imbalanced data problem several cost-sensitive boosting algorithms have also been proposed by combining the cost factors in the boosting procedure [76] [77] [78]. Zhou and Liu have shown that threshold-moving is a good choice to train cost sensitive Neural Networks amongst over-sampling, under-sampling, and threshold-moving. Threshold moving moves the threshold such that higher cost examples are harder to be misclassified [76]. Zheng [79] has taken two different approaches to incorporate cost sensitive analysis using AdaBoost with Neural Networks. To make AdaBoost cost sensitive, 1) Apply threshold-moving at the last stage of AdaBoost algorithm (CSBNN-TM). 2) Include cost matrix into the weight updating process [78]. He has compared the algorithms performance on the basis of misclassification rate, Type I error, Type II error, and NECM(Normalized Expected Cost Minimization). The results of his study shows that Cost-sensitive boosting Neural Networks with threshold-moving results in lower misclassification cost.

Our work focuses on a comparative study of the three classifiers proposed by Zheng [79] namely, CSBNN-TM, CSBNN-WU1 and CSBNN-WU2 with the traditional machine learning algorithms which are reported as top performing classifiers in defect prediction. For the traditional machine learning algorithms, cost-sensitivity is included as suggested by Moser et al. [13]. we compare Cost-sensitive boosting Neural Networks with other algorithms using NECM to determine whether CSBNN-TM still outperforms other traditional machine learning algorithms used in defect prediction.

3.3.2 Metrics and Datasets

We have conducted experiments on software defect data belonging to 22 open source software systems. It is made sure that these software systems are diverse in terms of their programming languages, technologies and development environments. The metrics used for defect prediction and the details of the data are listed in Table 3.5.

We have obtained the datasets from tera-PROMISE repository. The first 6 datasets are

Table 3.5: Summary of Datasets

S.No	Project	Size	pndf	pdf	Language	Metrics
1	Ant-1.7	745	0.78	0.22	JAVA	CK-OO
2	Poi -3.0	442	0.63	0.37	JAVA	CK-OO
3	Camel-1.6	965	0.81	0.19	JAVA	CK-OO
4	Xalan-2.6	885	0.54	0.46	JAVA	CK-OO
5	tomcat	858	0.91	0.09	JAVA	CK-OO
6	Prop-6	660	0.9	0.1	JAVA	CK-OO
7	EclipsePDEUI	1497	0.86	0.14	JAVA	Change, CK-OO and Static Code
8	EclipseJDTCore	997	0.79	0.21	JAVA	Change, CK-OO and Static Code
9	Mylyn	1862	0.87	0.13	JAVA	Change, CK-OO and Static Code
10	Lucene	691	0.91	0.09	JAVA	Change, CK-OO and Static Code
11	KC1	2109	0.85	0.15	C++	McCabe and Halstead
12	KC2	522	0.8	0.2	C++	McCabe and Halstead
13	KC3	194	0.82	0.18	JAVA	McCabe and Halstead
14	CM1	498	0.9	0.1	C	McCabe and Halstead
15	PC1	705	0.91	0.09	C	McCabe and Halstead
16	PC2	745	0.98	0.02	C	McCabe and Halstead
17	PC3	1077	0.88	0.12	C	McCabe and Halstead
18	PC4	1458	0.88	0.12	C	McCabe and Halstead
19	JM1	7782	0.79	0.21	C	McCabe and Halstead
20	Eclipse-2.0	6729	0.85	0.15	JAVA	Complexity
21	Eclipse-2.1	7888	0.89	0.11	JAVA	Complexity
22	Eclipse-3.0	10592	0.85	0.15	JAVA	Complexity

taken from Marian Jureckzo's CK and OO metrics set and this dataset consists of 21 metrics (wmc, dit, noc, cbo, rfc, lcom, ca, ce, npm, lcom3, loc, dam, moa, mfa, cam, ic, cbm, amc, max cc, avg cc and bug) [93] [8]. The next 4 datasets(7-10) belong to the Eclipse project and consist of 37 change, CK-OO and static code metrics (Entropy, WEntropy, LinEntropy, LogEntropy, ExpEntropy, numberOfVersions, numberOfFixes, numberOfRefactorings, numberOfAuthors, linesAdded, maxLinesAdded, avgLinesAdded, linesRemoved, maxLinesRemoved, avgLinesRemoved, codeChurn, maxCodeChurn, avgCodeChurn, age, weightedAge cbo, dit, fanIn, fanOut, lcom, noc, numberOfAttributes numberOfAttributesInherited numberOfLinesOfCode, numberOfMethods numberOfMethodsInherited, numberOfPrivateAttributes, numberOfPrivateMethods, numberOfPublicAttributes, numberOfPublicMethods, rfc, wmc and bugs). We have taken 9 datasets (11-19) from the mission critical projects of NASA which is part of NASA Metrics Program. They consist of 21 McCabe [9] and Halstead [10] metrics. These metrics mainly represent the complexity of the software. We request the reader to refer [94] for the details of NASA datasets. The last 3 datasets(20-22) that we have considered is published by Zimmerman et al. [42]. Though it consists of 198 features, we made use of only 31 complexity metrics (ACD, FOUT avg, FOUT max, FOUT sum, MLOC avg, MLOC max, MLOC sum, NBD avg,

NBD max, NBD sum, NOF avg, NOF max, NOF sum, NOI, NOM avg, NOM max, NOM sum, NOT, NSF avg, NSF max, NSF sum, NSM avg, NSM max, NSM sum, PAR avg, PAR max, PAR sum, TLOC, VG avg, VG max, VG sum and bugs) for our study. The *pndf* and *pdf* columns in the table indicate the percentage of non-defective and defective modules in each dataset respectively.

3.3.3 Experimental Setup

AdaBoost is an adaptive boosting algorithm first introduced in 1995 by Freund [95], and is the most widely used boosting algorithm. It sequentially trains the individual base classifiers, changing the weights of each example in each training round such that a correctly classified example has its weight decreased while incorrectly classified example has its weight increased. It then combines the base classifiers to form the composite classifier by simplistic or weighted voting scheme.

Algorithm 1 Boosting Neural Networks with AdaBoost

Require: A Training set T containing N examples $(x_n, y_n), n = 1, 2, ..., N$ where x_n is a vector of attribute values and $y_n \epsilon \{1, -1\}$ is the class label
Initialization: Let the weight vector: $W_1(n) = \frac{1}{N}$ for $n = 1, 2, ..., N$, $y \epsilon Y - \{y_n\}$
for $t = 1, 2, .., T$ **do**
Train neural network using weight distribution W_t and obtain the hypothesis $h_t \epsilon [-1, 1]$.
Calculate the error of h_t: $\varepsilon_t = \sum_{n:h_t(x_n) \neq y_n} W_t(n)$
if $\varepsilon_t > 0.5$ set $T = t - 1$ and abort loop
Set the weight updating parameter $\alpha_t = \frac{1}{2} \log\left(\frac{1-\varepsilon_t}{\varepsilon_t}\right)$
Update the normalized weight vector $W_{t+1}(n) = \frac{W_t(n) e^{-\alpha_t h_t(x_n) y_n}}{Z_t}$
where $n \leftarrow 1 \, to \, N$, Z_t is a normalization factor chosen so that W_{t+1} becomes a proper distribution.
end for
Ensure: The final classifier: $h_f(x) = sign\left(\sum_{t=1}^{t} \alpha_t h_t(x)\right)$

Cost Sensitive Boosting of Neural Networks (CSBNN) is one of the approaches to improve the performance of Neural Network classifiers. CSBNN uses Neural Networks as the weak classifier in the AdaBoost algorithm. It alters the weight distribution of the training sample during each training process such that the weights of misclassified examples are increased and those of the correctly classified examples are decreased. This kind of weight adjustment makes the learner to concentrate on different examples in each training round

which leads to diverse classifiers. Finally, the individual classifiers are combined to form the composite classifier by weighted or simple voting schemes. We refer the reader Zheng [79] for detailed description of the algorithm. However the same is discussed in Algorithm 1 for the sake of continuity and completeness of our work.

The costs associated with the misclassification are represented as $C(i, j)$ where the module belonging to class i has been misclassified as class j. Threshold moving is incorporated at the last step of the AdaBoost algorithm to generate a classification model which would give the minimum misclassification cost. In CSBNN-WU1 and CSBNN-WU2, a separate cost term is added while updating the weights of the individual examples. The cost term is kept 1 if the example had been correctly classified. In case of misclassification it is replaced by the misclassification cost. The parameters of the algorithm are tuned properly to fit the various systems such that sufficient emphasis is given to the minority class and also to ensure that the algorithm does not become overly biased towards the minority class.

- **CSBNN-TM:** This classifier uses Threshold Moving for making the Neural Network classifier cost sensitive. It takes into consideration the cost matrix to move the output threshold of the Neural Network classifier such that the high cost examples(Type II error) are harder to misclassify.

- **CSBNN-WU1:** This classifier introduces the cost matrix directly into the weight updating process. It boosts weight for samples with higher misclassification costs such that performance over those samples can be improved. It does not consider the weight updating parameter in the formulation.

- **CSBNN-WU2:** This classifier works exactly like CSBNN-WU1 except that it includes the weight updating parameter while boosting the weights for samples.

  While the first method doesn't demand retraining base Neural Network classifiers, other two methods demand retraining all base Neural Networks when the cost changes.

For the implementation of CSBNN-TM, CSBNN-WU1 and CSBNN-WU2 algorithms, as

suggested by Zheng [79], we use a Back Propagation Neural Network consisting of three layers. The input layer has as many nodes as the number of metrics used for prediction. The nodes in hidden layer are decided empirically by changing the number of nodes in the hidden layer. Another parameter found through experiments is the maximum number of iterations for the Neural Network. A combination of both the parameter values is sought, which would give the most optimal classification for each system. For most of the systems, we kept the hidden nodes to be 11 and fixed the maximum number of iterations to be 300. Only in case of some very large systems like the Eclipse versions 2.0, 2.1 and 3.0, the optimal number of hidden nodes is 30. This experiment is conducted to ensure that the Neural Network gives its best performance. The output layer consists of one node to indicate if the software module is defect prone or not.

The approach suggested by Moser et al. [13], is implemented using Bayesian Network, J48 variant of Decision Tree, K-Nearest Neighbors, Logistic Regression, Naïve Bayes, Neural Network and Random Forest. For the implementation of Bayesian Network, the hill climbing algorithm is used. Similar to the CSBNN-TM, the Neural Network implementation used here is a Back Propagation Neural Network with three layers. For training and testing purposes, each dataset is divided into 70:30 ratio randomly and all the experiments are averaged over 5 runs. All the above mentioned implementations is carried out using R.

To compare the performances of the classifiers, we have chosen the Normalized Expected Cost of Misclassification (NECM). For the binary classification problem, there exists two types of errors, Type I error and the Type II error. The Type I error is the error of misclassifying a module which is actually defect free to be defect prone. The Type II error is the error of misclassifying a module which is actually defect prone to be defect free. The costs associated with these two errors are CI and CII respectively. It is most often the case that $CII >> CI$.

$$Misclassification\,Rate(MR) = \frac{FP + FN}{TP + TN + FP + FN} \tag{3.2}$$

$$Type\,I\,error(Err_I) = \frac{FP}{TN + FP} \tag{3.3}$$

$$Type\,II\,error(Err_{II}) = \frac{FN}{TP + FN} \tag{3.4}$$

$$Expected\,Cost\,of\,Misclassification(ECM) = C_I Err_I P_{ndf} + C_{II} Err_{II} P_{df} \tag{3.5}$$

$$Normalized\,Expected\,Cost\,of\,Misclassification(NECM) = Err_I P_{ndf} + \frac{C_{II}}{C_I} Err_{II} P_{df} \tag{3.6}$$

where P_{ndf} and P_{df} are the prior probabilities of the not-defect prone and defect-prone modules in the dataset. The NECM is a unified measure which takes the misclassification costs of two classes into account along with their prior probabilities. We refer the reader [15] for complete derivation of NECM. The costs are however expressed in the form of a ratio of C_{II} to C_I. The cost ratio varies with the type project and organization.

3.3.4 Results and Discussion

We have built defect prediction models using 10 learning algorithms - CSBNN-TM, CSBNN-WU1, CSBNN-WU2, Logistic Regression, Naïve Bayes, Random Forest, Bayesian Network, Neural Networks, k-Nearest Neighbors and Decision Tree. We have built these models for each of 22 data sets. The results are plotted with cost ratio as x-axis and NECM as y-axis for all 22 systems and they are shown in Figure 3.5 through Figure 3.8. We have also shown the average ranking of the classifiers for cost ratio of 5 and 10 in Table 3.11.

To compare the performance differences amongst classifiers Nemenyi tests were performed. It calculates the critical distance from the average rank of each classifier and states that for any two classifiers, their performances are significantly different if their average ranks lie at least critical distance apart from each other. The critical difference is calculated as

$$CD = q_{\alpha,\infty,L}\sqrt{\frac{L\,(L+1)}{12K}} \tag{3.7}$$

To prove the significance of the results obtained in terms of NECM, we carry out Friedman Test on the results obtained for cost ratio 5 and cost ratio 10 separately. The null

hypothesis of the test being that no two classifiers are significantly different from each other. For the test carried out on results corresponding to cost ratio 5, we obtain a p-value of 2.2e-16 at a confidence level of 95%. Since the p-value is less than 0.05, we strongly reject the null hypothesis and state that at least for two classifiers, the results obtained are significantly different. The same procedure was repeated for results corresponding to a cost ratio of 10. The p-value for Friedman Test came out to be $< 2.2e - 16$. To find out the classifiers which gave significantly different performances, we next carried out the Nemenyi test.

From the above results we observe that the performance of the classifiers vary as shown in Figure 3.9. However, a trend is followed when we observe the average ranks of the classifiers at cost ratios 5 and 10. Overall for the entire range of cost ratios, excepting few cases, we observe that Random Forest is giving the best performance in terms of NECM closely followed by Logistic Regression and Bayesian Network. The significance tests show that there is no significant difference with the performances of Random Forest with Logistic Regression and Bayesian Network. However, the average rank of Random Forest at 5% cost ratio is 1.8636 while the value of next best classifier (Logistic Regression) is 2.6818 which is quite high compared to RF. We also observe that the cost sensitive boosting Neural Networks classifiers give a significantly poorer performance when compared to Random Forest.

Among cost sensitive learners, CSBNN-TM attains lower mean rank at 5 and 10 cost ratios, respectively 6.1364 and 5.6818. Another point to be observed is that there is a considerable improvement in the performances of CSBNN-TM, CSBNN-WU1 and CSBNN-WU2 when the cost ratio goes up from 5 to 10. We realize that this improvement in terms of average ranking based on NECM is obeserved because the classifiers, namely, KNN, Decision Tree and Naïve Bayes have higher Type II error rates in comparison to the other classifiers. However, the cost sensitive boosting Neural Network algorithms still perform significantly lower than traditional machine learning algorithms like Random Forest, Logistic Regression and Bayesian Networks. Hence, we conclude that cost sensitive Neural Network may not be an effective approach to build software defect prediction models.

Table 3.6: Eclipse - Complexity Metrics: Cost ratio and NECM

Projects	Cost Ratio	CSBNN TM	BN	DT	KNN	LR	NB	NNET	RF	CSBNN WU1	CSBNN WU2
Eclipse 2.0	1	0.1595	0.1341	0.1475	0.1439	0.1306	0.1448	0.3421	0.1247	0.1762	0.1511
	2	0.3367	0.2360	0.2445	0.2601	0.2310	0.2393	0.5395	0.2173	0.3112	0.3193
	3	0.4515	0.3205	0.3396	0.3361	0.3162	0.3344	0.6418	0.2941	0.4226	0.3949
	4	0.5247	0.4024	0.4215	0.4121	0.3868	0.4285	0.6492	0.3459	0.4790	0.4609
	5	0.5671	0.4563	0.4999	0.4721	0.4502	0.5217	0.7500	0.3935	0.5235	0.5221
	6	0.6045	0.5171	0.5610	0.5118	0.4989	0.6148	0.7664	0.4371	0.5747	0.5307
	7	0.6239	0.5661	0.6166	0.5492	0.5448	0.7088	0.7772	0.4770	0.5987	0.5976
	8	0.6353	0.5829	0.6779	0.5866	0.5923	0.8010	0.8110	0.5092	0.6275	0.5929
	9	0.6506	0.6030	0.7323	0.6232	0.6261	0.8929	0.8690	0.5421	0.6223	0.6201
	10	0.6658	0.6148	0.7896	0.6599	0.6281	0.9853	0.8507	0.5631	0.6467	0.6620
Eclipse 2.1	1	0.1327	0.1079	0.1274	0.1157	0.1064	0.1312	0.3205	0.1080	0.1450	0.1256
	2	0.3558	0.2053	0.2192	0.2390	0.2048	0.2144	0.7050	0.2072	0.3612	0.4169
	3	0.4939	0.2960	0.3050	0.3170	0.2957	0.2966	0.8193	0.2946	0.4467	0.5663
	4	0.5911	0.3777	0.3939	0.3930	0.3758	0.3784	0.8124	0.3700	0.6635	0.6264
	5	0.6440	0.4521	0.4706	0.5087	0.4500	0.4603	0.8581	0.4308	0.7406	0.6759
	6	0.6896	0.5022	0.5419	0.5588	0.5143	0.5412	0.8362	0.4731	0.7251	0.6781
	7	0.7224	0.5569	0.6065	0.6041	0.5603	0.6228	0.8172	0.5206	0.7492	0.7272
	8	0.7341	0.6037	0.6681	0.6481	0.6062	0.7037	0.8801	0.5622	0.8066	0.7768
	9	0.7455	0.6414	0.7356	0.6909	0.6509	0.7851	0.9014	0.6025	0.8116	0.7847
	10	0.7682	0.6708	0.7922	0.7339	0.6837	0.8659	0.8779	0.6348	0.8223	0.7881
Eclipse 3.0	1	0.1646	0.1452	0.1564	0.1614	0.1443	0.1615	0.4505	0.1344	0.1826	0.1640
	2	0.1516	0.1470	0.1936	0.1570	0.1482	0.2085	0.3049	0.1427	0.1821	0.1574
	3	0.1497	0.1486	0.2228	0.1682	0.1491	0.2542	0.3377	0.1435	0.1774	0.1531
	4	0.1486	0.1493	0.2514	0.1794	0.1501	0.2993	0.4303	0.1437	0.1895	0.1574
	5	0.1488	0.1494	0.2817	0.1543	0.1508	0.3442	0.4753	0.1450	0.1803	0.1616
	6	0.1480	0.1504	0.2920	0.1563	0.1501	0.3889	0.5440	0.1444	0.1713	0.1626
	7	0.1480	0.1508	0.3099	0.1583	0.1508	0.4327	0.5768	0.1443	0.1856	0.1670
	8	0.1480	0.1507	0.3311	0.1604	0.1504	0.4758	0.6152	0.1448	0.1763	0.1647
	9	0.1480	0.1513	0.3503	0.1624	0.1502	0.5211	0.5368	0.1449	0.1732	0.1735
	10	0.1480	0.1520	0.3693	0.1644	0.1495	0.5652	0.8574	0.1452	0.1757	0.1609

Table 3.7: Eclipse - Change and Static Code Metrics: Cost ratio and NECM

Projects	Cost Ratio	CSBNN TM	BN	DT	KNN	LR	NB	NNET	RF	CSBNN WU1	CSBNN WU2
Eclipse PDE	1	0.1662	0.1362	0.1634	0.1577	0.1376	0.1729	0.3735	0.1381	0.2135	0.1588
	2	0.3959	0.2553	0.2657	0.2851	0.2467	0.2609	0.6100	0.2340	0.4340	0.3836
	3	0.5687	0.3352	0.3703	0.3692	0.3395	0.3461	0.7261	0.3273	0.5117	0.4796
	4	0.6456	0.4288	0.4723	0.4533	0.4211	0.4287	0.7935	0.4065	0.6708	0.5876
	5	0.7184	0.5003	0.5797	0.5621	0.5085	0.5136	0.8226	0.4655	0.6949	0.7321
	6	0.7487	0.5705	0.6706	0.6055	0.5505	0.5989	0.8682	0.5152	0.7567	0.7594
	7	0.7650	0.6090	0.7604	0.6474	0.5825	0.6801	0.8403	0.5294	0.8039	0.7676
	8	0.7771	0.6543	0.8567	0.6894	0.6454	0.7645	0.9008	0.5629	0.8397	0.8130
	9	0.8008	0.7108	0.9545	0.7314	0.6768	0.8457	0.8508	0.5941	0.8517	0.8294
	10	0.8152	0.7313	1.0494	0.7734	0.7104	0.9302	0.8855	0.6125	0.8148	0.7997
Eclipse JDT	1	0.1796	0.1452	0.1949	0.1717	0.1417	0.1730	0.4854	0.1353	0.2361	0.1842
	2	0.3783	0.2481	0.3180	0.2902	0.2437	0.2934	0.6462	0.2252	0.4223	0.3385
	3	0.4883	0.2973	0.4418	0.3652	0.3152	0.4115	0.7008	0.3163	0.5218	0.4415
	4	0.6181	0.3632	0.5635	0.4403	0.3414	0.5317	0.7840	0.3587	0.6202	0.6183
	5	0.6768	0.4394	0.6859	0.4854	0.4213	0.6513	0.7139	0.4083	0.6665	0.5956
	6	0.7046	0.4906	0.7925	0.5269	0.4826	0.7709	0.7415	0.4307	0.7229	0.7124
	7	0.7420	0.5438	0.8916	0.5669	0.5453	0.8905	0.7614	0.4545	0.7264	0.7030
	8	0.7496	0.6137	1.0037	0.6070	0.5841	1.0101	0.8007	0.4863	0.7735	0.7348
	9	0.7755	0.6129	1.1195	0.6471	0.6393	1.1304	0.8996	0.4870	0.7477	0.7372
	10	0.7780	0.6124	1.2336	0.6871	0.6650	1.2499	0.8107	0.5085	0.7607	0.7836
Mylyn	1	0.1477	0.1272	0.1580	0.1352	0.1291	0.1556	0.3381	0.1229	0.1529	0.1518
	2	0.3155	0.2359	0.2505	0.2559	0.2345	0.2396	0.6689	0.2133	0.2816	0.2810
	3	0.4421	0.3130	0.3417	0.3334	0.3127	0.3174	0.7006	0.2896	0.3832	0.3510
	4	0.5156	0.3652	0.4285	0.4109	0.3646	0.3948	0.7292	0.3370	0.4695	0.4392
	5	0.5959	0.4262	0.5183	0.4593	0.4081	0.4700	0.7946	0.3826	0.5019	0.5060
	6	0.6215	0.4527	0.5913	0.4935	0.4244	0.5488	0.8212	0.4158	0.5932	0.5289
	7	0.6528	0.4559	0.6517	0.5273	0.4693	0.6210	0.8515	0.4480	0.5915	0.5525
	8	0.6617	0.5033	0.7196	0.5612	0.5057	0.6989	0.9007	0.4787	0.6543	0.6208
	9	0.6828	0.5314	0.7900	0.5950	0.5404	0.7709	0.8612	0.5133	0.6522	0.6311
	10	0.7006	0.5698	0.8583	0.6288	0.5724	0.8406	0.9016	0.5305	0.6534	0.6790
Lucene	1	0.1111	0.0825	0.0982	0.0868	0.0782	0.1405	0.2621	0.0817	0.1030	0.1218
	2	0.2768	0.1547	0.1617	0.1537	0.1403	0.1903	0.6454	0.1528	0.1906	0.2064
	3	0.4039	0.2205	0.2235	0.2070	0.2022	0.2391	0.5086	0.2206	0.2884	0.2879
	4	0.4678	0.2922	0.2848	0.2602	0.2691	0.2889	0.6467	0.2725	0.3823	0.3579
	5	0.5649	0.3498	0.3461	0.4130	0.3342	0.3377	0.6353	0.3253	0.4480	0.4335
	6	0.6022	0.3927	0.3844	0.4467	0.3636	0.3856	0.7850	0.3571	0.4679	0.4847
	7	0.6441	0.4435	0.4394	0.4843	0.3981	0.4335	0.7821	0.3853	0.5449	0.5104
	8	0.6951	0.4792	0.4867	0.5220	0.4235	0.4823	0.7517	0.4219	0.6063	0.5197
	9	0.7254	0.5038	0.5331	0.5596	0.4788	0.5301	0.9035	0.4603	0.6136	0.6371
	10	0.7294	0.5346	0.5863	0.5972	0.5156	0.5780	0.8931	0.4701	0.6577	0.5942

Table 3.8: NASA - McCabe and Halstead Metrics: Cost ratio and NECM

Projects	Cost Ratio	CSBNN TM	BN	DT	KNN	LR	NB	NNET	RF	CSBNN WU1	CSBNN WU2
KC1	1	0.1553	0.1472	0.1527	0.1674	0.1424	0.1766	0.2916	0.1424	0.1656	0.1595
	2	0.3595	0.2561	0.2634	0.2969	0.2544	0.2743	0.6629	0.2463	0.4383	0.4113
	3	0.4979	0.3283	0.3662	0.3865	0.3534	0.3711	0.7262	0.3187	0.5554	0.5222
	4	0.5561	0.4104	0.4706	0.4732	0.4091	0.4671	0.7702	0.3726	0.6025	0.5592
	5	0.5768	0.4495	0.5697	0.5002	0.4646	0.5601	0.7507	0.4169	0.6218	0.5856
	6	0.5899	0.4948	0.6579	0.5487	0.5007	0.6565	0.7411	0.4457	0.7050	0.5881
	7	0.5992	0.5182	0.7138	0.5910	0.5405	0.7531	0.8146	0.4780	0.7048	0.6233
	8	0.6188	0.5422	0.7635	0.6448	0.5482	0.8495	0.8062	0.5023	0.6642	0.6319
	9	0.6274	0.5459	0.7980	0.6909	0.5551	0.9462	0.7655	0.5217	0.6460	0.6259
	10	0.6361	0.5727	0.8256	0.7418	0.5721	1.0395	0.8016	0.5320	0.6858	0.6635
KC2	1	0.1704	0.1753	0.1774	0.1889	0.1614	0.1695	0.4279	0.1674	0.2474	0.1815
	2	0.3130	0.2700	0.2882	0.3005	0.2757	0.2866	0.6356	0.2768	0.5469	0.3624
	3	0.4309	0.3219	0.3891	0.4002	0.3396	0.4063	0.7236	0.3531	0.4966	0.5437
	4	0.5569	0.3693	0.4941	0.4974	0.3905	0.5260	0.7403	0.3933	0.7287	0.5735
	5	0.5967	0.3906	0.5671	0.5043	0.4045	0.6398	0.7817	0.4405	0.6620	0.6579
	6	0.6203	0.4400	0.6552	0.5585	0.4501	0.7583	0.7589	0.4623	0.7394	0.6817
	7	0.6782	0.4691	0.7504	0.6146	0.4864	0.8755	0.7484	0.4873	0.7362	0.7232
	8	0.7024	0.4986	0.8455	0.6549	0.4976	0.9927	0.7741	0.4858	0.7252	0.7462
	9	0.7219	0.5354	0.9360	0.7034	0.5260	1.1099	1.0469	0.5022	0.7673	0.8056
	10	0.7539	0.5682	1.0209	0.7584	0.5458	1.2284	1.0346	0.5374	0.7783	0.7699
KC3	1	0.1856	0.1658	0.1990	0.2200	0.1636	0.2048	0.4371	0.2105	0.2730	0.1890
	2	0.3970	0.2807	0.3319	0.4196	0.2711	0.3203	0.8144	0.3010	0.8144	0.5036
	3	0.5820	0.3751	0.4543	0.5394	0.4124	0.4314	0.8144	0.4051	0.8144	0.7099
	4	0.7258	0.5012	0.5871	0.6592	0.5003	0.5490	0.8144	0.5479	0.8144	0.7856
	5	0.8696	0.6352	0.7166	0.8539	0.6146	0.6601	0.8110	0.6719	0.8144	0.8144
	6	1.0134	0.7652	0.8360	0.9228	0.6817	0.7712	0.8043	0.6812	0.8144	0.7987
	7	1.0576	0.8108	0.9419	0.9917	0.6820	0.8891	0.8144	0.7472	0.8144	0.8144
	8	1.1690	0.8193	1.0646	1.0606	0.6892	0.9732	0.8443	0.6896	0.8144	0.8916
	9	1.2803	0.8246	1.1182	1.1294	0.7200	1.0843	0.8144	0.7072	0.8144	0.8043
	10	1.3917	0.8393	1.1499	1.1983	0.7465	1.1920	0.8144	0.7588	0.8144	0.8144
CM1	1	0.1228	0.1052	0.1027	0.0998	0.1035	0.1364	0.3087	0.1131	0.2102	0.1150
	2	0.3501	0.1916	0.1960	0.2076	0.1933	0.2156	0.7517	0.2174	0.4789	0.4591
	3	0.4953	0.2669	0.2893	0.2907	0.2655	0.2882	0.8900	0.3173	0.5616	0.5665
	4	0.5120	0.3485	0.3826	0.3739	0.3435	0.3559	0.7576	0.4018	0.6349	0.5020
	5	0.5981	0.3985	0.4620	0.5008	0.4188	0.4307	0.7993	0.4630	0.6826	0.5930
	6	0.6244	0.4326	0.5435	0.5583	0.4610	0.5069	0.8654	0.4720	0.7145	0.6577
	7	0.6325	0.5005	0.6218	0.6040	0.5006	0.5804	0.7921	0.5328	0.8034	0.6778
	8	0.6636	0.5333	0.6979	0.6510	0.5164	0.6350	0.9053	0.5474	0.7906	0.7098
	9	0.6712	0.5455	0.7710	0.6966	0.5367	0.7076	0.8652	0.6003	0.7614	0.7314
	10	0.6786	0.5886	0.8264	0.7423	0.4992	0.7775	0.9034	0.6287	0.8036	0.7193

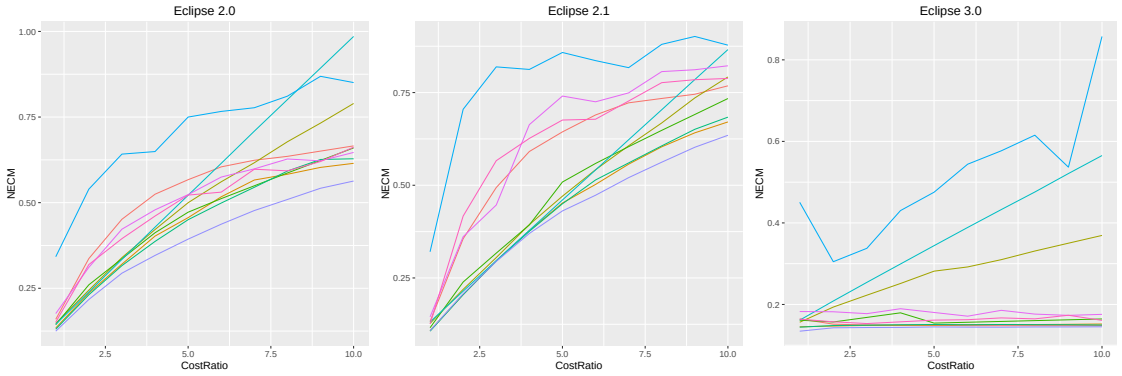
Table 3.9: NASA - McCabe and Halstead Metrics: Cost ratio and NECM (continued..)

Projects	Cost Ratio	CSBNN TM	BN	DT	KNN	LR	NB	NNET	RF	CSBNN WU1	CSBNN WU2
PC1	1	0.0924	0.0876	0.1005	0.0931	0.0893	0.1211	0.5827	0.0810	0.1757	0.1096
	2	0.2081	0.1585	0.1661	0.2153	0.1599	0.1735	0.9135	0.1400	0.2703	0.2945
	3	0.2892	0.2130	0.2308	0.2970	0.2029	0.2259	0.9039	0.2003	0.4717	0.3118
	4	0.3185	0.2755	0.2956	0.3788	0.2550	0.2782	0.8782	0.2783	0.6648	0.4999
	5	0.3540	0.2960	0.3603	0.5364	0.2943	0.3297	0.9135	0.3290	0.5906	0.6269
	6	0.4293	0.3296	0.4103	0.5871	0.3321	0.3811	0.8381	0.3429	0.9032	0.6191
	7	0.4543	0.3857	0.4650	0.6379	0.3973	0.4325	0.9135	0.3538	0.7954	0.5359
	8	0.4953	0.4148	0.5040	0.6886	0.4454	0.4839	0.8163	0.3598	0.8273	0.8630
	9	0.5203	0.4163	0.5424	0.7394	0.4800	0.5354	0.9135	0.3859	0.7107	0.8362
	10	0.5982	0.4132	0.5855	0.7901	0.4638	0.5868	0.8741	0.3938	0.8942	0.8749
PC2	1	0.0259	0.0241	0.0268	0.0215	0.0241	0.1353	0.2403	0.0215	0.0305	0.0264
	2	0.0988	0.0474	0.0483	0.0456	0.0456	0.1617	0.9305	0.0492	0.0750	0.1248
	3	0.1876	0.0689	0.0697	0.0671	0.0680	0.1827	0.8937	0.0742	0.2323	0.1444
	4	0.2230	0.0921	0.0921	0.0886	0.0903	0.2046	0.8659	0.1046	0.2788	0.1889
	5	0.2700	0.1172	0.1136	0.1955	0.1118	0.2238	0.9026	0.1305	0.2826	0.2166
	6	0.2920	0.1395	0.1368	0.2159	0.1333	0.2420	0.6782	0.1512	0.3926	0.2274
	7	0.3619	0.1628	0.1583	0.2363	0.1557	0.2612	0.8244	0.1712	0.4795	0.2707
	8	0.3714	0.1878	0.1798	0.2567	0.1771	0.2785	0.8952	0.1923	0.2600	0.4198
	9	0.4288	0.2111	0.2012	0.2771	0.1986	0.2968	0.8366	0.2062	0.2111	0.3208
	10	0.4385	0.2326	0.2227	0.2975	0.2210	0.3151	0.9127	0.1895	0.2991	0.4857
PC3	1	0.1410	0.1331	0.1633	0.1324	0.1373	0.5825	0.5764	0.1364	0.3053	0.1389
	2	0.2732	0.2549	0.2651	0.2916	0.2514	0.6212	0.8522	0.2347	0.4338	0.3113
	3	0.3494	0.3372	0.3642	0.3842	0.3373	0.6447	0.7649	0.3109	0.5821	0.4149
	4	0.4333	0.4024	0.4495	0.4768	0.4150	0.6625	0.8570	0.3740	0.8187	0.6249
	5	0.5182	0.4443	0.5394	0.5542	0.4562	0.6854	0.8756	0.4136	0.7896	0.6349
	6	0.5985	0.4762	0.5733	0.5997	0.4795	0.6995	0.8756	0.4398	0.8756	0.6405
	7	0.6875	0.4942	0.6553	0.6452	0.5012	0.7123	0.8756	0.4758	0.7757	0.6181
	8	0.7549	0.5212	0.6919	0.6907	0.5177	0.7270	0.8756	0.4961	0.7767	0.8248
	9	0.8195	0.5377	0.7249	0.7362	0.5091	0.7417	0.8744	0.5106	0.7766	0.8194
	10	0.8790	0.5700	0.7869	0.7817	0.5135	0.7493	0.8714	0.5284	0.8107	0.7383
PC4	1	0.0610	0.0078	0.0550	0.0276	0.0062	0.1064	0.3136	0.0197	0.0778	0.0540
	2	0.1481	0.0343	0.0905	0.1045	0.0317	0.1117	0.8833	0.0707	0.1717	0.1642
	3	0.2261	0.1050	0.0937	0.1045	0.0769	0.1179	0.8227	0.1232	0.3133	0.2204
	4	0.2775	0.1826	0.0937	0.1050	0.1353	0.1211	0.9007	0.1601	0.3897	0.3308
	5	0.3181	0.2477	0.0937	0.3735	0.1857	0.1252	0.8879	0.1897	0.3945	0.2806
	6	0.3481	0.3017	0.0952	0.3771	0.2336	0.1268	0.9949	0.2167	0.5374	0.4445
	7	0.4127	0.3631	0.0963	0.3776	0.2668	0.1273	0.8511	0.2375	0.4995	0.3330
	8	0.4452	0.4110	0.1015	0.3781	0.3007	0.1289	0.9097	0.2614	0.6069	0.4803
	9	0.4573	0.4433	0.1088	0.3781	0.3231	0.1315	0.7469	0.2823	0.5972	0.5113
	10	0.4755	0.4619	0.1088	0.3781	0.3433	0.1325	0.7000	0.2937	0.6370	0.4387
JM1	1	0.2258	0.2119	0.2291	0.2429	0.2104	0.2185	0.5427	0.2089	0.4469	0.2174
	2	0.4282	0.3990	0.3924	0.4709	0.3908	0.3952	0.7741	0.3726	0.7020	0.6471
	3	0.5596	0.5370	0.5346	0.5998	0.5193	0.5717	0.7787	0.5052	0.7855	0.7576
	4	0.6563	0.6457	0.6588	0.7285	0.6249	0.7461	0.7687	0.6023	0.7813	0.7740
	5	0.7378	0.7490	0.7486	0.7896	0.7094	0.9209	0.7973	0.6662	0.7855	0.7806
	6	0.7525	0.7897	0.8226	0.8493	0.7847	1.0954	0.7878	0.7119	0.7930	0.7878
	7	0.7674	0.7899	0.8682	0.9083	0.8005	1.2703	0.7866	0.7525	0.7940	0.7862
	8	0.7802	0.7897	0.8949	0.9673	0.7875	1.4451	0.8505	0.7709	0.7912	0.7857
	9	0.7886	0.7909	0.9285	1.0264	0.7851	1.6181	0.7870	0.7978	0.7897	0.7854
	10	0.7925	0.7868	0.9334	1.0854	0.7851	1.7904	0.8182	0.8232	0.7935	0.7883

Table 3.10: CK and OO Metrics: Cost ratio and NECM

Projects	Cost Ratio	CSBNN TM	BN	DT	KNN	LR	NB	NNET	RF	CSBNN WU1	CSBNN WU2
Ant 1.7	1	0.2099	0.1749	0.1916	0.2045	0.1846	0.2030	0.3004	0.1700	0.2485	0.1884
	2	0.3730	0.2744	0.2922	0.3061	0.2872	0.3011	0.6309	0.2585	0.3772	0.3549
	3	0.4583	0.3541	0.3918	0.3846	0.3692	0.3978	0.6926	0.3386	0.5048	0.4482
	4	0.5183	0.3836	0.4841	0.4631	0.4041	0.4958	0.7238	0.3857	0.5188	0.5058
	5	0.5484	0.4641	0.5819	0.5156	0.4474	0.5878	0.7618	0.4287	0.6173	0.5381
	6	0.5773	0.4998	0.6778	0.5548	0.5104	0.6678	0.7562	0.4643	0.6160	0.6016
	7	0.6111	0.5188	0.7513	0.5869	0.5060	0.7547	0.8900	0.4955	0.6311	0.6028
	8	0.6473	0.5328	0.8445	0.6191	0.5190	0.8396	0.7197	0.5176	0.6994	0.6486
	9	0.6613	0.5509	0.9377	0.6512	0.5914	0.9276	0.7217	0.5641	0.7429	0.6676
	10	0.6781	0.5850	0.9343	0.6833	0.6497	1.0165	0.8085	0.5846	0.7173	0.6999
Camel 1.6	1	0.2301	0.1916	0.2085	0.2252	0.1937	0.2026	0.4229	0.1906	0.2324	0.2287
	2	0.4614	0.3654	0.3649	0.4485	0.3648	0.3563	0.6646	0.3532	0.4550	0.4468
	3	0.5769	0.5228	0.5298	0.5700	0.4857	0.5049	0.7092	0.4604	0.6102	0.5336
	4	0.6181	0.6135	0.6452	0.6921	0.5958	0.6538	0.7188	0.5445	0.6199	0.5559
	5	0.6537	0.6919	0.7700	0.7278	0.6547	0.7909	0.7692	0.5879	0.6699	0.6668
	6	0.6623	0.7534	0.8815	0.7708	0.6398	0.9410	0.8169	0.6338	0.7362	0.6993
	7	0.6713	0.7915	0.9033	0.8226	0.6965	1.0905	0.8516	0.6628	0.7635	0.6959
	8	0.6807	0.8045	0.9270	0.8744	0.7686	1.2406	0.9021	0.6841	0.7362	0.7252
	9	0.6908	0.8052	0.9737	0.9255	0.7707	1.3826	0.8217	0.6851	0.7022	0.7614
	10	0.7157	0.8052	1.0101	0.9766	0.7716	1.5306	0.8698	0.6984	0.8110	0.7438
Poi 3.0	1	0.2132	0.2944	0.2151	0.2352	0.2266	0.4477	0.4167	0.1905	0.2548	0.2570
	2	0.2764	0.3439	0.3211	0.3022	0.3339	0.8190	0.3800	0.2590	0.3029	0.2943
	3	0.3147	0.3697	0.4242	0.3396	0.3840	1.1925	0.8647	0.3096	0.3759	0.3215
	4	0.3388	0.3891	0.5018	0.3771	0.3848	1.5300	0.4784	0.3438	0.4312	0.3806
	5	0.3469	0.3949	0.5770	0.3535	0.3960	1.8815	0.5470	0.3567	0.3837	0.4397
	6	0.3563	0.4052	0.5783	0.3757	0.3995	2.2349	0.6762	0.3665	0.4852	0.4497
	7	0.3607	0.4074	0.6031	0.3898	0.4085	2.6010	0.5909	0.3895	0.4281	0.4153
	8	0.3536	0.3814	0.6278	0.4038	0.4174	2.9182	0.9782	0.3948	0.4112	0.4345
	9	0.3597	0.3843	0.6734	0.4178	0.4036	3.2553	1.0405	0.3839	0.3658	0.4329
	10	0.3692	0.3886	0.7190	0.4318	0.4009	3.5980	0.7693	0.3771	0.3905	0.4022
Xalan 2.6	1	0.2870	0.2969	0.2641	0.2759	0.2601	0.2770	0.4368	0.2107	0.3044	0.2814
	2	0.4081	0.5027	0.3940	0.3746	0.3815	0.5082	0.5120	0.3361	0.4593	0.4317
	3	0.4364	0.5348	0.5152	0.4555	0.4717	0.7314	0.5880	0.3597	0.4972	0.4489
	4	0.4643	0.5356	0.6361	0.5353	0.5207	0.9525	0.6522	0.3752	0.5781	0.4870
	5	0.4768	0.5356	0.7706	0.4880	0.5335	1.1676	0.6285	0.3895	0.5252	0.5236
	6	0.4876	0.5356	0.8537	0.5297	0.5420	1.3772	0.7724	0.4158	0.5200	0.5915
	7	0.5006	0.5356	0.8739	0.5635	0.5356	1.5942	0.7894	0.4337	0.6549	0.5536
	8	0.5034	0.5356	0.9255	0.5934	0.5356	1.7894	0.7081	0.4236	0.5652	0.5836
	9	0.5032	0.5356	0.9451	0.6210	0.5356	1.9669	0.7034	0.4337	0.5635	0.5439
	10	0.5020	0.5356	1.0060	0.6804	0.5356	2.1728	0.7047	0.4441	0.5422	0.6006
Tomcat	1	0.1128	0.0854	0.1129	0.1035	0.0900	0.1467	0.2313	0.0907	0.1525	0.1138
	2	0.2473	0.1558	0.1868	0.1925	0.1647	0.2009	0.5890	0.1794	0.2517	0.2340
	3	0.3590	0.2275	0.2568	0.2555	0.2350	0.2559	0.5980	0.2373	0.2900	0.2916
	4	0.4250	0.2812	0.3127	0.3184	0.2838	0.3069	0.7976	0.2728	0.4334	0.3734
	5	0.4613	0.3088	0.3502	0.4275	0.3392	0.3582	0.7029	0.3032	0.5398	0.4173
	6	0.5212	0.3294	0.3980	0.4694	0.3592	0.4095	0.5961	0.3280	0.4920	0.4660
	7	0.5514	0.3669	0.4506	0.5082	0.3961	0.4571	0.8200	0.3526	0.5109	0.5090
	8	0.5810	0.3926	0.5090	0.5470	0.4069	0.4961	0.8933	0.3691	0.5265	0.5249
	9	0.6182	0.4271	0.5522	0.5858	0.4545	0.5390	0.8202	0.3856	0.6264	0.5240
	10	0.6489	0.4548	0.6024	0.6246	0.5057	0.5890	0.7888	0.3939	0.5559	0.5933
Prop 6	1	0.1263	0.1020	0.1095	0.1129	0.1037	0.5387	0.4481	0.1149	0.1298	0.1070
	2	0.3145	0.2037	0.1848	0.2641	0.2061	0.6327	0.8357	0.2149	0.3925	0.4107
	3	0.4406	0.3037	0.2622	0.3424	0.3049	0.6804	0.7597	0.3083	0.7336	0.6130
	4	0.5992	0.4151	0.3376	0.4218	0.3823	0.7157	0.8789	0.3625	0.8496	0.7332
	5	0.6464	0.4642	0.4130	0.5025	0.4682	0.7473	0.9024	0.4106	0.8360	0.8145
	6	0.7641	0.5396	0.4942	0.5483	0.5540	0.7701	0.8750	0.4585	0.9074	0.8046
	7	0.7881	0.6207	0.5617	0.5940	0.6147	0.8017	0.9005	0.4921	0.8367	0.8909
	8	0.8030	0.6701	0.6202	0.6397	0.6575	0.8282	0.8889	0.4863	0.8397	0.8773
	9	0.8067	0.7242	0.6868	0.6834	0.7033	0.8528	0.9234	0.5190	0.9000	0.8868
	10	0.8418	0.7813	0.7535	0.7291	0.7438	0.8793	0.9784	0.5495	0.9205	0.9169

Figure 3.5: Eclipse 2.0 2.1 and 3.0 - Complexity Metrics

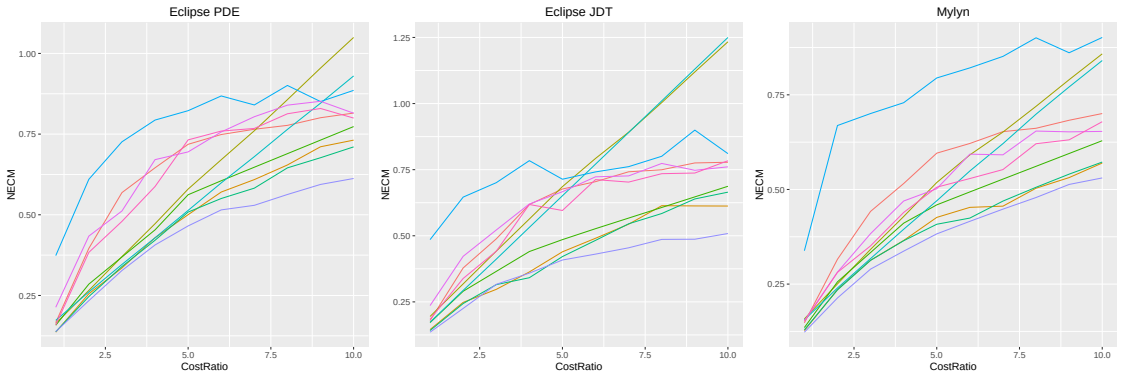


(a) Eclipse 2.0 **(b)** Eclipse 2.1 **(c)** Eclipse 3.0

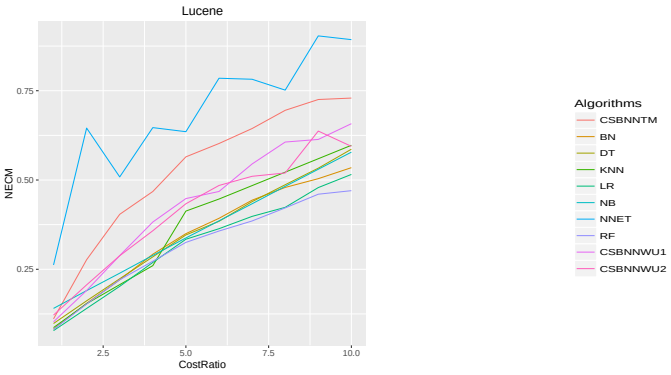
Table 3.11: Average Rank of classifiers at cost ratio 5 and 10

Cost Ratio	Random Forest	Logistic Regression	Bayesian Network	KNN	Decision Tree	CSBNN-TM	Naïve Bayes	CSBNN-WU2	CSBNN-WU1	NNET
5	1.8636	2.6818	3.2273	5.4091	5.5909	6.1364	6.1818	6.7045	7.7045	9.5000
10	1.5909	2.6136	3.2045	5.9091	7.1818	5.6818	7.9545	5.7727	6.4091	8.6818

Figure 3.6: Change, CK-OO and Static Code Metrics

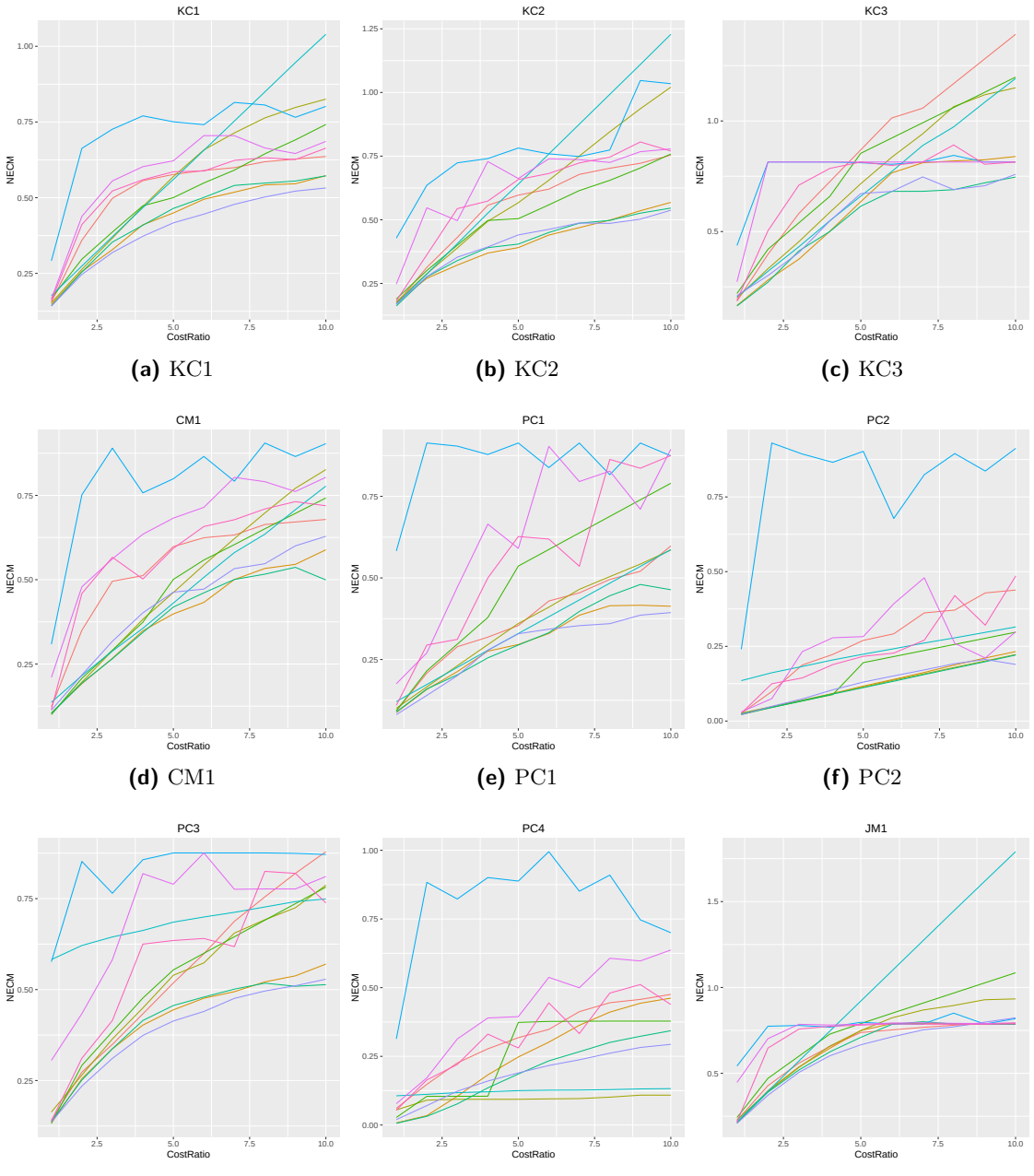


(a) Eclipse PDE **(b)** Eclipse JDT **(c)** Mylyn



(d) Lucene **(e)** Algorithms

Figure 3.7: NASA - McCabe and Halstead Metrics

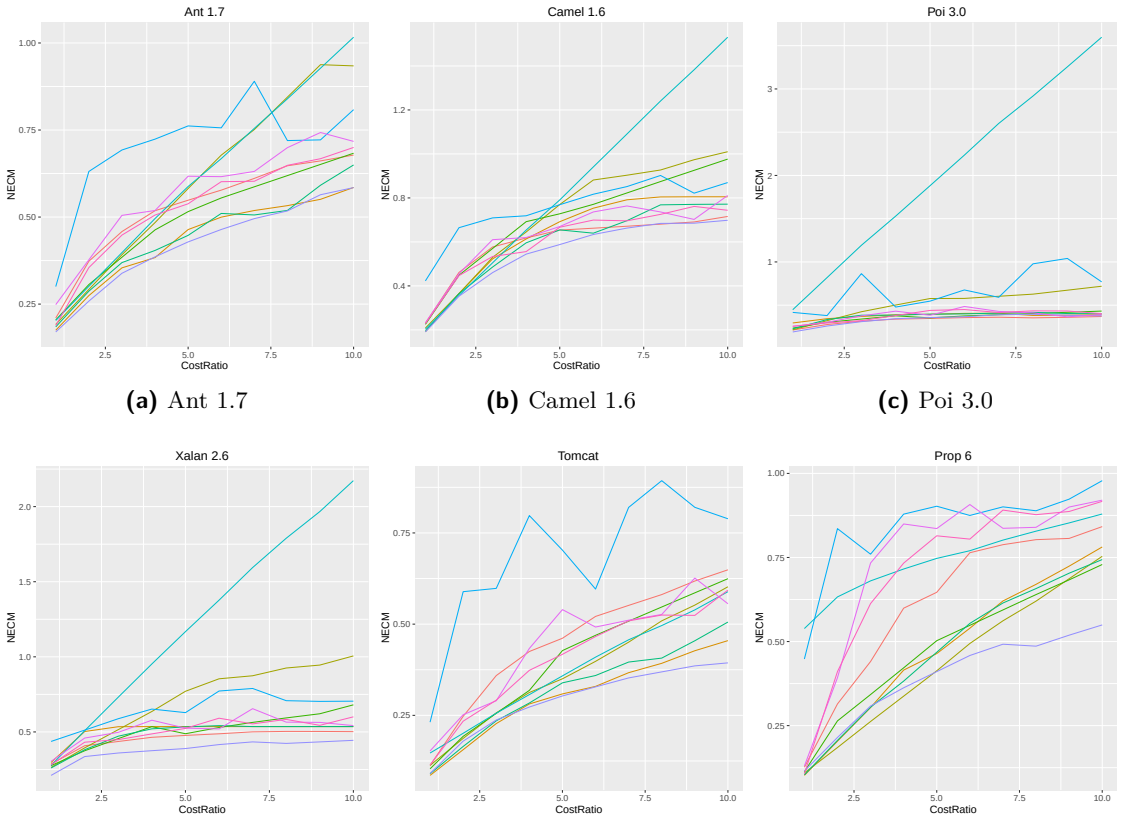


(a) KC1 **(b)** KC2 **(c)** KC3

(d) CM1 **(e)** PC1 **(f)** PC2

(g) PC3 **(h)** PC4 **(i)** JM1

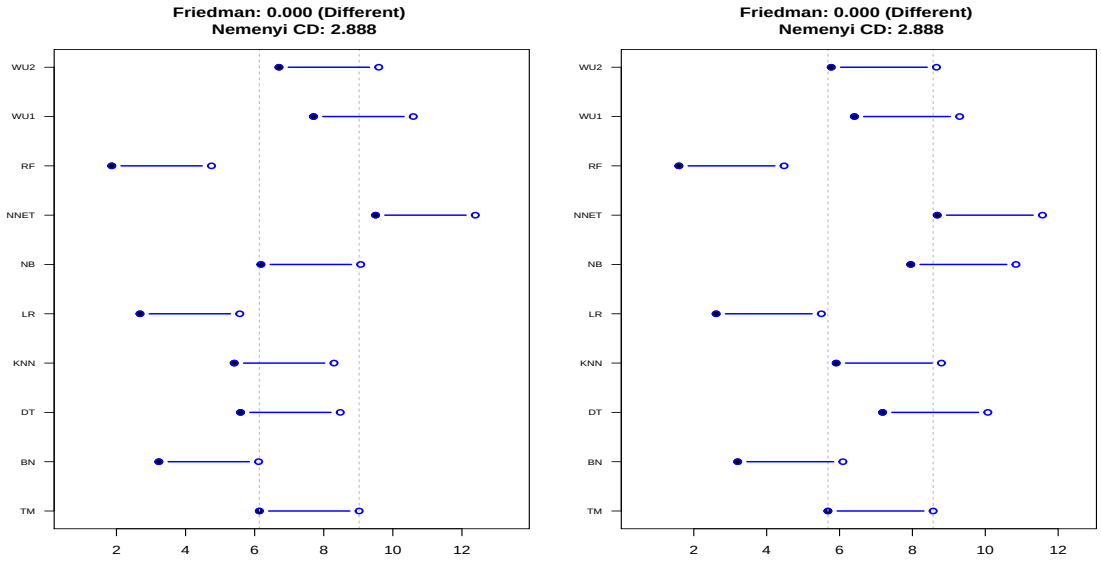
Figure 3.8: CK-OO Metrics



(a) Ant 1.7 **(b)** Camel 1.6 **(c)** Poi 3.0

(d) Xalan 2.6 **(e)** Tomcat **(f)** Prop 6

Figure 3.9: Nemenyi Diagram



(a) Nemeny 5% **(b)** Nemenyi 10%

3.3.5 Contributions

Random Forest is shown to be the outperforming algorithm in terms of NECM closely followed by Logistic Regression and Bayesian Network. Though the significance tests show that there is no difference amongst the performances of Random Forest, Logistic Regression and Bayesian Network, the mean rank values are very low for Random Forest as compared to others. Among the cost sensitive boosting Neural Network algorithms, we reconfirm that CSBNN-TM yields lower mean rank compared to CSBNN-WU1 and CSBNN-WU2. We conclude that Random Forest is shown to be significantly better than these three cost sensitive boosting Neural Network methods and traditional machine learning algorithms - Naïve Bayes, Neural Networks, k-Nearest Neighbors and Decision Tree.

3.4 Effort-aware Defect prediction

Quality has always been a non-compromising priority task for all IT companies and post release defects or defects experienced by end customer hampers the quality heavily. Quality assurance activities, such as tests or code reviews, are an expensive, but vital part of the software development process. Any support that makes this phase more effective may thus improve software quality and reduce development costs.

The project team might perform quality assurance activities on defect-prone files to catch bugs earlier to release of the software. The quality assurance activities could be peer code reviews, expert code reviews, testing etc. The team can choose one or more quality assurance activities that are apt for the project so that maximum number of defects will be uncovered. Also, project team has to allocate resources/revenues to complete these activities. Though it is ideal to get assurance activities done for all of the defect-prone files, it might not be feasible due to cost implications. Hence project team would act on top x% of defect-prone files based on the costs that can be borne by the team.

The top x% of files will be selected by sorting files in the descending order of number of predicted bugs and this approach has been adopted by researchers Ohlsson et al. [96],

Khoshgoftaar et al. [47]. We illustrate the limitations of this approach with the following example. Generally it is assumed that after performing the quality assurance activities like code review of files, all bugs in the file will be uncovered. Suppose project team would like to take up code review of top x% of files and there are two files File A, File B with the following characteristics.

File	LoC	# of predicted bugs	Effort per bug
File A	100	2	50
File B	1200	4	300

File B appears prior to File A as per the above mentioned sorted order and File A might have been ignored when project team selects top x% of files. The Effort per bug indicates the effort required to uncover one bug and is defined as the ratio of LoC and the number of bugs. The effort per bug for File A (50) is quite low as compared to File B (300) whereas the classical ordering ranks File B on top of File A. This issue has been addressed by Thilo Mende et al. [16]

Thilo Mende et al. [16] proposed a strategy to include the notion of effort into defect prediction models. They propose to rank files with respect to their effort per bug. They use McCabe's cyclomatic complexity as the surrogate measure for effort. They evaluate their new model against the naive model which just ranks files on the predicted number of bugs, by adopting to effort aware measures like CE [97]. They apply Demšar's non-parametric tests to conclude that their new model is significantly cost effective than the naive model.

Contributions: In this work we argue that the measure of effort should not be a generic measure such as cyclomatic complexity but instead it should be one that is specific to the kind of activity involved in the quality assurance process. We identify two most popular quality assurance activities namely code review and unit testing. We use lines of code to measure the effort involved in code reviews and the number of test cases to measure the effort in case of unit testing. We compare the cost effectiveness of our specific effort based models to generic effort based models.

3.4.1 Related Work

Many approaches in the literature use the source code metrics and change metrics as predictors of defects. In this study we have used the CK metrics suite [8], object oriented metrics, lines of code and some change metrics as features to build the defects prediction model. Basili et al. [98], Tang et al. [99], Cartwright and Shepperd [100], Subramanyam and Krishnan [101] explored the relationship between CK metrics and defect proneness of files. Basili et al. [98] found that WMC, RFC, CBO, DIT and NOC are correlated with faults while LCOM is not associated with faults based on experiments on eight student projects. Tang et al. [99] validated CK metrics using three industrial real time systems and suggest that WMC and RFC can be good indicators of defects. Cartwright and Shepperd [100] found DIT and NOC as fault influencers. Subramanyam and Krishnan [101] investigated the relationship between a subset of CK metrics and the quality of software measured in terms of defects. Though they proved the association, they conclude that this observation is not consistent with different OO languages like C++ and Java.

Nagappan et al. used a catalog of source code metrics to predict post release defects at the module level on five Microsoft systems [102]. Ostrand et al. [7] conclude that lines of code is a significant influence on the faults. Also their simple model based only on lines of code achieves roughly 73% bugs in the top 20% of files which is only 10% less than their full model.

Ohlsson et al. [96], Ostrand et al. [7] advocate models which rank files based on their fault proneness. Khoshgoftaar et al. [47] coined the term Module-Order-Model (MOM). It enables to select a fixed percentage of modules for further treatment which is a more realistic scenario for projects with a fixed quality assurance budget. MOMs can be evaluated by assessing the percentage of defects detected at fixed percentages of modules. Ostrand et al. [7] propose a model which found up to 83% of the defects in 20% of the files. One way to graphically evaluate MOMs are lift charts, sometimes known as Alberg diagrams [96]. Ostrand et al. validates Pareto principle in defect prediction and their results show that 20% of files have 84% of defects. Ostrand et al. [7] conclude that a prediction using defect densities is able to find more defects in a fixed percentage of code, but argue that the testing costs are, at least for system tests, not related to the size of a file.

Thilo Mende et al. [103] compare the performance of various classifiers over lines of code based lift charts. They conclude that performance measures should always take into account the percentage of source code predicted as defective. In their subsequent work [16], they propose two strategies to incorporate the treatment effort into defect prediction models. But both their strategies include cyclomatic complexity as the effort for quality assurance. In this work we propose that quality assurance activity specific effort measures perform better than generic effort based models.

3.4.2 Metrics and Datasets

We have considered a hybrid of source code metrics which include the Chidamber and Kemerer metric suite [8], few object oriented metrics and change metrics. Table 3.12 describes details about metrics considered. We need to have the number of test cases for each source file to achieve the objectives of this work. The benchmarked open source datasets, that are used for defect prediction research, do not have the number of test cases against each file. Hence we, acquired data from a private software company for three projects in which a record of all testing activities for every file is recorded. For the sake of convenience and anonymity, let us consider them as datasetA, datasetB and datasetC. The details about the number of files and number of defects in each dataset are shown in Table 3.14.

3.4.3 Experiment and Results

We build defect prediction models using the metrics mentioned in Table 3.12 for each of the three datasets in Table 3.14. We use stepwise linear regression algorithm to build our model and predict the number of faults for a file. Regression algorithm takes the number of faults into consideration while building the model whereas a classification algorithm such as Naïve Bayes classifier only considers the label of the file. Since we intend to predict effort per bug of a file we use stepwise linear regression algorithm. Nagappan et al. [43] also used stepwise linear regression to predict file defect density. We adopt the method of building defect prediction models using 2/3rd data and testing the models with remaining

Table 3.12: Source Code Metrics

WMC	Weighted Methods per Class
DIT	Depth Of Inheritance
NOC	Number of Children
CBO	Coupling between object classes
RFC	Response for a class
LCOM	Lack of cohesion in methods
Ce	Efferent coupling
NPM	Number of public methods
CC	Cyclomatic complexity is a popular procedural software metric equal to the number of decisions that can be taken in a procedure.
NOF	Number of fields in the class
NOI	Number of interfaces implemented by the class
LOC	Number of lines of code in the file
NOM	Number of methods in the class
Fan-in	Number of classes that reference the class
Fan-out	Number of classes referenced by the class
PC	Percentage of the file commented

Table 3.13: Change Metrics

noOfBugs	Number of bugs found and fixed during development
noOfStories	Number of stories this file is part of
noOfSprints	Number of sprints this file is part of
revisionCount	Number of commits this file is involved in
noOfAuthors	Number of authors who worked on this file during the release
LocA	Number of lines of code added in total to this file during development
MaxLocA	Max Number of lines of code added among all commits to this file during development
AvgLocA	Average Number of lines of code added among all commits to this file during development
LocD	Total Number of lines of code deleted among all commits to this file during development
MaxLocD	Max Number of lines of code deleted among all commits to this file during development
AvgLocD	Average Number of lines of code deleted among all commits to this file during development
LocAD	Total Number of lines of code added-deleted among all commits to this file during development
MaxLocAD	Max Number of lines of code added-deleted among all commits to this file during development
AvgLocAD	Average Number of lines of code added-deleted among all commits to this file during development

Table 3.14: Datasets

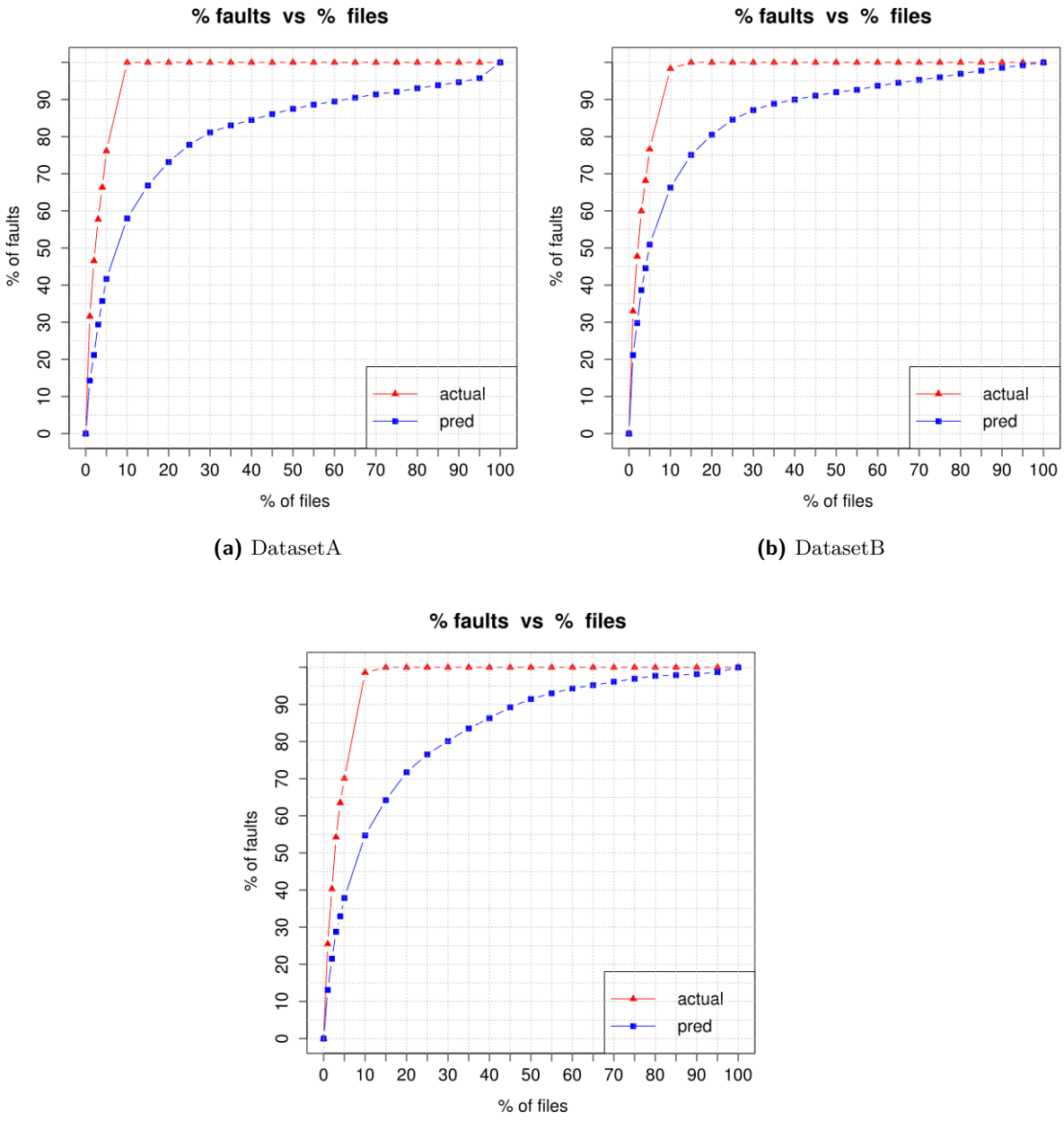
Projects	Number of Files	Post Release Defects
DatasetA	2257	252
DatasetB	2557	652
DatasetC	2151	349

1/3rd data. The division of training and testing data is being done randomly. The data is split randomly fifty times and the average of values were recorded as final results in experiments conducted for this work.

Khoshgoftaar et al. [47] proposed software quality model, MOM (Module Order Model) that is generally used to predict the rank-order of modules or files according to a quality factor [47]. We take this quality factor to be number of defects in our study. We apply stepwise linear regression algorithm and order files as per their predicted value of bugs. MOMs can be evaluated by assessing the percentage of defects detected against the fixed percentages of files. One way to graphically evaluate MOMs are lift charts, sometimes known as Alberg diagrams [96]. They are created by ordering files according to the score assigned by a prediction model and denoting for each fixed percentage of files on the x-axis, the cumulative percentage of defects identified, on the y-axis. Thus for any selected percentage of files, one can easily identify the percentage of predicted defects. We have drawn Alberg Diagrams [96] for two module-order models namely actual and simple prediction for each of the project and the plots are depicted in Fig. 3.11a through Fig. 3.11c.

The optimal or ideal or actual model (represented as red curve in the above plot) is drawn by arranging the modules in the decreasing order of actual defects. The simple prediction model(represented as blue curve in the plots) is drawn by arranging the files in the decreasing order of predicted number of faults. The predicted number of faults is value outputted by the stepwise linear regression algorithm for each file. The optimal or ideal model represents the best possible model that any defect prediction model can aspire for. The goodness of any prediction model depends on its closeness to the optimal curve, closer to the optimal curve better the prediction model is. And the major advantage of this method is that the model is flexible enough to predict bugs depending on the costs that can be borne by project team towards quality assurance activities.

Figure 3.10: Actual and Simple Prediction



(a) DatasetA

(b) DatasetB

(c) DatasetC

The Pareto principle for this problem is '20% of files should have 80% of bugs' and we have evaluated the number of bugs that can be found in top 20% of files for the prediction models across all the datasets and these values are tabulated in Table 3.15. The percentage of bugs found using these models across projects is found to be varying between 39.56% and 80%. These results are depicted visually in Fig. 3.11a-3.11c. Now, we shall explain the results of datasetB. The results, suggest that if we do quality assurance activities on the 20% of files we can catch 80% of the bugs. Although this is an excellent result, we shall now explore the amount of effort it takes to review/test these files. In practice there are couple of quality assurance activities that can be performed soon after the bugs are predicted through the learning model. We consider two such popular quality assurance activities namely code review and unit testing.

We assume that effort to do code review is directly proportional to the lines of code one has to review and for unit testing, effort is proportional to the number of test cases one has to perform manually. The effort required to conduct the two quality assurance activities for the top 20% files mentioned in Table 3.15 are tabulated in Table 3.16. From these results, we can infer the following: for datasetB, by reviewing the top 20% of the predicted files, one can uncover 80% of the bugs and the effort required for these files is reviewing 64.72% of the total lines of code or performing 76.91% of the test cases. This shows that, although the number of files predicted as defect-prone is only 20%, the amount of effort required to uncover the bugs from these files is very large which is only slightly better than a random inspection model which gives n% of bugs for n% of effort. This result is also observed in the remaining two datasets. In view of effort, the standard Pareto principle should be modified as follows: '20% of effort should give 80% of bugs'.

Table 3.15: Percentage of Bugs in Top 20% of Files

Projects	Actual	Predicted
datasetA	100	73.18
datasetB	100	80.48
datasetC	100	71.72

3.4.3.1 Effort Based Evaluation

In-order to minimize the effort involved to uncover the bugs in files, Thilo Mende et. al [16] propose a new ranking order. They rank the files based on the ratio: $\frac{predicted\ number\ of\ bugs}{effort}$ instead of predicted number of bugs alone. They use cyclomatic complexity as the surrogate measure for effort with the assumption that the lines of code as well as testcases correlate highly with cyclomatic complexity. In our datasets we have found the previous statement to be not true. Although the cyclomatic complexity is correlating highly with the lines of code, but it is not so with testcases. The Spearman correlations between cyclomatic complexity and lines of code, cyclomatic complexity and test cases is tabulated in Table 3.17.

Thus, instead of using a generic measure of effort such as cyclomatic complexity, we use lines of code and the number of testcases as our measures of effort. The predicted number of bugs term in the ratio is the value outputted by any learning algorithm which is in this case stepwise linear regression algorithm. The effort term in the ratio can be substituted with the lines of code, if quality assurance activity is code review or it can be number of testcases, if quality assurance activity is unit testing. This ratio gives a higher rank to files which have less effort per bug and a lower rank to files which have more effort per bug. Using our proposed ranking orders, the percentage of defects captured by 20% of effort are recorded in Table 3.18 and Table 3.19. The results are depicted visually by a plot of % of effort vs % defects in Figs. 3.12a-3.14b. Let us consider the Fig. 3.13a and Fig. 3.13b which correspond to datasetB. In both figures, the plots of $\frac{predicted\ number\ of\ bugs}{lines\ of\ code}$ and $\frac{predicted\ number\ of\ bugs}{testcases}$ is always higher than the plot of simple prediction. This indicates that, to uncover a certain amount of bugs, our proposed ranking orders require less effort than the traditional simple prediction rank order. Although the simple prediction model

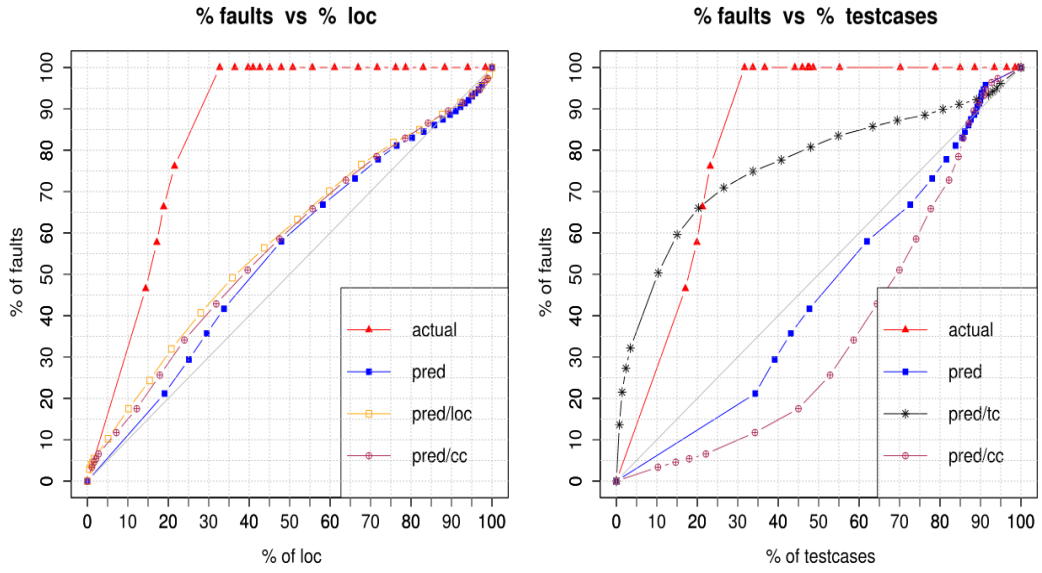
Table 3.16: Percentage of Effort Required for the Top 20% of Files

Projects	Model	% Lines of Code	% Testcases
DatasetA	Actual	39.75	36.67
	predicted	66.15	78.06
DatasetB	Actual	47.76	56.97
	predicted	64.72	76.91
DatasetC	Actual	41.66	40.05
	predicted	63.81	66.4

Table 3.17: Spearman Correlation

Projects	Cyclomatic Complexity and Lines of Code	Cyclomatic Complexity and Number of Testcases
DatasetA	0.9387	0.2703
DatasetB	0.9379	0.5487
DatasetC	0.9358	0.3687

Figure 3.11: Effort-Aware Prediction: DatasetA



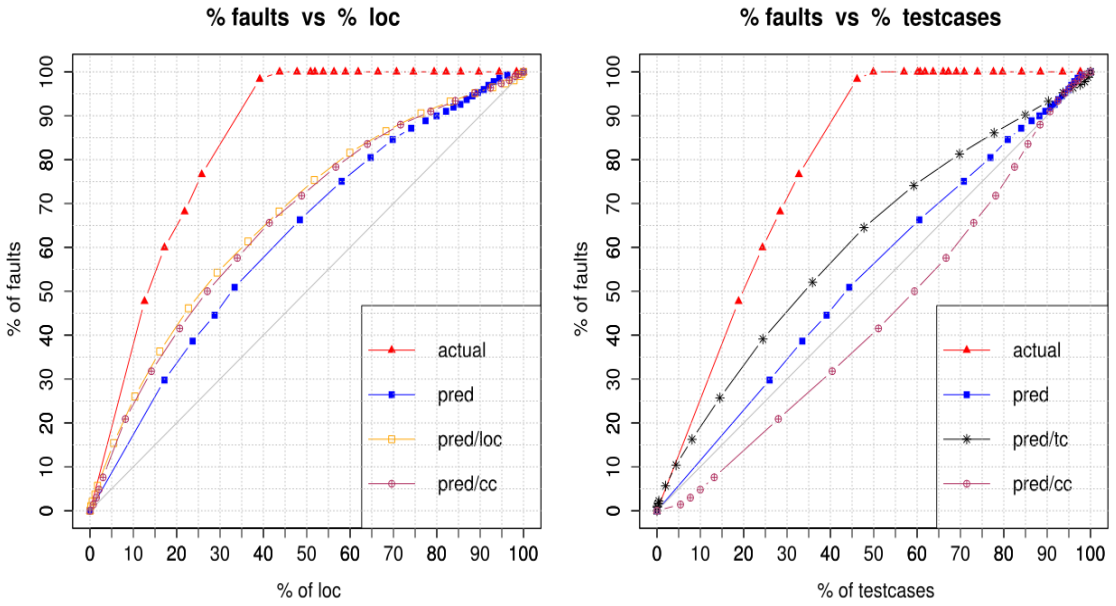
(a) DatasetA **(b)** DatasetA

is better than our proposed model when considering only the % of bugs caught vs the % of files (Fig. 3.11b), it fails to perform better when considering the effort required to uncover the bugs (Fig. 3.13a and Fig. 3.13b). This result is observed in all the three datasets. Also, common to all the three datasets is the observation that the specific effort based models i.e $\frac{predicted\ number\ of\ bugs}{lines\ of\ code}$ and $\frac{predicted\ number\ of\ bugs}{testcases}$ are performing better than the $\frac{predicted\ number\ of\ bugs}{cyclomatic\ complexity}$. In order to quantify the performance of each classifier we consider an evaluation scheme described in the next section.

Table 3.18: Percentage of Defects Caught by 20% of Effort(LOC)

Projects	Actual	Simple Prediction	$\frac{predicted\ \#bugs}{LOC}$	$\frac{predicted\ \#bugs}{CC}$
DatasetA	70	22	33	30
DatasetB	65	35	42	40
DatasetC	60	25	40	38

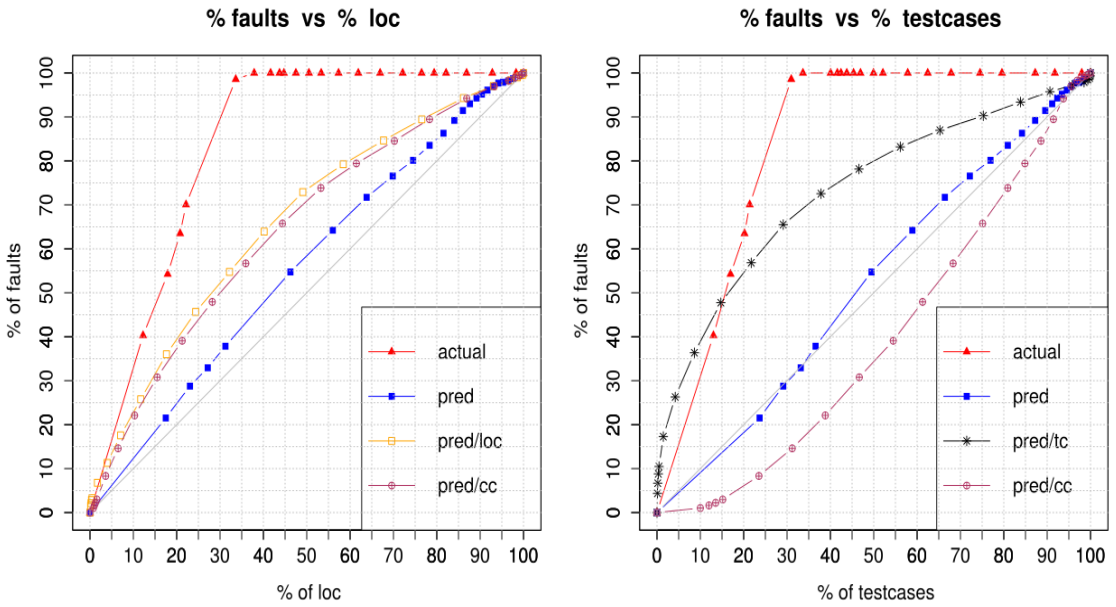
Figure 3.12: Effort-Aware Prediction: DatasetB



(a) DatasetB

(b) DatasetB

Figure 3.13: Effort-Aware Prediction: DatasetC



(a) DatasetC

(b) DatasetC

3.4.3.2 Percentage of Cost Effectiveness

Arisholm et. al [104] proposed an effort evaluation measure CE which stands for Cost Effectiveness. They calculate a cost effectiveness estimation based on the assumption that a random selection of n% of source lines contains n% of the defects.
A defect prediction model is cost effective only when the files predicted as defective contain a larger percentage of defects than their percentage of lines of code. Their performance measure CE can be obtained by calculating the area under the prediction model's curve which lies above a line of slope one. Thilo Mende et. al. [103] defined the measure p_{opt} which measures how close a model is to the ideal curve. Closer the curve to the ideal curve, the higher the value of p_{opt}. This measure takes into consideration the effort as well as the actual distribution of faults by benchmarking against a theoretically optimal model.

We consider the evaluation measure POA, Percentage of Area. This measure attempts to combine the aspects of both CE and p_{opt}, which has information about the lower bound of cost effectiveness, the random model and the upper bound, the theoretically optimal value. POA is the ratio of the CE of a model to the CE of the theoretically optimal model. This value gives the fraction of cost effectiveness a model achieves out of the theoretically maximum possible cost effectiveness.

$$POA = \frac{CE\,of\,model}{CE\,of\,the\,ideal\,model}$$

We present the POA values for the models when effort is lines of code in Table 3.20 and when effort is testcase in Table 3.21. From the results it is clear that specific effort based models outperform generic effort based measure cyclomatic complexity.

Table 3.19: Percentage of Defects Caught by 20% of Effort(Testcases)

Projects	Actual	Simple Prediction	$\frac{predicted\ \#bugs}{TC}$	$\frac{predicted\ \#bugs}{CC}$
DatasetA	59	10	66	6
DatasetB	50	24	33	14
DatasetC	64	16	55	6

3.4.3.3 Statistical Significance

We now test whether the differences between the three models are statistically significant using the non parametric tests mentioned by Demšar [38]. Demšar uses the Friedman test [105] to check whether the null hypothesis, i.e that all models perform equal on the datasets, can be rejected. It is calculated as follows:

$$\chi_F^2 = \frac{12N}{k(k+1)} \left(\sum_j R_j^2 - \frac{k(k+1)^2}{4} \right)$$

$$F_F = \frac{(N-1)\chi_F^2}{N(k-1)-\chi_F^2}$$

where k denotes the number of models, N the number of datasets, and R_j the average rank of model j on all data sets. F_F is distributed according to the F-Distribution with $k-1$ and $(k-1)(N-1)$ degrees of freedom. Once computed, we can check F_F against critical values for the F-Distribution and then accept or reject the null hypothesis. When the Friedman test rejects the null hypothesis, we can use the Nemenyi post-hoc test to check whether the difference of performance between two models is statistically significant. The test uses the average ranks of each model and checks for each pair of models whether the difference in their average ranks is greater than the critical difference CD.

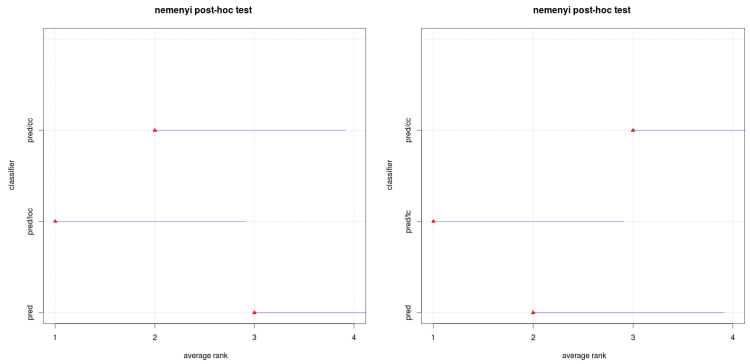
$$CD = q_\alpha \sqrt{\frac{k(k+1)}{6N}}$$

where k and N are number of models and datasets respectively. q_α is a critical value which are based on the Studentized range statistic divided by $\sqrt{2}$. The Studentized range

Table 3.20: POA - Lines of Code

Model	DatasetA	DatasetB	DatasetC
Ideal	100	100	100
simple prediction	14.23	35.59	16.03
$\frac{Predicted\ Number\ of\ Bugs}{Lines\ of\ Code}$	22.61	52.2	46.55
$\frac{Predicted\ Number\ of\ Bugs}{Cyclomatic\ Complexity}$	19.56	49.84	41.73
Random Model	0	0	0

Figure 3.14: Nemenyi Diagram for LOC and Testcases



(a) Nemenyi Plot - Lines of Code **(b)** Nemenyi Plot - Testcases

statistic depends on the alpha value which in this cases we take as 0.05 and the number of models k. For our setup, we used k=3, α=0.05 and q_α=2.85.

We use Lessmann et al. [5] modified version of Demšar's significance diagrams to depict the results of Nemenyi's post-hoc test: For each classifier on the y-axis, the average rank across the datasets is plotted on the x-axis, along with a line segment whose length encodes CD. All classifiers that do overlap in this plot do not perform significantly different and those that do not overlap, perform significantly different. The Nemenyi's post-hoc significance plots are presented in Fig. 3.15a and Fig. 3.15b.

Consider Fig. 3.15a: $\frac{predicted\ number\ of\ bugs}{lines\ of\ code}$ ranks the best, followed by $\frac{predicted\ number\ of\ bugs}{cyclomatic\ complexity}$ and $predicted\ number\ of\ bugs$. Although, the $\frac{predicted\ number\ of\ bugs}{lines\ of\ code}$ performs better, the difference between it and $\frac{predicted\ number\ of\ bugs}{cyclomatic\ complexity}$ is not significant.

In Fig. 3.15b: $\frac{predicted\ number\ of\ bugs}{test\ cases}$ ranks the best, followed by $predicted\ no.\ of\ bugs$

Table 3.21: POA - Testcases

Model	DatasetA	DatasetB	DatasetC
Ideal	100	100	100
simple prediction	0.75	13.18	7.06
$\frac{Predicted\ Number\ of\ Bugs}{Testcases}$	75.13	38.65	67.22
$\frac{Predicted\ Number\ of\ Bugs}{Cyclomatic\ Complexity}$	0.73	0.3	0.19
Random Model	0	0	0

and $\frac{predicted\ number\ of\ bugs}{cyclomatic\ complexity}$. $\frac{predicted\ number\ of\ bugs}{lines\ of\ code}$ model performs significantly better than $\frac{predicted\ number\ of\ bugs}{cyclomatic\ complexity}$.

3.4.3.4 Threats to Validity

Since open source softwares are not developed in controlled environment and unavailability of proper testcases and considering the difficulties of linking them with appropriate files, We decided to do the experiments in proprietary software datasets. Similar experiments in different datasets may not yield similar results.

3.4.4 Contributions

The consistent and repetitive results of three projects and generic Demšar test indicate us that the testing based effort aware models perform significantly better than generic effort-aware models. The code review based effort-aware model performs slightly better than generic model individually for each project though Demšar test show that there is no significant improvement. Hence, we conclude that effort-aware models are sensitive to the type of quality assurance activity one undertakes and the effort required for these activities should be considered while building prediction models.

4 Multi-objective Defect Prediction

4.1 Introduction

Prediction models are built with training data and they are evaluated for performance on the testing data. In literature, defect prediction models are built using three prediction techniques that differ based on the source of training data and testing data. The different prediction techniques are: 1) Cross Validation prediction: In this approach, training and testing data are taken from the same version of a project. The defect prediction models are built by taking 70% of data as the training data and tested on remaining 30% of the data, or by using 10-fold cross validation technique; 2) Cross Version prediction: In this approach, the data of the previous version of a software project is taken as the training data to build prediction model and it is tested on the data of current version of the same project; 3) Cross Project prediction: The prediction models are built by taking data from different projects as the training data and tested on the data of the project under study.

It is our intuitive understanding that using data of the previous version of the same project is more appropriate and can provide better results instead of using data from different projects to build defect prediction models. The main architectural or design characteristics of the project will remain more or less the same across different releases. The pattern of bug occurrences of the current version might also get influenced by the bug occurrences in the previous version. Hence, for predicting defect prone files of the current version, the most suitable training data is the data of previous version of the same project. These factors motivated us to explore further into cross version defect prediction model.

Harman suggested that the defect prediction problem can be viewed as a search problem, which can be solved using evolutionary algorithms [106]. The defect prediction problem can be formulated as a multi-objective optimization problem with contrasting objectives. We have attempted to solve two multi-objective defect prediction problems with competing objective functions in cross version setup.

We formulate our first multi-objective defect prediction problem with the following contrasting objectives

- Maximize effectiveness of the model
- Minimize misclassification cost

Effectiveness is the ratio of the number of components correctly predicted as defective to the number of actual defective components (recall). The misclassification cost is the cost incurred due to quality assurance activities required on wrongly classified files, i.e. cost of reviewing/testing the False Positive (predicted to be defective but not actually defective) files and cost of the False Negative (predicted to be non-defective but actually defective) files during post-release phase. Ideally any good model attempts to minimize the misclassification cost and maximize the effectiveness of model. Hence we have chosen them as objective functions.

There are many evolutionary algorithms which can solve multi-objective optimization problems. In this study, we have used NSGA-II, a multi-objective genetic algorithm proposed by Deb et al. [107]. Any multi-objective genetic algorithm requires a fitness function that guides the search process to find optimal or near optimal solutions. We have considered logistic regression function as fitness function to find out cost and effectiveness. Hence we are denoting this approach as Multi-Objective Logistic Regression (MOLR).

We build a cross version defect prediction model using multi-objective logistic regression (MOLR) for our first problem and this model is denoted by M1. We also build prediction

models using four traditional single objective algorithms, namely, Logistic Regression, Naive Bayes, Decision Tree and Random Forest. We try to answer the following research question.

RQ_1**:** How does the proposed M1 perform as compared to traditional single objective defect prediction models in the cross version defect prediction?

We formulate second multi-objective defect prediction problem with the following contrasting objectives.

- Maximize effectiveness of the model
- Minimize LOC cost

Canfora et al. proposed this objective function to solve multi-objective defect prediction problem in cross project setup [1,108]. Effectiveness is the same as it is defined in M1. The cost borne by project team to perform review/test/any QA activity on all defect prone files, is taken to be another objective. As the effort required is directly proportional to the sum of lines of code of all defect prone files, cost is taken as the sum of lines of code (LOC) of all the files which are predicted defective. This includes both true positives (predicted to be defective and actually defective) and false positives. Canfora et al. showed that the multi-objective cross project prediction is more cost-effective than single objective algorithms when there is lack of training data for a project, which is true for relatively newer projects [1,108]. We consider these objective functions to implement multi-objective defect prediction models across versions of the same project.

We build cross version defect prediction model using multi-objective logistic regression (MOLR) for second problem and this model is denoted by M2. We try to answer the following research question.

RQ_2**:** How does the proposed M2 perform as compared to traditional single objective defect prediction models in the cross version defect prediction?

We have used 11 software projects from the PROMISE repository [109], having a total of 41 different versions to answer RQ1 and RQ2. We had 30 pairs of training and testing versions. For each pair we compared multi-objective logistic regression with the traditional classification algorithms in terms of cost-effectiveness.

4.2 Related Work

Defect prediction models have been built using various sets of static code attributes and process metrics. Metrics like lines of code (LOC), Halstead, CK and OO, change metrics and past bugs are mined from the code base, version control system and bug trackers. They are used to predict defect prone files either within a project or across the projects. Ambros et al. tried to consolidate the defect prediction work, using the metrics mentioned above [27]. Though they found that WCHU (Weighted Churn of source code metrics) and LDHH (Linearly Decayed Entropy of source code metrics) are better performing metrics for defect prediction, they concluded that there is not a single metric that predicts defect prone files well across software projects . But they agree to many other past works which say that past bugs and source code metrics are right alternatives in terms of overall prediction and computational requirements. Researchers often say most of the source code metrics are proxies of size. Zhang *et al.* investigated the relationship between size of files and defect, with an assumption that large code base may correlate with more defects [110]. Their study concludes that the defect proneness increased with the size of the classes, but they suggested spending more resources on smaller classes which were found to be more problematic than larger classes. Kim *et al.* predicted defects using cached history [41]. They assumed that the defects will not occur alone, but rather in bursts of several related faults. So they cached locations that are likely to have bugs. Basili *et al.* found that CK metrics are useful in finding defect proneness in early phases of the software development [98]. Subramanyam *et al.* showed that CK metrics are associated with defects even after controlling for the size of the software [111]. A few

other studies endorsed the defect prediction models built with change metrics, as they gave better prediction performance in classifying defect prone files [13, 19, 20].

Apart from finding better prediction metrics, coming up with competitive prediction techniques for defect prediction is also an interesting research area. Classifiers like logistic regression, Naïve Bayes, decision tree, support vector machine, random forest, etc. are applied by different researchers in the past [41, 44, 103, 112, 113]. Czibula et al. used relational association rule mining to predict defective modules in software systems. Their model, Defect Prediction using Relational Association Rules (DAPR) gives better predictive performance compared to the existing defect prediction techniques [114]. Marian et al. proposed unsupervised machine learning method based on self-organizing maps which puts defective and non-defective files in two clusters [115]. Lessmann *et al.* performed a comparative study on defect prediction experiments with 22 classifiers applied on 10 public domain datasets from NASA repository and concluded that though the metrics based classification was useful in this domain, the importance of classification algorithm was not significant. They did not find any significant performance difference between top 17 out of 22 classifiers used in the study [5]. But Ghotra *et al.* argue that by making use of cleaned versions of NASA, PROMISE corpus and different classification algorithms, it is possible to produce defect prediction models with significant differences in performance [116].

With a perspective different from traditional approaches, Harman suggested to reformulate the classic software engineering problems as a search problem. This will help the community in finding solutions to difficult problems with competing constraints (e.g. quality, cost) [106]. Harman and Clark showed that, many metrics can be used as the guiding force behind the search for optimal or near optimal solutions to many software engineering problems [117]. Taking a clue from Harman's work some researchers have come up with multi-objective defect prediction techniques. Canfora *et al.* proposed MODEP (Multi-Objective DEfect Predictor) based on multi-objective forms of machine learning techniques, logistic regression and decision tree, trained using genetic algorithm [1, 108]. Carvalho *et al.* came up with Multi-Objective Particle Swarm Optimization (MOPSO), which generates a model composed of rules with specific properties, which are intuitive and comprehensible [118].

Research studies mentioned above, were focused on either within project cross-validation or cross-project defect prediction to build defect prediction models. According to our understanding, there is not much comprehensive work done on the defect prediction across multiple versions of a software project. Zimmerman *et al.* showed that defect prediction models learned from earlier releases can be used to predict defects for future releases [42]. For instance, the model trained from release 1.0 can be used to predict defects in release 2.0. But they concluded that their results were far from being perfect. Yang *et al.* built a regression model based on the data of previous version to predict defect proneness of components in the current version and ranked them based on their defect proneness [119]. They compared many traditional regression algorithms with their newly proposed Learning-To-Rank (LTR) approach and concluded that LTR performed better as compared to other algorithms. One of the recent works in cross version defect prediction found that network measures are more effective in cross version defect prediction. But they conclude that it does not improve the prediction performance in a bigger way, especially when ranking fault-prone modules [120]. Our work treats cross version defect prediction within a project as a multi-objective optimization problem, with competing constraints like cost and effectiveness. We have presented a comprehensive comparison of multi-objective algorithm with traditional algorithms. We have conducted experiments in a large number of projects with multiple releases, with the motivation of providing more generalizable results.

4.3 Datasets and Metrics

We are using six CK metrics(Chidamber and Kemerer) [121] and LOC as features to build defect prediction models. The choice of the metrics is based on the fact that all projects used in our study are object oriented projects and CK metrics has been widely used as quality indicators for object oriented softwares [27, 93, 112, 113, 122]. Table 4.1 gives brief description about predictors used in our study.

We have considered 11 open source projects that are having data for 3 or more versions in PROMISE repository. The details of these versions, namely number of classes and

percentage of defective classes are presented in the Table 4.2. As the table shows, different versions of the projects have 109 to 965 classes with average number of classes being 385. The choice of projects is done with a view of evaluating performance of multi-objective algorithm across projects having wide diversity, so that the results can be generalized.

Table 4.1: Metrics [1]

Name	**Description**
Lines of Code (LOC)	Number of non-commented lines of code for each software component (e.g., in a class)
Weighted Methods per Class (WMC)	Number of methods contained in a class including public, private and protected methods
Coupling Between Objects (CBO)	Number of classes to which a class is coupled
Depth of Inheritance (DIT)	Maximum inheritance path from the class to the root class
Number Of Children (NOC)	Number of immediate sub-classes of a class
Response For a Class (RFC)	Number of methods that can be invoked for an object of given class
Lack of Cohesion among Methods (LCOM)	Number of methods in a class that are not related through the sharing of some of the class fields

4.4 Formulation of multi-objective defect prediction models

We formulate defect prediction problem as multi-objective optimization problem with contrasting objectives. We propose two multi-objective prediction problems each with distinct set of objective functions.

Most of the machine learning algorithms are single objective in nature. That is, their final goal is to estimate one solution that minimizes the prediction error. For example logistic regression minimizes prediction error, i.e., Root Mean Square Error (RMSE) where RMSE is defined as follows,

Table 4.2: Projects under study

Project	Version	Number of Classes	% of Defective Classes
Ant	1.3	125	16%
	1.4	178	22.47%
	1.5	293	10.92%
	1.6	351	26.21%
	1.7	745	22.28%
Camel	1.0	339	3.83%
	1.2	608	35.53%
	1.4	872	16.63%
	1.6	965	19.48%
Ivy	1.1	111	56.76%
	1.4	241	6.64%
	2.0	352	11.36%
Jedit	3.2	272	33.09
	4.0	306	24.51%
	4.1	312	25.32%
	4.2	367	13.08%
	4.3	492	2.23%
Log-4j	1.0	135	25.18%
	1.1	109	33.94%
	1.2	205	92.19%
Lucene	2.0	195	46.67%
	2.2	247	58.3%
	2.4	340	59.7%
Poi	1.5	237	59.49%
	2.0	314	11.78%
	2.5	385	64.41%
	3.0	442	63.57%
Synapse	1.0	157	10.19%
	1.1	222	27.03%
	1.2	256	33.59%
Velocity	1.4	196	75%
	1.5	214	66.35%
	1.6	229	34.06%
Xalan	2.4	723	15.21%
	2.5	803	48.19%
	2.6	885	46.44%
	2.7	909	98.79%
Xerces	init	162	47.53%
	1.2	440	16.14%
	1.3	453	15.23%
	1.4	588	74.32%

$$RMSE = \sqrt{\sum_{i=1}^{n} \left(f(c_i) - dp(c_i)\right)^2} \quad (4.1)$$

where $f(c_i)$ and $dp(c_i)$ take either 1 or 0, $f(c_i)$ represents whether c_i is defective class or not and $dp(c_i)$ represents whether c_i is predicted to be defective or not.

It is always good to have defect prediction models with high recall so as to minimize post-release defects. And it is easy to build models with recall value of 1. A dummy model which predicts all files to be defect prone will have a recall value of 1. But this model is of no use because recall is maximized without considering cost of misclassification. Hence we would like to view defect prediction problem as multi-objective optimization problem rather than single objective optimization problem. There will be two types of costs associated with defect prediction models. We will discuss building multi-objective defect prediction models that maximizes effectiveness and minimizes cost(misclassification cost/LOC cost) in the next two subsections.

4.4.1 Optimize misclassification cost and effectiveness

There are two types of errors for any prediction model. For defect prediction problem, Type I error is predicting non-defective file to be defect prone and Type II error is predicting defective file to be non-defective. In fact the number of Type I errors and Type II errors are number of false positive files and number of false negative files, denoted by FP and FN. The cost incurred due to Type I errors is the effort spent by project team on quality assurance (QA) activities like reviewing, testing etc. of False Positive files. The cost incurred to fix Type II errors is the effort spent by project team to fix the defective file in post release phase.

$$Misclassification\ Cost = Cost\ of\ Type\ I\ errors + Cost\ of\ Type\ II\ errors \quad (4.2)$$

It is evident that cost of Type II error is much more than the cost of Type I error for our problem as fixing defective file during post release phase takes huge effort as compared to reviewing / testing a file before release. It is difficult to say how much Type I error

is costly as compared to Type II error. Cost factor denotes how much is the cost of misclassifying a defective class as non-defective compared to misclassifying a non-defective class as defective. The cost factor α can be written as follows:

$$\alpha = Cost\ of\ Type\ II\ error\ /\ Cost\ of\ Type\ I\ error \tag{4.3}$$

If cost of $Type\ I$ error is c then cost of $Type\ II$ error is αc.

$$Misclassification\ Cost = FPc + (FN)\alpha c \tag{4.4}$$

We can normalize misclassification cost as follows:

$$Normalized\ Misclassification\ Cost = (FP + \alpha(FN))c/nClasses \tag{4.5}$$

where $nClasses$ is number of classes in given version. It is always recommended to have defect prediction model that maximizes the effectiveness and minimizes misclassification cost. As $nClasses$ and c are constants for a project, minimizing the misclassification cost is same as minimizing normalized misclassification cost. And minimizing normalized misclassification is same as minimizing $(FP + \alpha(FN))/nclasses$. Hence c can be taken to be 1 for these kinds of optimization problems. Now, we formulate our first multi-objective defect prediction problem with the following contrasting objectives.

- ***maximize recall and minimize normalized misclassification cost.***

We take effectiveness of model to be recall, misclassification cost as normalized misclassification cost where

$$Recall = \frac{TP}{TP + FN} \tag{4.6}$$

$$Normalized\ Misclassification\ Cost = \frac{FP + \alpha \cdot FN}{nClasses} \tag{4.7}$$

4.4.2 Optimize cost of QA activities and effectiveness

In cross version defect prediction, the defect prone files of the current version will be predicted by making use of prediction models built using previous version data. The project team performs QA activities on the files which are predicted defective. The predicted defective files involves both TP and FP. Hence, the cost incurred by project team is cost of QA activities on TP and FP files. The cost of QA activities is proportional to LOC of TP and FP files. Thus cost is measured in terms of lines of code and effectiveness in terms of recall. The main motive behind these objectives is to find set of models such that they minimize cost of reviewing/testing files while achieving best possible effectiveness. And amongst these models, the most effective model can be selected based on the cost borne by project team.

We formulate second mutli-objective defect prediction problem with the following contrasting objectives.

- ***maximize recall and minimize LOC cost.***

We take effectiveness of model to be recall and it is the same as defined in Equation 4.6. We believe that the cost of QA activities of defect prone class(c_i) is proportional to number of lines of $c_i(LOC(c_i))$ and this is also confirmed by Rahman et al. [123]. We can find LOC cost using the following Equation,

$$LOC\ cost = \sum_{i=1}^{n} P(c_i) \cdot LOC(c_i) \tag{4.8}$$

where $P(c_i)$ denotes whether i^{th} class, c_i, is defect prone or not. $P(c_i)$ is set to 1 if the predicted probability $p(c_i) > 0.5$, otherwise it is set to 0.

4.5 Proposed Approach

In this section we illustrate our approach to build multi-objective prediction models. Let $C = \{c_1, c_2, ..., c_n\}$ be classes of a given version V_k of a project P with each class having 'm' attributes. So training data takes form of $n \times m$ matrix. A matrix entry $x_{i,j}$ denotes the value of j^{th} attribute for i^{th} class. The mathematical formulation of logistic regression is given as,

$$p(c_i) = \frac{e^{w_0+w_1x_{i,1}+w_2x_{i,2}+...+w_mx_{i,m}}}{1+e^{w_0+w_1x_{i,1}+w_2x_{i,2}+...+w_mx_{i,m}}} \tag{4.9}$$

where $p(c_i)$ denotes probability of i^{th} class being defective, while the set of scalars $W = \{w_0, w_1, w_2, ..., w_m\}$ represents the coefficients for the attributes $\{x_{i,1}, ..., x_{i,m}\}$. The objective of traditional logistic regression approach is to find out a set of coefficients $W = \{w_0, w_1, w_2, ..., w_m\}$ that minimize the prediction error. It is usually done with the help of gradient descent algorithm. This model is built using training data, and it is used to predict defective classes in the test data. A class c_i is predicted to be defective if $p(c_i) > 0.5$, based on the coefficients found during training process and the metrics $\{x_{i,1}, x_{i,2}, ..., x_{i,m}\}$.

The goal of multi-objective problem stated above is to find out a set of coefficients W, which optimizes two objectives. As there are two contrasting objectives we will get multiple models with different trade-offs between two objectives. This problem of multi-objective optimization can be solved by using genetic algorithm.

Now we briefly describe some basic concepts used to solve multi-objective optimization problem. The definitions are presented here for the sake of continuity. The set of all possible values that can be taken by solutions, is defined as the *feasible region.* In our case the set of values that can be taken by the coefficients W forms the feasible region, which is the set of real numbers, as there are no constraints on the values that coefficients can take.

In multi-objective optimization problems concepts of *Pareto dominance* and *Pareto optimality* are used to define the optimality of solutions [124]. These terms are defined for two contrasting objectives.

A solution x dominates another solution y (also written $x <_p y$) if and only if the values of the objective functions satisfy the following conditions:

$$objective1(x) \leq objective1(y) \text{ and } objective2(x) > objective2(y)$$

$$or$$

$$objective1(x) < objective1(y) \text{ and } objective2(x) \geq objective2(y)$$

Let us assume that $objective1$ is the cost (LOC cost or misclassification cost) and $objective2$ is effectiveness (recall) of a prediction model. In simple words above definition indicates that x *is better than* y if and only if, at the same level of cost x achieves greater effectiveness than that of y; or at the same level of effectiveness x incurs lower cost than that of y. A solution x is Pareto optimal if and only if it is not dominated by any other solution in the feasible region. The set of solutions (coefficient vectors W_i) that are not dominated by any other solutions is said to form *Pareto optimal set*, while the corresponding objective vectors, cost and effectiveness values of the solution set W, said to form *Pareto frontier*.

The final set of solutions on the Pareto frontier give different cost-effectiveness trade-offs. It is up to software project team to decide upon how much amount of time (cost) they can spend and then choose appropriate model. For example, with one month away from the release, owing to shortage of time, the software project manager typically needs to plan the cost of quality assurance activities on topmost $n\%$ of defective components. The value of n can vary based on quality requirements and cost borne by project and organization. In this case, there is a need of having a model which can provide multiple solutions with different cost-effectiveness trade-offs. So multi-objective approach presented here can be of great help in similar situations.

To search for coefficient vectors in the solution space we have used a Multi-Objective

Genetic Algorithm (MOGA) presented by Goldberg [125]. In general terms a genetic algorithm works as follows:

- It starts with random set of solutions called *population* of size p. Each individual in the population is known as *chromosome*.

- The population evolves through set of iterations, known as *generations*.

- From the given generation new population called *offspring*, is created using *crossover* and *mutation* operations. Crossover operator combines two individuals to generate new offspring. Mutation operator modifies the internal structure of an individual.

- A fitness function is applied on each individual of the current population to find the values of objective functions. From the current population, fittest set of p individuals are selected to be part of next generation using a selection operator. The fittest set of individuals are selected based on their objective values. The selection process follows the concepts of Pareto dominance and crowding distance. This is done to keep the size of population as constant in each generation.

- This process is repeated until termination criteria is met.

The intuition behind the genetic algorithm is that at the end of every generation only the fittest set of individuals make it to the next generation. So there is some improvement in the solution at the end of every generation. After many generations, the population approach to an ideal solution or approximately ideal solution.

In our implementation we have used NSGA-II proposed by Deb *et al.* [107]. For multi-objective logistic regression used in our study, one chromosome represents the coefficient vector $W = \{w_0, w_1, w_2, ..., w_m\}$, which forms one logistic regression model. Initially a process starts with a population size of say p. For each model, the fitness function computes the values of objectives. Based on the definition of Pareto optimality and crowding distance, the best p set of coefficient vectors are selected for the next generation. The process of selecting best chromosomes depends on implementation of multi-objective ge-

netic algorithm. NSGA-II uses fast non-dominated sorting algorithm and the concept of crowding distance for selection process. Complete explanation of the selection process, used in NSGA-II, is beyond the scope of this thesis. One can refer to the original paper by Deb *et al.* [107]. The process of generating new population and selection of fittest set of individuals repeats in each iteration, until optimal set of coefficients is found or maximum number of generations is reached. At the end of training process we get optimal set of coefficients, i.e. logistic regression models.

4.5.1 Data Pre-processing

We describe the steps required to build and evaluate the prediction models built using single objective and multi-objective algorithms. We pre-process the data of training version and testing version using data standardization. We have used CK metrics [121] and lines of code (LOC) as predictor metrics. Since the values of different metrics have different ranges, metrics are standardized to reduce the heterogeneity. Mathematically a metric m is normalized as follows:

$$m_z(i, c_j, V_k, P) = \frac{m(i, c_j, V_k, P) - \mu(i, V_k, P)}{\sigma(i, V_k, P)} \tag{4.10}$$

where $m(i, c_j, V_k, P)$ is the value of i^{th} metric computed on class c_j of version V_k of project P. Mean $\mu(i, V_k, P)$ and standard deviation $\sigma(i, V_k, P)$ have been calculated on all the classes of version V_k of the project P for the i^{th} metric. $m_z(i, c_j, V_k, P)$ is the normalized value of i^{th} metric. We apply this normalization on data of both the training and testing versions, while building and predicting the use of any modeling technique.

4.5.2 Training Process

1. We train MOLR for our first multi-objective optimization problem as mentioned above on normalized data of the training version V_{k-1}. At the end of training we get multiple MOLR models and four single objective models. From the final set of MOLR

models, we find out the best model. The best model is the one which is closest to the ideal model. The ideal model has misclassification cost=0 and recall=1. We find the closest model with help of Euclidean distance measure as shown in Algorithm 2. If more than one models are equidistant to the ideal model then we chose the model with least misclassification cost.

2. We train MOLR for our second problem as mentioned above and single objective models on normalized data of the training version V_{k-1}. At the end of training we get multiple MOLR models and four single objective models. We retain all the MOLR models and apply it on the data of the test version.

4.5.3 Testing Process

We illustrate testing process for both of our problems in this subsection.

4.5.3.1 Testing Process for M1

The best possible MOLR model and other four single objective models are applied on the data of test version. We compare all the single objective algorithms with MOLR in terms of misclassification cost, recall and F-measure.

Misclassification cost changes based on the value of cost factor α. So, for both MOLR and single objective algorithms, misclassification cost is calculated each time cost factor changes. For any model (single objective or MOLR model) misclassification cost and recall can be calculated as per Equations 4.6 and 4.7 . As recall is one of the objectives in MOLR, its value changes based on cost factor. F-measure is the harmonic mean of recall and precision. It denotes the balance achieved by the model between recall and precision values. This, in turn, shows how a model achieves the balance between false negatives and false positives. This is also a useful measure to find out effectiveness of the prediction model. For MOLR, F-measure changes as cost factor value changes, recall being one of

Algorithm 2 Algorithm to select best MOLR model among several models of final population produced by genetic algorithm.

function FINDBESTMODEL($X, train\_data, \alpha$)

/* X is a $p \times (m+1)$ matrix which represents set of coefficient vectors obtained at the end of execution of multi-objective genetic algorithm. Here p is the number of coefficient vectors in X (final population) and m is the number of predictor metrics (6 CK metrics and LOC).*/

/* train_data represents $n \times m$ matrix denoting predictor metrics values for n classes in the training data.*/

/* α is the cost factor.*/

Y = CALCULATEMISCLASSCOSTRECALL($X, train\_data, \alpha$)

/* calculateMisclassCostRecall function will find out misclassification cost and recall for each of the coefficient vectors in X based on the current training data and the cost factor. Y is $p \times 2$ matrix.*/

/* Initializing $min\_distance$, max_recall, $min\_misclass\_cost$ variables to find out best model which is closest to ideal model with misclassification cost 0 and recall 1. */

$best\_model\_index \leftarrow (-1)$
$min\_dist \leftarrow \infty$
$min\_misclass\_cost \leftarrow \infty$
$max\_recall \leftarrow 0$

for i=0 to p **do**
  $dist \leftarrow$ SQRT($(Y[i,1])^2 + (1 - Y[i,2])^2$)
  if $dist < min\_dist$ **then**
    $best\_model\_index \leftarrow i$
    $min\_dist \leftarrow dist$
    $min\_misclass\_cost \leftarrow Y[i,1]$
    $max\_recall \leftarrow Y[i,2]$

  else if $dist = min\_dist$ & $Y[i,1] < min\_misclass\_cost$ **then**
    /* Choosing the model with lesser misclassification cost in case the distance is same as current minimum distance */
    $best\_model\_index \leftarrow i$
    $min\_dist \leftarrow dist$
    $min\_misclass\_cost \leftarrow Y[i,1]$
    $max\_recall \leftarrow Y[i,2]$
  end if
end for

$best\_model \leftarrow X[best\_model\_index]$
return *best_model*
end function

the objectives of MOLR. For single objective models it remains the same. F-measure can be calculated using the following equation.

$$F\text{-}measure = \frac{2 \cdot precision \cdot recall}{precision + recall} \tag{4.11}$$

As explained earlier, after choosing the best MOLR model from the training process we apply it on the data of test version. The evaluation measures(recall, misclassification cost and F-measure) are calculated for the MOLR and single objective models.

We perform two tailed Wilcoxon signed rank test for each pair of results between MOLR and other single objective algorithms to determine whether the following null hypothesis can be rejected.

- H_{01}*: There is no significant difference between evaluation measures of MOLR and single objective models.*

This comparison is different for all four single objective algorithms and for different cost factors. We have conducted experiments with different cost factors - 5, 10, 15, 20. One can choose appropriate cost factor as required by project, company and past experiences. For all the tests, the significance level is assumed to be 0.05, i.e. probability of rejecting the null hypothesis is 5%, when it should not be rejected.

4.5.3.2 Testing Process for M2

From the training process we get multiple models for MOLR. In this approach we apply all the training models on the data of test version. We compare all the single objective algorithms with multi-objective algorithm in terms of LOC cost and effectiveness. LOC cost and effectiveness can be found using Equations 4.8 and 4.6. After complete training and testing process we plot LOC cost and effectiveness obtained from data of testing version on the same graph for comparison purpose.

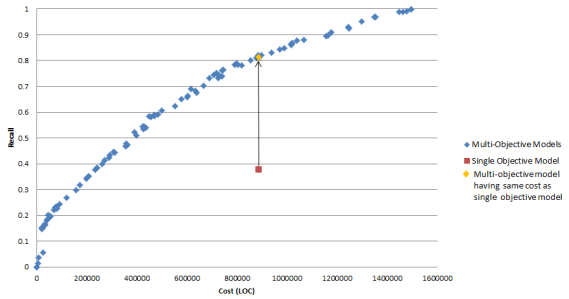


Figure 4.1: Example showing selection of multi-objective objective model having same cost as single objective model

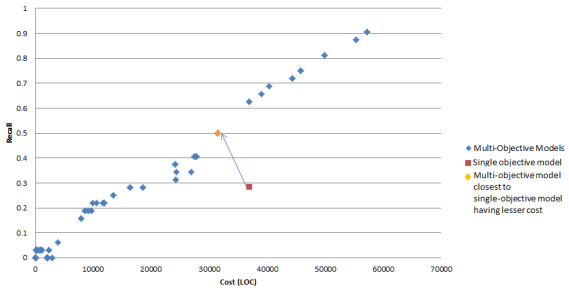


Figure 4.2: Example showing selection of closest multi-objective objective model having lesser cost than single objective model

From the definition of Pareto dominance described in the previous section, for each single objective model, we try to find out one MOLR model that has the same or lesser LOC cost than that of the single objective model. In particular we try to compare single and multi-objective models in terms of effectiveness at the same LOC cost that we have to spend with the single objective model. Figure 4.1 depicts the process of selecting multi-objective model corresponding to single objective model, having same LOC cost as that of single objective model. For the situation shown in Figure 4.1, multi-objective model has more effectiveness than single objective model.

Due to inherent randomness of NSGA-II algorithm, we may not get a multi-objective model having the same LOC cost as that of the single objective model. So we try to find nearest multi-objective model having lesser cost than single objective model. We find out how multi-objective model performs compared to the single objective model by spending lesser LOC cost than that to be spent with use of single objective model. The results show that multi-objective model is more effective even at lesser LOC cost. One example for this

situation is depicted in Figure 4.2. Here multi-objective model has more effectiveness than single objective model even at lesser LOC cost than that of the single objective model.

For M2, we compare single objective and multi-objective models in terms of effectiveness. As explained above, we find out MOLR model having same or lesser LOC cost as compared to given single objective model. After finding such a MOLR model for each of the single objective algorithm we compare recall values of both the models. We find out how much effective the MOLR model is compared to single objective model at the same LOC cost.

To prove significance of the results statistically, we compare recall values of MOLR and single objective model using two tailed Wilcoxon paired test to determine whether the following null hypotheses could be rejected.

- H_{02}*: There is no significant difference between recall values of MOLR and single objective predictors for cross version defect prediction at the same LOC cost.*

This comparison is different for all four single objective algorithms. In other words, we find out closest multi-objective model with respect to each single objective models, using the process described above. For all tests, the significance level is assumed to be 0.05, i.e. probability of rejecting the null hypothesis is 5%, when it should not be rejected.

4.5.4 Implementation Settings

MOLR is implemented using MATLAB Global Optimization Toolbox [126]. Other single objective algorithms are also implemented in MATLAB. The implementation settings are kept the same for all experiments and for both the approaches, so that we can compare results in the same conditions. Otherwise one can choose appropriate parameters according to project requirement.

The implementation details about single objective algorithms is as follows:

- **Logistic Regression:** *glmfit* function has been used for logistic regression imple-

mentation. The *distribution* parameter has been set to *binomial* and *link* parameter has been set to *logit*.

- **Naïve Bayes:** *NaïveBayes.fit* function has been used for Naïve Bayes implementation. Implementation of Naïve Bayes function requires unnormalized data as input. Rest of the functions requires normalized data as described in data preprocessing process (Section 4.5.1).

- **Decision Tree:** *classregtree* function has been used for decision tree implementation. The *method* parameter has been set to *classification*.

- **Random Forest:** *TreeBagger* function has been used for random forest implementation. The *method* parameter has been set to *classification*. Number of trees (*ntrees* parameter) has been set to 100. We experimented with different number of trees and chose the one having the least variation in results.

Table 4.3: Multi-Objective Genetic Algorithm Parameter Configuration

Parameter	Value
Population size	200
Initial Population	Random values between [-10,10]
Number of Generations	500
Crossover Function	Arithmetic crossover with crossover probability $p_c = 0.9$
Mutation function	Uniform Mutation with probability $p_m = 1/n$ where n is size of chromosomes
Stopping Criteria	If the average Pareto spread is less than 10^{-6} in the subsequent 50 generations, then the execution of GA is terminated.

gamultiobj function has been used for NSGA-II implementation. Selection of parameters for NSGA-II implementation has been done with reference to some of the previous studies like [1, 108, 124, 127] and experimentation. For the parameters chosen by experimentation we have taken *spread* value, defined by Deb, as the evaluation criteria [128]. Spread value gives an indication of how well the solutions are spaced on the final Pareto front. The greater the spread value, the better the solutions on Pareto front. Krall *et al.* [127]

have also used spread as one of the parameters to evaluate the multi-objective algorithm proposed by them. Spread value has been calculated on the Pareto fronts obtained after each run of NGSA-II on the data of training version. The parameters that are used for NSGA-II implementation are described in Table 3.

For the first multi objective defect prediction problem M1, NSGA-II algorithm has been executed 30 times during training process. This is done to account for the inherent randomness of GAs. After each run we choose a best model as described with algorithm 1. At the end of 30 runs we take best model among all 30 models obtained during each run. This is also done with help of Algorithm 2, i.e. final logistic regression model is the one which is closest to ideal model (0 misclassification cost, 1 recall) among all the 30 models obtained in 30 runs. Final logistic regression model obtained from the training process is applied on the data of the testing version.

For our second problem, NSGA-II algorithm has been executed 31 times during training process. The reason behind choosing an odd number (31 runs) is to choose coefficient vectors with respect to median Pareto front. We chose Pareto front with median spread value as the final solution, i.e. the coefficient vectors corresponding to median spread value among these 31 runs. These coefficient vectors represent final logistic regression models obtained from the training process and these models will be applied on the data of the testing version.

4.6 Results and Discussion

We discuss the results obtained for both the multi-objective defect prediction models and four single objective approaches in this section. We have conducted 30 experiments in 11 projects amongst 41 versions. We have taken projects which had at least 3 versions to put more strength to cross version study, with the hope of achieving more generalizable results. Each experiment will be referred as *'project_name train_version-test_version'*. For example 'Ant 1.3-1.4' denotes that data of version 1.3 of Ant project is used as training

data and data of version 1.4 is used as testing data. We use this notation to denote each experiment throughout this section.

RQ1: How does the proposed M1 perform as compared to traditional single objective defect prediction models in the cross version defect prediction?

In this section we discuss about the results obtained from our experiments and try to answer research question RQ_1. As explained earlier we compare the performance in terms of misclassification cost, recall and F-measure.

We have built multi-objective defect prediction model MOLR with cost factors being 5, 10, 15, 20 and the results are presented in Table 4.4, Table 4.5, Table 4.6 and Table 4.7 respectively. At the end of training phase of MOLR, we will have the best possible MOLR model that has (misclassification cost, recall) closest to (0,1) as explained in Section 4.5.2. For each of the experiments, misclassification cost and recall values of MOLR and four single objective prediction models are recorded. For each of the experiment, we have boldfaced misclassification value if it is the lowest amongst all other misclassification values. Similarly we have boldfaced recall value if it is the largest amongst all other recall values. For example, misclassification cost is the lowest and recall is the highest for MOLR for the experiment 'Ant 1.3-1.4' as shown in in Table 4.4. And in this scenario, we can always claim that MOLR is preferred to other four single objective models.

From the Table 4.4, we can observe that MOLR achieves lesser misclassification cost and higher recall as compared to other single objective algorithms in most of the experiments. Overall, in 20 out of 30 experiments, MOLR achieve better performance in terms of misclassification cost and recall combined. As cost factor increases, the performance of MOLR keeps improving. We can observe that, for the cost factors 10, 15 and 20, MOLR model dominate 24, 25, and 28 out of 30 experiments respectively in terms of both evaluation measures (misclassification cost and recall).

With increase in cost factors, FN file is penalized more than FP file. Minimizing misclassification cost takes care of controlling False Negative files in prediction and hence recall

improves. Hence MOLR performs better than other single objective models with increase in cost factor value.

There are only 2 experiments namely 'Ivy 1.1-1.4' and 'Jedit 4.2-4.3' where MOLR is not able to achieve least misclassification cost. One of the reasons can be that the test versions of both experiments have very few classes which are actually defective.

For Camel '1.2-1.4', 'Ivy 1.1-1.4', 'Poi 1.5-2.0' and 'Velocity 1.5-1.6' experiments, recall values achieved by MOLR is 1. This means MOLR is able to identify all the defective classes accurately.

Table 4.4: Misclassification Cost and Recall Comparison for $\alpha = 5$

Project	Train	Test	MOLR		LR		NB		DT		RF	
			MC Cost	Recall	MC Cost	Recall	MC Cost	Recall	MC Cost	Recall	MC Cost	Recall
Ant	1.3	1.4	**0.9551**	**0.3000**	1.0169	0.1500	1.0674	0.1250	0.9719	0.2250	1.0843	0.1250
	1.4	1.5	0.6894	**0.9063**	0.4676	0.1563	0.7884	0.6563	**0.3686**	0.5625	0.5256	0.1250
	1.5	1.6	**0.8262**	**0.4239**	1.1225	0.1522	1.0342	0.2500	1.1909	0.1087	1.0712	0.1957
	1.6	1.7	**0.5557**	**0.8855**	0.6765	0.4398	0.8040	0.3735	0.6832	0.5121	0.6268	0.5121
Camel	1.0	1.2	**1.4227**	**0.2824**	1.7516	0.0139	1.5526	0.1574	1.7204	0.0370	1.7549	0.0139
	1.2	1.4	0.7924	**1.0000**	0.7397	0.1517	0.7144	0.2345	0.7592	0.3931	**0.7099**	0.4483
	1.4	1.6	**0.6705**	**0.7234**	0.9098	0.0798	0.8860	0.1489	0.8642	0.2394	0.8342	0.1702
Ivy	1.1	1.4	0.9253	**1.0000**	0.5560	0.8125	**0.4398**	0.3125	0.6805	0.6875	0.6141	0.7500
	1.4	2.0	**0.3977**	**0.7750**	0.4858	0.1750	0.5341	0.1250	0.5483	0.1250	0.5653	0.0250
Jedit	3.2	4.0	**0.5784**	**0.8667**	0.7745	0.4667	1.0523	0.2000	0.6928	0.5333	0.6569	0.5733
	4.0	4.1	**0.5064**	**0.8734**	0.8237	0.3671	1.0417	0.2152	0.6058	0.6329	0.6795	0.5063
	4.1	4.2	0.4823	**0.9375**	0.3924	0.5417	0.5259	0.3125	**0.3760**	0.8542	0.4005	0.6042
	4.2	4.3	0.3862	**0.6364**	**0.1077**	0.4546	0.1240	0.2727	0.1728	0.3636	0.1280	0.3636
Log4j	1.0	1.1	**0.5780**	**0.7568**	0.9083	0.4865	0.9908	0.4595	0.6147	0.6757	0.7798	0.5676
	1.1	1.2	**1.3854**	**0.7090**	3.5659	0.2275	3.2488	0.2963	3.4732	0.2487	3.7122	0.1958
Lucene	2.0	2.2	**0.6559**	**0.9028**	1.7692	0.4236	2.2915	0.2292	1.7692	0.4306	1.5830	0.4861
	2.2	2.4	**0.4412**	**0.9803**	0.7265	0.8670	2.2353	0.2906	1.1676	0.6897	0.9147	0.7636
Poi	1.5	2.0	0.8599	**1.0000**	0.8567	0.5946	**0.6879**	0.3243	0.8280	0.7838	0.8280	0.8919
	2.0	2.5	**2.5403**	**0.2581**	3.1351	0.0282	2.8649	0.1169	2.9558	0.0927	2.9870	0.0766
	2.5	3.0	**0.3778**	**0.9893**	0.6538	0.8577	2.5317	0.2135	1.1380	0.7011	0.8869	0.7829
Synapse	1.0	1.1	**0.9505**	0.4167	1.3514	0.0000	1.0991	**0.4333**	1.1486	0.1667	1.2072	0.1167
	1.1	1.2	**0.7344**	**0.7791**	1.3594	0.2093	1.5352	0.1279	1.1836	0.3488	1.2930	0.2558
Velocity	1.4	1.5	**0.5701**	**0.9225**	0.6729	0.8873	1.4112	0.6620	0.8037	0.8451	0.6121	0.9085
	1.5	1.6	0.6332	**1.0000**	**0.5721**	0.9615	1.2882	0.3205	0.6900	0.8590	0.5852	0.8718
Xalan	2.4	2.5	**1.2304**	**0.5736**	2.2864	0.0543	2.1644	0.1137	2.1270	0.1344	2.2379	0.0801
	2.5	2.6	**0.5356**	**0.9903**	1.4147	0.4453	2.0362	0.1484	0.8429	0.7348	1.1119	0.6034
	2.6	2.7	**0.7701**	**0.8463**	3.3333	0.3252	3.6084	0.2695	2.6095	0.4733	2.7393	0.4454
Xerces	1.0	1.2	0.8295	**0.9718**	0.7614	0.4085	0.8932	0.4507	**0.7159**	0.2535	0.7591	0.3380
	1.2	1.3	0.7550	**0.6232**	0.7174	0.0580	**0.6313**	0.3188	0.7638	0.1739	0.7395	0.1449
	1.3	1.4	**1.9711**	**0.4783**	3.4201	0.0801	3.1207	0.1625	3.1667	0.1510	3.4456	0.0732
Average				**0.7603**		**0.3625**		**0.2774**		**0.4346**		**0.4005**

F-measure is the harmonic mean of recall and precision and it explains how the model is balanced against false positives and false negatives. For each value of cost factor - 5, 10, 15, 20, F-measure achieved by the respective models are reported in Table 4.8. And

Table 4.5: Misclassification Cost and Recall Comparison for $\alpha = 10$

Project	Train	Test	MOLR		LR		NB		DT		RF	
			MC Cost	Recall	MC Cost	Recall	MC Cost	Recall	MC Cost	Recall	MC Cost	Recall
Ant	1.3	1.4	**1.4326**	**0.5000**	1.9719	0.1500	2.0506	0.1250	1.8427	0.2250	2.0506	0.1250
	1.4	1.5	0.7509	**0.9688**	0.9283	0.1563	0.9761	0.6563	**0.6075**	0.5625	0.9727	0.1563
	1.5	1.6	**0.7664**	**0.8261**	2.2336	0.1522	2.0171	0.2500	2.3590	0.1087	2.1538	0.1848
	1.6	1.7	**0.6349**	**0.9157**	1.3007	0.4398	1.5020	0.3735	1.2268	0.5121	1.0966	0.5482
Camel	1.0	1.2	**2.5707**	**0.3287**	3.5033	0.0139	3.0493	0.1574	3.4309	0.0370	3.4885	0.0185
	1.2	1.4	**0.7947**	**1.0000**	1.4450	0.1517	1.3509	0.2345	1.2638	0.3931	1.0940	0.4621
	1.4	1.6	**0.6984**	**0.9362**	1.8062	0.0798	1.7150	0.1489	1.6052	0.2394	1.6435	0.1702
Ivy	1.1	1.4	0.8755	**1.0000**	**0.6183**	0.8125	0.6680	0.3125	0.7842	0.6875	0.7261	0.8125
	1.4	2.0	**0.4773**	**0.8000**	0.9545	0.1750	1.0313	0.1250	1.0455	0.1250	1.0938	0.0500
Jedit	3.2	4.0	**0.6993**	**0.8800**	1.4281	0.4667	2.0327	0.2000	1.2647	0.5333	1.1078	0.6133
	4.0	4.1	**0.7115**	**0.8861**	1.6250	0.3671	2.0353	0.2152	1.0705	0.6329	1.2468	0.5317
	4.1	4.2	0.5995	**0.9583**	0.6921	0.5417	0.9755	0.3125	**0.4714**	0.8542	0.5886	0.6458
	4.2	4.3	0.4817	**0.6364**	**0.1687**	0.4546	0.2053	0.2727	0.2439	0.3636	0.1850	0.4546
Log4j	1.0	1.1	**1.0183**	**0.7568**	1.7798	0.4865	1.9083	0.4595	1.1651	0.6757	1.5138	0.5676
	1.1	1.2	**2.4341**	**0.7407**	7.1268	0.2275	6.4927	0.2963	6.9366	0.2487	7.4683	0.1905
Lucene	2.0	2.2	**0.9393**	**0.9028**	3.4494	0.4236	4.5385	0.2292	3.4292	0.4306	3.2186	0.4653
	2.2	2.4	**0.5294**	**0.9754**	1.1235	0.8670	4.3529	0.2906	2.0941	0.6897	1.6794	0.7537
Poi	1.5	2.0	0.8822	**1.0000**	1.0955	0.5946	1.0860	0.3243	0.9554	0.7838	**0.8758**	0.8919
	2.0	2.5	**4.3818**	**0.3427**	6.2649	0.0282	5.7091	0.1169	5.8779	0.0927	6.0104	0.0686
	2.5	3.0	**0.4887**	**0.9715**	1.1063	0.8577	5.0317	0.2135	2.0882	0.7011	1.6176	0.7758
Synapse	1.0	1.1	**1.6757**	**0.4500**	2.7027	0.0000	1.8649	0.4333	2.2748	0.1667	2.3604	0.1333
	1.1	1.2	**0.9453**	**0.8488**	2.6875	0.2093	3.0000	0.1279	2.2773	0.3488	2.5078	0.2674
Velocity	1.4	1.5	**0.5187**	**0.9718**	1.0467	0.8873	2.5327	0.6620	1.3178	0.8451	0.8178	0.9225
	1.5	1.6	0.6419	**1.0000**	**0.6376**	0.9615	2.4454	0.3205	0.9301	0.8590	0.7860	0.8718
Xalan	2.4	2.5	**1.9738**	**0.6408**	4.5654	0.0543	4.3001	0.1137	4.2130	0.1344	4.4060	0.0904
	2.5	2.6	**0.5458**	**0.9951**	2.7028	0.4453	4.0136	0.1484	1.4588	0.7348	1.8000	0.6545
	2.6	2.7	**0.3311**	**0.9677**	6.6667	0.3252	7.2167	0.2695	5.2112	0.4733	5.5006	0.4432
Xerces	1.0	1.2	**0.8500**	**0.9718**	1.2386	0.4085	1.3364	0.4507	1.3182	0.2535	1.1818	0.4085
	1.2	1.3	**1.0088**	**0.6812**	1.4349	0.0580	1.1501	0.3188	1.3929	0.1739	1.4062	0.1304
	1.3	1.4	**3.1412**	**0.5835**	6.8384	0.0801	6.2330	0.1625	6.3214	0.1510	6.8912	0.0732
Average				**0.8146**		**0.3625**		**0.2774**		**0.4346**		**0.4161**

Table 4.6: Misclassification Cost and Recall Comparison for $\alpha = 15$

Project	Train	Test	MOLR		LR		NB		DT		RF	
			MC Cost	Recall	MC Cost	Recall	MC Cost	Recall	MC Cost	Recall	MC Cost	Recall
Ant	1.3	1.4	**2.0787**	**0.4750**	2.9270	0.1500	3.0337	0.1250	2.7135	0.2250	3.1124	0.1000
	1.4	1.5	**0.8464**	**0.9375**	1.3891	0.1563	1.1638	0.6563	0.8464	0.5625	1.4437	0.1563
	1.5	1.6	**1.0399**	**0.8152**	3.3447	0.1522	3.0000	0.2500	3.5271	0.1087	3.2678	0.1739
	1.6	1.7	**0.8148**	**0.9036**	1.9248	0.4398	2.2000	0.3735	1.7705	0.5121	1.5960	0.5482
Camel	1.0	1.2	**3.6266**	**0.3565**	5.2549	0.0139	4.5461	0.1574	5.1414	0.0370	5.2566	0.0139
	1.2	1.4	**0.7936**	**1.0000**	2.1502	0.1517	1.9874	0.2345	1.7683	0.3931	1.6594	0.4207
	1.4	1.6	**0.7741**	**0.9628**	2.7026	0.0798	2.5440	0.1489	2.3461	0.2394	2.3648	0.2021
Ivy	1.1	1.4	0.8589	**1.0000**	**0.6805**	0.8125	0.8963	0.3125	0.8880	0.6875	0.8589	0.7500
	1.4	2.0	**0.6278**	**0.8000**	1.4233	0.1750	1.5284	0.1250	1.5426	0.1250	1.6364	0.0500
Jedit	3.2	4.0	**0.9183**	**0.8667**	2.0817	0.4667	3.0131	0.2000	1.8366	0.5333	1.5196	0.6267
	4.0	4.1	**0.8301**	**0.9241**	2.4263	0.3671	3.0288	0.2152	1.5353	0.6329	1.8397	0.5317
	4.1	4.2	0.5940	**0.9792**	0.9918	0.5417	1.4251	0.3125	**0.5668**	0.8542	0.8965	0.6042
	4.2	4.3	0.5346	**0.6364**	**0.2297**	0.4546	0.2866	0.2727	0.3150	0.3636	0.2663	0.3636
Log4j	1.0	1.1	**1.2569**	**0.8108**	2.6514	0.4865	2.8257	0.4595	1.7156	0.6757	2.1009	0.5946
	1.1	1.2	**4.5805**	**0.6720**	10.6878	0.2275	9.7366	0.2963	10.4000	0.2487	11.0537	0.2011
Lucene	2.0	2.2	**1.2834**	**0.8958**	5.1296	0.4236	6.7854	0.2292	5.0891	0.4306	4.6599	0.4792
	2.2	2.4	**0.5559**	**0.9803**	1.5206	0.8670	6.4706	0.2906	3.0206	0.6897	1.9147	0.8128
Poi	1.5	2.0	0.8822	**1.0000**	1.3344	0.5946	1.4841	0.3243	1.0828	0.7838	**0.8089**	0.9730
	2.0	2.5	**6.1195**	**0.3871**	9.3948	0.0282	8.5532	0.1169	8.8000	0.0927	8.9351	0.0766
	2.5	3.0	**0.5045**	**0.9822**	1.5588	0.8577	7.5317	0.2135	3.0385	0.7011	2.2285	0.7865
Synapse	1.0	1.1	**2.6126**	0.4000	4.0541	0.0000	2.6306	**0.4333**	3.4009	0.1667	3.5901	0.1167
	1.1	1.2	**0.5977**	**0.9884**	4.0156	0.2093	4.4648	0.1279	3.3711	0.3488	3.8477	0.2442
Velocity	1.4	1.5	**0.6028**	**0.9718**	1.4206	0.8873	3.6542	0.6620	1.8318	0.8451	1.1589	0.9155
	1.5	1.6	**0.6245**	**1.0000**	0.7031	0.9615	3.6026	0.3205	1.1703	0.8590	1.0087	0.8718
Xalan	2.4	2.5	**2.4620**	**0.7003**	6.8443	0.0543	6.4359	0.1137	6.2989	0.1344	6.6389	0.0853
	2.5	2.6	**0.8203**	**0.9586**	3.9910	0.4453	5.9910	0.1484	2.0746	0.7348	2.3480	0.6910
	2.6	2.7	**0.6062**	**0.9599**	10.0000	0.3252	10.8251	0.2695	7.8130	0.4733	8.1683	0.4488
Xerces	1.0	1.2	**0.8886**	**0.9155**	1.7159	0.4085	1.7795	0.4507	1.9205	0.2535	1.6318	0.4225
	1.2	1.3	**1.2450**	**0.6812**	2.1523	0.0580	1.6689	0.3188	2.0221	0.1739	2.0839	0.1449
	1.3	1.4	**4.5340**	**0.5973**	10.2568	0.0801	9.3452	0.1625	9.4762	0.1510	10.3078	0.0755
Average				**0.8186**		**0.3625**		**0.2774**		**0.4346**		**0.4160**

Table 4.7: Misclassification Cost and Recall Comparison for $\alpha = 20$

Project	Train	Test	MOLR		LR		NB		DT		RF	
			MC Cost	Recall	MC Cost	Recall	MC Cost	Recall	MC Cost	Recall	MC Cost	Recall
Ant	1.3	1.4	**2.2303**	**0.5750**	3.8820	0.1500	4.0169	0.1250	3.5843	0.2250	3.8989	0.1500
	1.4	1.5	**0.8498**	**0.9375**	1.8498	0.1563	1.3515	0.6563	1.0853	0.5625	1.7850	0.2188
	1.5	1.6	**1.2251**	**0.8261**	4.4558	0.1522	3.9829	0.2500	4.6952	0.1087	4.3476	0.1739
	1.6	1.7	**0.9879**	**0.8976**	2.5490	0.4398	2.8980	0.3735	2.3141	0.5121	2.1987	0.5241
Camel	1.0	1.2	**4.8059**	**0.3519**	7.0066	0.0139	6.0428	0.1574	6.8520	0.0370	7.0740	0.0046
	1.2	1.4	**0.7936**	**1.0000**	2.8555	0.1517	2.6239	0.2345	2.2729	0.3931	2.1021	0.4345
	1.4	1.6	**0.7627**	**0.9734**	3.5990	0.0798	3.3731	0.1489	3.0870	0.2394	3.2394	0.1755
Ivy	1.1	1.4	0.8797	**1.0000**	**0.7427**	0.8125	1.1245	0.3125	0.9917	0.6875	0.8423	0.8125
	1.4	2.0	**0.6165**	**0.8500**	1.8920	0.1750	2.0256	0.1250	2.0398	0.1250	2.2898	0.0000
Jedit	3.2	4.0	**0.9444**	**0.9333**	2.7353	0.4667	3.9935	0.2000	2.4085	0.5333	2.0523	0.6133
	4.0	4.1	**0.9840**	**0.9114**	3.2276	0.3671	4.0224	0.2152	2.0000	0.6329	2.3045	0.5570
	4.1	4.2	**0.6294**	**0.9792**	1.2916	0.5417	1.8747	0.3125	0.6621	0.8542	1.0599	0.6458
	4.2	4.3	0.6138	**0.6364**	**0.2907**	0.4546	0.3679	0.2727	0.3862	0.3636	0.3333	0.3636
Log4j	1.0	1.1	**1.2752**	**0.8649**	3.5229	0.4865	3.7431	0.4595	2.2661	0.6757	2.9725	0.5676
	1.1	1.2	**6.1902**	**0.6667**	14.2488	0.2275	12.9805	0.2963	13.8634	0.2487	14.8342	0.1958
Lucene	2.0	2.2	**1.2632**	**0.9236**	6.8097	0.4236	9.0324	0.2292	6.7490	0.4306	6.1741	0.4792
	2.2	2.4	**0.9029**	**0.9557**	1.9176	0.8670	8.5882	0.2906	3.9471	0.6897	3.0941	0.7586
Poi	1.5	2.0	**0.8822**	**1.0000**	1.5732	0.5946	1.8822	0.3243	1.2102	0.7838	1.0223	0.8919
	2.0	2.5	**2.2468**	**0.8468**	12.5247	0.0282	11.3974	0.1169	11.7221	0.0927	12.0130	0.0686
	2.5	3.0	**0.4593**	**0.9893**	2.0113	0.8577	10.0317	0.2135	3.9887	0.7011	2.8190	0.7936
Synapse	1.0	1.1	**3.2658**	**0.4333**	5.4054	0.0000	3.3964	0.4333	4.5270	0.1667	4.7928	0.1167
	1.1	1.2	**0.9375**	**0.9419**	5.3438	0.2093	5.9297	0.1279	4.4648	0.3488	4.9688	0.2674
Velocity	1.4	1.5	**0.7850**	**0.9648**	1.7944	0.8873	4.7757	0.6620	2.3458	0.8451	1.3411	0.9225
	1.5	1.6	**0.6288**	**1.0000**	0.7686	0.9615	4.7598	0.3205	1.4105	0.8590	1.2271	0.8718
Xalan	2.4	2.5	**2.4135**	**0.7830**	9.1233	0.0543	8.5716	0.1137	8.3848	0.1344	8.8406	0.0853
	2.5	2.6	**0.5480**	**0.9951**	5.2791	0.4453	7.9684	0.1484	2.6904	0.7348	3.0927	0.6886
	2.6	2.7	**0.8482**	**0.9577**	13.3333	0.3252	14.4334	0.2695	10.4147	0.4733	10.8691	0.4499
Xerces	1.0	1.2	**0.9341**	**0.9437**	2.1932	0.4085	2.2227	0.4507	2.5227	0.2535	1.8932	0.5070
	1.2	1.3	**1.1280**	**0.8551**	2.8698	0.0580	2.1876	0.3188	2.6512	0.1739	2.6556	0.1449
	1.3	1.4	**4.1395**	**0.7254**	13.6752	0.0801	12.4575	0.1625	12.6310	0.1510	13.8129	0.0709
Average				**0.8573**		**0.3625**		**0.2774**		**0.4346**		**0.4185**

Table 4.8: F-measure comparison for different cost factor values

Project	Train	Test	F-measure							
			MOLR ($\alpha = 5$)	MOLR ($\alpha = 10$)	MOLR ($\alpha = 15$)	MOLR ($\alpha = 20$)	LR	NB	DT	RF
Ant	1.3	1.4	0.29268	0.34783	0.33333	0.38333	0.21053	0.16667	0.26866	0.15873
	1.4	1.5	0.23387	0.22711	0.21429	0.2214	0.25641	0.18341	0.40909	0.16
	1.5	1.6	0.5	0.54874	0.54152	0.54676	0.25455	0.34586	0.18182	0.31034
	1.6	1.7	0.46519	0.46697	0.43924	0.41913	0.52518	0.40391	0.47887	0.54313
Camel	1.0	1.2	0.33243	0.355	0.37288	0.36715	0.0274	0.23944	0.06957	0.02715
	1.2	1.4	0.29562	0.29502	0.29532	0.29532	0.22335	0.2753	0.26887	0.30303
	1.4	1.6	0.38256	0.38344	0.35806	0.36346	0.13889	0.20664	0.25568	0.26122
Ivy	1.1	1.4	0.12549	0.13169	0.13389	0.13115	0.17568	0.13889	0.13253	0.15385
	1.4	2.0	0.37349	0.4	0.36994	0.39766	0.26415	0.17241	0.15873	0.04444
Jedit	3.2	4.0	0.48689	0.49811	0.4797	0.41916	0.47619	0.26786	0.52632	0.54088
	4.0	4.1	0.53906	0.49822	0.45483	0.45283	0.50435	0.30631	0.57803	0.58824
	4.1	4.2	0.35294	0.31293	0.31544	0.30719	0.48148	0.32967	0.42708	0.44961
	4.2	4.3	0.07447	0.06512	0.06335	0.05833	0.25641	0.17143	0.12308	0.18605
Log-4j	1.0	1.1	0.6747	0.65116	0.60606	0.59259	0.61017	0.54839	0.72464	0.66667
	1.1	1.2	0.80723	0.8284	0.78154	0.77778	0.3691	0.45528	0.39496	0.32599
Lucene	2.0	2.2	0.71038	0.71038	0.70685	0.72087	0.53744	0.35106	0.53219	0.59574
	2.2	2.4	0.74812	0.74576	0.74953	0.74046	0.7169	0.39073	0.65882	0.72261
Poi	1.5	2.0	0.21512	0.21083	0.21083	0.21083	0.17391	0.17143	0.2028	0.2129
	2.0	2.5	0.34595	0.4359	0.45714	0.746	0.05447	0.20351	0.16197	0.13971
	2.5	3.0	0.782	0.7913	0.78298	0.79202	0.78887	0.33803	0.70232	0.7483
Synapse	1.0	1.1	0.41322	0.4186	0.3871	0.39695	0	0.325	0.26667	0.2
	1.1	1.2	0.54472	0.53875	0.55016	0.52769	0.32143	0.1913	0.43165	0.36975
Velocity	1.4	1.5	0.77059	0.78632	0.79083	0.78963	0.75904	0.63087	0.74074	0.76558
	1.5	1.6	0.51827	0.51485	0.52174	0.52	0.55762	0.37594	0.54032	0.5913
Xalan	2.4	2.5	0.57513	0.59759	0.60559	0.63924	0.10145	0.19383	0.22034	0.14253
	2.5	2.6	0.63994	0.63757	0.61755	0.64664	0.51841	0.23282	0.66083	0.59903
	2.6	2.7	0.91127	0.9775	0.97346	0.9723	0.49076	0.42456	0.6391	0.61633
Xerces	1.0	1.2	0.27879	0.27935	0.29748	0.28571	0.25778	0.21262	0.25899	0.24742
	1.2	1.3	0.26543	0.26629	0.26857	0.26879	0.10959	0.30986	0.16901	0.16807
	1.3	1.4	0.62857	0.70932	0.72099	0.80457	0.14799	0.2768	0.25882	0.13617
Average			0.4761	0.4876	0.4800	0.4931	0.3436	0.2946	0.3814	0.3658

Table 4.9: Results of Wilcoxon signed rank test. Table shows p-value obtained by comparing performance measures of MOLR and other algorithms

Cost	Mis-classification Cost				Recall				F-measure			
Factor	LR	NB	DT	RF	LR	NB	DT	RF	LR	NB	DT	RF
5	0.00096	0.0002	0.00128	0.0015	0	0	0	0	0.00278	0	0.00496	0.00298
10	0	0	0	0	0	0	0	0	0.00168	0	0.00528	0.00466
15	0	0	0	0	0	0	0	0	0.00386	0	0.01174	0.01108
20	0	0	0	0	0	0	0	0	0.00528	0	0.01552	0.01242

also F-measure of four single objective prediction models are also shown in the same table. F-measure of MOLR models are relatively better than other single objective algorithms.

For 'Xalan 2.6-2.7' experiment, F-measure is greater than 0.9 for all values of cost factosr. Unlike other experiments F-measure does not always increase with increase in cost factor. We can observe that on an average, for different values of cost factor, MOLR achieve better F-measure as compared to four single objective prediction models.

Wilcoxon signed test is applied to test whether MOLR model is significantly better than single objective optimization model. If p-value is less than 0.05 we can conclude that MOLR is significantly better than the single objective prediction model. For each evaluation measure (recall, misclassification cost, F-measure) and cost factor value (5, 10,15, 20) the MOLR model is tested to check whether it is significantly better than the four single objective prediction models. And p-values are shown in Table 4.9.

From the results, one can observe that p-value is less than 0.05 for all evaluation measures and cost factors. In fact, for comparison in terms of recall value, p-value is zero for all the experiments. Even for misclassification cost, p-value is zero for the cost factor greater than or equal to 10. This shows that MOLR is working quite dominantly. As p-value is less than 0.05 for all evaluation measures and cost factors, we can easily reject null hypothesis H_{01}. This confirms domination of MOLR over single objective prediction models.

RQ2: How does the proposed M2 perform as compared to traditional single objective defect prediction models in the cross version defect prediction?

In this sub-section we discuss about results of our experiments to answer our second research question RQ_2. As explained earlier, we compare the performance of MOLR and four single objective prediction models in terms of cost and recall.

At the end of the training phase of building multi-objective defect prediction model, there are several models with varying cost and effectiveness. As described in Section 4.5.3.2, for each single objective prediction model we select the closest multi-objective model having

the same or lesser LOC cost. We find out how effective (in terms of recall) the multi-objective model is compared to single objective model at the same or lesser LOC cost.

The comparative results of single objective logistic regression model and the corresponding multi-objective prediction model are shown in Table 4.10. We also report LOC cost difference along with recall difference of two models since we compare effectiveness at the same or lesser LOC cost. From the Table 4.10, it can be observed that, for the experiment Ant 1.6 - 1.7, MOLR gained recall of 0.1446 by incurring lesser cost (-1050) than that of Single Objective Logistic Regression.

The comparative results of multi-objective prediction model with single objective prediction models - Naïve Bayes classifier, decision tree, random forest are shown in Table 4.11, Table 4.12, and Table 4.13 respectively. We take a deeper look on each comparison table. For each comparison table, we discuss the gain in recall values achieved by MOLR and the special cases where MOLR did not achieve any gain compared to single objective algorithm. From the tables it can be seen that recall values of MOLR are better than single objective algorithms for most of the cases.

The comparative results of MOLR and Single Objective logistic regression are shown in Table 4.10. For 25 out of 30 experiments, MOLR is able to achieve better recall than single objective logistic regression. MOLR is able to achieve 1% to 65% gain in recall. In Xalan project all 3 experiments achieved more than 40% gain in recall, with Xalan 2.6-2.7 experiment achieving highest recall gain of 65.14%. The LOC cost difference is highest for Xerces 1.3-1.4 experiment (LOC difference of 11412), but recall gain achieved by MOLR model is 63.39%.

There are 5 cases when recall value of MOLR is lesser than or equal to SOLR values (Ant 1.3-1.4, Ant 1.4-1.5, Ant 1.5-1.6, Jedit 4.2-4.3, and Synapse 1.0-1.1). In Synapse 1.0-1.1 experiment SOLR achieves zero LOC cost and effectiveness. The reason for this can be that the train version Synapse-1.0 had very few defective classes (10% of 157 classes), that gave poor performance when tested on Synapse version 1.1. The SOLR predictor classified all classes as non-defective, resulting in zero LOC cost and zero effectiveness. The loss in recall varies from 0% to 10% for these 5 cases, which is very less compared to gains we

achieve for rest of the 25 experiments. For three projects recall remains the same but for only two project MOLR was not able to get better recall with lesser cost.

Average recall gain across all experiments is 22.77%. This shows how effectively MOLR model is able to predict defect prone classes compared to SOLR.

The recall and LOC cost values for MOLR and Naïve Bayes classifier are reported in Table 4.11. MOLR is able to outperform Naïve Bayes in all 30 experiments. The gain in recall varies from 1% to 76%. In particular, there are 17 cases when MOLR is able to achieve more than 30% gain in recall compared to Naïve Bayes classifier. In this case also, maximum cost difference is reported for Xerces 1.3 - 1.4 experiment (LOC difference of 13834). The gain in recall value for this experiment is 76.89%, which is maximum among all experiments. MOLR is able to achieve average recall gain of 32.65% across all experiments.

The recall and LOC cost values for MOLR and decision tree are reported in Table 4.12. In most of the cases, decision tree achieves good recall values. The decision tree is able to achieve more than 60% recall for ten experiments, but still MOLR achieves better recall values in most of the cases. There are 5 cases where the recall values of MOLR are less than or equal to decision tree (Ant 1.4-1.5, Ant 1.5 1.6, Jedit 4.2-4.3, Log4j 1.0-1.1, and Synapse 1.0-1.1). The loss in recall varies from 0% to 13% for these 5 cases. For rest of the 25 cases recall values of MOLR are higher than decision tree. The gain in recall value varies from 2% to 72%. Maximum recall gain of 72.31% is reported for Xerces 1.3-1.4 experiment. And maximum cost difference is reported for Xerces init-1.2 case (LOC difference of 25530). In this case MOLR achieves recall gain of 36.62%. MOLR is able to achieve average recall gain of 21.08% across all experiments.

The comparative results of MOLR and Random Forest classifier are shown in Table 4.13. There are only 3 cases when recall values of MOLR is lesser than random forest recall values (Camel 1.0-1.2, Synapse 1.0-1.1, and Xerces init-1.2). For rest of the 27 cases, MOLR achieves recall gain up to 77%. Maximum recall gain of 77.12% is achieved for Xerces 1.3-1.4 experiment. MOLR is able to achieve average recall gain of 21.83% across all experiments. Overall recall gains are not as significant as other single objective algorithms like

Table 4.10: Single Objective Logistic Regression vs Multi-Objective Logistic Regression

Project	Train Version	Test Version	SOLR		MOLR		Difference	
			Cost (LOC)	Recall	Cost (LOC)	Recall	Cost (LOC)	Recall
Ant	1.3	1.4	15115	0.1500	**10091**	**0.1500**	-5024	0
	1.4	1.5	4614	0.1563	4447	0.0625	-167	-0.0938
	1.5	1.6	17451	0.1522	14907	0.0978	-2544	-0.0543
	1.6	1.7	102300	0.4398	**101250**	**0.5843**	-1050	+0.1446
Camel	1.0	1.2	360	0.0139	**281**	**0.0278**	-79	+0.0139
	1.2	1.4	24617	0.1517	**23840**	**0.5172**	-777	+0.3655
	1.4	1.6	18121	0.0798	**16678**	**0.4894**	-1443	+0.4096
Ivy	1.1	1.4	49984	0.8125	**43924**	**0.8750**	-6060	+0.0625
	1.4	2.0	19163	0.1750	**17360**	**0.2000**	-1803	+0.0250
Jedit	3.2	4.0	62288	0.4667	**61911**	**0.8000**	-377	+0.3333
	4.0	4.1	54274	0.3671	**52027**	**0.6582**	-2247	+0.2911
	4.1	4.2	75135	0.5417	**68974**	**0.6667**	-6161	+0.1250
	4.2	4.3	56442	0.4546	**50416**	**0.4546**	-6026	0
Log-4j	1.0	1.1	8870	0.4865	**8588**	**0.5946**	-282	+0.1081
	1.1	1.2	19768	0.2275	**18850**	**0.6561**	-918	+0.4286
Lucene	2.0	2.2	46442	0.4236	**45586**	**0.9097**	-856	+0.4861
	2.2	2.4	89882	0.8670	**78787**	**0.9557**	-11095	+0.0887
Poi	1.5	2.0	70112	0.5946	**69570**	**0.8649**	-542	+0.2703
	2.0	2.5	28501	0.0282	**26640**	**0.4597**	-1861	+0.4315
	2.5	3.0	112170	0.8577	**104730**	**0.9751**	-7440	+0.1174
Synapse	1.0	1.1	0	0.0000	0	0.0000	0	0
	1.1	1.2	16487	0.2093	**13637**	**0.3954**	-2850	+0.1861
Velocity	1.4	1.5	28875	0.8873	**28727**	**0.9366**	-148	+0.0493
	1.5	1.6	55801	0.9615	**55062**	**1.0000**	-739	+0.0385
Xalan	2.4	2.5	60444	0.0543	**55952**	**0.6021**	-4492	+0.5478
	2.5	2.6	258200	0.4453	**255950**	**0.8832**	-2250	+0.4380
	2.6	2.7	354260	0.3252	**345740**	**0.9766**	-8520	+0.6514
Xerces	1.0	1.2	112560	0.4085	**105950**	**0.6197**	-6610	+0.2113
	1.2	1.3	30597	0.0580	**28773**	**0.5797**	-1824	+0.5217
	1.3	1.4	56835	0.0801	**45423**	**0.7140**	-11412	+0.6339
Average				0.3625		**0.5902**		+0.2277

Table 4.11: Naïve Bayes vs Multi-Objective Logistic Regression

Project	Train Version	Test Version	NB		MOLR		Difference	
			Cost (LOC)	Recall	Cost (LOC)	Recall	Cost (LOC)	Recall
Ant	1.3	1.4	15333	0.1250	**10091**	**0.1500**	-5242	+0.0250
	1.4	1.5	58017	0.6563	**54519**	**0.8438**	-3498	+0.1875
	1.5	1.6	38203	0.2500	**37861**	**0.4022**	-342	+0.1522
	1.6	1.7	86944	0.3735	**85265**	**0.4940**	-1679	+0.1205
Camel	1.0	1.2	12036	0.1574	**10890**	**0.1667**	-1146	+0.0093
	1.2	1.4	30307	0.2345	**29462**	**0.5724**	-845	+0.3379
	1.4	1.6	30205	0.1489	**29745**	**0.6117**	-460	+0.4628
Ivy	1.1	1.4	32032	0.3125	**30380**	**0.7500**	-1652	+0.4375
	1.4	2.0	16486	0.1250	**15265**	**0.2500**	-1221	+0.1250
Jedit	3.2	4.0	45376	0.2000	**45040**	**0.6267**	-336	+0.4267
	4.0	4.1	45988	0.2152	**44391**	**0.5443**	-1597	+0.3291
	4.1	4.2	68278	0.3125	**67847**	**0.6667**	-431	+0.3542
	4.2	4.3	59442	0.2727	**50416**	**0.4546**	-9026	+0.1818
Log-4j	1.0	1.1	7685	0.4595	**7672**	**0.5676**	-13	+0.1081
	1.1	1.2	17090	0.2963	**16824**	**0.5450**	-266	+0.2487
Lucene	2.0	2.2	25216	0.2292	**24233**	**0.7500**	-983	+0.5208
	2.2	2.4	35849	0.2906	**34735**	**0.7291**	-1114	+0.4384
Poi	1.5	2.0	36223	0.3243	**36180**	**0.4865**	-43	+0.1622
	2.0	2.5	40844	0.1169	**35412**	**0.4758**	-5432	+0.3589
	2.5	3.0	40658	0.2135	**40369**	**0.6619**	-289	+0.4484
Synapse	1.0	1.1	28270	0.4333	**25356**	**0.5000**	-2914	+0.0667
	1.1	1.2	12121	0.1279	**11763**	**0.3023**	-358	+0.1744
Velocity	1.4	1.5	22826	0.6620	**22658**	**0.8732**	-168	+0.2113
	1.5	1.6	43271	0.3205	**42496**	**0.9872**	-775	+0.6667
Xalan	2.4	2.5	71233	0.1137	**70314**	**0.6615**	-919	+0.5478
	2.5	2.6	70599	0.1484	**68896**	**0.4818**	-1703	+0.3333
	2.6	2.7	306030	0.2695	**299830**	**0.9577**	-6200	+0.6882
Xerces	1.0	1.2	56364	0.4507	**55448**	**0.7606**	-916	+0.3099
	1.2	1.3	123940	0.3188	**114900**	**0.9130**	-9040	+0.5942
	1.3	1.4	86894	0.1625	**73060**	**0.9314**	-13834	+0.7689
			Average	0.2774		**0.6039**		+0.3265

Table 4.12: Decision Tree vs Multi-Objective Logistic Regression

Project	Train Version	Test Version	D. Tree		MOLR		Difference	
			Cost (LOC)	Recall	Cost (LOC)	Recall	Cost (LOC)	Recall
Ant	1.3	1.4	22266	0.2250	**22112**	**0.3500**	-154	+0.1250
	1.4	1.5	30936	0.5625	30751	0.4375	-185	-0.1250
	1.5	1.6	17600	0.1087	14907	0.0978	-2693	-0.0109
	1.6	1.7	104610	0.5121	**101250**	**0.5843**	-3360	+0.0723
Camel	1.0	1.2	2937	0.0370	**2626**	**0.0556**	-311	+0.0185
	1.2	1.4	34238	0.3931	**33828**	**0.6000**	-410	+0.2069
	1.4	1.6	31877	0.2394	**31470**	**0.6277**	-407	+0.3883
Ivy	1.1	1.4	46099	0.6875	**43924**	**0.8750**	-2175	+0.1875
	1.4	2.0	19995	0.1250	**17360**	**0.2000**	-2635	+0.0750
Jedit	3.2	4.0	51516	0.5333	**51047**	**0.6933**	-469	+0.1600
	4.0	4.1	78999	0.6329	**77452**	**0.8608**	-1547	+0.2279
	4.1	4.2	118510	0.8542	**112650**	**0.9375**	-5860	+0.0833
	4.2	4.3	44481	0.3636	**39275**	**0.3636**	-5206	0
Log-4j	1.0	1.1	11925	0.6757	10162	0.6216	-1763	-0.0541
	1.1	1.2	20216	0.2487	**20019**	**0.6720**	-197	+0.4233
Lucene	2.0	2.2	39645	0.4306	**38183**	**0.8958**	-1462	+0.4653
	2.2	2.4	70512	0.6897	**70427**	**0.9360**	-85	+0.2463
Poi	1.5	2.0	71231	0.7838	**69570**	**0.8649**	-1661	+0.0811
	2.0	2.5	41642	0.0927	**41048**	**0.5282**	-594	+0.4355
	2.5	3.0	91731	0.7011	**90774**	**0.9537**	-957	+0.2527
Synapse	1.0	1.1	8531	0.1667	4909	0.1000	-3622	-0.0667
	1.1	1.2	20402	0.3488	**20390**	**0.4884**	-12	+0.1395
Velocity	1.4	1.5	27690	0.8451	**25848**	**0.9085**	-1842	+0.0634
	1.5	1.6	52329	0.8590	**42496**	**0.9872**	-9833	+0.1282
Xalan	2.4	2.5	84023	0.1344	**83456**	**0.7158**	-567	+0.5814
	2.5	2.6	343230	0.7348	**336330**	**0.9562**	-6900	+0.2214
	2.6	2.7	352990	0.4733	**345740**	**0.9766**	-7250	+0.5033
Xerces	1.0	1.2	102530	0.2535	**77000**	**0.6197**	-25530	+0.3662
	1.2	1.3	29330	0.1739	**28773**	**0.5797**	-557	+0.4058
	1.3	1.4	64336	0.1510	**59471**	**0.8741**	-4865	+0.7231
			Average	0.4346		**0.6454**		+0.2108

Table 4.13: Random Forest vs Multi-Objective Logistic Regression

Project	Train Version	Test Version	Random Forest		MOLR		Difference	
			Cost (LOC)	Recall	Cost (LOC)	Recall	Cost (LOC)	Recall
Ant	1.3	1.4	18778	0.1750	**18699**	**0.3500**	-79	+0.1750
	1.4	1.5	10176	0.1563	**9342**	**0.1875**	-834	+0.0313
	1.5	1.6	30515	0.2174	**27254**	**0.3044**	-3261	+0.0870
	1.6	1.7	112400	0.5301	**110760**	**0.6566**	-1640	+0.1265
Camel	1.0	1.2	56	0.0093	22	0.0046	-34	-0.0046
	1.2	1.4	38565	0.4207	**37281**	**0.6207**	-1284	+0.2000
	1.4	1.6	22497	0.2021	**20718**	**0.5372**	-1779	+0.3351
Ivy	1.1	1.4	48820	0.7500	**43924**	**0.8750**	-4896	+0.1250
	1.4	2.0	10135	0.0750	**6557**	**0.1250**	-3578	+0.0500
Jedit	3.2	4.0	65254	0.5733	**64131**	**0.8400**	-1123	+0.2667
	4.0	4.1	61109	0.5570	**59899**	**0.7215**	-1210	+0.1646
	4.1	4.2	111950	0.6458	**111940**	**0.9375**	-10	+0.2917
	4.2	4.3	56148	0.3636	**50416**	**0.4546**	-5732	+0.0909
Log-4j	1.0	1.1	9247	0.5405	**9153**	**0.6216**	-94	+0.0811
	1.1	1.2	20158	0.2169	**20019**	**0.6720**	-139	+0.4550
Lucene	2.0	2.2	43632	0.4514	**42251**	**0.9306**	-1381	+0.4792
	2.2	2.4	86894	0.7783	**78787**	**0.9557**	-8107	+0.1773
Poi	1.5	2.0	88547	0.9189	**83322**	**0.9730**	-5225	+0.0541
	2.0	2.5	38128	0.0685	**35412**	**0.4758**	-2716	+0.4073
	2.5	3.0	106020	0.7687	**104730**	**0.9751**	-1290	+0.2064
Synapse	1.0	1.1	5916	0.1167	4909	0.1000	-1007	-0.0167
	1.1	1.2	16572	0.2558	**13637**	**0.3954**	-2935	+0.1395
Velocity	1.4	1.5	29876	0.9155	**29841**	**0.9578**	-35	+0.0423
	1.5	1.6	50271	0.8462	**42496**	**0.9872**	-7775	+0.1410
Xalan	2.4	2.5	73382	0.0904	**70314**	**0.6615**	-3068	+0.5711
	2.5	2.6	291530	0.7470	**281600**	**0.9027**	-9930	+0.1557
	2.6	2.7	351450	0.4298	**345740**	**0.9766**	-5710	+0.5468
Xerces	1.0	1.2	115990	0.6901	105950	0.6197	-10040	-0.0704
	1.2	1.3	31204	0.1594	**28773**	**0.5797**	-2431	+0.4203
	1.3	1.4	45145	0.0801	**45130**	**0.8513**	-15	+0.7712
			Average	0.4250		**0.6417**		+0.2167

Table 4.14: % Increase in Average Recall achieved by MOLR

Algorithm	Average Recall	Difference in Average Recall	% Difference in Recall compared to Single Objective algorithm
Logistic Regression	0.3625		
MOLR	0.5902	0.2277	62.81%
Naïve Bayes	0.2774		
MOLR	0.6039	0.3265	117.72%
Decison Tree	0.4346		
MOLR	0.6454	0.2108	48.51%
Random Forest	0.425		
MOLR	0.6417	0.2167	50.98%

logistic regression and Naïve Bayes. One of the reasons might be that, being an ensemble algorithm, random forest forms a model consisting of multiple decision trees. This reduces overfitting and enhances performance compared to single decision tree model.

Overall, there are very few cases when MOLR is achieving lesser effectiveness than single objective algorithms.

We summarize our findings in Table 4.14. The average recall of each of single objective prediction model and MOLR is reported in this table. And also the percentage of increase achieved by MOLR as compared to average recall value of single objective algorithm is reported. Overall MOLR achieved more than 48% increase in all four cases, with maximum of 117% increase in case of comparison with Naïve Bayes algorithm. This shows the dominance of multi-objective approach as compared to single objective algorithms. Single objective algorithms only optimize prediction error as their objective and they do not consider cost of prediction. MOLR models are trained to minimize LOC cost and maximize effectiveness as their objectives. This is one of the major reasons why MOLR outperformed single objective algorithms. MOLR is able to identify more defect prone classes than single objective algorithms at same or lesser LOC cost.

To confirm these findings statistically, we performed Wilcoxon two tailed paired test [32] for all four cases. p-values for all four types of experiments i.e., SOLR vs MOLR, Naïve Bayes vs MOLR, Decision Tree vs. MOLR and Random Forest vs. MOLR are 0.00000131, 0.00000000, 0.00000117 and 0.00000008 respectively. As p-values are significantly lower than threshold value 0.05, we can easily reject null hypothesis H_{02}. This confirms domination of MOLR over single objective algorithms in terms of LOC cost and recall.

4.7 Threats to Validity

This section discusses various threats to validity that may impact the analysis of the proposed approach and the experimental study presented here.

Threats to Construct Validity

The choice of recall as performance measure is widely adopted in previous studies. The choice of cost factor is based on the fact that it is more costly to fix defects during post-release phase as compared to performing QA activities on non-defective files in pre-release phase [13]. One can choose appropriate cost factor based on the project and organization. And also different values of cost factor may yield different results. For our second problem the choice of LOC cost is inspired from the fact that it is related to amount of time that will be spent to review or test the code. And the same measure has been used in [1, 108, 123]. But one can use other measures as well. One more threat to construct validity can be the choice of metrics and datasets. The CK metrics are one of the standard sets of metrics used for object oriented projects. We have considered datasets from widely used PROMISE repository, but we are not denying the fact that the datasets are prone to imperfection and incompleteness.

Threats to Internal Validity

One of the biggest threat is the choice of parameter settings for implementation. These parameters are chosen from experimentation and past research work [1, 108, 124, 127]. For the parameters chosen from experimentations, we have used spread as the evaluation criteria for measuring goodness of Pareto optimal solutions, as defined by Deb [128]. The choice of different evaluation criteria may lead to different parameter settings. The choice of best parameters may vary from one dataset to another. We chose a uniform set of parameters to compare the results at the same scale. To mitigate inherent randomness of GA we ran training process multiple times. For the problem M1 we ran NSGA-II 30 times to get the best model of MOLR which is used for testing purpose. For the problem M2 we ran and took coefficients corresponding to median Pareto front. The choice of median Pareto front corresponds to median spread value obtained among all 31 runs. The parameters for other algorithms are standard ones available in MATLAB. We have used logistic regression as fitness function to multi-objective genetic algorithm and four traditional machine learning algorithms for the comparison purpose. All of these algorithms have been used in many of the past works [41, 44, 103, 112, 113], but the

results may not hold true for other machine learning algorithms. Our aim was to compare cost-effectiveness of multi-objective algorithm with traditional algorithms for cross version defect prediction, rather than comparing different machine learning algorithms with each other.

Threats to Conclusion Validity

We have performed two tailed Wilcoxon paired test to statistically compare the difference in the performance measures obtained from MOLR and traditional single objective algorithms. Wilcoxon test is a non-parametric test, which does not make any assumptions about input value distributions. We have confirmed our findings at 5% significance level.

Threats to External Validity

We have experimented with 11 open source projects from the PROMISE repository having different versions. One may find different results with the projects developed in industry, where certain standards are followed. So the findings presented by us may or may not hold good for industrial software projects. We have used CK metrics as predictors in our study. The choice of different metrics may yield different results.

4.8 Contributions

In this work, we have formulated cross version defect prediction as multi-objective optimization problem with two distinct set of objective functions, and the same was solved using multi-objective genetic algorithm. We compared multi-objective logistic regression with four single objective algorithms for cross version defect prediction. We applied the proposed approach to 11 projects and a total of 30 train version-test version pairs. Our results indicate following benefits of multi-objective approach:

1. The multi-objective defect prediction model (M1) is able to identify more defects at the same or lesser misclassification cost incurred by all four single objective defect prediction models. And this observation holds good for four different values of cost factor 5, 10, 15, 20.

2. The multi-objective defect prediction model (M2) incurs the same or lesser LOC cost to achieve better recall as compared to all four single objective defect prediction models. This proves that M2 is able to identify more defect prone classes at the same or lesser LOC cost than the cost incurred with single objective algorithms.

In summary, multi-objective approach can yield better results for cross version defect prediction compared to traditional single objective approaches.

5 Probability Distribution of Defects and Object-oriented Metrics

5.1 Introduction

The well-known and widely applicable Pareto principle is verified to be holding good for bugs across files in software projects and the principle in this context means that eighty percent of bugs are contained in twenty percent of files [129] [130]. The more insightful research question is to find out the underlying probability distribution that models bugs across files in software projects. If the generic model for bug distribution across projects and domains is known, quality assurance activities like testing and review of future releases can be planned efficiently. An understanding of the underlying distribution will help in enhancing the applications of the Pareto and other related principles in managing quality assurance activities effectively. The generative process of bugs can also be well understood if the underlying probability distribution is known.

Studies in the past have proposed the Pareto distribution to model bugs in software systems. However, several other probability distributions such as the Weibull, Bounded Generalized Pareto, Double Pareto, Log Normal and Yule-Simon distributions have also been proposed and each of them has been evaluated for their fitness to model bugs in different studies. In Section 5.2, we investigate this problem further by making use of information theoretic (criterion based) approaches to model selection by which issues like over-fitting etc., that are prevalent in previous works, can be handled elegantly. In this study, we have made an attempt to answer the hypothesis that the software bugs follow a

particular distribution by comparing various distributions applied over a large number of projects using a recognized statistical framework.

In Section 5.3, we have made an attempt to answer the hypothesis that the software metrics follow a particular distribution by comparing various distributions applied over a large number of projects using a recognized statistical framework. The past studies have shown that the software projects frequently follow power law. Availability of more datasets and choice of better goodness-fit-measures motivated us to study further about the appropriate distribution to model metrics. We have used Nemenyi test and Friedman test to provide concrete results on whether a model is significantly outperforming other models. And thereby we eliminate the threats of human error and statistical inaccuracy.

5.2 Probability Distribution of Defects

In this work we seek to add to and improve the existing work regarding the choice of most appropriate distribution to model software bugs. We compare six models which have been proposed in the context of modelling bug distributions in current literature. The Bounded Generalised Pareto, Weibull and Pareto distributions are compared by Chin-Yu Huang et al. [131] [132]. In another study, the authors consider the Pareto and Weibull distributions for comparison [133] while the Yule-Simon, Double Pareto, Lognormal and Weibull distributions are compared yet in another study [134]. We discuss the intuition for considering these six models as relevant to our problem in the third section.

The contemporary work on choosing the most appropriate model for the underlying bug distribution relies on parameter estimation methods and goodness-of-fit statistics. For example, in the papers mentioned above, one to all of Maximum Likelihood, Least Squares and Method of Moments procedures are used for parameter estimation and models are compared based on goodness-of-fit statistics such as R^2, K-S tests and χ^2 tests etc. This approach is found to have some drawbacks from both, practical and theoretical standpoint, which we shall discuss in fourth section. We note that using goodness-of-fit statistics for model selection does not take into account model complexity.

In this work we seek to strengthen the model selection methodology and make it less vulnerable to threats such as over-fitting by making use of information criterion based approaches. We make use of three prevalent information criteria, namely the Bayesian Information criterion, Akaike Information Criterion & Hannan-Quinn Information Criterion to compare models. In general, the number of datasets used in previous works is not large enough to perform a quantitative summarization of results. In this work we use a number of datasets, large enough to warrant a quantitative method of results summarization. We use a method proposed by Demsar et al. [38] to summarize results which make our conclusions stronger and less prone to human error. In addition, BIC-based model selection allows us to quantify the relative fit of two models based on the difference between the BIC values for each of them. We show that this complements the approach specified so far and use the relative fit values to draw additional conclusions. Using these methods, we seek to compare models objectively and figure out model(s) that fits bug distribution well.

5.2.1 Related Work and Motivation

Many studies have shown the applicability of the Pareto principle to bugs in software systems. Fenton and Ohlsson [129] and Andersson and Runeson [130] studied the distribution of bugs in software systems and came to the conclusion that a small number of modules contains majority of faults and that the Pareto principle was applicable to software fault data. Daskalantonakis [135] showed that the Pareto principle was applicable to Motorola's software systems and further used Pareto analysis to identify the major causes of requirement changes. Ostrand and Weyuker's [136] findings also show that there is strong support for the applicability of Pareto analysis to model file level fault data.

In contrast, Zhang studied the distribution of both pre-release and post-release faults in the Eclipse software system and found that the Weibull distribution modelled the data better than a traditional Pareto distribution [133]. They analyse the package level data by fitting the Pareto and Weibull distributions to the data using nonlinear regression methods.They use R^2 and Standard Error of Estimates to measure the goodness-of-fit of the models.

In another study Chih-Song Kuo et al. proposed the Bounded Generalised Pareto distribution as a possible model for bug distributions in software systems [132]. They evaluate their model on Eclipse pre-release data [42] as well as data from three versions of Bugzilla. They computed parameters of the Pareto, Weibull and Bounded Generalised Pareto distribution using two methods, Maximum Likelihood Estimation and Least Squares Estimation. The resulting models are evaluated using the R^2, Standard Error of Estimate, Average Relative Percentage error and Bias measures. They conclude that both the Bounded Generalised Pareto and Weibull distribution model software fault data well.

Chin-Yu Huang et al. extended this study by (a) including an additional method of parameter estimation(Method of Moments Estimator) (b) performing a study over fault data collected from Ant, Thunderbird and Mozilla Firefox in addition to Bugzilla and (c) including more measures of goodness-of-fit [131]. In their extended study Chin-Yu Huang et al. reduced the magnitude of several internal threats to validity present in earlier work. They conclude that the Bounded Generalised Pareto distribution models bug data better than other two distributions.

Giulio Concas et al. study the bug distribution at the granularity of modules across six versions of eclipse [134]. They propose four models, namely, the Yule-Simon, Double Pareto, Lognormal and Weibull distributions for modelling bug distribution. For the Yule-Simon, Double Pareto and Lognormal models, the authors discuss associated generative models and their applicability to bug generation process. The distributions are fit to the data using least squares estimation and the goodness-of-fit is measured using the R^2 value. Giulio Concas et al. believe that the associated generative model of the Yule-Simon distribution is more appropriate to model bugs in general. The authors note that while the graphical evidence reflects that the Double Pareto model fits the data the best, the R^2 statistic indicates that the Yule-Simon model fits the data the best.

Tihana Galinac Grbac et al. compare the Pareto, Lognormal, Weibull, Double Pareto and Yule-Simon distributions to model bugs in a large scale proprietary software systems [137]. They fit models to the data using non linear regression methods and use R^2, and standard error of estimates to select the best distribution. The results show that Double Pareto model fits the data better than any other model.

The process adopted by contemporary researchers for selecting the most appropriate distribution is (a) finding point estimates of the parameters of the distributions and (b) comparing the resulting models by means of some goodness-of-fit measures. From a statistical point of view, there are arguments against this approach. First, we note that using goodness-of-fit measures for model selection can lead to significant over-fitting. Consider a situation where two models are compared, one simple and another much more complex. Here, even if the data was originally generated from the simpler model, the complex model might perform better in terms of goodness-of-fit statistics purely by virtue of its complexity. The second issue is that by making a point estimate of the parameters, before performing model comparison, we are actually comparing specific instances of models rather than models. And also model parameters have been chosen by optimizing some selection criteria, such as, maximum likelihood over the data which might indirectly lead to over-fitting of the model that is evaluated by making use of goodness-of-fit statistics. We believe that these issues are serious and hence we attempt to explore better ways of comparing models.

While the earliest work relies on fault data from relatively fewer software systems, we seek to generalize the results by using a large collection of datasets consisting of Eclipse dataset [42] and the datasets collected by Jureczko et al. [93]. In total, we use data from 62 versions of various open source and proprietary software systems. We elaborate further on the nature of the data in the following section.

The related work on this problem makes use of qualitative methods to summarize conclusions across multiple datasets, while goodness-of-fit measures are used to quantitatively compare the fitness of different models for a dataset. In addition, as the number of datasets grow as it does in our study, analysis using this approach becomes intractable. Here we use statistical methods like Nemenyi and Friedman tests to summarize results across the multiple datasets [5], [38]. The details of the methodology and its theoretical motivations are provided in the model selection section.

5.2.2 Probability Distributions and Datasets

Several previous works have shown the applicability of power law distributions for modeling software fault data. And also, generative models associated with some distributions have been shown to be applicable to bug generation process. Here, we choose the distributions that have been justified in literature so far from either of the approaches. In total, we will consider 6 models, details of which are mentioned below. We will also briefly discuss associated generative models. In our study we follow the same notation as used by Huang et.al. and Kuo et al. [131] [132]. And thus x is explained as the normalized proportion of cumulative software modules where the modules are ordered by decreasing number of faults. P(x) denotes cumulative density function and it represents the normalized cumulative number of faults induced by a given proportion of software modules.

- The Pareto distribution (PD) is a highly skewed and heavy tailed distribution based on the Pareto principle. The Probability Density Function (PDF) of the Pareto distribution in its simplest form is defined as

$$p\left(x\right) = \frac{\alpha k^{\alpha}}{x^{\alpha+1}} \tag{5.1}$$

  where $k <= x < \infty; \alpha, k > 0$

- The Weibull distribution (WD) has been proposed by several previous software reliability studies as a model for bug distributions. The PDF of two parameter general Weibull distributions is defined as

$$p(x) = \gamma x^{(\gamma-1)} \exp(-(x^{\gamma})) \qquad x \geq 0; \gamma > 0 \tag{5.2}$$

- The Bounded Generalized Pareto Distribution (BGPD) is a derivative of the Generalized Pareto family that has been formulated to have theoretical properties suitable to the problem of modeling bug distributions [131] [132]. The PDF of the Bounded Generalized Pareto Distribution is defined as

$$p(x) = \frac{1}{\beta}(1-x)^{((1/\beta)-1)} \tag{5.3}$$

where $\beta > 0, x\epsilon\,[0,1]$

- The Yule-Simon distribution (YS) was developed based on the preferential attachment model to explain the power law tails in some empirical distributions. It is a discrete probability distribution supported in the non-negative integers. The discrete Yule-Simon distribution is defined in terms of k which is the rank of the file when arranged in decreasing order of number of bugs. The PDF of the Yule-Simon distribution is defined as

$$p(k) = p_0 \frac{B(c+k,\alpha)}{B(c,\alpha)} \tag{5.4}$$

where $c > 0, \alpha > 0$, $0 < p_0 < 1$ is the probability of a module with no bugs and k is defined as the rank of the file, arranged in decreasing order of the number of bugs.

- The Lognormal distribution (LGNRM) is characterized by the fact that its natural logarithm is the normal distribution. It is a continuous distribution supported for $x > 0$. The PDF of the Log-normal distribution is

$$p(x) = \frac{1}{x\sqrt{2\pi\sigma^2}} \exp\Big[-\frac{(\ln x - \mu)^2}{2\sigma^2}\Big] \tag{5.5}$$

where $x > 0, \sigma > 0$ and μ is real.

- The double pareto distribution (DP) was designed such that it can closely match the body of Log-normal distribution and the tail of a Pareto distribution. The Double Pareto distribution is a continuous distribution supported on $0 < x \leq x_m$. The PDF of the Double Pareto distribution is

$$p(x) = \Big(\frac{\gamma}{t}\Big)\frac{[1+(x_m/t)^{-\beta}]^{\gamma/\beta}}{[1+(x/t)^{-\beta}]^{1+\gamma/\beta}}\Big(\frac{x}{t}\Big)^{-(1+\beta)} \tag{5.6}$$

where $0 < t < x_m, \beta > 0$ and $\gamma > 0$. Here the parameter t determines the crossover point where the behavior of the function changes from Log-normal body to power law tail.

The Pareto and Weibull distributions have been commonly used in modeling reliability of systems and were adopted to the bug distribution problem due to the fact that power law tails were observed in the data. The BGPD model was proposed to deal with several limitations of applying the Pareto model to bug distributions. And one of the major issues with Pareto distribution in this context is that the probability density function may take negative values if parameters of the distributions are estimated by standard methods like maximum likelihood, MoM, least squared etc. The BGPD is an alternate model derived from the Generalized Pareto Family such that it remains valid when used to model bug distributions and is additionally defined on the relevant domain of $[0, 1]$.

Here we note that in case of the other three distributions, namely the Yule-Simon, Double Pareto and Lognormal models, there is a generative model that backs up the distributions which may be applicable to the process of bug generation in software systems. In case of the Log-normal distribution, the generative process is that bugs introduced into a file are proportional to the number of bugs that are present in the previous version of the file, and each file has the same probability of being selected for an update. If the bug generation process follows this generative model, we can say that the resulting bug counts will be distributed according to the Log-normal distribution [138]. One major drawback of this model is that it cannot fit the number of bugs introduced into a file/module (or any other granularity at which we consider the distribution of bugs) with no bugs as yet.

The Double Pareto distribution is able to fit a power law tail while at the same time it is more flexible and able to fit the body better than pure power law distributions. The body section of a Double Pareto distribution behaves similar to a Lognormal distribution while having power law behavior in the tail. The Double Pareto distribution has been shown to be applicable to a wide variety of data showing power law behavior, such as, the distribution of file sizes in a file system. Several generative models have been proposed for the Double Pareto model such as Downey's multiplicative file size model and Mitzenmacher's Recursive forest model [139], [140]. The authors of these works also suggest that the generative models might be applicable to other areas such as software engineering and this might well be the case for bug generation.

The generative model of the Yule-Simon distribution is based on the preferential atta-

chment mechanism. This means that in the current version of a system of n files, all new files introduced in this version are initialized with bug value h_0. And files are selected based on probability proportional to their current bug count and the number of bugs is incremented for the selected file. If bugs in the current version are likely to follow this procedure, it can be shown that the resulting empirical distribution will have the form of a Yule-Simon distribution [141]. The Yule-Simon model is thus a discrete distribution and models both the inclusion of new files into the system and their bug count values effectively. We also see that Yule-Simon model's generative process is the only one that can deal effectively with files with no bugs. The interpretation and validation of these generative models using fault data have been partly done in the previous works [134].

In this work, we use fault data from 34 releases of 12 different open source software systems and 28 releases of 6 different proprietary software systems. This includes fault data from 3 versions of Eclipse [42] and 11 different open source software systems [93]. The Eclipse fault data was collected by the University of Saarland while the fault data from the remaining open source and proprietary software projects was collected by Jureczko et al. In both cases, the authors looked for keywords in the metadata of commits to identify bug fixes. The bug count of every file/class was set according to the number of bug fixes associated with that file/class.

Of the six proprietary softwares projects under consideration, five software systems are built for companies in the insurance domain and the sixth is a standard tool that supports quality assurances in software development. All six were built by the same company. The number of files and bugs in each version of projects - prop1, prrop2, prop3, prop4, prop43 and prop5 are reported in Table 5.1 and Table 5.2. The open source data was collected from Eclipse (2.0 , 2.1 , 3.0), Apache Ant (1.6 - 1.7), Apache Camel (1.2,1.4,1.6), Apache Ivy (1.1), JEdit (3.2 - 4.2), Apache Log4j (1.0 - 1.2), Apache Lucene (2.0 - 2.4), Apache POI (1.5 - 3.0), Apache Tomcat (6.0), Apache Velocity (1.4 - 1.6.1), Apache Xalan-Java (2.4.0 - 2.7.0), Apache Xerces (1.1.0 - 1.4.4). The number of files and defects for each version of these projects are reported in Table 5.3 and Table 5.4. These projects are briefly described below to provide some additional insight into the kind of data studied.

- **Eclipse** is an integrated development environment (IDE). It contains a base

Table 5.1: Number of Files and Bugs - Proprietary Software Projects[1]

Project	prop43			prop5					
Version	2	5	6	1	2	3	4	5	6
Files	1740	2172	2265	3445	2863	3260	3514	3815	3509
Bugs	52	126	208	748	722	487	345	743	2427

workspace and an extensible plug-in system for customising the environment. Written mostly in Java, Eclipse can be used to develop applications. By means of various plug-ins, Eclipse may also be used to develop applications in other programming languages. Development environments include Eclipse Java development tools (JDT) for Java and Scala, Eclipse CDT for C/C++ and Eclipse PDT for PHP, among others.

- **Apache Ant** is a Java library and command-line tool that can be used to pilot any type of process which can be described in terms of targets and tasks.

  The main known usage of Ant is to facilitate the build of Java applications. Ant supplies a number of built-in tasks allowing to compile, assemble, test and run Java applications. Ant can also be used effectively to build non Java applications, for instance C or C++.

- **Apache Camel** is an open-source integration framework based on known Enterprise Integration Patterns. Camel can be used to define routing and mediation rules in a variety of domain-specific languages, including a Java-based Fluent API, Spring or Blueprint XML Configuration files, and a Scala DSL.

- **Apache Ivy** is a popular dependency manager focusing on flexibility and simplicity. An external XML file defines project dependencies and lists the resources necessary to build a project. Ivy resolves and downloads resources from an artifact repository which is either a private repository or one publicly available on the Internet.

Table 5.2: Number of Files and Bugs - Proprietary Software Projects[2]

Project	prop1						prop2						prop3				prop4		
Version	1	2	3	4	5	6	1	2	3	4	5	7	1	2	3	4	1	2	3
Files	3619	3541	3692	4081	4455	3670	1864	2403	1931	2025	2372	2472	1709	2330	2156	2440	2906	2802	2865
Bugs	388	546	107	563	264	2234	234	111	140	729	424	534	253	401	131	560	269	1308	353

Table 5.3: Number of Files and Bugs - Open Source Software Projects[1]

Project	Eclipse			Log4j			Lucene			Tomcat		Velocity		Xalan			
Version	2.0	2.1	3.0	1.0	1.1	1.2	2.0	2.2	2.4	2.0	1.4	1.5	1.6	2.4	2.5	2.6	2.7
Files	6729	7888	10593	135	109	205	288	381	536	1162	224	246	261	862	945	1170	1194
Bugs	1692	1182	2679	61	86	498	268	414	632	114	210	331	190	156	531	625	1213

- **jEdit** is a mature programmer's text editor with hundreds (counting the time developing plug-ins) of person-years of development behind it. It was originally developed as a proprietary software and its development was transferred to the open source community in 2006.

- **Apache log4j** is a logging library for Java. Apache log4j was originally written by a single developer and has since been developed by a team of Apache Software Foundation.

- **Apache Lucene** is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search.

- **Apache POI** was developed to create and maintain Java APIs for manipulating various file formats based on Office Open XML standards (OOXML) and Microsoft's OLE 2 Compound Document format (OLE2). In short, to facilitate reading and writing MS Excel files using Java. In addition, it facilitates reading and writing MS Word and MS PowerPoint files using Java.

- **Apache Tomcat** is an open-source web server and servlet container developed by the Apache Software Foundation (ASF). Tomcat implements several Java EE specifications including Java Servlet, JavaServer Pages (JSP), Java EL, and WebSocket, and provides a "pure Java" HTTP web server environment for Java code to run in.

Table 5.4: Number of Files and Bugs - Open Source Software Projects[2]

Project	Ivy	Jedit				POI			ANT		Camel			Xerces			
Version	1.1	3.2	4.0	4.1	4.2	1.5	2.5	3.0	1.6	1.7	1.2	1.4	1.6	1.2	1.3	1.4	init
Files	111	272	306	312	367	237	385	531	498	1066	765	1122	1252	440	453	588	162
Bugs	233	382	226	217	106	342	496	500	170	338	522	335	500	115	193	1596	167

- **Apache Velocity** is a Java-based template engine that provides a template language to reference objects defined in Java code. It aims to ensure clean separation between the presentation tier and the business tiers in a Web application.

- **Apache Xalan** was originally created by IBM under the name LotusXSL and implements the XSLT 1.0 XML transformation language and the XPath 1.0 language. The Xalan XSLT processor is available for both the Java and C++ programming languages of which we will be using the data from the Java version.

- **Apache Xerces** is Apache's collection of software libraries for parsing, validating, serialising and manipulating XML. The library implements a number of standard APIs for XML parsing, including DOM, SAX and SAX2.

5.2.3 Model Selection

5.2.3.1 Comparing Models across Multiple Datasets

In our experiments, we compare six different models and perform our tests on multiple software systems. While we discuss information criterion based approaches for model selection in the following subsections (4.2 to 4.5), we also need statistical methods to summarize our results across multiple datasets and six different models. We discuss the relevant statistical methods for summarization of results in this subsection.

Demsar has addressed the problem of ranking classifiers across various datasets [38] which have also been reviewed and recommended in the context of software fault prediction [5]. These recommendations are summarized below and will be used in this work to compare multiple models across datasets.

The null hypothesis being tested for our problem is that all models are equally good and there is no significant difference between them. One of the common methods to solve this problem is through testing the significance of the difference between mean accuracies (or alternate measures) across datasets. This approach relies on analysis of variance (ANOVA)

and hence makes assumptions such as 1) Performance differences between models are distributed normally, 2) All classifiers exhibit the same variance in predictive performance over all data sets (homogeneity of variance), and 3) Variance in performance differences across two classifiers is identical to all possible pairs of classifiers (sphericity assumption). Violation of these assumptions especially sphericity assumptions, has been shown to be detrimental to the performance of ANOVA and consequently subsequent post hoc tests are recommended [142]. Due to this, Demsar specifically discourages the use of ANOVA and instead relies on the Friedman test which is based on ranked performance and makes less restrictive assumptions.

Friedman's test is based on ranked performance of the competing classifiers rather than actual performance estimates which make it less susceptible to outliers. Classifiers are ranked according to their performance for each dataset and their mean ranks are computed across all datasets. Friedman test statistic is defined as

$$\chi_F^2 = \frac{12K}{L(L+1)}\left[\sum_{i=1}^{L} AR_i^2 - \frac{L(L+1)^2}{4}\right] \tag{5.7}$$

Where L represents the overall number of models, K denotes the number of datasets and AR_i represents the mean rank of model i over the datasets. And χ_F^2 is Chi-Squared distributed with $L-1$ degrees of freedom. A suitable Pearson's Chi-Squared test can be done to reject the null hypothesis at any desired level of significance. If the null hypothesis is rejected, it can be concluded that there is significant difference among the models being considered.

The post hoc test recommended by Demsar to establish discrimination among models, once it has been established that there is a significant difference between models, is the Nemenyi test. The Nemenyi test for any pair of models tests the null hypothesis that their mean ranks are equal, and the null hypothesis can be rejected if the difference between their mean ranks exceeds critical difference, which is:

$$CD = q_{a,\infty,L}\sqrt{\frac{L(L+1)}{12K}} \tag{5.8}$$

where $q_{a,\infty,L}$ is the studentised range statistic which is widely tabulated.

The performance measure, used to rank the models in order to perform the Friedman and Nemenyi tests, will be discussed in next three subsections.

5.2.3.2 Information criteria for model selection; Why not GoF statistics?

As we have noted earlier, most of the contemporary work on comparing distributions for modeling bugs in software projects make use of the Goodness of Fit (GoF) statistics such as R^2 , K-S test, Chi-square Tests etc. The parameters of the chosen models have been determined by procedures like MLE , least of squares, MoM. We elaborate on arguments that are prevalent on limitations of these approaches in this section and motivate information criterion based measures. It should be noted that model complexity is not being considered in the above mentioned GoF statistics. A complex model is able to fit random variations in the data and maximize the goodness-of-fit statistic. This means that even though data might be generated from a simpler model, the complex model may be wrongly selected. Hence any conclusion on the best fitting distributions can be misleading if the above mentioned goodness-of-fit statistic is used to compare models.

Another subtle issue is that the parameter estimation method and the goodness-of-fit statistic might not be suitable to each other, and lead to a biased result overall. Recently Crawley et al. have shown that any method, involving optimization of a selection criterion over a finite sample is prone to over-fitting [143]. They also show that the effects of such over-fitting are significant and might mislead a model selection algorithm. Hence, if we choose a parameter estimation method like MLE and a goodness-of-fit measure like R^2 to evaluate the resulting model, we cannot guarantee that the goodness-of-fit statistic is not biased. Overall goodness-of-fit statistics are complex and often involve controversial issues causing some researchers to question their validity [144]. These issues are not present when using an information criterion based approach which we will explain further in the following sections.

We note that in earlier works, these issues are discussed as threats to internal validity and Chin-Yu Huang et al. attempt to combat some of these by including three methods of parameter estimation and ten goodness-of-fit measures [131]. However, multiple para-

meter estimation methods and goodness-of-fit statistics does not address the over-fitting problems. Thus it is unclear how their method addresses the issues highlighted above. Further, they resort to qualitative means to summarize the results across the multiple dimensions of variations i.e. the goodness-of-fit statistics and methods of parameter estimation.

The Akaike, Bayesian and Hannan Quinn information criteria penalize the model for complexity in terms of the number of parameters. The more parameters a model has more likely it is to over-fit, and this is reflected in the increasing penalty term that seeks to eliminate this overfitting. We illustrate the origin and meaning of these information criteria in brief in the following sections. We note that all three of these information criteria represent relative fit of the models and do not provide any information on the absolute fit of the models. Hence, after computing the information criteria we perform the Nemenyi and Friedman tests to check whether the null hypothesis holds good or not. As the information criteria cannot be used to judge the absolute fit of the models, we have extensively justified the selection of models to be compared in Section 3. We also note that these models have shown to produce very high absolute goodness-of-fit measures in many previous works.

5.2.3.3 Akaike Information Criterion

The Akaike Information Criterion (AIC) was derived as a model selection measure based on minimum information loss concepts from information theory [145], [146]. We briefly discuss the original arguments of AIC. We have omitted all mathematical rigor to keep the it brief and to refer the interested reader to Akaike's original papers [145], [146] for further information. The task of measuring the fit of a model can be accomplished by measuring its discrepancy from the unknown true distribution. To accomplish this Akaike makes use of the Kullback-Leibler Distance which is an information theoretic measure of the directed distance between two probability distributions. He then proceeds to show that in the resulting expression, all models will contain a constant term dependent on the unknown true distribution. As all models have this constant term, it can be omitted for model comparison purposes and the left-over expression can be regarded as the relative

distance of the model under consideration from the underlying true distributions. This is an ideal measure for model selection, provided, it can be calculated. To this end Akaike shows that negative log likelihood is a biased estimator of this relative distance criterion and that under certain conditions this bias can be approximated by the number of free parameters, k of the model. This leads to the expression for the AIC as

$$AIC = -2\log(P(D/\theta_D)) + 2k \tag{5.9}$$

where θ_D represents the maximum likelihood estimate of parameters and k represents the number of parameters in the model. Here, $P(D/\theta_D)$ is the likelihood of the data conditioned on the parameters θ_D. Here on we use the notation $P(A/B)$ to denote the likelihood of A conditioned on B. We also note that the first term gives the goodness-of-fit of the model and the second term penalizes the model for complexity in terms of the number of free parameters, consistent with the intuitive arguments for model parsimony we have made.

5.2.3.4 Bayesian Information Criterion

The Bayesian Information Criterion (BIC) was first derived by Schwartz et al. as an unbiased model selection criterion [147]. We note that BIC can be viewed as an approximation of the Bayesian model selection approach, using a flat non informative prior [148]. Here we briefly explain this approach. The Bayesian approach to model selection is based on the posterior odds ratio $P(M_1|D)/P(M_2|D)$ which is equal to the Bayes factor $P(D|M_1)/P(D|M_2)$ when the both the models are considered equiprobable or the prior odds ratio is one [149]. Here we have considered that M_1 & M_2 are two competing model families and D represents the data observed. Considering the two model families to be parametrized by some parameter vectors θ_1 & θ_2, we have Bayes factor as

$$\frac{P(M_1|D)}{P(M_2|D)} = \frac{\int P(D|M_1,\theta_1)P(\theta_1|M_1)d\theta_1}{\int P(D|M_2,\theta_2)P(\theta_2|M_2)d\theta_2} \tag{5.10}$$

As can be seen from the expression for the Bayes factor, the computation of the marginal

likelihood(or model evidence) involves inclusion of a prior distribution over the parameters $P(\theta|M_k)$.

However, in a model selection scenario where it would be desirable to make each member of the model family equally like a priori, an uninformative flat prior would be a good choice. The BIC value for a model is the log of the marginal likelihood, and is derived by using a flat prior and by approximating the model evidence integral using the Laplace approximation. The resulting expression is

$$BIC = -2\log(P(D/\theta_D)) + k\log n \tag{5.11}$$

where θ_D represents maximum likelihood estimate of the parameters, k represents number of parameters in the model and n represents number of observations. Using this approximation, the log of bayes factor for two models is simply the difference of their BIC values. The interpretation of the strength of evidence based on the bayes factor values is a well studied problem and a table was proposed by Harold Jeffreys to do the same [149], [150]. Based on this, a table to interpret the BIC differences as strength of evidence for a model was introduced by Rafetry [151].

The hypothesis tests we use are dependent on the ranked performance of models and while this has several advantages, it also abstracts away the information contained in the magnitude of BIC values. We note that this information could be used to further strengthen our conclusions based on Nemenyi and Friedman tests. To this end we make use of BIC difference tables mentioned earlier to calculate the strength of evidence for models.

5.2.3.5 Hannan-Quinn Information Criterion

The Hannan-Quinn Information Criterion (HQIC) was first derived while exploring the model selection problem relating to the determination of the order of auto-regression by Hannan and Quinn [152]. However HQIC is applicable as a general model selection criterion and it has become one of the most cited information criterion for model selection.

HQIC is defined as

$$HQIC = -2\log(P(D/\theta_D)) + 2k\log\log n \tag{5.12}$$

where θ_D represents maximum likelihood estimate of the parameters, k represents number of parameters in the model and n represents number of observations. Once we calculate the HQIC values for models, we rank them using these values and perform hypothesis tests similar to the AIC and BIC cases.

5.2.4 Experiments and Results

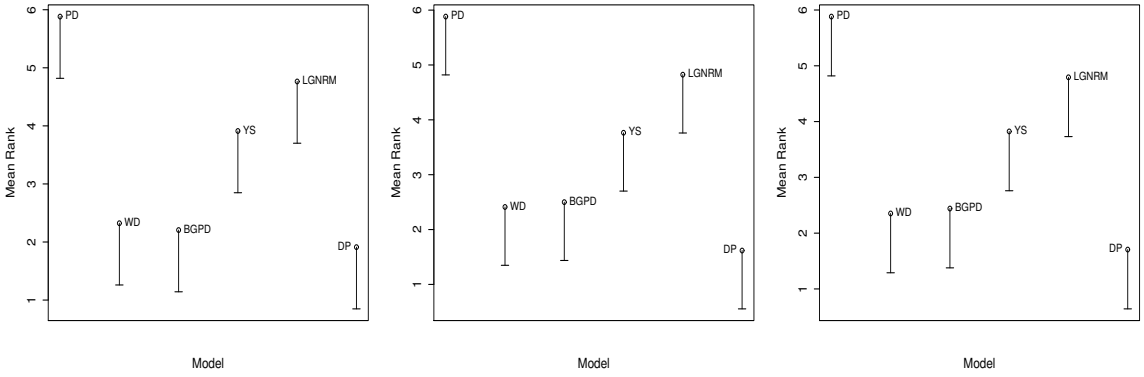
5.2.4.1 Open Source Software

The experiments and results of open source projects using each of the information criteria are discussed in this section. For each version, we rank six models as per an information criterion metric, say BIC. We performed Friedman's test to reject or accept the null hypothesis. And null hypothesis is that there is no significant difference among six models. Our experiments conclude that null hypothesis can be rejected at 0.01 level of significance.

As there is a significant difference between the models, we perform the post hoc test i.e the Nemenyi test. The Nemenyi test results are plotted on a graph with the Mean rank indicated on the y-axis and the models on the x-axis. The critical difference interval for any model is denoted on the graph as an extended downward line with length CD from the mean rank. The Nemenyi plot drawn for BIC, AIC and HQIC are shown in Figure 5.1. If any model M_1 has a mean rank lesser than the lower bound of the critical difference interval of another model M_2, we can say that M_1 outperforms M_2 with statistical significance. For example in Figure 5.2a we can see that the Double Pareto model outperforms the Pareto model with statistical significance.

From the Nemenyi diagram generated using the BIC values, we can see that the top three ranked models are the Weibull, Bounded Generalized Pareto and Double Pareto distributions. While the Bounded Generalized Pareto model performs slightly better than the Weibull Model as indicated by its lower mean rank, it fails to do so with statistical signifi-

Figure 5.1: Nemenyi plot comparing models using (a) BIC, (b) AIC and (c) HQIC to perform the ranking for open source data



(a) BIC **(b)** AIC **(c)** HQIC

cance. Similar observations can be made for the Double Pareto and Bounded Generalized Pareto models. Overall, the top three ranked models fail to outperform one another with statistical significance and are very close in terms of mean ranks for open source fault data. However all the top three models outperform the Yule-Simon, Pareto and Lognormal models with statistical significance. Pareto distribution is the worst performing distribution and is outperformed by all the other distributions.

For the remaining metrics AIC and HQIC, we performed friedman's test and found that null hypothesis is rejected. Consequently we performed post hoc tests, and Nemenyi plot for each of these metric is shown in 5.2b and 5.2c

The AIC criterion is more lenient than BIC in penalizing the complexity of the model and this can be observed from the Nemenyi plots. Here again we observe that the top three ranked models are the Double Pareto, Weibull and Bounded Generalized Pareto models. The mean ranks of the top three models are slightly shifted. We note that this shift can be explained by the change in complexity penalty between AIC and BIC. First, the Double Pareto models mean rank shifts downwards as the complexity penalty for the Double Pareto is lower than the corresponding BIC case. This means that for projects, where Double Pareto model was outperformed by another model with a margin smaller than this change in penalty, the Double Pareto distribution becomes the best performing model in AIC case. However we observe that this shift is not enough to statistically

Table 5.5: Open Source Software BIC difference and interpretation tables

Software	DP vs WD			DP vs BGPD			WD vs BGPD		
	ΔBIC	**Model**	**Evidence**	**ΔBIC**	**Model**	**Evidence**	**ΔBIC**	**Model**	**Evidence**
eclipse-2.0	464.62	DP	V Strong	489.42	DP	V Strong	24.81	WD	V Strong
eclipse-2.1	385.35	DP	V Strong	361.40	DP	V Strong	-23.95	BGPD	V Strong
eclipse-3.0	668.61	DP	V Strong	687.43	DP	V Strong	18.81	WD	V Strong
log4j-1.0	2.33	DP	Positive	-3.61	BGPD	Positive	-5.93	BGPD	Positive
log4j-1.1	-9.43	WD	Strong	-16.51	BGPD	V Strong	-7.07	BGPD	Strong
log4j-1.2	64.67	DP	V Strong	17.65	DP	V Strong	-47.02	BGPD	V Strong
lucene-2.0	-23.38	WD	V Strong	-21.86	BGPD	V Strong	1.52	WD	Weak
lucene-2.2	38.23	DP	V Strong	65.26	DP	V Strong	27.03	WD	V Strong
lucene-2.4	40.52	DP	V Strong	30.84	DP	V Strong	-9.68	BGPD	Strong
tomcat	20.45	DP	V Strong	14.74	DP	V Strong	-5.71	BGPD	Positive
velocity-1.4	-35.44	WD	V Strong	-57.05	BGPD	V Strong	-21.61	BGPD	V Strong
velocity-1.5	13.03	DP	V Strong	-0.20	BGPD	Weak	-13.22	BGPD	V Strong
velocity-1.6	5.47	DP	Positive	-1.90	BGPD	Weak	-7.37	BGPD	Strong
xalan-2.4	35.59	DP	V Strong	28.34	DP	V Strong	-7.25	BGPD	Strong
xalan-2.5	169.24	DP	V Strong	131.59	DP	V Strong	-37.65	BGPD	V Strong
xalan-2.6	153.28	DP	V Strong	118.64	DP	V Strong	-34.64	BGPD	V Strong
xalan-2.7	304.29	DP	V Strong	187.67	DP	V Strong	-116.61	BGPD	V Strong
ivy-1.1	-17.71	WD	V Strong	1.20	DP	Weak	18.91	WD	V Strong
jedit-3.2	-25.08	WD	V Strong	-1.42	BGPD	Weak	23.66	WD	V Strong
jedit-4.0	-35.42	WD	V Strong	-27.40	BGPD	V Strong	8.02	WD	Strong
jedit-4.1	-12.54	WD	V Strong	-16.92	BGPD	V Strong	-4.38	BGPD	Positive
jedit-4.2	-1.81	WD	Weak	-7.51	BGPD	Strong	-5.69	BGPD	Positive
poi-1.5	-15.20	WD	V Strong	-2.63	BGPD	Positive	12.57	WD	V Strong
poi-2.5	111.18	DP	V Strong	59.37	DP	V Strong	-51.81	BGPD	V Strong
poi-3.0	114.35	DP	V Strong	156.49	DP	V Strong	42.13	WD	V Strong
ant-1.6	14.35	DP	V Strong	6.23	DP	Strong	-8.12	BGPD	Strong
ant-1.7	27.98	DP	V Strong	18.25	DP	V Strong	-9.73	BGPD	Strong
camel-1.2	57.56	DP	V Strong	50.84	DP	V Strong	-6.72	BGPD	Strong
camel-1.4	30.26	DP	V Strong	25.53	DP	V Strong	-4.73	BGPD	Positive
camel-1.6	22.80	DP	V Strong	44.85	DP	V Strong	22.05	WD	V Strong
xerces-1.2	10.67	DP	V Strong	14.03	DP	V Strong	3.36	WD	Positive
xerces-1.3	13.58	DP	V Strong	22.04	DP	V Strong	8.47	WD	Strong
xerces-1.4	-22.79	WD	V Strong	386.00	DP	V Strong	408.78	WD	V Strong
xerces-init	31.38	DP	V Strong	15.58	DP	V Strong	-15.80	BGPD	V Strong

outperform the Weibull and Bounded Generalized Pareto models. The Weibull models mean rank moves down and it replaces the Bounded Generalized Pareto Model as the second best performing model. However, change in mean rank is not big enough to produce a statistically significant difference between the two models.

The HQIC criterion falls somewhere in-between the AIC and BIC criteria in terms of penalty for model complexity. Consistent with this, HQIC based mean ranks of models are shifted with respect to BIC based mean ranks. And reason for shift is the same as discussed in AIC case. The observations for all models remain consistent with BIC-based observations as the change in mean ranks is not large enough to induce any statistically significant changes.

As indicated in Section 5.2.3, we also seek to find the relative fit of the models. To this end, we have discussed Bayes factors using BIC values and mentioned its applicability to model selection. Using BIC values, the relative fit criterion as defined in the Section 5.2.3 is computed for all projects. We restrict ourselves to the top three ranked models for this analysis to understand the relative performance of the these models. The relative fit criterion values along with the selected model and the associated strength of evidence are recorded for the top three ranked models in Table 5.5. The strength of the evidence based on the relative fit criterion is derived from the tables proposed by Raftery et al. [151]. From Table 5.5 we can see that the Double Pareto distribution is selected 22 times with very strong evidence and twice with positive evidence when pitted against the Weibull distribution. The Weibull model is selected ten times out of which the evidence is weak once, strong once, very strong eight times. We observe that the Double Pareto distribution is selected more number of times with very strong evidence. However even though the Weibull distribution is selected comparatively fewer number of times, it is also selected with very strong evidence in most of the cases. Given the number of cases where this is observed, there is an ambiguity between the Weibull and Double Pareto models as selecting either of the two leads to significant risk of inferior fit to the data. These conclusions are in line with the Nemenyi test which fails to find a distribution that is significantly better than the other distribution.

Comparing BGPD and Weibull distributions, we see that in 21 cases, BGPD is a better fit

while in 13 cases Weibull distribution fits the data better. In the cases where the Weibull distribution fits the data, the evidence is very strongly in favor of the Weibull distribution 9 times while it is strong twice, positive once and weak in one case. Out of cases where the BGPD distribution is favored, evidence in favor of BGPD is very strong 9 times, strong seven times and positive five times. All arguments indicating ambiguity in the previous case are applicable here also. We, therefore conclude that it is not possible to choose one of the distributions over the other. We again find that Nemenyi diagram supports our conclusions and indicates that there is no statistically significant difference between the two distributions.

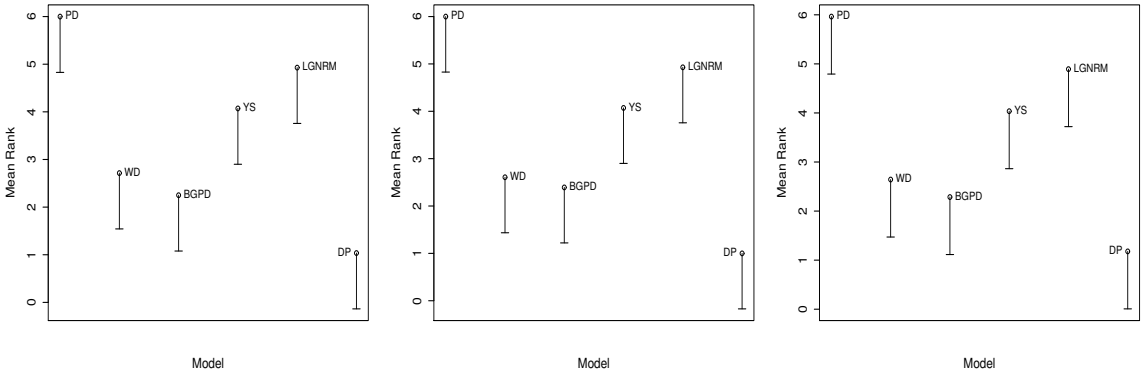
Between the BGPD and DP models, in 23 cases DP model is preferred where evidence is very strong 14 times, strong 6 times and positive thrice. Out of the 11 times the BGPD model is preferred, evidence is very strong 4 times, strong thrice, positive thrice and weak once. Again we put forth the same arguments made previously to conclude that we cannot make a concrete conclusion in favor of either distribution. This is also confirmed by the Nemenyi plot where there is no statistically significant difference between mean ranks of two distributions.

5.2.4.2 Proprietary Software

We repeat the same experiments as explained in the previous section on fault data of proprietary software systems [93] and the results are reported in a similar form. Here also, we found that null hypothesis is rejected at 0.01 level of significance and hence conclude that there is a significant difference between the models. Consequently we proceed to perform the post hoc Nemenyi test.

We observe that the top three ranked models are the Weibull, Pareto and the Double Pareto models using all three information criteria. And this observation is in line with the results of open source projects. However, here the Double Pareto model outperforms both the WD and the BGPD models with statistical significance and is the best performing model with all three information criteria. The BGPD model performs marginally better than the WD model in terms of average rank but it fails to pass the test for statistically

Figure 5.2: Nemenyi plot comparing models using (a) BIC, (b) AIC and (c) HQIC to perform the ranking for proprietary data



(a) BIC **(b)** AIC **(c)** HQIC

significant difference in all three cases. Consistent with the earlier results, the Yule-Simon, Lognormal and Pareto models are ranked, fourth, fifth and sixth respectively. The YS, PD and Lognormal models are outperformed with significance by the top three models using all three information criteria. Overall, we can see that all three information criteria produce similar results using the Nemenyi diagram, and the Double Pareto model performs the best consistently.

The relative fit criterion values along with the selected model and the associated strength of evidence are recorded in Table 5.6. We can see that the DP model is selected unanimously when pitted against the WD model. In all but one case, the evidence for DP model is very strong. Comparing DP and BGPD models, we can see that DP model is selected in all but one case. In case where the BGPD model is selected, the evidence is positive whereas in cases where the DP model is selected, the evidence is 'very strong' except once where it is strong. From these values we can conclude that by selecting DP model over BGPD and WD models, we do not risk a significant loss of fit. This observation is in agreement with the Nemenyi diagram which indicates that DP model is significantly better than WD and BGPD.

Comparing the WD and BGPD models from Table 5.6, we can see that BGPD model was selected 20 times while the WD model was preferred 8 times. Out of the 20 times BGPD model was selected, the evidence is 'strong' 6 times, 'very strong' thrice, 'positive'

Table 5.6: Proprietary Software BIC difference and interpretation tables

Software	DP vs WD			DP vs BGPD			WD vs BGPD		
	ΔBIC	**Model**	**Evidence**	**ΔBIC**	**Model**	**Evidence**	**ΔBIC**	**Model**	**Evidence**
P1-ver1	78.79	DP	V Strong	79.61	DP	V Strong	0.82	WD	Weak
P1-ver2	126.77	DP	V Strong	127.91	DP	V Strong	1.13	WD	Weak
P1-ver3	25.24	DP	V Strong	23.09	DP	V Strong	-2.15	BGPD	Positive
P1-ver4	170.73	DP	V Strong	164.4	DP	V Strong	-6.33	BGPD	Strong
P1-ver5	37.61	DP	V Strong	33.52	DP	V Strong	-4.09	BGPD	Positive
P1-ver6	604.65	DP	V Strong	680.09	DP	V Strong	75.43	WD	V Strong
P2-ver1	43.26	DP	V Strong	36.49	DP	V Strong	-6.77	BGPD	Strong
P2-ver2	23.36	DP	V Strong	17.93	DP	V Strong	-5.43	BGPD	Positive
P2-ver3	29.98	DP	V Strong	25.03	DP	V Strong	-4.95	BGPD	Positive
P2-ver4	298.32	DP	V Strong	251.76	DP	V Strong	-46.56	BGPD	V Strong
P2-ver5	66.5	DP	V Strong	61.18	DP	V Strong	-5.32	BGPD	Positive
P2-ver7	135.99	DP	V Strong	138.29	DP	V Strong	2.30	WD	Positive
P3-ver1	63.54	DP	V Strong	55.1	DP	V Strong	-8.44	BGPD	Strong
P3-ver2	59.5	DP	V Strong	50.44	DP	V Strong	-9.06	BGPD	Strong
P3-ver3	14.05	DP	V Strong	8.97	DP	Strong	-5.08	BGPD	Positive
P3-ver4	182.28	DP	V Strong	191.57	DP	V Strong	9.29	WD	Strong
P4-ver1	64.47	DP	V Strong	65.39	DP	V Strong	0.91	WD	Weak
P4-ver2	415.71	DP	V Strong	347.72	DP	V Strong	-67.99	BGPD	V Strong
P4-ver3	82.34	DP	V Strong	79.89	DP	V Strong	-2.45	BGPD	Positive
P43-ver2	2.27	DP	Positive	-2.03	BGPD	Positive	-4.29	BGPD	Positive
P43-ver5	16.09	DP	V Strong	14.55	DP	V Strong	-1.55	BGPD	Weak
P43-ver6	43.35	DP	V Strong	36.39	DP	V Strong	-6.96	BGPD	Strong
P5-ver1	171.87	DP	V Strong	177.98	DP	V Strong	6.11	WD	Strong
P5-ver2	92.64	DP	V Strong	80.69	DP	V Strong	-11.96	BGPD	V Strong
P5-ver3	67.86	DP	V Strong	58.27	DP	V Strong	-9.59	BGPD	Strong
P5-ver4	108.65	DP	V Strong	100.14	DP	V Strong	-8.51	BGPD	Strong
P5-ver5	191.84	DP	V Strong	187.13	DP	V Strong	-4.72	BGPD	Positive
P5-ver6	204.71	DP	V Strong	363.78	DP	V Strong	159.07	WD	V Strong

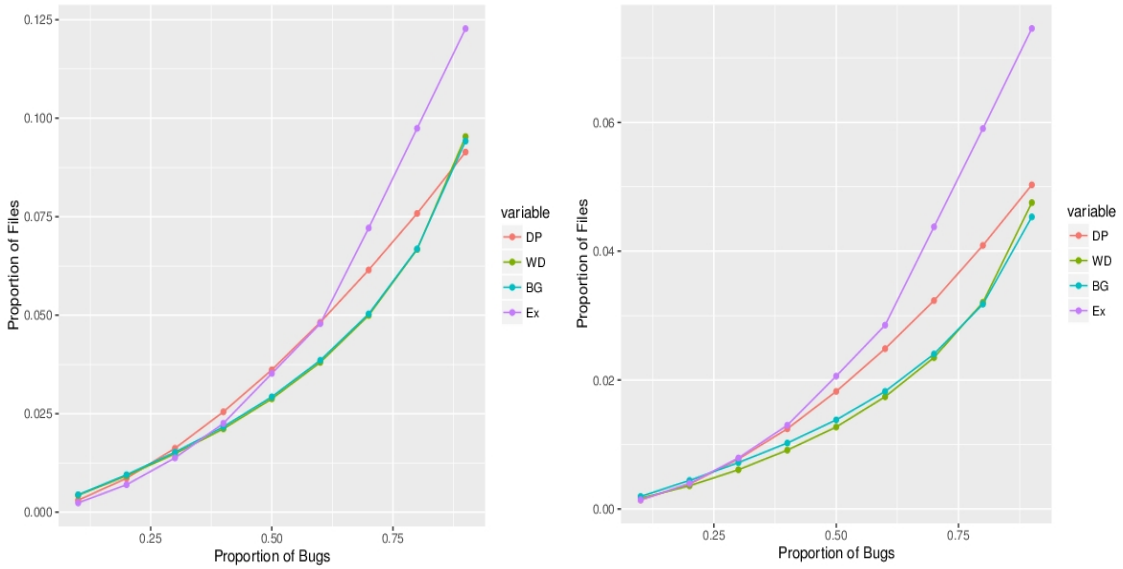
8 times and 'weak' once. Out of the 8 times WD model was selected, the evidence is 'very strong' twice, 'strong' twice, 'positive' once and 'weak' thrice. Here we can see that even though BGPD model was selected majority of the times, the evidence for the model is not strong enough to get to any conclusions. This means that the WD model might be able to provide a comparable fit in many of the cases where the BGPD model was selected. A similar observation can be made about the instances where WD model was selected. This indicates that the models cannot be separated clearly as they are able to provide very similar fits to the data under consideration. This is consistent with the Nemenyi diagram which also says that these models do not outperform each other with statistical significance.

It is always interesting and practically very useful to know the percentage of bugs induced by a portion of software modules. This helps to plan quality assurance activities like testing and code review better. We demonstrate two cases where the percentage of bugs can be estimated better by making use of the best model proposed by this study i.e., Double Pareto Distribution. Using the top three models in our study, we estimate percentage of bugs for various proportion of software files for Eclipse 2.1 and Prop1. These are then compared with the actual values. For these two projects, the estimated percentage of bugs by top three models and the actual percentage of bugs for several proportion of files is shown in Figure 5.3. From these figures, it can be observed that the Double Pareto Model seems to fit better and provide a better estimate in both the projects. The difference seems to be more pronounced in case of Prop1.

5.2.5 Contributions

We summarize our work, highlight the contributions and outline our conclusions. Several papers have studied the distribution of bugs in software systems and proposed various distributions to model them. The previous studies use goodness-of-fit statistics to compare models and results are summarized across datasets by qualitative means. We note that using goodness-of-fit statistics for model selection has been shown to be vulnerable to threats, such as, over-fitting and we attempt to tackle these issues by using an information criteria based approach to model selection. We make use of three prevalent informa-

Figure 5.3: Proportion of Files vs Bugs - Eclipse and Prop1 Projects.



(a) Eclipse **(b)** Prop1

tion criteria, namely, the Bayesian Information criterion, Akaike Information Criterion & Hannan-Quinn Information Criterion to compare models.

The distribution of software bugs across files has been shown to follow power law tails and hence several power law distributions have been considered for the problem so far. In this work we consider the Pareto, Weibull and BGPD models. In addition, the Yule Simon, Double Pareto and Lognormal distributions have been shown to be applicable in similar situations and also have generative models that have been shown to be applicable to software bug behavior hence they are included in this study.

We use a large collection of data comprising of fault data from 62 versions of a wide variety of open and proprietary software systems. We study data from 34 versions of open source software systems and 28 versions of proprietary software systems. We note that the number of projects considered in this study is larger than the number of projects used in previous works and, thus, this work is widespread study in terms of data. We conduct experiments and analyze results on data sets that have not been used previously to model bug distributions, to the best of our knowledge, and hence add to the body of published empirical results. We do away with the qualitative accumulation of results in favor of a statistically proven methods, Friedman and Nemenyi tests. Specifically, first we use the

Friedman test to establish that there is significant difference between models and then use the Nemenyi test to check if they outperform each other.

Experiments with open source data showed that the Double Pareto, Weibull and Bounded Generalized Pareto models are the top three best fitting models. Each of these models is significantly better than YS, LGNRM and PD models. And there is no significant difference between YS and LGNRM models but both of them are significantly better than PD. Further, we also note that DP, BGPD and WD models fail to outperform each other with statistical significance. However it is found out that DP models fare better than BGPD and WD models though not significantly. And these conclusions hold good for all three measures-BIC, AIC and HQIC used in our study. The experiments based on Bayes factor and BIC relative fit values reveal that DP model is found to be better than WD model and BGPD model with Very Strong or Positive evidence in nearly 70% of open source projects.

DP model is found to be significantly better than all other five models in experiments conducted on proprietary software projects. There is no significant difference between WD and BGPD models but they are significantly better than YS, BGPD and PD models. And there is no significant difference between YS and LGNRM models but they are significantly better than PD model. The same results are observed irrespective of the three measures-BIC, AIC and HQIC used in our study. The experiments based on Bayes factor and BIC relative fit values reveal that DP model is found better than WD model and BGPD model with Very Strong or Positive evidence in nearly 100% of proprietary software projects. Hence we confidently conclude that the Double Pareto distribution fits bugs better than other distributions for propriety software systems.

5.3 Probability Distribution of Object-oriented Metrics

The distributions followed by the object oriented metrics is a starting point to explore the stochastic process that the development of a software project follows. Though the process of software development adopts a design and coding process with the primary goal of imple-

menting the functionalities effectively and efficiently, the entire process has an apparent randomness, however the resulting software system metrics have statistical regularities. The distribution followed by metrics indicates the underlying generative process involved. Each of the distributions have their own generative processes. Mitzenmacher [153] has listed six unrelated, independent generative models that can result in power Law. Similarly Concas et al. [154] has provided the generative process for Log-normal distribution and Normal Distribution. We have considered these generative models and have attempted to reason if these generative processes could indeed explain the process of the development of the project attributes.

We have considered five models in our study. Past studies have shown that the software projects frequently follow power law. They occasionally seem to follow Log-normal or Gamma distribution [154] [155] [156]. Apart from these three models we have also considered Weibull distribution and Generalized Pareto Distribution (GPD). The intuitions for considering the distributions have been discussed in Section 5.3.2.2. The number of projects that were considered in past studies are not sufficient to infer a more generalizable conclusion. In this study, an attempt has been made to perform an exhaustive comparison over a large number of projects that shall provide a more definitive conclusion to the hypothesis that the metric follows a particular distribution or set of distributions. The CK metrics that have been considered in the study are Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Response For Class (RFC), Lack of Cohesion of Methods (LCOM) , Coupling Between Objects (CBO) and Number of Children (NOC). Apart from these metrics we have also considered Lines of Code (LOC). The process of choosing the most appropriate model involves parameter fitting followed by comparison of models using one of the goodness-of-fit statistics. Studies [154] [155] [156] generally involve parameter fitting using one or more of the following methods: Maximum Likelihood Estimate (MLE), Least Square Estimate (LSE) or Method of Moments (MoM) and compare the models using the goodness-of-fit measures like KS Distance, R^2(Coefficient of Determination) or Pearson χ^2 test. However using these measures have few drawbacks and Akaike Information Criterion (AIC) is a better statistic to compare the models [157], and this shall be discussed in Section 5.3.2.3.

Therefore the objectives of the study are to propose the distribution that best fits each of

CK metrics and to reason out the generative model followed by the metric that may have resulted in the distribution.

5.3.1 Related Work

The pervasiveness of power law in Software systems has been actively explored in various studies. Louridas et al. [158] have explored in depth the ubiquity of power laws in various levels of abstraction of software systems. It was established that power law was not only existent at class and function level but also in the distribution of packages and libraries in different systems and languages. The measurement of procedural complexity of software system has been done using fan-in and fan-out. The power law distribution was fitted using linear regression on the log-log data plots and the benchmark used to determine the goodness-of-fit was R^2. Studies focused on the distribution based on not only file-dependencies but other attributes as well. Baxter et al. [159] studied the key structural attributes of about 56 Java programs and fitted power law using weighted least squares fit. It was concluded that while most of the 17 attributes followed power law, few of them followed Log-normal or stretched exponential. It was checked whether each fit was consistent with the data at different confidence intervals, and then the best fit was decided.

Concas et al. [154] focused on the CK metric properties of the implementation of a Smalltalk system (VisualWorks Smalltalk). They validated whether the data follow a Log-normal or a Pareto distribution using Maximum Likelihood Estimations. They have found that most studied metrics followed a power law distribution except for the number of methods metric (i.e., the WMC metric), the number of all instance variables metric, and the CBO metric, which follow a Log-normal distribution. Though the study performed was quite detailed, the number of projects considered was too less to arrive at concrete conclusions and also the study was performed on SmallTalk programs, the OOP principles of which vary considerably from the more commonly used languages of Java and C++. Shatnawi et al. [155] focused on fitting CK metrics to power law distribution using Maximum Likelihood Estimation and have proposed initial statistical tests (based on Kurtosis and skewness) that can be used as initial screening test to rule out power law Distribution.

They have considered 5 Java software projects. Herraiz et al. [156] based their study on the Qualitas Corpus, measuring the CK metrics suite values for every file of every Java project in the corpus. They concluded that the range of high values for the different metrics follows a power law distribution, whereas the rest of the range follows a Log-normal distribution. They fitted the distribution using the Kolmogorov-Smirnov distance as a measurement of the goodness-of-fit.

Among the distributions that follow power law, the prominent one is Pareto distribution. Other specific distributions have been considered quite often in studies pertaining to software systems. Grbac et al. [137] have considered Weibull distribution and Pareto distribution among others while studying the probability distribution of faults in software systems. Adamic et al. [160] provide the analytical derivation of Pareto 80-20 principle by converting power law to rank size distribution. This has been used by Louridas et al. [158] more appreciable interpretations of the fitted distribution by providing conclusions like the richest 'a' percent of the items have 'b' percent of the resources where a and b are analytically calculated.

Studies have also attempted to explore the generative models of various distributions. Mitzenmacher [153] discovered that Log-normal and power law distributions are connected quite naturally, and hence it is not surprising to state that Log-normal distributions could be a possible alternative to power law distributions across many fields contributing significantly to the debate over whether file size distributions are best modeled by a power law distribution or a Log-normal distribution. Further, these model to have been used to make an attempt at explaining the stochastic process involved in Software Development. Concas et al. [154] have also tried to justify the possible generation process that could have resulted in the statistical distribution. The models considered by them are: Random and Independent increments leading to Normal distribution, Random and proportional increments paving way to Log-normal distribution and finally Preferential Attachment resulting in power law.

While most of these studies almost ubiquitously used goodness-of-fit tests like KS Distance or Chi-Square test, Vose [157] provides reasoning to why Akaike Information Criterion (AIC) is a better goodness-of-fit measure. Among the reasons stated, the prominent one

is that AIC is based on calculating the log likelihood of the fitted distribution producing the set of observations. This allows one to use maximum likelihood as the fitting method and be consistent with the goodness-of-fit statistic. Also, the information criteria penalize distributions with greater number of parameters, and thus help avoid the over-fitting problem [146]. Luo Li et al. [161] also adopt AIC as the goodness-of-fit measure while choosing the distribution that models the number of defect occurrences in each time period over the lifetime of a release of a widely deployed software project. After comparing the AIC values it was concluded that Weibull performed the best and Theil statistics values were derived and analyzed to validate the Weibull model.

In this work, we have considered 5 possible distribution which have been fit over each of the CK metrics of 50 projects. When the number of datasets is huge as in our case, analysis becomes intractable. Therefore statistical methods like Nemenyi tests are employed to summarize results across the multiple datasets [38]. The process was employed by Lessman et al. [5] in order to determine the significantly better classifier in a large-scale empirical comparison of 22 classifiers over 10 datasets while benchmarking classification models for software defect prediction.

Nemenyi test and Friedman test provide a concrete results on whether a model is significantly outperforming other models. And thereby we can eliminate the threats of human error and statistical inaccuracy. The details of the methodology used and its theoretical motivations are provided in the model selection section. We aim to check if a model is significantly better in fitting the metric distribution over all the project. In the instances where such a model exists, we have discussed the possible stochastic generative process that could have resulted in the distribution and also reasoned if such an inference is valid.

5.3.2 Background

5.3.2.1 Datasets

Most of the earlier studies have relied on CK metric data from relatively fewer software systems. However we attempt to generalize the results using larger datasets consisting of

the public Eclipse dataset [162] and the public data collected by Jureczko et al. [93]. We have used the CK Metric values of 50 versions of various software systems that have been listed in Table 5.8.

Table 5.7: Metrics

Name	Description
Lines of Code (LOC)	Number of non-commented lines of code for each software component (e.g., in a class)
Weighted Methods per Class (WMC)	Number of methods contained in a class including public, private and protected methods
Coupling Between Objects (CBO)	Number of classes to which a class is coupled
Depth of Inheritance (DIT)	Maximum inheritance path from the class to the root class
Number Of Children (NOC)	Number of immediate sub-classes of a class
Response For a Class (RFC)	Number of methods that can be invoked for an object of given class
Lack of Cohesion among Methods (LCOM)	Number of methods in a class that are not related through the sharing of some of the class fields

Apart from the open source projects, six proprietary softwares have been considered. Of them, five are custom software systems built for companies in the insurance domain and the sixth is a standard tool that supports quality assurances in software development. All six were built by the same company.

5.3.2.2 Probability Distributions

In total we have considered five models which have been discussed below.

- **Pareto Distribution**

  The first distribution that has been considered in the study is the heavy tailed Pareto distribution, which follows power law [138]. Named after the economist Vilfredo Pareto, the Pareto Principle was first conceived to model the distribution of resources in a population, the famous 80-20 principle [163]. Formally the Pareto Distribution has the following probability distribution,

Table 5.8: Dataset

S.No	Project Name	Version	Number of Modules	S.No	Project Name	Version	Number of Modules
1	Apache Ant	1.3	126	**23**	Apache Lucene	2.2	248
2		1.4	179	**24**		2.4	341
3		1.5	294	**25**	Apache POI	1.5	238
4		1.6	352	**26**		2	315
5		1.7	746	**27**		2.5	386
6		1	340	**28**		3	443
7	Apache Camel	1.2	609	**29**	Apache Synapse	1	158
8		1.4	873	**30**		1.1	223
9		1.6	966	**31**		1.2	257
10	Apache Ivy	1.1	112	**32**	Apache Velocity	1.4	197
11		1.2	242	**33**		1.5	215
12		1.3	353	**34**		1.6	230
13	jEdit	3.2	273	**35**	Xalan Java	2.4	724
14		4	307	**36**		2.5	804
15		4.1	313	**37**		2.6	886
16		4.2	368	**38**		2.7	910
17		4.3	493	**39**	Apache Xerces	init	163
18	Apache Tomcat	1	859	**40**		1.2	441
19	Apache Log4j	1	136	**41**		1.3	454
20		1.1	110	**42**		1.4	589
21		1.2	206	**43**	Redaktor	1	176
22	Apache Lucene	2	196	**44**	ArcPlatform	1	234

$$\frac{\alpha x_m^a}{x^{\alpha+1}} \quad for \quad x \geq x_m \tag{5.13}$$

where $x_m > 0$ is the scale (real) parameter and $\alpha > 0$ is the shape (real) parameter.

Its CDF can be described as,

$$1 - \left(\frac{x_m}{x}\right)^{\alpha} \quad for \quad x \geq x_m \tag{5.14}$$

Pareto distribution is reported to be ubiquitous when it comes to the properties of software systems. Fenton et al. [129] performed a quantitative analysis of fault distributions and confirmed that the number of defects in software systems follow Pareto distribution. The same was replicated and confirmed by Andersson et al. [130]. Concas et al. [154] have concluded that the software properties systematically followed Pareto distribution in the SmallTalk project, Visual Works.

The Pareto distribution is the most common power law distribution. Its statistical distribution can thus be generated by the different stochastic processes listed by Mitzenmacher [153]. Preferential Attachment is the simplest and most common explanation of all different stochastic processes [154].

Preferential Attachment It is the phenomena where new objects tend to attach to popular objects. That means, the probability of choosing an entity to modify (increment) is directly proportional to the present size of the entity and the increment value is independent of the entity size. In the software systems, the objects can be files (classes) and the properties are the metric values. The probability of modifying a larger file can intuitively be considered to be higher. The preferential attachment may thus apply in our context.

- **Log-normal Distribution**

The next distribution that we have considered, a common contender to fit the software metrics, is Log-normal distribution. Previous studies have considered Log-normal distribution and found that few metrics follow this distribution [154] [155] [156]. The Log-normal distribution has the following probability distribution:

$$\frac{1}{x\sigma\sqrt{2\pi}}e^{-\frac{(ln\,x-\mu)^2}{2\sigma^2}} \tag{5.15}$$

where $\mu \epsilon \mathbb{R}$ is the location parameter and $\sigma > 0$ is the scale parameter. And its CDF is as follows.

$$\frac{1}{2}+\frac{1}{2}erf\left[\frac{ln\,x-\mu}{\sqrt{2\sigma}}\right] \tag{5.16}$$

Law of Proportional Effect: Gibrat's law proposed by Gibrat [164], also known as the law of Proportional Effect provides the stochastic process that results in Log-normal distribution. It states that the probability of change (increment) is same for

all modules (files), independent of their size, however the increment is proportional to their present property value. In our context, it means that the probability of a particular file being chosen for increment is the same as other files. However the magnitude of the increment of metric is proportional to its current value. A major drawback of this stochastic model is that it cannot fit the metric value to a file/module (or any other granularity) which is newly introduced and has no metric value yet.

- **Weibull Distribution**

Another distribution we have considered is the Weibull distribution, named after the Swedish mathematician Waloddi Weibull. The distribution has found widespread application in the field of reliability engineering when we fit the distribution of failure times of a product. In such cases the shape parameter (k), provides the rate of failure. The value of k is used to determine if the product faces 'Infant Mortality' or 'Aging'. If the shape parameter k < 1, then failure rate decreases with time. And Weibull distribution is said to be fat-tailed in this case. This is the case that we come across in our study.

Weibull distribution is also used to describe the strength of material with shape factor being the shape module. Weibull distribution is widely adopted in predicting the reliability of software systems in the software development life cycle where the cumulative number of faults was plotted against weeks after deployment [161]. Further, Zhang et al. [133] explored the modeling the distribution of faults over file with Weibull distribution and have found it to perform better than Pareto distribution. As the software metrics are correlated with the fault distribution in modules [165], we consider Weibull distribution in our study. The probability distribution of Weibull is:

$$f(x) = \begin{cases} \frac{k}{\lambda}\left(\frac{x}{\lambda}\right)^{k-1} e^{-\left(\frac{x}{\lambda}\right)^{k}} & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{5.17}$$

where $\lambda \epsilon (0, +\infty)$ is the scale parameter and $k \epsilon (0, +\infty)$ is the shape parameter. While the CDF of Weibull distribution is given as:

$$CDF = \begin{cases} 1 - e^{-\left(\frac{x}{\lambda}\right)^k} & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{5.18}$$

- **Generalized Pareto Distribution (GPD)**

We have also considered Generalized Pareto Distribution (GPD) as another possible candidate distribution. The GPD was introduced by Pickands [166] and since then it found applications in reliability engineering. The GPD is mainly used for the analysis of extreme events as it provides the complex model to describe the full range of data, especially the tail data. The probability distribution function of GPD is given by:

$$\frac{1}{\sigma}(1 + \xi z)^{-\left(\frac{1}{\xi+1}\right)} \tag{5.19}$$

where $z = \frac{x-\mu}{\sigma}$, where $\mu \epsilon (-\infty, \infty)$ is the location parameter, $\sigma \epsilon (0, \infty)$ and $\xi \epsilon (-\infty, \infty)$ is the shape parameter and CDF is given by

$$1 - (1 + \xi z)^{-\frac{1}{\xi}} \tag{5.20}$$

The distribution provides a wide and continuous range of possible shapes with exponential and Pareto distributions as the special cases. The form of GPD depends upon the value of its shape parameter (k).

  - when k = 0, GPD reduces to exponential function

  - when k = 1, GPD reduces to uniform distribution

– when k < 0, GPD reduces to Pareto distribution of second kind

GPD too is not new to the field of software engineering as well, Kolassa et al. [167] proposed GPD to define commit sizes in open source projects.

- **Gamma Distribution**

Another distribution that has been considered is Gamma distribution. In their study, Concas et al. [154] observed that one of the software metrics followed Gamma distribution.

Let's consider a Poisson process with 'points' occurring randomly in time. The sequence of inter-arrival times in the process is a sequence of independent random variables, each of which has exponential distribution. And the distribution of the n^{th} interval is said to follow Gamma distribution. This formal generative process is however unintuitive to realize in our study. The probability density function is given as

$$\frac{1}{\Gamma(k)\,\theta^k} x^{k-1} e^{-\frac{x}{\theta}} \tag{5.21}$$

The CDF is given as

$$\frac{1}{\Gamma(k)} \gamma\left(k, \frac{x}{\theta}\right) \tag{5.22}$$

In the following section we explain in detail how the models are fit to the dataset and what was the basis of selection of the best model.

5.3.2.3 Model Selection

For this problem, we put forth the same methodology/argument as in Section 5.2 for comparing models across multiple datasets and the reasons for making use of information criteria (AIC, BIC).

Root Mean Square Error (RMSE) is a frequently used statistic to predict the difference between the observed values and predicted values.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(X_i - O_i)^2} \tag{5.23}$$

where O_i is the set of Observed values and X_i is the set of respective predicted values. The above equation suggests that a low value of the metric indicates better fit. We use the statistic not to compare various models but to check how well the best model as proposed by the AIC actually fits the given data distribution.

5.3.2.4 Experiments

We have conducted experiments on 50 versions of software systems and the details of these projects are described in Section 5.3.2.1. For each of the version, AIC and BIC values are computed for each of five models and six metrics. We then performed Friedman's test to reject or accept the null hypothesis which states that *"there is no significant difference between the five models"*. We reject the null hypothesis at 0.01 level of significance. If null hypothesis is rejected, we perform the post-hoc test (Nemenyi) and plot the Nemenyi Diagram. The diagram has the models on X axis and their mean rank on y axis. The Critical Difference (CD), which is used to determine if a model is performing better than the rest, is denoted as a vertical line extending above the mean rank. If a model M1 has a higher mean rank than the highest point of the CD interval line of another model M2 then we can say that M2 outperforms M1 with statistical significance. The Nemenyi plots for different metrics have been plotted and discussed in the next section.

Table 5.9: Average: AIC

Distribution/ Metrics	Gamma	GPD	LNORM	Pareto	Weibull
CBO	12529.01	12554.24	12900.62	12554.16	12555.20
DIT	6006.38	7113.30	6077.51	7100.89	6012.88
LCOM	8647.76	7649.41	7579.94	7652.89	7821.39
LOC	22590.38	21002.44	21005.10	21002.43	21138.05
NOC	592.32	-2220.51	-2765.89	-2220.51	-699.13
RFC	12131.79	12209.30	12373.95	12208.77	12160.73
WMC	9584.81	9601.52	9343.66	9602.98	9662.59

Table 5.10: Average: BIC

Distribution/ Metrics	Gamma	GPD	LNORM	Pareto	Weibull
CBO	12537.32	12562.55	12908.93	12562.47	12563.51
DIT	6014.70	7121.61	6085.83	7109.20	6021.19
LCOM	8655.79	7657.44	7587.97	7660.92	7829.42
LOC	22598.69	21010.75	21013.41	21010.75	21146.36
NOC	600.76	-2212.07	-2757.45	-2212.07	-690.70
RFC	12139.94	12217.46	12382.11	12216.92	12168.89
WMC	9593.12	9609.83	9351.97	9611.30	9670.90

Table 5.11: Average: RMSE

Distribution/ Metrics	Gamma	GPD	LNORM	Pareto	Weibull
CBO	0.048	0.055	0.052	0.055	0.052
DIT	0.156	0.120	0.157	0.119	0.151
LCOM	0.093	0.074	0.067	0.075	0.086
LOC	0.156	0.170	0.150	0.170	0.183
NOC	0.030	0.022	0.032	0.022	0.018
RFC	0.041	0.042	0.054	0.042	0.040
WMC	0.043	0.045	0.036	0.045	0.043

5.3.3 Results and Discussions

The following results have been obtained for each of the metrics:

5.3.3.1 Weighted Method per Class (WMC)

For WMC, the null hypothesis is rejected. This means that not all the models have performed equally and thus we can conclude that one or few of the models performs significantly better than others. Nemenyi plots are shown in Figure 5.4. On performing the Nemenyi test, we observe that the Log-normal model fits better, closely followed by Gamma and Generalized Pareto models.

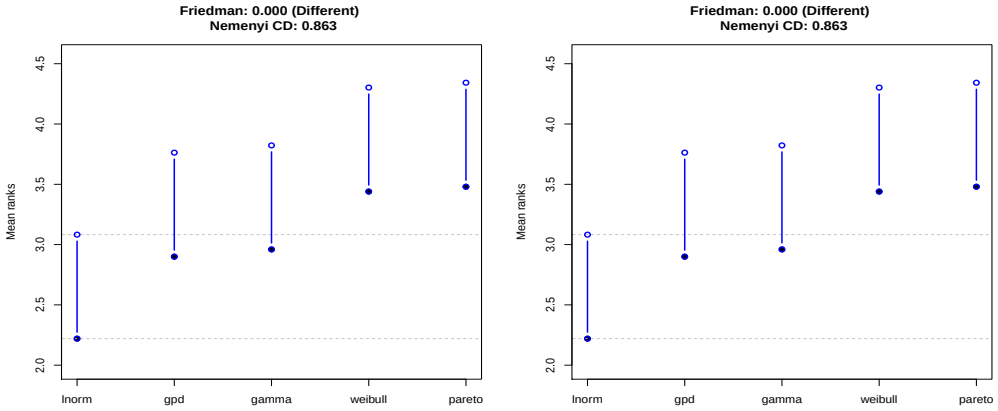
Wheedlen et al. observed that WMC follows power law behavior [168]. Concas et al. observed that the distribution of WMC for a Small Talk project followed a Log-normal distribution. Our result is in agreement with the observations [134]. However, Law of Proportionate increment is too rough to model the generative process for the metric distribution. Most of the methods are written at one go according to the class design that was drawn initially. Therefore it is too much of a generalization if we say that the class is selected at random and the increment in the number of methods in class is proportional to the existing number of methods.

5.3.3.2 Coupling Between Objects (CBO)

The null hypothesis is not rejected for CBO and hence there is no significant difference among five models.

Many attempts have made to represent a software project as a graph, and observe the scale free behavior shown by the distribution of edges, which is the characteristic of any system following power law. Louridas et al. [158] have software project with such a perspective, links representing the use of a class by another class. And have concluded it to follow power law.

Figure 5.4: Nemenyi plots WMC



(a) AIC **(b)** BIC

CBO is a metric closely related to these links. It is the count of the number of classes that are coupled to a particular class which means that the methods of one class call and access the variables of other. These calls are counted in both the directions. Therefore CBO of a class A includes both, the classes referenced by this class and also the classes referencing class A. CBO is (Number of Links)/(Number of Classes in the module) where the links represent Dependence and Composition between modules. Therefore it was intuitively expected that CBO metric shall follow power law, with Pareto model fitting the distribution well. Our experiment shows all the models (including Pareto) perform equally and no model performs significantly better than the rest.

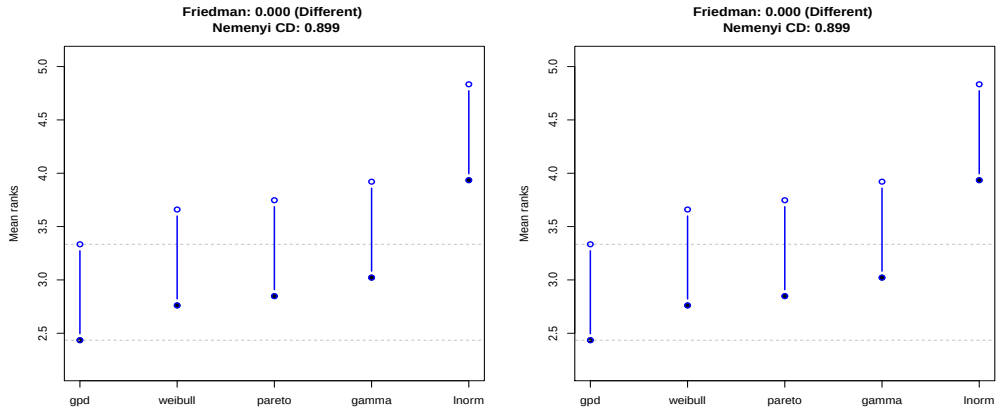
5.3.3.3 Response For Class (RFC)

Response for Class is the number of distinct methods and constructors invoked by a class. It includes the methods defined in the class and also the methods called by the methods of the class. The null hypothesis is rejected for RFC, suggesting significant difference in the performance of the models. Nemenyi diagram is shown in Figure 5.5. From the diagram we observe that Gamma, Pareto, Weibull and Generalized Pareto models fit RFC better than Log-normal.

Like CBO, RFC is a metric that deals with the relationship between classes. In our

experiment, we observe that the four out of five models, performed no better than each other. Thus we shall not be able to pinpoint the generative model that may have led to the current distribution.

Figure 5.5: Nemenyi plots RFC



(a) AIC **(b)** BIC

5.3.3.4 Depth of Inheritance Tree (DIT)

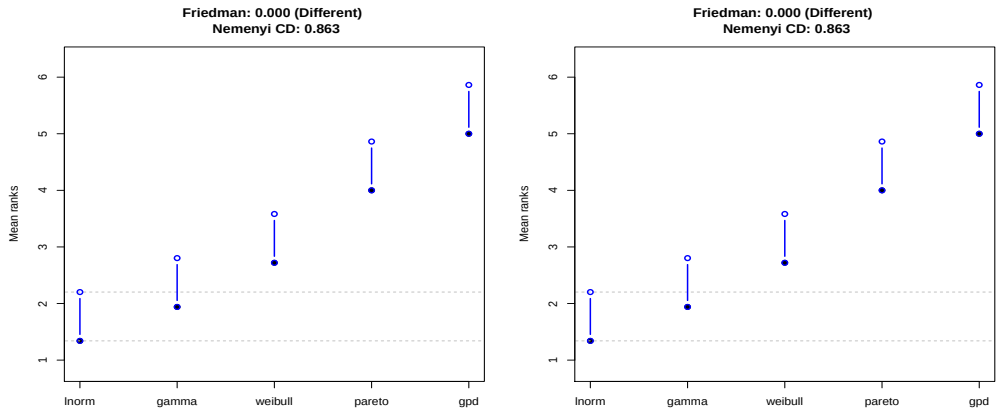
The null hypothesis is rejected and therefore we proceeded to perform the Nemenyi test. Figure 5.6 shows the Nemenyi diagram generated using AIC, BIC values for models, fitting DIT metrics of various projects.

From the Nemenyi diagram we can see that the two best fitting models are Log-normal and Gamma. Their respective RMSE values are 0.0156 and 0.0157 respectively.

The law of proportionate effect doesn't seem to be applicable though our experiments reveal that DIT follows Log-normal. It would be highly impractical to say that the increment of the depth in inheritance tree is proportional to the present depth value. Every time a programmer makes the inheritance tree of a class deeper, it is by a single unit. Therefore increments are constant rather than proportional. It seems unrealistic to say that all the files will be chosen with equal probability. And at the same time it is equally extreme to say that the probability of choosing a file is proportional to its existing value. Thus, though experiments show Lognormal and Gamma Distributions to be fitting the

metric distribution well, their generative processes are too complicated to be intuitively conceived. Also, since the dynamics exhibited by the metric data is very limited, the result may be misleading. Hence we cease to explore further possibilities

Figure 5.6: Nemenyi plots DIT



(a) AIC **(b)** BIC

5.3.3.5 Lack of Cohesion of Methods (LCOM)

LCOM gives the correlation between the methods and local variables of a class. High cohesion suggests good class division. The null hypothesis is rejected, indicating that there is significant difference between the performance of models. Therefore Nemenyi test is performed and it is observed that Log-normal distribution is the best performer among the models. Figure 5.7 shows the plotted graph. The average value of RMSE for the fit of Log-normal distribution is 0.067

Pani and Concas observed that the Log-normal model fit the LCOM metric distribution the best in case of the Java project Eclipse [169]. In our study we notice that this observation can be generalized to all Java projects.

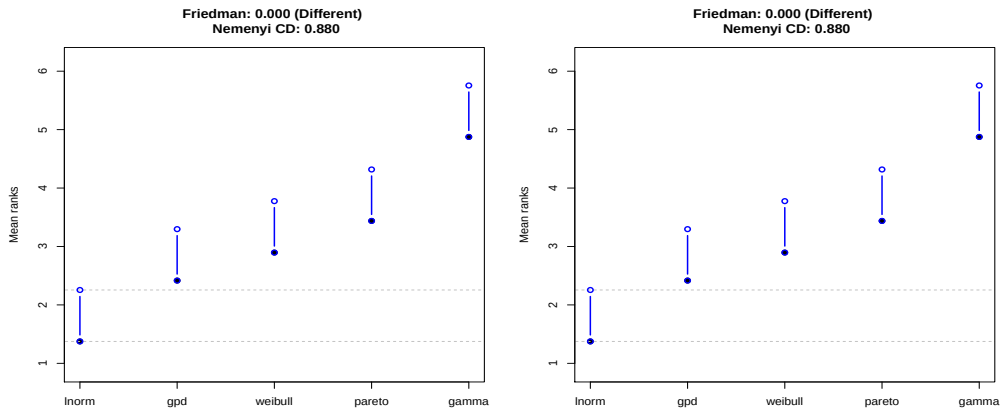
LCOM metric as originally proposed by Chidamber and Kemerer is as follows [8]. Let a class C have n methods $M_1, M_2, M_3, ...M_n$. Let I_i be the set of instance variables used by method M_i. Let $P = \{(I_i, I_j) \,|I_i \cap I_j = \phi\}$ and $Q = \{(I_i, I_j) \,|I_i \cap I_j \neq \phi\}$ then,

$$\begin{aligned} LCOM &= |P| - |Q|\,,\, if\ |P| > |Q| \\ &= 0 \quad otherwise \end{aligned} \tag{5.24}$$

The above definition shows that a higher LCOM value suggests the splitting of a class as there is a lack of cohesion. This suggests that $|P| > |Q|$ which means that more pairs of methods don't have a common instance variable. An increase in the LCOM value can be achieved in the following ways: Increasing $|P|$, i.e., introducing a method that doesn't share a common instance variable with more than half of the existing methods. Decreasing $|Q|$, i.e., deleting a method that shares a common instance variable with more that half of the existing methods. In the process of software development, deletion of methods is highly rare when compared to addition of new methods. Therefore we focus on the 1st scenario.

Consider 2 classes C1 and C2 with the LCOM value 10 and 20 respectively. This means that C1 has 10 more pairs of incoherent methods than coherent ones and C2 has 20 more pairs. This means that the C2 has more 'clusters' of methods that are incoherent with each other. If we introduce a incoherent method to each of these 'clusters', the magnitude of increase in the LCOM value of C2 will be higher than that of C1. The possible magnitude of increase is approximately proportional to the LCOM value of the class. Hence we can identify the presence of 'Law of Proportional Effect' which leads to the Log-normal distribution that the metric follows.

Figure 5.7: Nemenyi plots LCOM



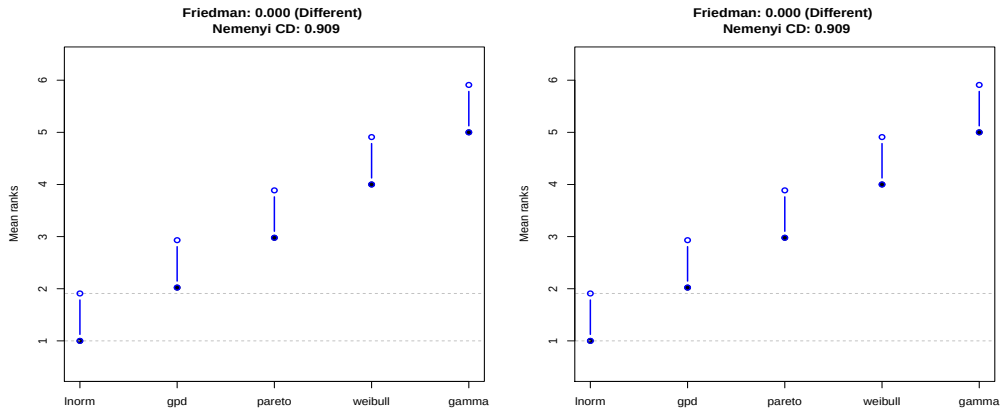
(a) AIC **(b)** BIC

5.3.3.6 Number of Children (NOC)

The null hypothesis is rejected and hence Nemenyi test is employed. The nemenyi plots are shown in Figure 5.8. It is observed that Log-normal fits NOC better than other models. The average RMSE for the Log-normal fit is 0.0323 which suggests that not only is the Lognormal model relatively better than other model, but also it fits the data distribution well in absolute sense too.

It seems unrealistic to say that the metric distribution might have been generated by Law of Proportional effect, i.e., the classes being chosen with equal probability and the magnitude of increment in number of children for a class being proportional to its present metric value. Thus, though experiments show Lognormal model to be fitting the metric distribution well, its generative process is too complicated to be intuitively obtained.

Figure 5.8: Nemenyi plots NOC



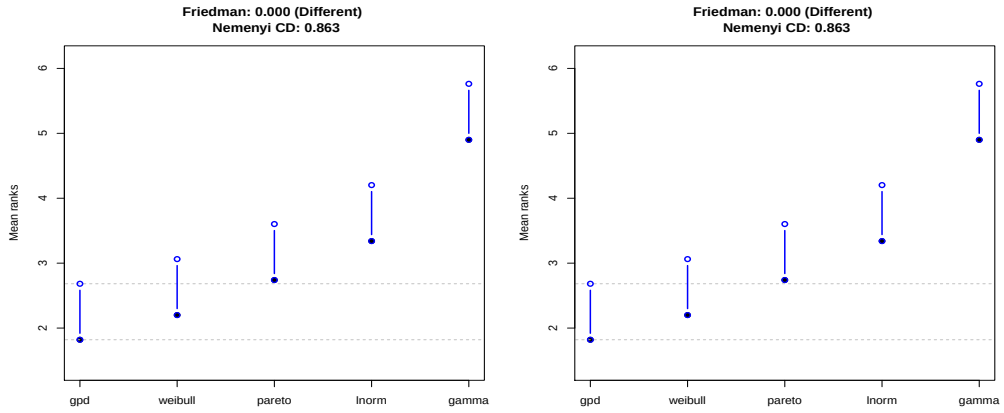
(a) AIC **(b)** BIC

5.3.3.7 Lines of Code (LOC)

Though this is not one of the original CK metric, Lines of Code is a commonly used metric to understand the size and growth of the software systems. In an attempt to find the distribution followed by the metric, we fit five models and compare them with their respective AIC values. The null hypothesis is rejected and hence one or few of the models fit significantly better than the rest. We further proceeded to perform the Nemenyi test

and found that GPD and Weibull models fit the data equally well and are better than the other models with a statistical significance. We then observed the RMSE values for the fits in order to estimate how well the models actually fit the data, and find the score to be 0.170 and 0.183 for GPD and Weibull respectively. This significant value suggests that though the models are better among the models that we have considered, they can not be considered to fit the given distribution well. Their actual performance seems to be poor.

Figure 5.9: Nemenyi plots LOC



(a) AIC **(b)** BIC

5.3.4 Contributions

We attempted to model six object oriented metrics. We have also considered the LOC metric. With respect to AIC, we observe that WMC, DIT LCOM and NOC metrics follow the Log-normal distribution. On the contrary RFC follows four of the distributions except the Log-normal distribution. There is no significant difference between five models in case of CBO. GPD and Weibull models fit the LOC metric significantly better than other three distributions. Understanding the distributions followed by CK metrics will help us in predicting the evolutions of software systems and also in the estimation of efforts required to maintain such systems.

6 Conclusion and Future Work

Defect prediction research has been evolving quite rapidly but its applicability to IT industry is far from reality. The efficacy of defect prediction models built with various combinations of metrics, algorithms and performance measures has been investigated in this work. Defect prediction models built with proposed approaches are compared with existing approaches.

6.1 Contributions and Findings

The goal of our work is to progress defect prediction research with the pre-processing techniques, novel features, Bayesian learning techniques, multi-objective optimization techniques and comparative study of classification algorithms. We have also made an attempt to model the probability distribution of past defects and object-oriented metrics. We summarize the contributions of our work as follows.

1. Our investigation on the impact of feature selection on defect prediction establishes that prediction accuracies can be enhanced significantly by applying feature selection techniques during pre-processing steps. We encourage to use greedy based algorithms for feature selection as they stand out and perform much better than the other competent algorithms. We re-emphasize that there may not be any common metrics amongst optimal feature subsets of similar projects. We have also found out that the impact of feature selection algorithm is independent of the underlying machine learning algorithms used for the classification.

2. We define a modus operandi to extract some popular change metrics from the Eclipse repository on Github, which can be generalized for any open-source Github repository. We have proposed couple of new change metrics, namely, Entropy and Mean Period of Change to figure out whether commits are uniformly distributed over the time-line between the two consecutive releases. We establish that a source file which was last changed in the early periods of the timeline and has been free of changes for the remaining periods can be considered relatively stable, as compared to a source file for which period of change concentration is localized closer to the date of release. We have used Logistic Regression, CART Decision Tree, Gaussian Naïve Bayes and Naïve Bayes Tree algorithm to build prediction models using the proposed and existing change metrics. We have found that Naïve Bayes Tree perform better than other commonly used algorithms for defect prediction problem.

3. Our initial experiments confirmed that Naïve Bayes classifier is one of the best performing classification techniques in defect prediction. It assumes conditional independence of features and for the defect prediction problem which is not entirely true. For example, lines of code (LOC) and number of operands of a source file can't be taken as independent. It motivated us to relax these conditional independence and to check whether there is an improvement in performance of classifiers. We defined augmentation as follows.

   A simple Bayesian Network which doesn't not have an arc from class node to every other node, is not an augmented. Bayesian Network is augmented if it has an arc from class node to every other node. For our experiments we have augmented Bayesian Network structures learnt using HC, TABU, GS, MMHC and RSMAX2.

   First, we have compared the performance of defect prediction models built using simple Bayesian Network classifiers with other traditional machine learning classifiers. In terms of AUC and H-Measure, Random Forest classifier is found to be better performing classifier than Bayesian Network classifiers. However, on defect prediction models built with our proposed augmented Bayesian Network structure, we observe that RSMAX2 and Grow-Shrink are consistently better, and there is no significant difference in performance between these classifiers and Random Forest. When we

compare classifiers using different cost ratio distributions leading to Risk Averse and Delay Averse scenarios, RSMAX2 and Grow-Shrink classifiers are performing very close to Random Forests and are found better than Naïve Bayes classifier.

4. The cost of fixing a software defect varies with the phase in which it is uncovered. Defects found during post-release phase costs much more than the defect that is uncovered in pre-release phase. Classifying a defective source file to be non-defective will result in a defect during post release phase and non-defective file to be a defective file will result in wastage of effort (due to QA activities). These two costs are called misclassification costs. Such misclassification cost has not been given much importance while comparing defect prediction models. Hence, we find it interesting to evaluate defect models using cost centric measure - Normalized Expected Cost of Misclassification (NECM).

   Though there are few studies that consider NECM as a model evaluation measure, they have built models using either cost sensitive neural networks or traditional machine learning algorithms. As there is no comprehensive comparative study on the effectiveness of defect prediction models which are built using cost sensitive neural networks and traditional machine laerning classifiers, we find it interesting to build models using both the approaches and compared their performances using NECM.

   Our experiments confirm Random Forest as an outperforming algorithm in terms of NECM closely followed by Logistic Regression and Bayesian Network. Random Forest performs significantly better than cost sensitive boosting Neural Network methods. We also observe that among the cost sensitive boosting Neural Network algorithms, Cost Sensitive Boosting of Neural Networks using Threshold Moving(CSBNN-TM) perform better than Cost Sensitive Boosting of Neural Networks with Weight Updation-I(CSBNN-WU1) and Cost Sensitive Boosting of Neural Networks with Weight Updation-II(CSBNN-WU2).

5. Researchers proposed effort-aware defect prediction models wherein files are ranked not only based on their defect-proneness but also on the effort that is required to

perform quality assurance activities like testing, code review etc. on defect prone files. Previous studies, considered Cyclomatic Complexity as a metric for the effort required irrespective of the assurance activity that will be undertaken during post prediction phases. We believe that effort-aware models are sensitive to the type of quality assurance activity one undertakes. We have proposed effort-aware defect prediction models that will consider the effort required to perform the specific quality assurance activities like code review and testing. We have considered lines of code and the number of testcases as our measures of effort for code review and testing respectively. We have empirically proven the supremacy of these models over the existing effort-aware model built with Cyclomatic Complexity.

- Our results indicate that the testing based effort-aware models perform significantly better than generic effort aware models.

- The code review based effort-aware model performs slightly better than generic model individually for each project but we could not find significant difference in models performance.

6. In multi-objective optimization approach we have formulated two problems with two sets of contrasting objectives as follows. M1: Maximize effectiveness of the model and Minimize misclassification cost and M2: Maximize effectiveness of the model and Minimize LOC cost.

   We have compared multi-objective logistic regression models with four single objective algorithms for cross version defect prediction. Our results indicate following benefits of multi-objective approach:

   - The multi-objective defect prediction model (M1) is able to identify more defects at the same or lesser misclassification cost incurred by all four single objective defect prediction models.

   - The multi-objective defect prediction model (M2) is able to identify more defect

prone classes at the same or lesser LOC cost than the cost incurred with single objective algorithms.

We recommend multi-objective approach for cross version defect prediction problems against traditional single objective approaches.

7. Understanding the distributions of CK metrics will help us in predicting the evolutions and efforts required to maintain such systems. Previously it has been shown that these metrics frequently follow power law. As these metrics are also shown to follow Log-normal or Gamma distribution occasionally, we have studied more number of software systems and applied more distribution models to infer more generalizable results

   In our attempt to model six object oriented metrics, we observe that WMC, DIT LCOM and NOC metrics follow the Log-normal distribution. RFC follows four of the distributions except the Log-normal distribution and CBO does not follows any distribution better than other distributions. We have also found that LOC metrics follow GPD and Weibull distributions.

8. Studies have proposed the Pareto distribution to model bugs in software systems. However, few other probability distributions such as the Weibull, Bounded Generalized Pareto, Double Pareto, Log Normal and Yule-Simon distributions have also been proposed to model bugs. The contemporary work on choosing the most appropriate model for the underlying bug distribution relies on parameter estimation methods and goodness-of-fit statistics. In this work we seek to strengthen the model selection methodology and make it less vulnerable to threats such as over-fitting by making use of information criterion based approaches.

   - Experiments with open source data showed that the Double Pareto, Weibull and Bounded Generalized Pareto models are the top three best fitting models. However it is found out that DP models fare better than BGPD and WD models though not significantly. It holds good for all three measures- BIC, AIC and HQIC used in our study. The experiments based on Bayes factor and BIC

relative fit values reveal that DP model is found to be better than WD model and BGPD model with Very Strong or Positive evidence in nearly 70% of open source projects.

- We also find that Double Pareto model significantly better than all other five models in experiments conducted on proprietary software projects. Similar results are observed irrespective of the three measures- BIC, AIC and HQIC used in our study. The experiments based on Bayes factor and BIC relative fit values reveal that DP model is found better than WD model and BGPD model with Very Strong or Positive evidence in nearly 100% of proprietary software projects. Hence we confidently conclude that the Double Pareto distribution fits bugs better than other distributions for propriety software systems.

6.2 Future Work

Software Analytics is gaining momentum as a result of involved empirical research in enhancing quality and productivity of software engineering activities. There have been rigorous research efforts in the areas of defect prediction, bug localization and effort estimation by making use of historical data. We recognize following problem areas where advanced machine learning and search-based software engineering techniques can improve the state-of-art and define them as our future work.

Bug-fix time prediction We find bug-fix time prediction as an interesting problem with lots of advantages to industry. In the event of any reported bug, generally the project management team approaches experienced team members to get an estimate of the bug-fix time and the response would be very much a personalized and subjective estimate. The accurate prediction of bug-fix times is useful in planning and management of resources that results in the lessened cumulative bug fix time. There have been attempts to solve defect prediction problem with machine learning approaches. Similar techniques can be leveraged to solve the problem of predicting bug-fix times.

By making use of the historical data about bug-fix times of previous bugs of the project, prediction models can be built to predict bug-fix time of future bugs.

In our initial attempt to predict bug-fix time, we have made use of three major features: report title, description and the reputation of the bug reporter. We introduce a new feature, the score of bug reporter, to predict bug fix time. We define the score or reputation of a reporter as follows: "Number of bugs reported by a reporter that were eventually fixed". For example, if a reporter logs 50 bugs out of which 30 are fixed, score of the reporter is 30. Our intuition to solve this problem is "Given two reports which are similar in complexity, the bug reported by the reporter with higher score will have a lower fix time and vice versa". Our dataset contains 91 reporters and around 567 bugs extracted from Jira issue tracking system of JBoss Project.

The prediction models built with 'reputation of the bug reporter' as a feature and other existing features is found to be performing significantly better than other existing bug fix time prediction models. We are interested in finding more features and suitable machine learning approaches to build more accurate bug fix-time prediction models.

Software effort-estimation Effort-estimation has been a very difficult problem in fast changing development environments. Underestimating results in under-staffing, limiting quality assurance activities and missing deadlines. Past projects data can be used to predict the effort required for future projects. Features like application domain, project duration, project size, programming language and tools, function points, geographical locations and project team experience in building similar systems can be extracted and used for effort prediction. We are interested to propose novel features and applying state of art learning techniques to enhance prediction accuracies.

Bibliography

[1] G. Canfora, A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella, "Defect prediction as a multiobjective optimization problem," *Software Testing, Verification and Reliability*, vol. 25, no. 4, pp. 426–459, 2015.

[2] J. M. Stecklein, J. Dabney, B. Dick, B. Haskins, R. Lovell, and G. Moroney, "Error cost escalation through the project life cycle," 2004.

[3] S. A. Sherer, "Software fault prediction," *Journal of Systems and Software*, vol. 29, no. 2, pp. 97–105, 1995.

[4] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4. ACM, 2004, pp. 86–96.

[5] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *Software Engineering, IEEE Transactions on*, vol. 34, no. 4, pp. 485–496, 2008.

[6] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 2–13, 2007.

[7] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *Software Engineering, IEEE Transactions on*, vol. 31, no. 4, pp. 340–355, 2005.

[8] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, no. 6, pp. 476–493, 1994.

[9] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308–320, 1976.

[10] M. H. Halstead, "Elements of software science," 1977.

[11] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change bursts

as defect predictors," in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on.* IEEE, 2010, pp. 309–318.

[12] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement.* ACM, 2012, pp. 171–180.

[13] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on.* IEEE, 2008, pp. 181–190.

[14] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *Software Engineering, IEEE Transactions on*, vol. 34, no. 4, pp. 485–496, 2008.

[15] T. M. Khoshgoftaar and N. Seliya, "Comparative assessment of software quality classification techniques: An empirical case study," *Empirical Software Engineering*, vol. 9, no. 3, pp. 229–257, 2004.

[16] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on.* IEEE, 2010, pp. 107–116.

[17] H. Liu, J. Li, and L. Wong, "A comparative study on feature selection and classification methods using gene expression profiles and proteomic patterns," *Genome Informatics Series*, pp. 51–60, 2002.

[18] Y. Yang and J. O. Pedersen, "A comparative study on feature selection in text categorization," in *ICML*, vol. 97, 1997, pp. 412–420.

[19] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering.* IEEE Computer Society, 2009, pp. 78–88.

[20] S. Krishnan, C. Strasburg, R. R. Lutz, and K. Goševa-Popstojanova, "Are change metrics good predictors for an evolving software product line?" in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering.* ACM, 2011, p. 7.

[21] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim, "Reducing features to improve code change-based bug prediction," *Software Engineering, IEEE Transactions on*, vol. 39, no. 4, pp. 552–569, 2013.

[22] R. Moser, W. Pedrycz, and G. Succi, "Analysis of the reliability of a subset of change metrics for defect prediction," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement.* ACM, 2008, pp. 309–311.

[23] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *Software Engineering, IEEE Transactions on*, vol. 34, no. 2, pp. 287–300, 2008.

[24] S. Mahmood, R. Lai, Y. Soo Kim, J. Hong Kim, S. Cheon Park, and H. Suk Oh, "A survey of component based system quality assurance and assessment," *Information and Software Technology*, vol. 47, no. 10, pp. 693–707, 2005.

[25] N. E. Fenton and M. Neil, "Software metrics: successes, failures and new directions," *Journal of Systems and Software*, vol. 47, no. 2, pp. 149–157, 1999.

[26] Y. Jiang, B. Cukic, and T. Menzies, "Can data transformation help in the detection of fault-prone modules?" in *Proceedings of the 2008 workshop on Defects in large software systems.* ACM, 2008, pp. 16–20.

[27] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on.* IEEE, 2010, pp. 31–41.

[28] S. Scott and S. Matwin, "Feature engineering for text classification," in *ICML*, vol. 99, 1999, pp. 379–388.

[29] M. Chapman, P. Callis, and W. Jackson, "Metrics data program," *NASA IV and V Facility, http://mdp.ivv.nasa.gov*, 2004.

[30] F. Provost and T. Fawcett, "Robust classification for imprecise environments," *Machine Learning*, vol. 42, no. 3, pp. 203–231, 2001.

[31] Z. Zhao, F. Morstatter, S. Sharma, S. Alelyani, A. Anand, and H. Liu, "Advancing feature selection research-asu feature selection repository," *School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe*, 2010.

[32] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust prediction of fault-proneness by random forests," in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on.* IEEE, 2004, pp. 417–428.

[33] G. C. Cawley and N. L. Talbot, "Gene selection in cancer classification using sparse

logistic regression with bayesian regularization," *Bioinformatics*, vol. 22, no. 19, pp. 2348–2355, 2006.

[34] S. K. Shevade and S. S. Keerthi, "A simple and efficient algorithm for gene selection using sparse logistic regression," *Bioinformatics*, vol. 19, no. 17, pp. 2246–2253, 2003.

[35] G. Biau, L. Devroye, and G. Lugosi, "Consistency of random forests and other averaging classifiers," *The Journal of Machine Learning Research*, vol. 9, pp. 2015–2033, 2008.

[36] D. W. Hosmer, S. Taber, and S. Lemeshow, "The importance of assessing the fit of logistic regression models: a case study." *American journal of public health*, vol. 81, no. 12, pp. 1630–1635, 1991.

[37] T. Fawcett, "An introduction to roc analysis," *Pattern recognition letters*, vol. 27, no. 8, pp. 861–874, 2006.

[38] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *The Journal of Machine Learning Research*, vol. 7, pp. 1–30, 2006.

[39] O. Vandecruys, D. Martens, B. Baesens, C. Mues, M. De Backer, and R. Haesen, "Mining software repositories for comprehensible software fault prediction models," *Journal of Systems and software*, vol. 81, no. 5, pp. 823–839, 2008.

[40] S. Zhong, T. M. Khoshgoftaar, and N. Seliya, "Analyzing software measurement data with clustering techniques," *Intelligent Systems, IEEE*, vol. 19, no. 2, pp. 20–27, 2004.

[41] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 489–498.

[42] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on.* IEEE, 2007, pp. 9–9.

[43] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on.* IEEE, 2005, pp. 284–292.

[44] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *Software Maintenance (ICSM), 2010 IEEE International Conference on.* IEEE, 2010, pp. 1–10.

[45] R. Kohavi, "Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid." in *KDD*, 1996, pp. 202–207.

[46] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting component failures at design time," in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. ACM, 2006, pp. 18–27.

[47] T. M. Khoshgoftaar and E. B. Allen, "Ordering fault-prone software modules," *Software Quality Journal*, vol. 11, no. 1, pp. 19–37, 2003.

[48] K. Dejaeger, T. Verbraken, and B. Baesens, "Toward comprehensible software fault prediction models using bayesian network classifiers," *Software Engineering, IEEE Transactions on*, vol. 39, no. 2, pp. 237–257, 2013.

[49] T. Wang and W.-h. Li, "Naive bayes software defect prediction model," in *Computational Intelligence and Software Engineering (CiSE), 2010 International Conference on*, 2010, pp. 1–4.

[50] N. Fenton, M. Neil, and D. Marquez, "Using bayesian networks to predict software defects and reliability," *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability*, vol. 222, no. 4, pp. 701–712, 2008.

[51] A. Okutan and O. T. Yıldız, "Software defect prediction using bayesian networks," *Empirical Software Engineering*, vol. 19, no. 1, pp. 154–181, 2014.

[52] N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian network classifiers," *Machine learning*, vol. 29, no. 2-3, pp. 131–163, 1997.

[53] J. P. Sacha, L. S. Goodenday, and K. J. Cios, "Bayesian learning for cardiac spect image interpretation," *Artificial Intelligence in Medicine*, vol. 26, no. 1, pp. 109–143, 2002.

[54] E. Keogh and M. Pazzani, "Learning augmented bayesian classifiers: A comparison of distribution-based and classification-based approaches," in *Proceedings of the seventh international workshop on artificial intelligence and statistics*. Citeseer, 1999, pp. 225–230.

[55] G. I. Webb, J. R. Boughton, and Z. Wang, "Not so naive bayes: aggregating one-dependence estimators," *Machine learning*, vol. 58, no. 1, pp. 5–24, 2005.

[56] M. Sahami, "Learning limited dependence bayesian classifiers." in *KDD*, vol. 96, 1996, pp. 335–338.

[57] Y. Jing, V. Pavlović, and J. M. Rehg, "Boosted bayesian network classifiers," *Machine Learning*, vol. 73, no. 2, pp. 155–184, 2008.

[58] J. Cheng and R. Greiner, "Comparing bayesian network classifiers," in *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 1999, pp. 101–108.

[59] K. B. Korb and A. E. Nicholson, *Bayesian artificial intelligence*. CRC press, 2010.

[60] S. Russell, P. Norvig, and A. Intelligence, "A modern approach," *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, vol. 25, 1995.

[61] D. Margaritis, "Learning bayesian network model structure from data," Ph.D. dissertation, US Army, 2003.

[62] G. F. Cooper and E. Herskovits, "A bayesian method for the induction of probabilistic networks from data," *Machine learning*, vol. 9, no. 4, pp. 309–347, 1992.

[63] D. Heckerman, D. Geiger, and D. M. Chickering, "Learning bayesian networks: The combination of knowledge and statistical data," *Machine learning*, vol. 20, no. 3, pp. 197–243, 1995.

[64] I. Tsamardinos, L. E. Brown, and C. F. Aliferis, "The max-min hill-climbing bayesian network structure learning algorithm," *Machine learning*, vol. 65, no. 1, pp. 31–78, 2006.

[65] M. Scutari, "Learning bayesian networks with the bnlearn r package," *arXiv preprint arXiv:0908.3817*, 2009.

[66] T. Menzies, J. S. Di Stefano, M. Chapman, and K. McGill, "Metrics that matter," in *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*. IEEE, 2002, pp. 51–57.

[67] K. B. Irani, "Multi-interval discretization of continuous-valued attributes for classification learning," 1993.

[68] D. J. Hand, "Measuring classifier performance: a coherent alternative to the area under the roc curve," *Machine learning*, vol. 77, no. 1, pp. 103–123, 2009.

[69] D. J. Hand and C. Anagnostopoulos, "A better beta for the h measure of classification performance," *Pattern Recognition Letters*, vol. 40, pp. 41–46, 2014.

[70] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *The Journal of Machine Learning Research*, vol. 7, pp. 1–30, 2006.

[71] K. Gao, T. M. Khoshgoftaar, H. Wang, and N. Seliya, "Choosing software metrics for defect prediction: an investigation on feature selection techniques," *Software: Practice and Experience*, vol. 41, no. 5, pp. 579–606, 2011.

[72] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 531–577, 2012.

[73] M. Soni, "Defect prevention: Reducing costs and enhancing quality," *iSixSigma. com*, vol. 19, 2006.

[74] Z.-H. Zhou and X.-Y. Liu, "Training cost-sensitive neural networks with methods addressing the class imbalance problem," *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 1, pp. 63–77, 2006.

[75] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 434–443, 2013.

[76] W. Fan, S. J. Stolfo, J. Zhang, and P. K. Chan, "Adacost: misclassification cost-sensitive boosting," in *Icml*, 1999, pp. 97–105.

[77] Y. Sun, M. S. Kamel, A. K. Wong, and Y. Wang, "Cost-sensitive boosting for classification of imbalanced data," *Pattern Recognition*, vol. 40, no. 12, pp. 3358–3378, 2007.

[78] K. M. Ting, "A comparative study of cost-sensitive boosting algorithms," in *In Proceedings of the 17th International Conference on Machine Learning.* Citeseer, 2000.

[79] J. Zheng, "Cost-sensitive boosting neural networks for software defect prediction," *Expert Systems with Applications*, vol. 37, no. 6, pp. 4537–4543, 2010.

[80] Y. Zhou and H. Leung, "Empirical analysis of object-oriented design metrics for predicting high and low severity faults," *IEEE Transactions on software engineering*, vol. 32, no. 10, pp. 771–789, 2006.

[81] G. J. Pai and J. B. Dugan, "Empirical analysis of software fault content and fault proneness using bayesian methods," *IEEE Transactions on software Engineering*, vol. 33, no. 10, pp. 675–686, 2007.

[82] T. M. Khoshgoftaar, D. L. Lanning, and A. S. Pandya, "A comparative study of pattern recognition techniques for quality evaluation of telecommunications software," *IEEE Journal on Selected Areas in Communications*, vol. 12, no. 2, pp. 279–291, 1994.

[83] T. M. Khoshgoftaar, E. B. Allen, J. P. Hudepohl, and S. J. Aud, "Application of neural networks to software quality modeling of a very large telecommunications system," *IEEE Transactions on neural networks*, vol. 8, no. 4, pp. 902–909, 1997.

[84] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *Software Engineering, IEEE Transactions on*, vol. 31, no. 10, pp. 897–910, 2005.

[85] R. W. Selby and A. A. Porter, "Learning from examples: generation and evaluation of decision trees for software resource analysis," *IEEE Transactions on Software Engineering*, vol. 14, no. 12, pp. 1743–1757, 1988.

[86] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. Hudepohl, "Classification tree models of software quality over multiple releases," in *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on.* IEEE, 1999, pp. 116–125.

[87] A. G. Koru and H. Liu, "An investigation of the effect of module size on defect prediction using static measures," vol. 30, no. 4. New York, NY, USA: ACM, May 2005, pp. 1–5.

[88] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *Journal of Systems and Software*, vol. 81, no. 5, pp. 649–660, 2008.

[89] C. Ebert, "Classification techniques for metric-based software development," *Software Quality Journal*, vol. 5, no. 4, pp. 255–272, 1996.

[90] B. W. Boehm, "Industrial software metrics top 10 list," *IEEE software*, vol. 4, no. 5, pp. 84–85, 1987.

[91] S. Dick and A. Kandel, "Data mining with resampling in software metrics," *Artificial Intelligence Methods in Software Testing*, vol. 56, p. 175, 2004.

[92] C. Seiffert, T. M. Khoshgoftaar, and J. Van Hulse, "Improving software-quality predictions with data sampling and boosting," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, vol. 39, no. 6, pp. 1283–1294, 2009.

[93] M. Jureczko and L. Madeyski, "Towards identifying software project clusters

with regard to defect prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, ser. PROMISE '10. New York, NY, USA: ACM, 2010, pp. 9:1–9:10. [Online]. Available: http://doi.acm.org/10.1145/1868328.1868342

[94] N. Niu and A. Mahmoud, "Enhancing candidate link generation for requirements tracing: The cluster hypothesis revisited," in *Requirements Engineering Conference (RE), 2012 20th IEEE International*, Sept 2012, pp. 81–90.

[95] Y. Freund, "Boosting a weak learning algorithm by majority." in *COLT*, vol. 90, 1990, pp. 202–216.

[96] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *Software Engineering, IEEE Transactions on*, vol. 22, no. 12, pp. 886–894, 1996.

[97] E. Arisholm and L. C. Briand, "Predicting fault-prone components in a java legacy system," in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. ACM, 2006, pp. 8–17.

[98] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *Software Engineering, IEEE Transactions on*, vol. 22, no. 10, pp. 751–761, 1996.

[99] M.-H. Tang, M.-H. Kao, and M.-H. Chen, "An empirical study on object-oriented metrics," in *Software Metrics Symposium, 1999. Proceedings. Sixth International*. IEEE, 1999, pp. 242–249.

[100] M. Cartwright and M. Shepperd, "An empirical investigation of an object-oriented software system," *Software Engineering, IEEE Transactions on*, vol. 26, no. 8, pp. 786–796, 2000.

[101] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *Software Engineering, IEEE Transactions on*, vol. 29, no. 4, pp. 297–310, 2003.

[102] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 452–461.

[103] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*. ACM, 2009, p. 7.

[104] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *Journal of Systems and Software*, vol. 83, no. 1, pp. 2–17, 2010.

[105] M. Friedman, "The use of ranks to avoid the assumption of normality implicit in the analysis of variance," *Journal of the American Statistical Association*, vol. 32, no. 200, pp. 675–701, 1937.

[106] M. Harman, "The relationship between search based software engineering and predictive modeling," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, ser. PROMISE '10. New York, NY, USA: ACM, 2010, pp. 1:1–1:13.

[107] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii," *Lecture notes in computer science*, vol. 1917, pp. 849–858, 2000.

[108] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Multi-objective cross-project defect prediction," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 252–261.

[109] T. Menzies, R. Krishna, and P. D., "The promise repository of empirical software engineering data," 2015, http://openscience.us/repo.

[110] D. Zhang, K. El Emam, H. Liu *et al.*, "An investigation into the functional form of the size-defect relationship for software modules," *Software Engineering, IEEE Transactions on*, vol. 35, no. 2, pp. 293–304, 2009.

[111] R. Subramanyam and M. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects," *Software Engineering, IEEE Transactions on*, vol. 29, no. 4, pp. 297–310, April 2003.

[112] Z. He, F. Peters, T. Menzies, and Y. Yang, "Learning from open-source projects: An empirical study on defect prediction," in *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*, Oct 2013, pp. 45–54.

[113] F. Peters, T. Menzies, and A. Marcus, "Better cross company defect prediction," in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, May 2013, pp. 409–418.

[114] G. Czibula, Z. Marian, and I. G. Czibula, "Software defect prediction using

relational association rule mining," *Inf. Sci.*, vol. 264, pp. 260–278, Apr. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.ins.2013.12.031

[115] Z. Marian, I. G. Czibula, G. Czibula, and S. Sotoc, "Software defect detection using self-organizing maps." *Studia Universitatis Babes-Bolyai, Informatica*, vol. 60, no. 2, 2015.

[116] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 789–800.

[117] M. Harman and J. Clark, "Metrics are fitness functions too," in *Software Metrics, 2004. Proceedings. 10th International Symposium on.* IEEE, 2004, pp. 58–69.

[118] A. B. De Carvalho, A. Pozo, and S. R. Vergilio, "A symbolic fault-prediction model based on multiobjective particle swarm optimization," *Journal of Systems and Software*, vol. 83, no. 5, pp. 868–882, 2010.

[119] X. Yang, K. Tang, and X. Yao, "A learning-to-rank approach to software defect prediction," *Reliability, IEEE Transactions on*, vol. 64, no. 1, pp. 234–246, 2015.

[120] W. Ma, L. Chen, Y. Yang, Y. Zhou, and B. Xu, "Empirical analysis of network measures for effort-aware fault-proneness prediction," *Information and Software Technology*, vol. 69, pp. 50–70, 2016.

[121] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, June 1994.

[122] S. Herbold, "Training data selection for cross-project defect prediction," in *Proceedings of the 9th International Conference on Predictive Models in Software Engineering*, ser. PROMISE '13. New York, NY, USA: ACM, 2013, pp. 6:1–6:10.

[123] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the "imprecision" of cross-project defect prediction," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 61:1–61:11.

[124] C. C. Coello, G. B. Lamont, and D. A. Van Veldhuizen, *Evolutionary algorithms for solving multi-objective problems.* Springer Science & Business Media, 2007.

[125] D. E. Goldberg, *Genetic algorithms.* Pearson Education India, 2006.

[126] MATLAB, *version 8.5.0 (R2015a)*. Natick, Massachusetts: The MathWorks Inc., 2015.

[127] J. Krall, T. Menzies, and M. Davies, "Gale: Geometric active learning for search-based software engineering," *IEEE Transactions on Software Engineering*, vol. 41, no. 10, pp. 1001–1018, 2015.

[128] K. Deb, *Multi-objective optimization using evolutionary algorithms*. John Wiley & Sons, 2001, vol. 16.

[129] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software engineering*, vol. 26, no. 8, pp. 797–814, 2000.

[130] C. Andersson and P. Runeson, "A replicated quantitative analysis of fault distributions in complex software systems," *IEEE Transactions on Software Engineering*, vol. 33, no. 5, pp. 273–286, 2007.

[131] C.-Y. Huang, C.-S. Kuo, and S.-P. Luan, "Evaluation and application of bounded generalized pareto analysis to fault distributions in open source software," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 309–319, 2014.

[132] C.-S. Kuo and C.-Y. Huang, "A study of applying the bounded generalized pareto distribution to the analysis of software fault distribution," in *Industrial Engineering and Engineering Management (IEEM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 611–615.

[133] H. Zhang, "On the distribution of software faults," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, p. 301, 2008.

[134] G. Concas, M. Marchesi, A. Murgia, R. Tonelli, and I. Turnu, "On the distribution of bugs in the eclipse system," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 872–877, 2011.

[135] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," in *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4. ACM, 2002, pp. 55–64.

[136] M. K. Daskalantonakis, "A practical view of software measurement and implementation experiences within motorola," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 998–1010, 1992.

[137] T. G. Grbac and D. Huljenić, "On the probability distribution of faults in complex software systems," *Information and Software Technology*, vol. 58, pp. 250–258, 2015.

[138] M. E. Newman, "Power laws, pareto distributions and zipf's law," *Contemporary physics*, vol. 46, no. 5, pp. 323–351, 2005.

[139] M. Mitzenmacher, "Dynamic models for file sizes and double pareto distributions," *Internet Mathematics*, vol. 1, no. 3, pp. 305–333, 2004.

[140] A. B. Downey, "The structural cause of file size distributions," in *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001. Proceedings. Ninth International Symposium on.* IEEE, 2001, pp. 361–370.

[141] K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra, "Empirical study of object-oriented metrics." *Journal of Object Technology*, vol. 5, no. 8, pp. 149–173, 2006.

[142] R. J. Boik, "A priori tests in repeated measures designs: Effects of nonsphericity," *Psychometrika*, vol. 46, no. 3, pp. 241–255, 1981.

[143] G. C. Cawley and N. L. Talbot, "On over-fitting in model selection and subsequent selection bias in performance evaluation," *Journal of Machine Learning Research*, vol. 11, no. Jul, pp. 2079–2107, 2010.

[144] S. Roberts and H. Pashler, "How persuasive is a good fit? a comment on theory testing." *Psychological review*, vol. 107, no. 2, p. 358, 2000.

[145] H. Akaike, "Information theory and an extension of the maximum likelihood principle," in *Selected Papers of Hirotugu Akaike.* Springer, 1998, pp. 199–213.

[146] H. Akaike, "A new look at the statistical model identification," *IEEE transactions on automatic control*, vol. 19, no. 6, pp. 716–723, 1974.

[147] G. Schwarz *et al.*, "Estimating the dimension of a model," *The annals of statistics*, vol. 6, no. 2, pp. 461–464, 1978.

[148] H. Bhat and N. Kumar, "On the derivation of the bayesian information criterion," *School of Natural Sciences, University of California*, 2010.

[149] R. E. Kass and A. E. Raftery, "Bayes factors," *Journal of the american statistical association*, vol. 90, no. 430, pp. 773–795, 1995.

[150] H. Jeffreys, *The theory of probability.* Oxford University Press, Oxford, 1939.

[151] A. E. Raftery, "Bayesian model selection in social research," *Sociological methodology*, pp. 111–163, 1995.

[152] E. J. Hannan and B. G. Quinn, "The determination of the order of an autoregression," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 190–195, 1979.

[153] M. Mitzenmacher, "A brief history of generative models for power law and lognormal distributions," *Internet mathematics*, vol. 1, no. 2, pp. 226–251, 2004.

[154] G. Concas, M. Marchesi, S. Pinna, and N. Serra, "Power-laws in a large object-oriented software system," *IEEE Transactions on Software Engineering*, vol. 33, no. 10, pp. 687–708, 2007.

[155] R. Shatnawi and Q. Althebyan, "An empirical study of the effect of power law distribution on the interpretation of oo metrics," *ISRN Software Engineering*, vol. 2013, 2013.

[156] I. Herraiz, D. Rodriguez, and R. Harrison, "On the statistical distribution of object-oriented system properties," in *2012 3rd International Workshop on Emerging Trends in Software Metrics (WETSoM)*. IEEE, 2012, pp. 56–62.

[157] D. Vose, "Fitting distributions to data," 2010.

[158] P. Louridas, D. Spinellis, and V. Vlachos, "Power laws in software," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 18, no. 1, p. 2, 2008.

[159] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero, "Understanding the shape of java software," in *ACM Sigplan Notices*, vol. 41, no. 10. ACM, 2006, pp. 397–412.

[160] L. A. Adamic and B. A. Huberman, "Zipf's law and the internet," *Glottometrics*, vol. 3, no. 1, pp. 143–150, 2002.

[161] P. L. Li, M. Shaw, J. Herbsleb, B. Ray, and P. Santhanam, "Empirical evaluation of defect projection models for widely-deployed production software systems," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 6. ACM, 2004, pp. 263–272.

[162] A. Murgia, G. Concas, M. Marchesi, R. Tonelli, and I. Turnu, "An analysis of bug distribution in object oriented systems," *arXiv preprint arXiv:0905.3296*, 2009.

[163] V. Pareto and A. N. Page, "Translation of manuale di economia politica ("manual of political economy")," *AM Kelley*, 1971.

[164] R. Gibrat, *Les inégalités économiques: applications: aux inégalitês des richesses,*

à la concentration des entreprises, aux populations des villes, aux statistiques des familles, etc: d'une loi nouvelle: la loi de l'effet proportionnel. Librairie du Recueil Sirey, 1931.

[165] M. Jureczko, "Significance of different software metrics in defect prediction," *Software Engineering: An International Journal*, vol. 1, no. 1, pp. 86–95, 2011.

[166] J. Pickands III, "Statistical inference using extreme order statistics," *the Annals of Statistics*, pp. 119–131, 1975.

[167] C. Kolassa, D. Riehle, and M. A. Salim, "A model of the commit size distribution of open source," in *International Conference on Current Trends in Theory and Practice of Computer Science*. Springer, 2013, pp. 52–66.

[168] R. Wheeldon and S. Counsell, "Power law distributions in class relationships," in *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*. IEEE, 2003, pp. 45–54.

[169] F. E. Pani and G. Concas, "Stochastic models of software development activities," in *WSEAS International Conference. Proceedings. Recent Advances in Computer Engineering Series*, no. 7. WSEAS, 2012.

List of Publications

Peer-reviewed Journals and Conferences

1. C K Shriram, K Muthukumaran, N L Bhanu Murthy "Empirical Study on the Distribution of Bugs in Software Systems". **International Journal of Software Engineering and Knowledge Engineering. World Scientific Publishing.** (to appear)

2. Swapnil Shukla, T Radhakrishnan, K Muthukumaran, N L Bhanu Murthy "Multi-objective cross-version defect prediction". **Soft computing. Springer-Verlag Berlin Heidelberg 2016.** (article in press)

3. K Muthukumaran, Suri Srinivas, Aruna Malapati, N L Bhanu Murthy "Software defect Prediction using Augmented Bayesian Networks". *8th International Conference on Soft Computing and Pattern Recognition 2016, Dec 19-21 (pp. 279-293).* **Volume 614, Advances in Intelligent Systems and Computing. Springer, Cham.**

4. K Muthukumaran, Amrita Dasgupta, Shirode Abhidnya, N L Bhanu Murthy "On the Effectiveness of Cost Sensitive Neural Networks for Software Defect Prediction". *8th International Conference on Soft Computing and Pattern Recognition 2016, Dec 19-21 (pp. 557-570).* **Volume 614, Advances in Intelligent Systems and Computing. Springer, Cham.**

5. Pranav Ramarao, K Muthukumaran, Siddharth Dash and N L Bhanu Murthy "Impact of Bug Reporter's Reputation on Bug-fix Times" 2016. ***2016 International Conference on Information Systems Engineering (pp. 57-61)***. **IEEE. Los Angeles, USA.**

6. K Muthukumaran, N L Bhanu Murthy, Karthik Reddy and Prateek Talishetti "Testing and Code Review Based Effort-Aware Bug Prediction Model". *In Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (pp. 17-30).* **Volume 653, Studies in Computational Intelligence. Springer International Publishing.**

7. K Muthukumaran, Choudhary A, N L Bhanu Murthy "Mining Github for Novel Change Metrics to Predict Buggy Files in Software Systems". In ***2015 International Conference on Computational Intelligence and Networks*** 2015 Jan (pp. 15-20). **IEEE. Bhubaneshwar, India.**

8. K Muthukumaran, Rallapalli A, N L Bhanu Murthy "Impact of Feature Selection Techniques on Bug Prediction Models". In Proceedings of the ***8th India Software Engineering Conference*** 2015 Feb 18 (pp. 120-129). **ACM. Bangalore, India.**

9. K Muthukumaran, N L Bhanu Murthy, Reddy GK, Aruna M "Comparative Study on Effectiveness of Standard Bug Prediction Approaches". In Proceedings of the ***5th IBM Collaborative Academia Research Exchange Workshop*** 2013 Oct 17 (Article No. 9). **ACM. New Delhi, India. (Best Paper Award)**

Prediction and Probability Distribution of Defects in Software Systems

THESIS

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

By

MUTHUKUMARAN K.

ID. No. 2011PHXF0415H

Under the Supervision of

Prof. N L Bhanu Murthy

Co-Supervision of

Dr. Aruna Malapati

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

2017

6 Conclusion and Future Work

Defect prediction research has been evolving quite rapidly but its applicability to IT industry is far from reality. The efficacy of defect prediction models built with various combinations of metrics, algorithms and performance measures has been investigated in this work. Defect prediction models built with proposed approaches are compared with existing approaches.

6.1 Contributions and Findings

The goal of our work is to progress defect prediction research with the pre-processing techniques, novel features, Bayesian learning techniques, multi-objective optimization techniques and comparative study of classification algorithms. We have also made an attempt to model the probability distribution of past defects and object-oriented metrics. We summarize the contributions of our work as follows.

1. Our investigation on the impact of feature selection on defect prediction establishes that prediction accuracies can be enhanced significantly by applying feature selection techniques during pre-processing steps. We encourage to use greedy based algorithms for feature selection as they stand out and perform much better than the other competent algorithms. We re-emphasize that there may not be any common metrics amongst optimal feature subsets of similar projects. We have also found out that the impact of feature selection algorithm is independent of the underlying machine learning algorithms used for the classification.

2. We define a modus operandi to extract some popular change metrics from the Eclipse repository on Github, which can be generalized for any open-source Github repository. We have proposed couple of new change metrics, namely, Entropy and Mean Period of Change to figure out whether commits are uniformly distributed over the time-line between the two consecutive releases. We establish that a source file which was last changed in the early periods of the timeline and has been free of changes for the remaining periods can be considered relatively stable, as compared to a source file for which period of change concentration is localized closer to the date of release. We have used Logistic Regression, CART Decision Tree, Gaussian Naïve Bayes and Naïve Bayes Tree algorithm to build prediction models using the proposed and existing change metrics. We have found that Naïve Bayes Tree perform better than other commonly used algorithms for defect prediction problem.

3. Our initial experiments confirmed that Naïve Bayes classifier is one of the best performing classification techniques in defect prediction. It assumes conditional independence of features and for the defect prediction problem which is not entirely true. For example, lines of code (LOC) and number of operands of a source file can't be taken as independent. It motivated us to relax these conditional independence and to check whether there is an improvement in performance of classifiers. We defined augmentation as follows.

   A simple Bayesian Network which doesn't not have an arc from class node to every other node, is not an augmented. Bayesian Network is augmented if it has an arc from class node to every other node. For our experiments we have augmented Bayesian Network structures learnt using HC, TABU, GS, MMHC and RSMAX2.

   First, we have compared the performance of defect prediction models built using simple Bayesian Network classifiers with other traditional machine learning classifiers. In terms of AUC and H-Measure, Random Forest classifier is found to be better performing classifier than Bayesian Network classifiers. However, on defect prediction models built with our proposed augmented Bayesian Network structure, we observe that RSMAX2 and Grow-Shrink are consistently better, and there is no significant difference in performance between these classifiers and Random Forest. When we

compare classifiers using different cost ratio distributions leading to Risk Averse and Delay Averse scenarios, RSMAX2 and Grow-Shrink classifiers are performing very close to Random Forests and are found better than Naïve Bayes classifier.

4. The cost of fixing a software defect varies with the phase in which it is uncovered. Defects found during post-release phase costs much more than the defect that is uncovered in pre-release phase. Classifying a defective source file to be non-defective will result in a defect during post release phase and non-defective file to be a defective file will result in wastage of effort (due to QA activities). These two costs are called misclassification costs. Such misclassification cost has not been given much importance while comparing defect prediction models. Hence, we find it interesting to evaluate defect models using cost centric measure - Normalized Expected Cost of Misclassification (NECM).

   Though there are few studies that consider NECM as a model evaluation measure, they have built models using either cost sensitive neural networks or traditional machine learning algorithms. As there is no comprehensive comparative study on the effectiveness of defect prediction models which are built using cost sensitive neural networks and traditional machine laerning classifiers, we find it interesting to build models using both the approaches and compared their performances using NECM.

   Our experiments confirm Random Forest as an outperforming algorithm in terms of NECM closely followed by Logistic Regression and Bayesian Network. Random Forest performs significantly better than cost sensitive boosting Neural Network methods. We also observe that among the cost sensitive boosting Neural Network algorithms, Cost Sensitive Boosting of Neural Networks using Threshold Moving(CSBNN-TM) perform better than Cost Sensitive Boosting of Neural Networks with Weight Updation-I(CSBNN-WU1) and Cost Sensitive Boosting of Neural Networks with Weight Updation-II(CSBNN-WU2).

5. Researchers proposed effort-aware defect prediction models wherein files are ranked not only based on their defect-proneness but also on the effort that is required to

perform quality assurance activities like testing, code review etc. on defect prone files. Previous studies, considered Cyclomatic Complexity as a metric for the effort required irrespective of the assurance activity that will be undertaken during post prediction phases. We believe that effort-aware models are sensitive to the type of quality assurance activity one undertakes. We have proposed effort-aware defect prediction models that will consider the effort required to perform the specific quality assurance activities like code review and testing. We have considered lines of code and the number of testcases as our measures of effort for code review and testing respectively. We have empirically proven the supremacy of these models over the existing effort-aware model built with Cyclomatic Complexity.

- Our results indicate that the testing based effort-aware models perform significantly better than generic effort aware models.

- The code review based effort-aware model performs slightly better than generic model individually for each project but we could not find significant difference in models performance.

6. In multi-objective optimization approach we have formulated two problems with two sets of contrasting objectives as follows. M1: Maximize effectiveness of the model and Minimize misclassification cost and M2: Maximize effectiveness of the model and Minimize LOC cost.

   We have compared multi-objective logistic regression models with four single objective algorithms for cross version defect prediction. Our results indicate following benefits of multi-objective approach:

   - The multi-objective defect prediction model (M1) is able to identify more defects at the same or lesser misclassification cost incurred by all four single objective defect prediction models.

   - The multi-objective defect prediction model (M2) is able to identify more defect

prone classes at the same or lesser LOC cost than the cost incurred with single objective algorithms.

We recommend multi-objective approach for cross version defect prediction problems against traditional single objective approaches.

7. Understanding the distributions of CK metrics will help us in predicting the evolutions and efforts required to maintain such systems. Previously it has been shown that these metrics frequently follow power law. As these metrics are also shown to follow Log-normal or Gamma distribution occasionally, we have studied more number of software systems and applied more distribution models to infer more generalizable results

   In our attempt to model six object oriented metrics, we observe that WMC, DIT LCOM and NOC metrics follow the Log-normal distribution. RFC follows four of the distributions except the Log-normal distribution and CBO does not follows any distribution better than other distributions. We have also found that LOC metrics follow GPD and Weibull distributions.

8. Studies have proposed the Pareto distribution to model bugs in software systems. However, few other probability distributions such as the Weibull, Bounded Generalized Pareto, Double Pareto, Log Normal and Yule-Simon distributions have also been proposed to model bugs. The contemporary work on choosing the most appropriate model for the underlying bug distribution relies on parameter estimation methods and goodness-of-fit statistics. In this work we seek to strengthen the model selection methodology and make it less vulnerable to threats such as over-fitting by making use of information criterion based approaches.

   - Experiments with open source data showed that the Double Pareto, Weibull and Bounded Generalized Pareto models are the top three best fitting models. However it is found out that DP models fare better than BGPD and WD models though not significantly. It holds good for all three measures- BIC, AIC and HQIC used in our study. The experiments based on Bayes factor and BIC

relative fit values reveal that DP model is found to be better than WD model and BGPD model with Very Strong or Positive evidence in nearly 70% of open source projects.

- We also find that Double Pareto model significantly better than all other five models in experiments conducted on proprietary software projects. Similar results are observed irrespective of the three measures- BIC, AIC and HQIC used in our study. The experiments based on Bayes factor and BIC relative fit values reveal that DP model is found better than WD model and BGPD model with Very Strong or Positive evidence in nearly 100% of proprietary software projects. Hence we confidently conclude that the Double Pareto distribution fits bugs better than other distributions for propriety software systems.

6.2 Future Work

Software Analytics is gaining momentum as a result of involved empirical research in enhancing quality and productivity of software engineering activities. There have been rigorous research efforts in the areas of defect prediction, bug localization and effort estimation by making use of historical data. We recognize following problem areas where advanced machine learning and search-based software engineering techniques can improve the state-of-art and define them as our future work.

Bug-fix time prediction We find bug-fix time prediction as an interesting problem with lots of advantages to industry. In the event of any reported bug, generally the project management team approaches experienced team members to get an estimate of the bug-fix time and the response would be very much a personalized and subjective estimate. The accurate prediction of bug-fix times is useful in planning and management of resources that results in the lessened cumulative bug fix time. There have been attempts to solve defect prediction problem with machine learning approaches. Similar techniques can be leveraged to solve the problem of predicting bug-fix times.

By making use of the historical data about bug-fix times of previous bugs of the project, prediction models can be built to predict bug-fix time of future bugs.

In our initial attempt to predict bug-fix time, we have made use of three major features: report title, description and the reputation of the bug reporter. We introduce a new feature, the score of bug reporter, to predict bug fix time. We define the score or reputation of a reporter as follows: "Number of bugs reported by a reporter that were eventually fixed". For example, if a reporter logs 50 bugs out of which 30 are fixed, score of the reporter is 30. Our intuition to solve this problem is "Given two reports which are similar in complexity, the bug reported by the reporter with higher score will have a lower fix time and vice versa". Our dataset contains 91 reporters and around 567 bugs extracted from Jira issue tracking system of JBoss Project.

The prediction models built with 'reputation of the bug reporter' as a feature and other existing features is found to be performing significantly better than other existing bug fix time prediction models. We are interested in finding more features and suitable machine learning approaches to build more accurate bug fix-time prediction models.

Software effort-estimation Effort-estimation has been a very difficult problem in fast changing development environments. Underestimating results in under-staffing, limiting quality assurance activities and missing deadlines. Past projects data can be used to predict the effort required for future projects. Features like application domain, project duration, project size, programming language and tools, function points, geographical locations and project team experience in building similar systems can be extracted and used for effort prediction. We are interested to propose novel features and applying state of art learning techniques to enhance prediction accuracies.