Incremental Development and Cost-based Evaluation of Software Fault Prediction Models

Yue Jiang

Dissertation submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in
Computer and Information Sciences

Dr. Bojan Cukic, Ph.D., Co-Chair
Dr. Tim Menzies , Ph.D., Co-Chair
Dr. Arun Ross, Ph.D.
Dr. Katerina Goseva-Popstojanova, Ph.D.
Dr. James Harner, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2009

Keywords: fault-proneness, prediction, requirement, design, software metrics machine learning, classification

ABSTRACT

Incremental Development and Cost-based Evaluation of Software Fault Prediction Models

Yue Jiang

It is difficult to build high quality software with limited quality assurance budgets. Software fault prediction models can be used to learn fault predictors from software metrics. Fault prediction prior to software release can guide Verification and Validation ($V\&V$) activity and allocate scarce resources to modules which are predicted to be fault-prone.

One of the most important goals of fault prediction is to detect fault prone modules as early as possible in the software development life cycle. Design and code metrics have been successfully used for predicting fault-prone modules. In this dissertation, we introduce fault prediction from software requirements. Furthermore, we investigate the advantages of the incremental development of software fault prediction models, and we compare the performance of these models as the volume of data and their life cycle origin (design, code, or their combination) evolution during project development. We confirm that increasing the volume of training data improves model performance. And that, models built from code metrics typically outperform those built using design metrics only. However, both types of models prove to be useful as they can be constructed in different phases of the life cycle. We also demonstrate that models that utilize a combination of design and code level metrics outperform models which use either one metric set exclusively.

In evaluation of fault prediction models, misclassification cost has been neglected. Using a graphical measurement, the cost curve, we evaluate software fault prediction models. Cost curves not only allow software quality engineers to introduce project-specific misclassification costs into model evaluation, but also allow them to incorporate module-specific misclassification costs into model evaluation. Classifying a software module as fault-prone implies the application of some verification activities, thus adding to the development cost. Misclassifying a module as fault free carries the risk of system failure, and is also associated with cost implications. Our results, through the analysis of more than ten projects from public repositories, support a recommendation to adopt cost curves as one of the standard methods for software fault prediction model performance evaluation.

Acknowledgements

I would like to thank my advisor, Dr. Bojan Cukic, for his guidance, advice, and continual encouragement. It is a pleasure to work under his supervision. Without him, this dissertation could not have come about. I would like to thank my other advisor, Dr. Tim Menzies, for his guidance, advice, and help in my study, research, and dissertation.

I would also like to thank my other committee members: Dr. Arun Ross, Dr. Katerina Goseva-Popstojanova, and Dr. James Harner for their help during my studies.

And finally, I thank my family members for their constant support, encouragement, and help.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Significant research has been conducted to study the detection of software modules which are likely to contain faults. The fault-proneness information not only points to the need for increased quality monitoring during the development but also provides important advice to assign verification and validation activities. A study has shown that software companies spend 50 to 80 percent of their software development effort on testing [58]. The identification of fault-prone modules might have a significant cost-saving impact on software quality assurance.

The earlier a fault is identified, the cheaper it is to correct it. Boehm and Papaccio advise that to fix a fault early in the lifecycle, it is cheaper by a factor of $50 - 200$ [12]. A panel of IEEE Metrics in 2002 found that fixing a severe software problem after delivery is often 100 times more expensive than that of fixing it early in requirement and design phase [78]. When it comes to software fault prediction, naturally, people would wonder whether metrics from the early lifecycle, such as requirement metrics and design metrics, are effective to predict software faults or not. In this dissertation, we will investigate:

- Q1: Are requirement metrics good fault-prone predictors for fault prediction models?
- Q2: Are design metrics good enough to be built into fault prediction models?
- Q3: Are code metrics good fault-prone predictors?

A wide range of software metrics have been proposed to be collected and used to predict

faults [31]. There are requirement metrics [38,40,59], design metrics [20,68,70,73,76,80,94], code metrics [17,33,45,62,67], and different social network metrics, for example, the developer information [85], messages about mailing list [55], and organizational structure complexity metrics [69]. Although there are various software metrics used in the prediction of fault-proneness, comparing the effectiveness of design and code metrics received limited attention. To the best of our knowledge, the paper by Zhao *et al.* is unique as it compares the performance of design and code metrics in the prediction of software fault content [91]. They compared fault prediction models built from design metrics, code metrics, and the combination of design and code metrics in the context of a large real-time telecommunication system. Their design metrics are a modified version of McCabe's cyclomatic complexity, extracted from Specification Description Language (SDL). They used regression equations to fit their three groups of metrics and R^2 statistic to evaluate the performance of the models. While their findings are based on the analysis of a single data set, we have access to more than 10 project data sets and we analyze the effect of design metrics, and code metrics such as:

- Q4: Do fault prediction models built from a combination of requirement, design, and code metrics provide better performance than models built from any metrics subset?

Besides metrics, how large is a training subset needed for building a meaningful software fault prediction model? This problem is seldom investigated although various sizes of training subsets are used in literature. For example, Menzies *et al.* used 90% data as the training subset in his work [62]. Lessmann *et al.* used $\frac{2}{3}$ data as the training subset in his study [53]. Guo *et al.* use $\frac{3}{4}$ data to build the training models [33]. In Jiang *et al.*'s work [40], 80% data is used to train fault prediction models. In this dissertation, we would like to explore the following problems:

- Q5: How large does a training subset needed to be in order to build meaningful fault prediction models?
- Q6: How large does a training subset needed to be in order to achieve statistically the best performance fault prediction models ?

When a software project evolves into its late lifecycle stages, i.e, testing phase, it is necessary to analyze misclassification costs of a project and even misclassification costs for module(s) in a project. The "traditional" software fault prediction models typically assume uniform misclassification cost. In other words, these models assume that the cost implications of wrongly

predicting a faulty module as fault free is the same as the cost of predicting that a fault free module may contain faults. In actuality, the cost implications of these two types of misclassifications are seldom equal in the real world. In high risk software projects, for example, safety-related spacecraft navigation instruments, the cost of mis-predicting a faulty module may have extreme consequence according to a loss of the entire mission. Therefore, in such projects significant resources are typically available for identifying and eradicating all faults because the cost of losing a mission is much higher. On the other hand, in low risk projects which aim to occupy a new market niche, time to market pressure may imply that only a minimal number of faulty modules can be analyzed. The cost of analyzing any significant number of misclassified fault free modules is, therefore, unacceptable.

An important factor that makes the identification of faulty modules a challenge is the reality that they usually form a minority class compared to fault free modules. We want to analyze problems associated with misclassification costs:

- Q7: How can we include misclassification costs in fault prediction models?
- Q8: How can we achieve the lowest misclassification cost for a given data set?
- Q9: From misclassification cost perspective, how can fault prediction models guide different risk level projects?
- Q10: How can we evaluate individual module's specific misclassification costs in fault prediction models?

The remainder of this dissertation is organized as follows. Chapter 2 discusses related work. First, after introducing a brief history, associated machine learning algorithms, and approaches in fault prediction models, we review the measurements used and the appropriate statistical tests applied in software fault-proneness prediction. Chapter 3 discusses fault prediction models built from requirement metrics, and the combination of requirement metrics and software module metrics (module metrics include design metrics and code metrics). Chapter 3 will address Question Q1 and partial of Q4. Chapter 4 presents our experiments and results on incremental development of software fault prediction models comparing $design$, and $code$ metrics, and their combination of both of them. Chapter 4 will answer Question Q2, Q3, Q4, Q5, and Q6. Chapter 5 describes our experiments and results to evaluate misclassification costs in fault prediction models and the cost curve to address project specific economic parameters. Chapter 5 will answer Question Q7, Q8, Q9, and Q10. Chapter 6 draws conclusions and proposes possible

further work.

Chapter 2

Related Work

Software quality engineering includes different quality assurance activities in the software development process such as testing, fault prevention, fault avoidance, fault tolerance, and fault prediction. Faults may be predicted before they cause failures. Once predicted, fault-prone modules are tested more intently than fault free modules. There are many benefits of software fault prediction such as more reasonable allocation of resources by focusing on faulty modules, improving test process, and eventually having a high quality system.

Many different techniques have been investigated in fault prediction including algorithms, statistical methods, software metrics, and software projects. Until now, researchers used different algorithms such as Genetic Programming, Decision Trees, Fuzzy Logic, Neural Networks, and Case-based Reasoning for fault prediction. Various metrics including design metrics, code metrics, test metrics, historical metrics, metrics extracted from social network associated with software products are used to predict software faults. All different kinds of statistical measurements are used to evaluate software fault prediction models including accuracy, probability of detection, Area Under the Curve of *ROC, F-Measure*, and *G-means* etc.

In this chapter, we will first introduce software fault prediction models including a brief history. Second, we will present different approaches in fault prediction models. Third, we will review various metrics used in fault prediction models. Fourth, different kinds of measurements in fault prediction models are presented. Fifth, we will briefly review the research works con-

ducted using MDP data sets. Finally, we will discuss what has been achieved by the current state-of-the-art techniques in fault prediction, and we also point out what needs to be done in the future.

2.1 Fault Prediction Modeling

The problem of predicting the quality of a software product before it is used is challenging. Many research efforts and different kinds of methods and models have been investigated. Fault prediction allows the tester to manipulate their resources more effectively and efficiently, which would potentially result in higher quality products and lower costs.

To the best knowledge of the author, the earliest fault prediction models have been built by Akiyama in 1971 [4]. In his study, Akiyama used four regression models to fit some simple metrics, i.e, lines of code (LOC) to estimate the total number of faults in a system at Fujitsu, Japan. Interestingly, fault prediction models based on regression models are still common and useful in the software engineering literature. Another early study has been conducted by Ferdinand [32] in 1974. Ferdinand proposed that the expected number of faults increases with the number of code segments.

Milestones were established by Halstead in 1975 [34] and McCabe in 1976 [60]. Halstead proposed a set of code size metrics, well known as Halstead complexity metrics in the software engineering community, to measure source codes. Halstead provided a solution to the most fundamental question in software "How big is a program (a project)?". Since then, Halstead metrics are not only used as a measurement for developing effort, testing effort, but also are used as a set of metrics to predict software faults. McCabe proposed another important set of metrics, referred to as McCabe's cyclomatic complexity [60]. McCabe's metrics represent the structure of software product by measuring the software product data-flow graphs and control-flow graphs (i.e, the number of decision points, the number of links, the number of nodes, the nesting depth). Nowadays, these sets of metrics are still the primary quantitative measures in use, and they are the foundation of software engineering. After the publication of Halstead and McCabe metrics, many software fault prediction models have utilized them [7,10,11,35,62,77].

With the appearance of Object-Oriented (OO) programming languages, another important

OO metrics, referred to as CK-metrics [20], were proposed by Chidamber and Kemerer in 1994. CK metrics are proved to be effective predictors for software faults [68, 80]. Halstead, McCabe, and CK metrics are all originally defined on source codes. Currently, other kinds of metrics are studied: for example, requirement metrics [40], design metrics [20], code churn [66], the historical data [74], the number of developers [85], and software product's social networks [69].

With the development of machine learning algorithms, such as decision trees [45], neural networks [46], and genetic algorithms [8] etc., software fault prediction models advanced at a faster pace.

2.1.1 Machine Learning

Machine learning is a process in which a machine (computer) improves its performance based on "rule" (pattern) learning from previous behaviors (data). There are two essential steps involved in machine learning. First, learning knowledge from existing data; secondly, predicting (suggesting) what to do in the future (on new data). Machine learning is successfully used in many different fields, such as medical diagnosis, bioinformatics, biometrics, credit card fraud, stock market, and software engineering. On the other hand, machine learning is also a multi-disciplinary science used by many different sciences, including statistics, artificial intelligence, information theory, computer science, control theory, biology, and other fields.

Major algorithms used in machine learning are decision trees, neural network, concept learning, genetic algorithm, reinforcement learning, and algorithms derived from them. All these algorithms have proved to be useful in the prediction of software faults. In this dissertation, we mainly use five different machine learning algorithms which will be introduced later: Naive Bayes, random forest, logistic regression, bagging, and boosting.

Naive Bayes

Naive Bayes is a classifier developed from the Bayes rule of conditional probability. In this dissertation, we are interested in whether a module contains a fault or not. Hence, we only consider binary classification for a module: faulty or fault-free (non-faulty). Let capital L

denoted the final class (L is faulty or non-faulty in our case). We use a set of metrics M = (M_1, M_2, ..., M_n) with n number of attributes (features, or variables) to predict the final classes L. Let $P(L)$ denote the prior probability of a module in class L given a training subset data. Let $P(M)$ denote the prior probability that the given training data is observed. Noted that, $P(M)$ is constant and will be eliminated in the final calculation step. Let $P(L|M)$ denote the probability of a new module is in class L in the condition of the training data. $P(L|M)$ is also called the posterior probability which reflects the confidence of final class after the training data is applied. $P(L|M)$ is what we are interested. Let $P(M_i|L)$ denote the probability of M_i metric (attribute) in class L for a given training subset. Thus, Bayes theorem is:

$$P(L|M) = \frac{P(L)\prod_{i=1}^{n} P(M_i|L)}{P(M)} \tag{2.1}$$

For example, we have a training subset shown as Table 2.1 which has 7 faulty modules and 14 fault-free modules. This training subset has two metrics, $LineofCode$ and $HalsteadVolume$. The last row in Table 2.1 shows a new module with 40 $LineofCode$ and $HalsteadVolume$ of 2555. Which class will the new module be classified into given the training subset?

First, we calculate mean, standard deviation, and the number of faulty and non-faulty modules of the given training subset, summarized in Table 2.2.

Because we deal with numeric attributes, the probability will be calculated according to the probability density function. The probability density function for a normal distribution with mean μ and standard deviation σ is given by the expression:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.2}$$

For the new module, with *LineOfCode = 40* and *HalsteadVolume=2555*, the probabilities of it belonging to faulty class and non-faulty class are calculated as follows.

$f(LineOfCode = 40|faulty) = \frac{1}{\sqrt{2\pi}*63.3} e^{\frac{(x-58.4)^2}{2*63.3^2}} = 0.00657$

$f(LineOfCode = 40|nonfaulty) = \frac{1}{\sqrt{2\pi}*10.2} e^{\frac{(x-14.1)^2}{2*10.2^2}} = 0.983$

$f(HalsteadVolume = 2555|faulty) = \frac{1}{\sqrt{2\pi}*2131.7} e^{\frac{(x-3684.7)^2}{2*2131.7^2}} = 0.000215$

$f(HalsteadVolume = 2555|nonfaulty) = \frac{1}{\sqrt{2\pi}*734.7} e^{\frac{(x-1240.3)^2}{2*734.7^2}} = 0.00269$

Table 2.1. An example of training subset

LineOfCode	HalsteadVolume	faulty
4	1126	FALSE
10	1140	FALSE
5	451	FALSE
16	2416	FALSE
11	1168	FALSE
7	669	FALSE
37	2881	FALSE
36	253	FALSE
12	1181	FALSE
13	816	FALSE
15	617	FALSE
12	1844	FALSE
14	1361	FALSE
5	1441	FALSE
7	461	TRUE
24	3029	TRUE
64	5581	TRUE
195	7140	TRUE
44	3104	TRUE
53	3533	TRUE
22	2945	TRUE
new module		
40	2555	?

Table 2.2. Summarized statistics of the example training subset.

	LineOfCode		HalsteadVolume		faulty	
	faulty	non-faulty	faulty	non-faulty	faulty	non-faulty
	7	4	461	1126	7	14
	22	5	3029	1140		
	24	5	5581	451		
	44	7	7140	2416		
	53	10	3104	1168		
	64	11	3533	669		
	195	12	2945	2881		
		12		253		
		13		1181		
		14		816		
		15		617		
		16		1844		
		36		1361		
		37		1441		
mean	58.4	14.1	3684.7	1240.3	7/21	14/21
std	63.3	10.2	2131.7	734.7		

Using these probabilities to predict the new module, we have:
Likelihood of faulty $=P(faulty)\frac{P(LineOfCode=40|faulty)*P(HalsteadVolumne=2555|faulty)}{P(M)}$
$=\frac{7}{21}*\frac{0.00657*0.000215}{P(M)}=\frac{4.72e-07}{P(M)}$
Likelihood of non-faulty $=P(nonfaulty)\frac{P(LineOfCode=40|nonfaulty)*P(HalsteadVolumne=2555|nonfaulty)}{P(M)}$
$=\frac{14}{21}*\frac{0.983*0.00269}{P(M)}=\frac{0.001763702}{P(M)}$

Which leads to the probabilities of the new module as follows:
Probability of faulty = $=\frac{4.72e-07}{4.72e-07+0.001763702}=0.00027$
Probability of non-faulty = $\frac{0.001763702}{4.72e-07+0.001763702}=0.99973$

Thus, using the Naive Bayes classifier, the new module with LineOfCode=40 and Halstead-Volume=2555 will be classified as fault free module.

NaiveBayes have been used extensively in fault-proneness prediction [39, 40, 62].

Logistic Regression

Regression analysis is used for explaining or modeling the relationship between a predicted variable (response variable, i.e, class L in our case) and one or more predictors, metrics M = (M_1, M_2, ..., M_n) ($n \geq 1$). When $n = 1$, it is called simple regression; when $n \geq 1$, it is called multiple regression (or multivariate regression). In our case, we have multiple predictors (multiple metrics), hence, our regression is multiple regression. When the predicted (response) variable is numeric, linear regression and logistic regression can be used. However, when the predicted variable is binomial (binary classification as fault or fault-free, denoted as 1 or 0), logistic regression will be adapted.

Referring back to the previous notation, $P(L|M)$ denotes the probability of a module in class L (faulty or non-faulty) given a set of metrics in a training subset. Then $P(faulty|M)$ would be the probability of a faulty module given a set of metrics. Because this is binary classification, then $1 - P(faulty|M)$ would be the probability of a fault-free module for a

given set of metrics. Abbreviated $P(faulty|M)$ as P, Logistic regression is defined as:

$$logit(P) = log\frac{P}{1-P}, with, P = \frac{1}{1+e^{-\beta_0-\beta_1 M_1-\beta_2 M_2-...-\beta_n M_n}} \tag{2.3}$$

where β_0 is the regression intercept and β_i is regression coefficient for Metric M_i ($i \geq 1$). Note that, in Equation 2.3, the interactions between variables are not considered. Using the training data in Table 2.1 as an example, we have $\beta_0 = 3.9455$, $\beta_1 = -0.0277$, and $\beta_2 = -0.0012$. The final model for the given training data in Table 2.1 is:

$$P = \frac{1}{1+e^{-3.9455+0.0277*LineOfCode+0.0012*HalsteadVolume}} \tag{2.4}$$

Taking the metrics of the new module into Equation 2.4, we have $P = 0.44$, that is, the probability of this new module being faulty is 44%. Then the probability of it being fault-free is $1 - 44\% = 56\%$. Using 50% as the classification boundary, the new module is classified as fault-free.

Logistic regression analysis is widely used to predict fault-proneness modules in software engineering [66, 67, 69]. The advantage of regression analysis is its flexibility of combining it to other algorithms, for example, a classification tree, to form a regression tree algorithm.

Bootstrap

Three ensemble classifiers, the random forest, bagging, and boosting all utilize the bootstrap method. Therefore, before we introduce them, we need to introduce the bootstrap method. The bootstrap method is based on sampling with replacement to form a training subset. The bootstrap method works as follows. Given a data set with n modules, bootstrap samples n modules from the original data set with replacement to act as training data. The sampled n modules will be repeated and there must be some modules in the original data set that are not picked into the new sampled data. Those unpicked modules will be left as testing data.

Each module in a data has the probability of $\frac{1}{n}$ being picked into a bootstrapped data. Thus, the probability of not being picked is $1 - \frac{1}{n}$. After n times of picking, the probability of a module which is not picked is

$(1 - \frac{1}{n})^n \approx e^{-1} = 0.368$
with e as natural logarithms (≈ 2.7183). Therefore, for each bootstrap, there are 36.8% of data that are not picked and is used as testing data. The training data, although it has n instances, only contains 63.2% of the original data. Thus, the bootstrap method is often called 0.632 bootstrap. The bootstrap method usually is repeated several times, and then the result will be determined by majority vote (in our case) or by average when the prediction outcome is numeric.

Random Forest

Random Forest is a decision tree-based ensemble classifier which is designed by Breiman [14]. As implied from its name, it builds an ensemble, i.e., the forest of classification trees using the following strategy:

1. Each decision tree has been built from a different bootstrap sample.
2. At each node, a subset of variables are randomly selected to split the node.
3. Each tree is grown to the largest extent possible without pruning.
4. When all trees (at least 500 as recommended by its designer) in the forest are built, test instances are fitted into all the trees and a voting process takes place.
5. The forest selects the classification with the most votes as the prediction of new instances.

Random forest as a machine learning classifier provides many advantages [14]. It also automatically generates the importance of each attribute in the process of classification. Hence, it is a good algorithm to perform Feature Subset Selection. In empirical studies, Random forest usually is one of the best learners in fault-proneness prediction [33, 39, 40].

Bagging

Bagging stands for *bootstrap aggregating*. It performs using the following procedure:

1. For a training subset with n number of samples, it resamples n number of samples from this training subset with replacement using the bootstrapping method.
2. Building a model using a base classifier (default as REPtree in Weka)
3. For a testing sample, the model predicts a class (faulty or fault-free, in our case). And the predicted class is stored for later usage.
4. Repeat the above steps for multiple times (default value is 10 in Weka)
5. Using the most predicted class for a testing sample.

We can see that bagging relies on an ensemble of different based models. Bagging shares a lot of similarity with random forest,

- The training data is resampled from the original data set with replacement.
- It is an ensemble of base classifiers. Bagging can have any kinds of basic algorithm, while random forest is only based on tree algorithm.
- The final prediction is voted by majority.

If bagging uses the same base classifier as the random forest, these two algorithms are essential the same. According to Witten and Frank [87], bagging typically performs better than single method models and almost never significantly worse. And it has good performance in our experiments [39, 40] too.

Boosting

The boosting algorithm combines multiple models by explicitly seeking models that complement one another [87]. The boosting as an ensemble algorithms has many similarities with bagging (or random forest):

- first of all, it is an ensemble of multiple based models.
- the ensemble of models are built on resampling with replacement from training subset.
- the prediction outcome is determined by voting.

But there are differences between bagging and boosting.

- After the learning process is begun, the base models for boosting can be built on different algorithms, while the base algorithm of bagging never changed once it is determined.
- The base classifier for boosting is iterative, that is, each new model is influenced by the performance of those built previously, while bagging builds each new base classifier separately.
- The final voting outcome is voted differently: boosting weights each model's prediction according to its performance while bagging treats every vote equally. By weighting instances, the boosting hopes to concentrate on a particular set of instances, for example, those with high weight.

The random forest, bagging and boosting ensemble learners have been developed in the past decade and have proved to have astonishingly good performance in data mining [87]. In our experiments of fault prediction modeling, they demonstrate superior performance than other learners [39, 40].

2.2 Different Approaches for Fault Prediction Models

A project manager needs to make sure a project met its timetable and budget plan without loss of quality. In order to help project managers to make a decision, fault prediction models play an important role to allocate software quality assurance resources. Existing research in software fault-prone models focus on predicting faults from these two perspectives:

- **The number of faults or fault density:** This technique predicts the number of faults (or fault density) in a module or a component. Project managers can use these predictions to determine the process of software timetable and $V\&V$ resources allocation. These models typically use data from historical versions (or pre-release parts) and predict the faults in the new version (or the new developed parts). For example, the fault data from historical releases can be used to predict faults in updated releases [36, 51, 74].
- **Classification:** Classification predicts which modules (components) contain faults and which modules don't. The goal of this kind of prediction distinguishes fault free subsystems from faulty subsystems. This allows project managers to focus resources to fix faulty

subsystems. Many software fault prediction models are this type of study [7,53,62,63,77] including this dissertation.

2.2.1 Supervised Learning

There are two methods to classify fault-prone modules from fault free modules: supervised learning and unsupervised learning. Both of them are used in different situations. When a new system without any previous release is built, for the new developed subsystems (modules, components, or classes), in order to predict fault-prone subsystems, unsupervised learning needs to be adopted. After some subsystems are tested and put into function, these pre-release subsystems can be used as training data to build software fault prediction models to predict new subsystems. This is the time when supervised learning can be used. The difference between supervised and unsupervised learning is the status of training data's class, if it is unknown, then the learning is unsupervised, otherwise, the learning is supervised learning.

A learning is called supervised because, the method operates under supervision provided with the actual outcome for each of the training examples. Supervised learning requires known fault measurement data (i.e, the number of faults, fault density, or fault-prone or not) for training data. Usually, Fault measurement data from previous versions [74], pre-release [65], or similar project [48] can act as training data to predict new projects (subsystems).

Most research reported in fault prediction is supervised learning including experiments in this dissertation. The learning result of supervised is easier to judge than unsupervised learning. This probably helps to explain why there are abundant reports on supervised learning in the literature and there are few reports on unsupervised learning. Like most research conducted in fault prediction, a data with all known classes is divided into training data and testing data: the classes for training data are provided to a machine algorithm, the testing data acts as the validation set and is used to judge the training models. The success rate on test data gives an objective measure of how well the machine learning algorithm performs. When this process is repeated multiple times with randomized divided training and testing sets, it is the standard data mining practice, called cross-validation. Like other research in data mining, randomization, cross-validation, and bootstrapping are often the standard statistical procedures for fault prediction in software engineering.

2.2.2 Unsupervised Learning

Sometimes we may not have fault data or we may have very little modules having previous fault data. For example, if a new project is developing or previous fault data is not collected, supervised learning approaches do not work because we do not have labeled training data. Therefore, unsupervised learning approaches such as clustering methods may be applied. However, research for this approach is seldom reported. As far as the author is aware, Zhong *et al.* [92, 93] are the first group who investigate this in fault prediction. They use Neural-Gas and K-means clustering to class software modules into several groups, with the help of human experts to identify fault-prone or not fault-prone to each groups. Their results indicate promising potentials for this unsupervised learning method. Interestingly, their two data sets are JM1 and KC2 from NASA MDP repository and they only use 13 metrics among 39 (or 40) total metrics.

2.3 Metrics Used in Software Fault Prediction Models

In this section, we will introduce various metrics which have been used to predict software faults. These metrics include requirement metrics, design metrics, code metrics, and other different kinds of metrics.

Requirement metrics have been used to predict fault prone software modules [38, 40, 59]. Malaiya *et al.* examined the relationship between requirement changes and fault density and found a positive correlation [59]. Javed et al. [38] investigate the impact of requirement instability on software faults. In 4 industrial e-commerce projects and 30 releases they found: (1) a significant relationship between pre/post release change requests and overall software faults; (2) insufficient and inadequate client communication during system design phase cause requirements changes and, consequently, software faults.

One of the earliest studies of design metrics in fault prediction models was conducted by Ohlsson and Alberg [73]. They predicted fault-prone modules prior to coding in a Telephone Switches system of 130 modules at Ericsson Telecom AB. Their design metrics are derived from graphs where functions and subroutines in a module are represented by one or more graphs. These graphs, called Formal Description Language (FDL) graphs, offer a set of direct and indi-

rect metrics based on the measures of complexity. Examples of direct metrics are the number of branches, the number of graphs in modules, the number of possible connections in a graph, and the number of paths from input to the output signals etc. Indirect metrics are the metrics calculated from the direct metrics such as McCabe cyclomatic complexity.

The suite of object oriented (OO) metrics, referred as CK metrics, has been first proposed by Chidamber and Kemerer [20]. They proposed six CK design metrics including Weight Method Per Class (WMC), Number of Children (NOC), Depth of Inheritance Tree (DIT), Coupling Between Object class (CBO), Response For a Class (RFC), and Lack of Cohesion in Methods (LCOM). Basili et al. [9] were among the first to validate these CK metrics using 8 C++ systems developed by students. They demonstrated the benefits of CK metrics over code metrics. In 1998, Chidamber, Darcy and Kemerer explored the relationship between the CK metrics and productivity, rework effort or design effort [19]. They show that CK metrics have better explanatory power than traditional code metrics. Predicting fault-prone software modules using metrics from design phase has recently received increased attention [62, 74, 76, 88]. In these studies, metrics are either extracted from design documents or, like in our work, by mining the source code using the reverse engineering techniques. Subramanyam and Krishnan predict faults from three design metrics: Weight Method Per Class (WMC), Coupling Between Object Class (CBO), and Depth of Inheritance Tree (DIT) [80]. The system they study is a large B2C e-commerce application suite developed using C++ and Java. They showed that these design metrics are significantly related to faults and that faults are strongly related to the language used. Nagappan, Ball and Zeller [68] predict component failures using OO metrics in five Microsoft software systems. Their results show that these metrics are suitable to predict software faults. They also show that the predictors are project specific, the suggestion also mentioned by Menzies et al. [62].

Recovering design from source code has been a hot topic in software reverse engineering [5, 6, 16]. Systa [81] recovered UML diagrams from source code using static and dynamic code analysis. Tonella and Potrich [82] were able to extract sequence diagrams from source code. Briand et al. demonstrated sequence diagrams, conditions and data flow can be reverse engineered from Java code through transformation techniques [15].

Recently, Schroter, Zimmermann, and Zeller [76] applied reverse engineering to recover design metrics from source code. They used 52 ECLIPSE plug-ins and found usage relation-

ships between these metrics and past failures. The relationship they investigate is the usage of import statements within a single release. The past failure data represents the number of failures for a single release. They collected the data from version archives (CVS) and bug tracking systems (BUGZILLA). They built predictive models using the set of imported classes of each file as independent variables to predict the number of failures of the file. The average prediction accuracy of the top 5% is approximately 90%. Zimmermann, Premraj and Zeller [94] further investigate ECLIPSE open source, extract object oriented metrics along with static code complexity metrics and point out their effectiveness to predict fault-proneness. Their data set is now posted in the PROMISE [13] repository. Neuhaus, Zimmermann, Holler and Zeller examine Mozilla code to extract the relationship of imports and function calls to predict software components vulnerabilities [70].

Besides requirement metrics, design metrics, and code metrics, various other measures have been used to predict fault prone software modules. Historical project characteristic, developer information and social networks have all been reported as effective predictors. Ostrand, Weyuker, and Bell [74] predict the files most likely to contain the largest numbers of faults in the next release using modification history from previous release and the code in the current release. Illes-Seifert and Paech investigate the relationship between software history characteristics and the number of faults [36]. After analyzing 9 open source Java projects, they conclude that some history characteristics, such as the number of changes and the number of distinct authors performing changes to a file, highly correlate with faults. Code churn, defined as the amount of code change taking place within a software unit [66], also called cached history [51], was also reported as an effective predictor of faults.

The role of developer social networks is currently receiving significant attention. Weyuker, Ostrand, and Bell [85] found that the addition of developer information improves the accuracy of fault prediction models. Li et al. analyzed 139 metrics collected from software product, development, deployment, usage, software and hardware configurations in OpenBSD [55]. They found that the number of messages to the technical discussion mailing list during the development period is the best predictor of the number of field faults. Nagappan et al. [69] collect 8 organizational structure complexity metrics which relate code binary to the organizational social networks, i.e, the number of engineers, the number of ex-engineers, edit frequency of source code, and organization intersection factors to predict failure-proneness. They compare

this model to models which use five groups of traditional metrics (code churn, code complexity, code coverage, dependency, and pre-release bugs). The use of organizational structure complexity metrics appears to hold a significant promise for fault prediction.

2.4 Model Evaluation Techniques

Many statistical techniques have been proposed to predict fault-proneness of program modules in software engineering. Choosing the "best" candidate among many available models involves performance assessment and detailed comparison. But these comparisons are not simple due to the applicability of varying performance measures. Classifying a software module as fault-prone implies the application of some verification activities, thus adding to the development cost. Misclassifying a module as fault free carries the risk of system failure, also associated with cost implications. Methodologies for precise evaluation of fault prediction models should be at the core of empirical software engineering research, but have attracted sporadic attention. In this chapter we overview model evaluation techniques. In addition to many techniques that have been used in software engineering studies before, we introduce and discuss the merits of cost curves.

2.4.1 Methodology for 2-class Classification

We are interested in the identification of fault-proneness information of software, that is, which parts contain faults. Requirement metrics, design metrics, code metrics, and the fusion of these metrics serve as predictors. And the predicted variable is whether a fault is detected in the given software units. Throughout the measurement in this dissertation, the confusion matrix of 2-class classification serves as the foundation of measurement. All kinds of measurement used in this disseartation are derived from the confusion matrix. Figure 2.1 describes the confusion matrix of prediction outcomes. The confusion matrix has four categories: True positives (TP) are modules correctly classified as faulty modules. False positives (FP) refer to fault-free modules incorrectly labeled as faulty modules. True negatives (TN) correspond to fault-free modules correctly classified as such. Finally, false negatives (FN) refer to faulty modules incorrectly classified as fault-free modules.

predicted	Real data		
		Fault	**No fault**
	Fault	TP	FP
	No fault	FN	TN

Figure 2.1. A confusion matrix of prediction outcomes.

$$
\begin{aligned}
PD = recall = & \quad \frac{TP}{TP+FN} \\
PF = & \quad \frac{FP}{FP+TN} \\
acc = & \quad \frac{TP+TN}{TN+TP+FP+FN} \\
p = precision = & \quad \frac{TP}{FP+TP} \\
sp = specificity = 1 - PF = & \quad \frac{TN}{TN+FP} \\
G - mean\_1 = & \quad \sqrt{PD * p} \\
G - mean\_2 = & \quad \sqrt{PD * sp} \\
F - measure = & \quad \frac{(\beta^2+1)*p*PD}{\beta^2*p+PD} \\
J\_coeff = & \quad PD - PF
\end{aligned}
$$

The Probability of Detection (PD), also called recall or specificity in some literature [33, 62]), is defined as the probability of the correct classification of a module that contains a fault. The Probability of False alarm (PF) is defined as the ratio of false positives to all non-faulty modules. Accuracy (acc) is widely used in all kind of data mining classifiers. It is the ratio of sum of true positive and true negative to total number of units studied, that is, the overall ratio of correctly predicted units. Thus, acc gives the overall classification performance of a classifier. However,it ignores the data distribution and cost information. Therefore, it can be a misleading criterion as faulty modules are likely to represent a minority of the modules in the dataset [57]. Precision (p) is the proportion of the predicted faulty software units that actually contain fault(s). It is shown that precision is not a suitable measurement to evaluate software quality model alone [61]. Specificity shows how well a classifier correctly identifies the non-faulty cases, that is, it is the proportion of true negatives of all non-faulty in a project.

Compared to overall accuracy, F-measure [54], geometric mean (G-mean) [52], and J coefficient (J_coeff) [89] tell a more honest story about model's performance. $G - mean_1$ is the

square root of the product of sensitivity (PD) and precision. $G-mean_2$ is the square root of the product of PD and specificity. In software quality prediction, it may be critical to identify as many fault-prone modules as possible (that is, a high sensitivity or PD). If two methodologies produce the same or similar PD, we prefer the one with higher specificity. So, G-mean2 is the geometric mean of two accuracies: one for the majority class and another for the minority class. Precision tells us among all the faulty modules the model may have discovered, how many are actually faulty. Both $G-mean_1$ and F-measure integrate PD and $precision$ in a performance index. F-measure offers more flexibility by including a weight factor β which allows us to manipulate the weight (cost) assigned to PD and precision. Parameter β may assume any non-negative value. If we set β to 1, equal weights are given to PD and precision. The weight of PD increases as the value of β increases.

Youden proposed J coefficient (J_coeff) to evaluate measurements in medical sciences [89]. El-Emam *et al.* were the first to use J coefficient to compare classification performance in software engineering [27]. The formula is $J\_coeff = sensitivity + specificity - 1 = PD - 1 + specificity = PD - (1 - specificity) = PD - PF$. We can see that J coefficient is a combination of PD and PF. When $J\_coeff = 0$, the probability of detecting a faulty module is equal to the false alarm rate. Such a classifier is not very useful. When $J\_coeff > 0$, PD is greater than PF, a desirable classification result. Hence, $J\_coeff = 1$ represents perfect classification, while $J\_coeff = -1$ is the worst case.

Variance is a measure of statistical dispersion of a random variable, computed by averaging the squares of the deviations. When evaluating performance of a classification experiment, the smaller the variance, the more "reliable" (stable) the classifier performs. Although the importance of variance in the supervised classification is known, it is seldom reported and analyzed in software fault prediction models. Assume μ stands for the mean, $\mu = \frac{\sum x_i}{n}$, variance, s^2, is calculated as follows:

$$s^2 = \frac{\sum (x_i - \mu)^2}{n - 1} \tag{2.5}$$

The standard deviation, σ, is the square root of variance, that is, $\sigma = \sqrt{s^2}$. Noted that, in the supervised classification, many commonly used performance indices, such as, *recall, precision, PF, and AUC*, are all from 0 to 1. As a consequence, variances are all less than 1. This does not imply that variance is small. From an empirical perspective, if variance is greater than 0.05,

then the classification has a larger variance [44].

2.4.2 Graphical Evaluation Methods

In this section, we present Receiver Operating Characteristic (ROC) curve, Precision and Recall (PR) curve, and Lift Chart etc. These graphs are closely related, being derived from the confusion matrix. Ling *et al.* discussed the relationship between ROC and lift chart [56]. In [84], Vuk and Curk establish a common mathematical framework between ROC and lift chart, and the Area Under the ROC Curve (AUC) and Area Under the Lift Chart. However, each curve reveals different aspects of classification performance, making them worth considering in software engineering projects.

ROC Curve

Many classification algorithms allow users to define and adjust a threshold parameter in order to generate an appropriate classifier. When predicting software quality a higher PD can be produced at the cost of higher PF and vice versa. The (PF, PD) pairs generated by adjusting the algorithm's threshold form a Receiver Operating Characteristic (ROC) curve. ROC analysis is a more general way to measure a classifier's performance than numerical indices [90]. An ROC curve provides a visual tool for examining the tradeoff between the ability of a classifier to correctly detect fault-prone modules (PD or $sensitivity$) and the number of non-fault-prone modules that are incorrectly classified (PF, or $1 - sp$).

The Area Under the ROC curve, referred to as AUC, is a numeric performance evaluation measure that is directly associated with an ROC curve. It is very common to use AUC to compare the performance of different classification methods based on the same data. However, large parts of the region under the curve are typically not of interest to software engineers. For instance, the regions associated with very low probability of detection (region C in Figure 2.2(a)) or very high probability of false alarm (region B in Figure 2.2(a)) or both (region D in Figure 2.2(a)), typically indicate poor performance. With a few possible exceptions (for example safety critical systems, in which risk aversion drives the development process), only the performance points associated with acceptable PD and PF rates (region A) are likely to

have a practical value for software engineers. Therefore, in software engineering studies, the area under the curve of region A (denoted AUCa) is typically a more meaningful method to make the comparison than the standard AUC.

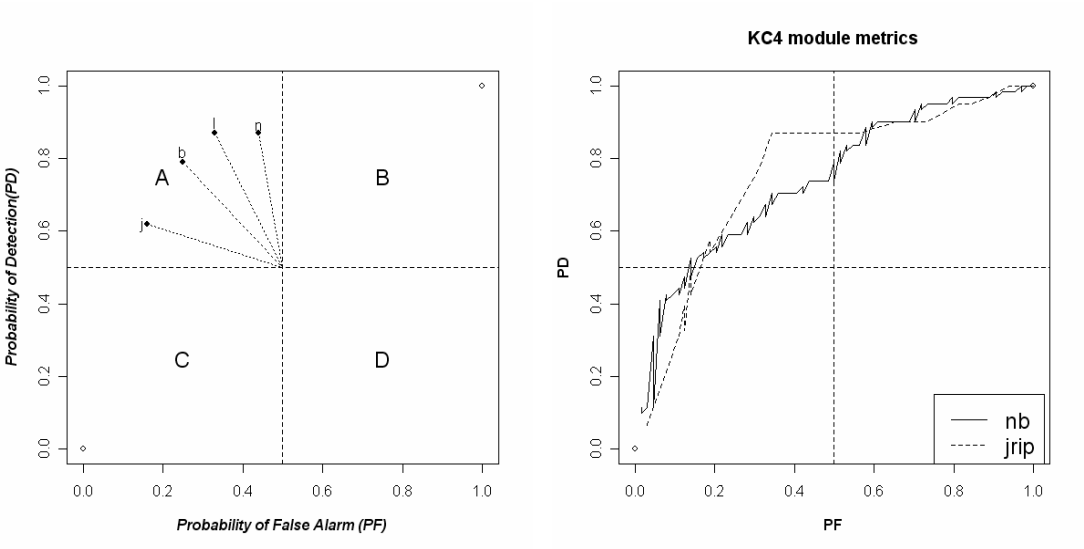

Figure 2.2. (a) Four possible regions represent various performance of software fault prediction models; (b) ROC curves from two classifiers Naive Bayes (nb) and jrip on project KC4.

The utility of this comparison method extends to classifiers which lack the flexibility of generating an ROC curve, since they typically lack the threshold parameter. Such algorithms are only capable of producing a single point in the space represented by a $(PF,\ PD)$ pair. Rather than measuring distances from an arbitrary reference point, one can calculate as the distance from the perfect classification point (0, 1) to the (PF, PD) pair point, defined as ED [62]:

$$ED = \sqrt{\theta * (1 - PD)^2 + (1 - \theta) * PF^2},$$

where θ is a parameter ranging from 0 to 1, used to control weights assigned to 1-sensitivity (i.e., 1-PD) and 1-specificity (i.e., PF). The smaller the distance, i.e., the closer the point is to

the perfect classification, the better the performance of the associated classifier. This distance from perfect point (0,1) is appropriate to choose a best point from a convex hull of an ROC curve too. This method can assist software engineers in determining the "best" threshold for a classifier, given a project data and an appropriate value of parameter .

Precision-Recall curve

Precision-Recall (PR) curve presents an alternate approach to the visual comparison of classifiers [22]. PR curve can reveal the difference between algorithms which is not apparent from an ROC curve. In a PR curve, x-axis represents recall and y-axis is precision. Recall is yet another term for PD.

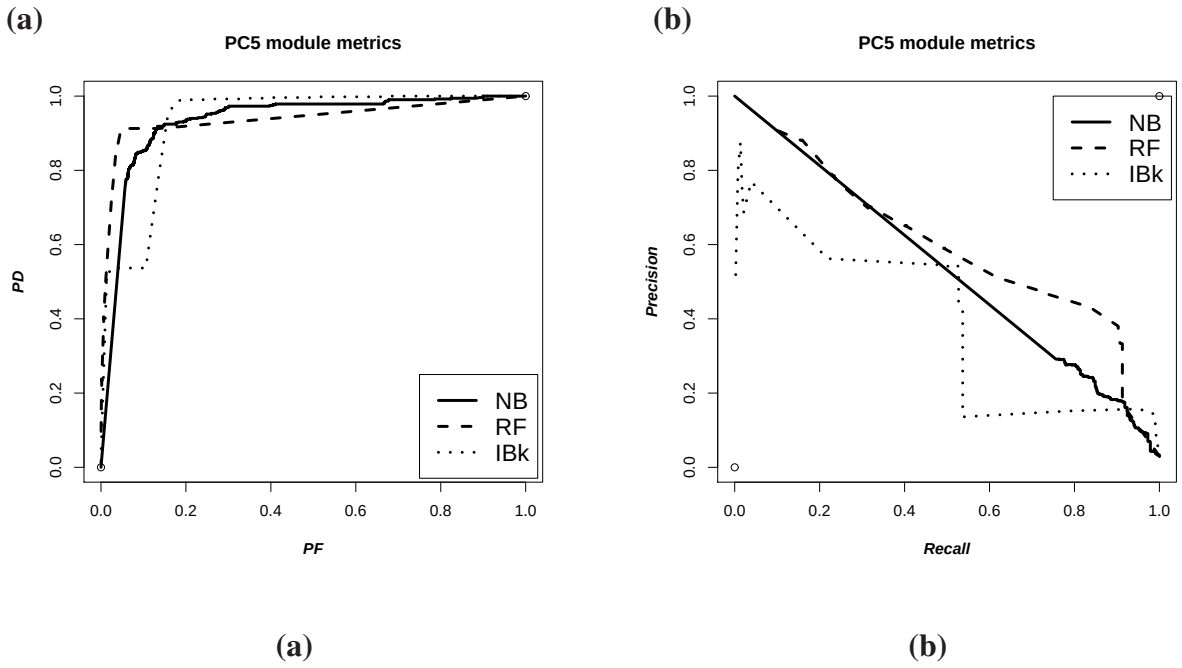

Figure 2.3. ROC curve (a) and PR curve (b) of models built from PC5 module metrics

Figure 2.3 shows an example. Looking at the ROC curves for project PC5, it is difficult to tell the difference among three classifiers: Naive Bayes (nb), Random Forest (RF), and IBk.

However, their PR curves allow us to understand the difference in their performance: Random Forest algorithm performs better than Naive Bayes and Naive Bayes has an advantage over IBk.

In ROC curves, the best performance indicates high PD and low PF in the left-hand upper corner of the curve. PR curves favor classifiers which offer high PD and high Precision, i.e., the ideal performance goal lays in the right-hand upper corner. In Figure 2.3(a), the performance of the three classifiers appears to approach close to the optimal left-hand upper corner. However, the PR curve of Figure 2.3(b) indicates that there is still plenty of room for the improvement of classification performance. We do not say that PR curves are better than ROC curves. We only recommend that when ROC curves fail to reveal differences in the performance of different classification algorithms, PR curves may provide adequate distinction.

Lift Chart

In practice, every software project has a finite time and budget constraints for verification and validation. Given the fault prediction model, the question that typically arises is how to utilize available resources to achieve most effective quality improvement. Lift charts [87], known in software engineering as Alberg diagrams [47, 72–74], are another visual aid for measuring classification performance. Lift is a measurement of the effectiveness of a classifier to detect faulty modules. It calculates the ratio of correctly identified faulty modules with and without the predictive model. Lift chart is especially useful in situations when the project has limited resources to apply verification activities to, say, 5% of the modules. Which model selects the 5% of the project's modules with a largest probability that these modules contain faults?

Lift chart analysis starts by ranking all the modules with respect to their chance of containing fault(s). The ranking methods can vary [47, 56, 72–74, 87]. For example, some algorithms (i.e. multiple linear regression models) may use the expected number of faults in a module. Alternatively, a classifier may output a score indicating the likelihood that the module belongs to a faulty class (i.e. Naive Bayes). An ensemble algorithm (random forest) may count the voting score. Once modules are ranked, we calculate the number of faulty modules in the specific rank (from 0 to 100%). In the lift chart, the x-axis represents the percentage of the modules considered, and the y-axis indicates the corresponding detection rate within this sample. The lift chart consists of a baseline and a lift curve. The lift curve shows the detection probability

resulting from the use of the predictive model, while the baseline indicates the proportion of faulty modules in the data set. The greater the area between the lift curve and the baseline is, the better the performance of the classifier is.

Boxplot Diagrams

A boxplot, also known as a box and whisker diagram, graphically depicts numerical data distributions using five first order statistics: the smallest observation, lower quartile (Q1), median, upper quartile (Q3), and the largest observation. The box is constructed based on the interquartile range (IQR) from Q1 to Q3. The line inside the box depicts the median which follows the central tendency. The whiskers indicate the smallest observation and the largest observation. Figure 2.4 shows an example boxplot of the best learners on the three groups of metrics on PC5 data set.

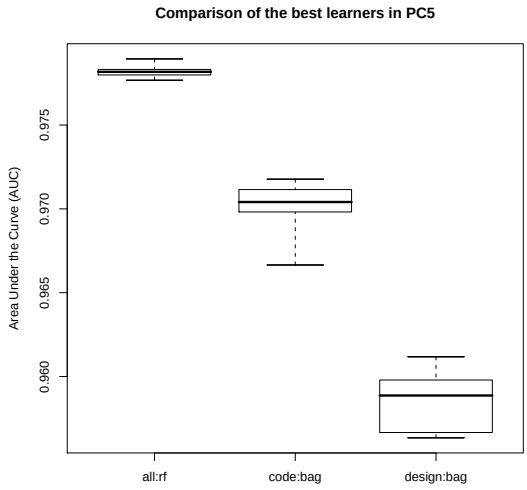

Figure 2.4. Boxplots of PC5 data set

2.4.3 Statistical Comparisons of Classification Models

Drawing sound decisions from performance comparison of different classification algorithms is not simple. Statistical inference is often required. The purpose of performance comparison is to select the best model(s) out of several candidates. Suppose there are k models to compare. The statistical hypotheses are:
H_0*: There is no difference in the performance among k classifiers.*
vs.
H_α*: At least two classifiers have significantly different performance.*

When more than two classifiers are under comparison, a multiple test procedure may be appropriate. If the null hypothesis of equivalent performance among k classifiers is rejected, we need to know which differences cause the rejection. In order to better understand why H0 is rejected we need to conduct pairwise comparison, that is, compare the performance difference between pairs of classifiers. We will then have the follow up statistical hypotheses for comparison between two classifiers:
H_0*: There is no difference in the performance between the two classifiers.*
vs.
H_α*: There is a significant difference in the performance of the two classifiers.*

The most popular method used to evaluate a classifier on a data set is 10 by 10 ways cross-validation (*10x10 CV*). The $10x10$ CV results in 10 values of the performance index of interest (*AUC, F-measure*, or any other). These 10 values are likely similar to each other, given that they come from the same population. But, given the 10 values it is difficult to establish whether they follow the normal distribution (i.e., indicate that they obey the central limit theorem). Therefore, using parametric statistical methods, which assume the normally distributed population, may not be justified. A prudent approach calls for the use of nonparametric methods. The loss of efficiency caused by using nonparametric tests is typically marginal [21, 79].

Demsar [23] overviewed theoretical work on statistical tests for classifier comparison. He recommended the Wilcoxon signed rank test for the comparison of two classifiers and the Fried-

man test with the corresponding post-hoc tests, the Nemenyi test, when the comparison includes more than two classifiers. The Wilcoxon signed rank test and the Friedman's test are nonparametric counterparts for paired t test and analysis of variance (ANOVA) parametric methods, respectively. Demsar advocates these tests largely due to the fact that nonparametric procedures make less stringent demands on the data. However, two issues need attention. First, nonparametric tests do not utilize all the information available. The actual data values (in our case performance indices such as AUC) are not used in the test procedure. Instead, the signs or ranks of the observations are used. Therefore, parametric procedures will be more powerful then their nonparametric counterparts, when justifiably used. The second point is that the signed rank tests are constructed for the null hypothesis that the difference of the performance measure is symmetrically distributed. For non-symmetric distributions, this test can lead to a wrong conclusion.

Demsar used a significance diagram to represent test result, called *Demsar's significance diagram*. An example is shown in Figure 2.5. The figure shows the result of Demsar's procedure on $cm1\ design$ metrics, comparing models from 5 different training sizes, measured by AUC. The statistical hypotheses are:
H_0:*The size of the model's training set has no influence on model performance among 10 different runs in cm1 design metrics.*
vs.
H_α: *Some (at least two) models developed using different training subset sizes result in significantly different performance among 10 different runs in cm1 design metrics.*

Let's look at Figure 2.5. The numbers in the scale represent the average rank; the higher the rank, the worse the performance of a training size. Therefore, from the worst towards the best, the order of models is from 10% to 90%. In Figure 2.5, the critical difference (CD) is 2.728. When the difference between two average ranks (or two models) is smaller than the value of CD, the difference in their performance is not significant, as connected by a bold straight line. Figure 2.5 indicates that our fault prediction models form two performance clusters: one is 10%, 25%, 50%, and 75%; the other is 25%, 50%, 75%, and 90%.

With more than one data set, we used a modified version of Demsar's significance diagrams. Let's compare Demsar's significance diagram shown in Figure 2.5 to the first item (the same

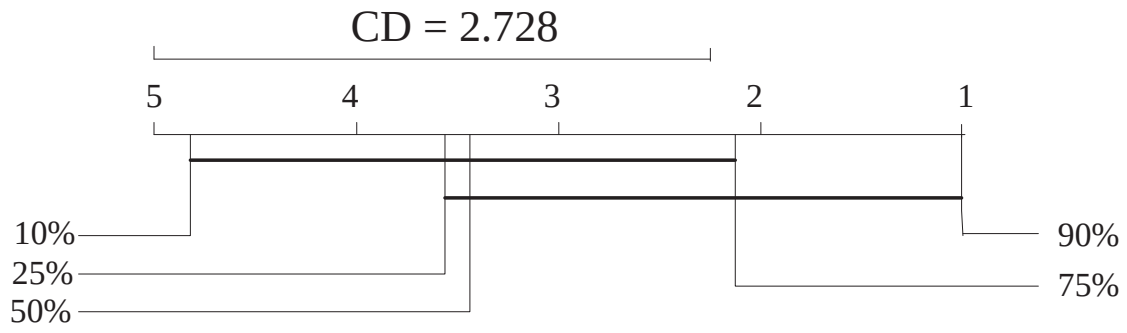

Figure 2.5. An example of Demsar's significance diagrams.

test on $cm1$ data) in Figure 2.6. The rank is still represented on a horizontal scale from 1 to 5: the higher the rank, the worse the performance of a training size. CD still represents the critical difference value for this statistical test. However, two straight lines are replaced with two bracket pairs: round and square bracket. The models inside each bracket do not have any statistical difference. The round bracket forms the worst performance models while the best performance models are enclosed inside the square bracket. In this way, Figure 2.6 is able to show the test result of 14 data sets.

2.5 MDP Data Sets and Prior Experiments

The 16 data sets used in this dissertation come from the NASA MDP repository [2] and PROMISE (3 data sets) [13] shown in Table 2.3. Metrics Data Program (MDP) is a software metrics repository provided by NASA $IV\&V$ and is available to general users through website http://mdp.ivv.nasa.gov/. MDP data stores and organizes the software product metrics data and associated error data at the module (functional/method) level. Currently, there are 13 projects data available. All MDP data are also available from PROMISE [13] public repository. Public fault data repository provide a possible platform for comparison of different approaches, different measurements, and different research groups worldwide. With the availability of public fault data sets, fault prediction are able to be investigated in a repeatable, or improvable, or even refutable way.

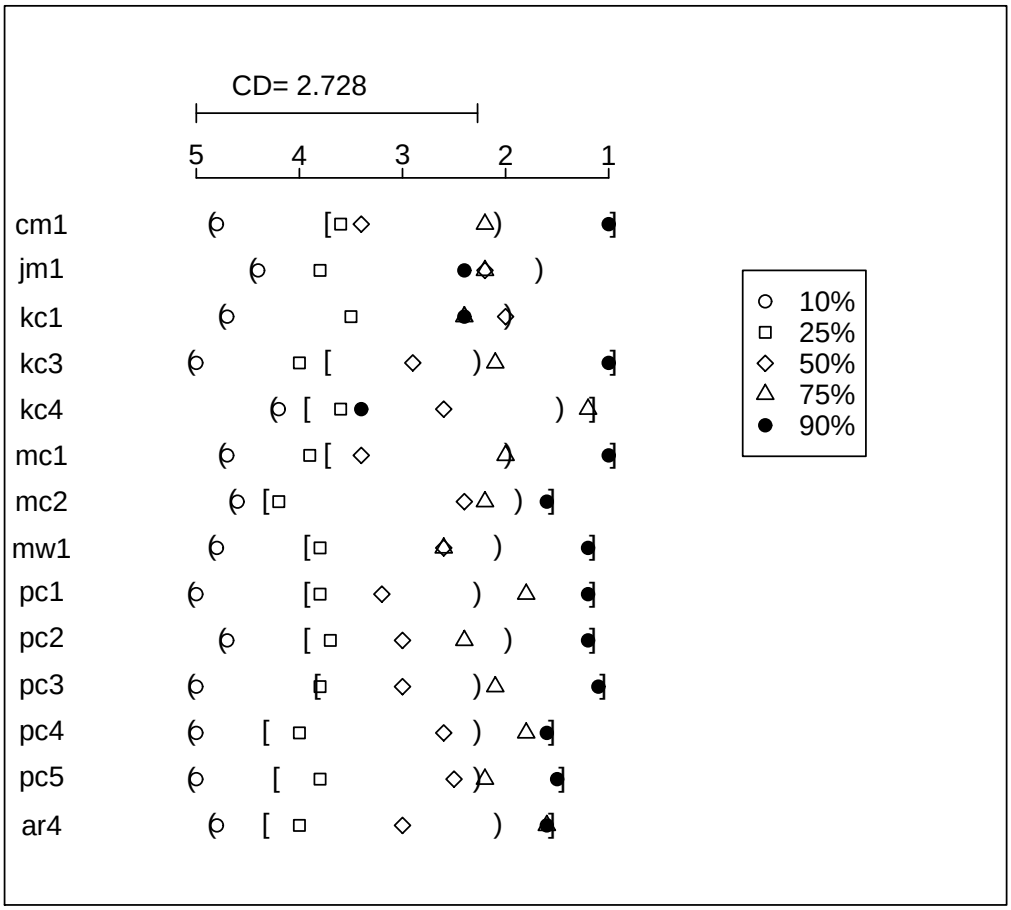

Figure 2.6. The Friedman test on $design$ metrics models using AUC for performance evaluation.

Table 2.3. Datasets used in this dissertation

Data	mod.#	% faulty	project description	lang.	source
JM1	10,878	19.3%	Real time predictive ground system	C	MDP
MC1	9466	0.64%	Combustion experiment of a space shuttle	(C)C++	MDP
PC2	5589	0.42%	Dynamic simulator for attitude control systems	C	MDP
PC5	17,186	3.00%	Safety enhancement system	C++	MDP
PC1	1109	6.59%	Flight software from an earth orbiting satellite	C	MDP
PC3	1563	10.43%	Flight software for earth orbiting satellite	C	MDP
PC4	1458	12.24%	Flight software for earth orbiting satellite	C	MDP
CM1	505	16.04%	Spacecraft instrument	C	MDP
MW1	403	6.7%	Zero gravity experiment related to combustion	C	MDP
KC1	2109	13.9%	Storage management for ground data	C++	MDP
KC3	458	6.3%	Storage management for ground data	Java	MDP
KC4	125	48%	Ground-based subscription server	Perl	MDP
MC2	161	32.30%	Video guidance system	C++	MDP
ar3	63	12.70%	Refrigerator	C	PROMISE
ar4	107	18.69%	Washing machine	C	PROMISE
ar5	36	22.22%	Dish washer	C	PROMISE

Inside these MDP data sets, JM1 and KC1 have 21 attributes that can be used as predictor variables, MC1 and PC5 have 39, the other data sets from MDP repository have 40. The 3 data sets from PROMISE [3] were collected from a Turkish white-goods manufacturer building controller software for a washing machine, a dishwasher and a refrigerator, with 29 variables. Although the names of variables from these 3 PROMISE data sets are different from MDP, they in fact are equivalent to a subset of metrics used in MDP repository. For example, metrics "total_loc", "unique_operands", and "halstead_time" in PROMISE correspond to "loc_total","num_operands", and "halstead_prog_time" in MDP. The metrics describing PROMISE project are presented in bold font in Table 2.4.

The metrics shown in Table 2.4 have been extracted using McCabe IQ 7.1, a reverse engineering tool that derives software quality metrics from code, visualize flowgraphs and generate report documents [1]. In all the data sets, available metrics are classified into three groups: $design$, $code$, and $other$, as indicated in Table 2.4. What separates design metrics from code metrics is the opportunity to extract them from design specification diagrams such as UML. For example, Ohlsson and Alberg extract design metrics such as McCabe cyclomatic complexity from Formal Description Language (FDL) graphs [72]. Their design metrics include $node\_count$, $branch\_count$, and McCabe cyclomatic complexity measures [72], also produced by McCabe IQ tool. McCabe complexity metrics are also used as design metrics in [91], the study that provides motivation and a point of comparison for this research. As a quick reminder, if graph G represents module's flowgraph, its cyclomatic complexity $v(G)$ is calculated as $v(G) = e - n + 2$, where e is the number of edges and n is the number of nodes.

The $code$ metrics are the features that can only be extracted from the source code. Static code metrics, such as $num\_operators$,$num\_oprands$, and the Halstead metrics are calculated from program statements [31, 34]. $other$ metrics are those related to both design and code. Most data sets have four metrics we classified as $other$. We do not use other metrics in isolation to build models, but we include them in the experiments in which fault prediction models are developed from all (also called module metrics) available attributes.

Guo *et al.* [33] build fault prediction modules using a set of classifiers on five MDP data sets(CM1, JM1, PC1, KC1, and KC2) and they find the random forest classifier has better performance than others inside this set of classifiers. Challagulla *et al.* [18] study 4 MDP data sets using 11 classifiers. Their findings show that there is no particular learning technique that

Table 2.4. Metrics used.

group	metrics	description or formula
code	**parameter_count**	Number of parameters to a given module
	num_operators:N1	The number of operators contained in a module
	num_operands:N2	The number of operands contained in a module
	num_unique_operators:μ_1	The number of unique operators contained in a module
	num_unique_operands:μ_2	The number of unique operands contained in a module
	halstead_content:μ	The halstead length content of a module $\mu = \mu_1 + \mu_2$
	halstead_length:N	The halstead length metric of a module $N = N_1 + N_2$
	halstead_level:L	The halstead level metric of a module $L = \frac{(2*\mu_2)}{\mu_1 * N_2}$
	halstead_difficulty:D	The halstead difficulty metric of a module $D = \frac{1}{L}$
	halstead_volume:V	The halstead volume metric of a module $V = N * \log_2(\mu_1 + \mu_2)$
	halstead_effort:E	The halstead effort metric of a module $E = \frac{V}{L}$
	halstead_prog_time: T	The halstead programming time metric of a module $T = \frac{E}{18}$
	halstead_error_est: B	The halstead error estimate metric of a module $B = \frac{E^{2/3}}{1000}$
	number_of_lines	Number of lines in a module
	loc_blank	The number of blank lines in a module
	loc_code_and_comment:NCSLOC	The number of lines which contain both code and comment in a module
	loc_comments	The number of lines of comments in a module
	loc_executable	The number of lines of executable code for a module (not blank or comment)
	percent_comments	Percentage of the code that is comments
	loc_total	The total number of lines for a given module
design	edge_count:e	Number of edges found in a given module from one module to another
	node_count:n	Number of nodes found in a given module
	branch_count	Branch count metrics
	call_pairs	Number of calls to functions in a module
	condition_count	Number of conditionals in a given module
	cyclomatic_complexity: v(G)	The cyclomatic complexity of a module $v(G) = e - n + 2$
	decision_count	Number of decision points in a module
	decision_density	$Condition\_count/Decision\_count$
	design_complexity:iv(G)	The design complexity of a module
	design_density	Design density is calculated as: $\frac{iv(G)}{v(G)}$
	essential_complexity:ev(G)	The essential complexity of a module
	essential_density	Essential density is calculated as: $\frac{(ev(G)-1)}{(v(G)-1)}$
	maintenance_severity	Maintenance Severity is calculated as: $\frac{ev(G)}{v(G)}$
	modified_condition_count	The effect of a condition affect a decision outcome by varying that condition only
	multiple_condition_count	Number of multiple conditions within a module
	pathological_complexity	A measure of the degree to which a module contains extremely unstructured constructs
others	**normalized_cylomatic_complexity**	$\frac{v(G)}{number\_of\_lines}$
	global_data_complexity:gdv(G)	the ratio of cyclomatic complexity of a module's structure to its parameter_count
	global_data_density	Global Data density is calculated as: $\frac{gdv(G)}{v(G)}$
	cyclomatic_density	$\frac{v(G)}{NCSLOC}$

performs the best for all the data sets. However, some combination methods are the better methods that showed relatively better consistency in prediction accuracy compared with others. Mertik *et al.* [64] have the same results. Mertik *et al.* use so-called multimethods which combine decision tree C4.5, support vector machine, and genetic algorithms to predict software fault severity of three MDP projects, PC4, KC3, and KC4. And they show that combination multimethods are generally better than single algorithm method.

Feature Subset Selection (FSS) is investigated in MDP data sets too. Turhan *et al.* [83] use 8 MDP data sets to built fault prediction models using metrics which have been filtered by feature subset selection (FSS) methods and those have not been filtered by FSS. They found the models which have been filtered by using FSS is better than those have not. Menzies *et al.* [62] and Guo [33] also apply FSS on MDP data sets, but they found in different data sets, the top five most important attributes vary with different data sets.

Menzies *et al.* [62] make the foundation of fault prediction studies on MDP data sets. By using a simple classifier naiveBayes, it is practical and useful to build fault prediction models from code metrics to guide software $IV\&V$ inspection, and they call for a better classifier for fault prediction. Lessmann *et al.* [53] investigated 22 classifiers over 10 data sets in MDP repository, they found that most classifiers (18 out of 22) demonstrate the same performance statistically.

Zhong *et al.* [92,93] are the first group who applied unsupervised learning in two MDP data sets: JM1 and KC2. They used Neural-Gas and K-means clustering to class software modules into several groups by using 13 module metrics. The clustered groups were then inspected by human experts to be fault-prone or not fault-prone. Their results indicate promising potential for the unsupervised learning method.

2.6 Heuristic

In this chapter, we review various metrics used in literature: requirement metrics, design metrics, code metrics, and different social network metrics, for example, the developer information, messages about mailing list, and organizational structure complexity metrics. All these metrics are proved to be effective predictors for faults (or fault-proneness information). But,

models built from these metrics are seldom combined and compared. Base on the best knowledge of the authors, the only work combining and comparing design and code metrics is Zhao *et al.* [91]. They conclude that the design metrics are as good as the code metrics; little improvement can be achieved if both design metrics and code metrics are used.... However, their conclusions were drawn from the data analysis from a single software project using only one fault prediction modeling technique (logistic regression). One of our goals is to reexamine their conclusion. In this dissertation, we use 14 publicly available project data sets. Each of these projects offers both design and static code metrics. We built fault prediction models using each metrics set separately (models denoted as $design$ and $code$, respectively), as well as from a combined metrics data set (denoted as all). Chapter 3 will introduce our experiments to compare combine requirement metrics and module metrics (design and code metrics). Chapter 4 will present our experiments and results on comparing and combining design and code metrics.

We also review all possible different types of evaluation methods for software fault prediction models. Inside these evaluation techniques, there have been attempts to include cost factors. *F-measure*, for example, offers a technique to account for the cost factor [39] when comparing different models. Khoshgoftaar and Allen [49] proposed the use of prior probabilities of misclassification to select classifiers which offer the most appropriate performance. Cost effectiveness measures described by Arisholm *et al.* [7] can account for the nonuniform cost of module-level $V\&V$. However, there is no existing evaluation techniques can address a project specific misclassification cost or a module specific misclassification cost needs. Hence, we introduce a cost curve evaluation technique in Chapter 5 to address this special cost evaluation in fault prediction models.

Chapter 3

Prediction from Requirement Metrics

In this chapter, we explore the following problems:

- Q1: Are requirement metrics good enough to build fault prediction models?
- Q4: Do fault prediction models built from a combination of requirement, design, and code metrics provide better performance than models built from any metric subset?

First, we begin with an introduction on the requirement metrics in MDP repository. Then, we show our experimental design and result. We conclude with a summary and discussion.

Therefore, in this chapter, we investigate whether metrics available early in the development lifecycle can be used to identify fault-prone software modules. More precisely, we build predictive models using the metrics that characterize structured textual requirements. We compare the performance of requirements-based prediction models against the performance of module-based models. Finally, we develop a methodology that combines requirement and module metrics. Since such models cannot be developed early in the lifecycle, we evaluate whether such combination can increase the prediction power in comparison to models that use module metrics only. Using a range of modeling techniques and the data from three projects included in NASA Metrics Data Program (MDP) [2], CM1, JM1, and PC1, our experiments indicate that the early lifecycle metrics can play an important role in project management, either by pointing to the need for increased quality monitoring during the development or by using the models to

assign verification and validation activities.

3.1 Static Requirements Features

The datasets used in this chapter is NASA Metrics Data Program (MDP) data repository [2]. Although MDP repository contains 13 projects, only 3 of them offer requirement metrics. These three projects are CM1, JM1, and PC1. From Table 2.3 we can see that CM1 project is a NASA spacecraft instrument, JM1 is a realtime ground system, PC1 is an earth orbiting satellite system. There are 10 attributes that describe requirements. One of them is the unique requirement identifier. The remaining 9 attributes assume non negative integer values. These metrics are shown in Table 3.1. We use them as attributes when building fault prediction models.

All the MDP requirement metrics follow the definitions from Wilson [86]. According to the reference, a tool "searches the requirements document for terms [the research at NASA-Goddard Software Assurance Research Center has] identified as quality indicators". ARM (*Automated Requirement Measurement*) tool [86] was an experiment in lightweight parsing of requirements documents. Rather than to tackle the complexities of full-blown natural language, the ARM research explored what could be easily automatically identified. The ARM work resulted in an automatic parser and some threshold guidelines regarding when to be concerned about a requirements document. Since our classification algorithms can automatically generate such thresholds, we ignored the ARM thresholds and just used the numerical values offered by the parser.

The ARM parser reports information at the level of individual requirement specifications. Specification statements are evaluated along the following dimensions:

- *Imperatives* (something that must be provided) count the phrases "shall", "must" or "must not", "is required to", "are applicable", "responsible for", "will", or "should".
- *Continuances* (connections between statements) count the phrases "as follows", "following", "listed", "in particular", or"support".
- *Directives* (to supporting illustrations) count the phrases "figure", "table", "for example", "note".
- *Options* give the developer latitude in satisfying the specification statement. This count

includes the phrases “can”, “may” and “optionally”.

- *Weak Phrases* (causing uncertainty, leave room for multiple interpretations) count the phrases “adequate”, “as a minimum”, “as appropriate”, “be able to”, “be capable”, “but not limited to”, “capability of”, “capability to”, “ effective”, “if practical”, “normal”, “ provide for”, “timely”, and “tbd”.

Table 3.1. Requirement Metrics.

Requirement	Definitions
action	Represents the number of actions the requirement needs to be capable of performing.
conditional	Represents whether the requirement will be addressing more than one condition.
continuance	Phrases that follow an imperative and precede the definition of lower level requirement specification.
imperative	Those words and phrases that command that something must be provided.
incomplete	Phrases such as ‘TBD’ or ‘TBR’. They are used when a requirement has yet to be determined.
option	Those words that give the developer latitude in the implementation of the specification that contains them.
risk_level	A calculated risk level metric based on weighted averages from metrics collected for each requirement.
source	Represents the number of sources the requirement will interface with or receive data from.
weak_phrase	Clauses that are apt to cause uncertainty and leave room for multiple interpretations.

3.2 Experimental Design and Result

3.2.1 Experimental Design

In this chapter, we report the development and evaluation of models to predict fault-prone software modules using the following information from NASA MDP datasets [2]:

1. Experiment 1: Available metrics describing unstructured textual requirements;

2. Experiment 2: Available software module metrics;

3. Experiment 3: A combination of requirement metrics and module metrics.

The goal of each of these experiments is different. In experiment 1, we try to assess how useful the early lifecycle data and the related metrics are in identifying potentially problematic modules. The second experiment is the least interesting from the research standpoint as the use of static module metrics in the prediction of faulty modules has been investigated extensively, even on the same datasets we use here. However, in our case, it sets the performance baseline for the third experiment. We will demonstrate that the use of combined requirements metrics and static module metrics performs extremely well in predicting fault-prone modules.

In order to enable these experiments, we had to be able to relate individual requirements with software modules. The generic structure of an entity relationship diagram that connects requirements with modules for a specific project entered in NASA MDP database is shown in Figure 3.1. All software requirements and modules are uniquely numbered. A requirement may be implemented in one or more modules. A module implementation may reflect one or more requirements. Further, a module may contain zero, one or more faults. For the purpose of our experiments, if a module contains any faults, it is considered fault-prone. Otherwise, it is fault free. Unfortunately, MDP datasets [2] reveal anomalies too. Some requirements are associated with no software modules and some modules cannot be traced to any stated requirement. As our research group has not been involved with the data collection, we could only point out to such inconsistencies. The extent of such inconsistencies is described later.

Figure 3.1. An entity-relationship diagram relates project requirements to modules and modules to faults

We combine requirement metrics and module metrics available for some of the projects in NASA Metrics Data Program (MDP) database [2]. Combining requirement metrics and module

metrics if there is a one-to-one relationship between the requirements and modules is trivial. But, due to many-to-many relationship, we need to utilize the $inner - join$ database operation. Inner-join, also referred in the literature as $equijoin$ [75], results in merging the records from two relational tables for which there is a matching value in the field(s) on which the tables are joined. It creates an all-to-all association between the corresponding entries in two database tables. This is the most common type of database join operation.

Referring to Figure 3.1, the inner-join takes all the records from CM1_product_requirement_metrics table and finds the matching record(s) in table CM1_requirement_product_relations based on the common predicate — Requirement_ID. The result is written into a temporary table. Inner-join further takes all the records from the temporary table and looks for the matching records in CM1_product_module_metrics via the join predicate Module_ID. Table 3.2 shows partial results of this operation.

Table 3.2. The result of inner join on CM1 requirement and module metrics.

Req_id	action	Continuance	Mod_id	loc_blank	Branch_count	⋯
100	1	0	25321	82	43	⋯
100	1	0	25333	34	17	⋯
101	3	1	25325	14	7	⋯
102	1	0	25321	82	43	⋯
102	1	0	25325	14	7	⋯
102	1	0	25333	34	17	⋯
⋯	⋯	⋯	⋯	⋯	⋯	

As mentioned earlier, partial requirement metrics are available for the three MDP datasets: CM1, JM1 and PC1. In addition, our analysis uncovered several data discrepancies. Table 3.3 summarizes the available data. In projects CM1 and PC1, only 22% and 18% of all modules, respectively, have their requirements identified. The extreme case is the dataset JM1 in which only 1% of modules are associated with any requirement. But, due to the fact that JM1 is the largest project, the number of modules with identified requirements (97) is similar to the corresponding value for PC1 (109). In experiments, we will consider only the subset of modules in CM1, JM1 and PC1 which have their requirements identified. We will call these datasets CM1_r, JM1_r and PC1_r, to separate them from their usage in existing literature [17, 33, 62].

Table 3.3. The associations between modules and requirements in CM1, JM1 and PC1.

	total modules	**modules with faults**	**modules have req.**	**faulty modules with req.**	**total # of req.**	**req. related to module(s)**	**# of req. related to faults**
CM1_r	505	81(16.03%)	109(22%)	58(53.21%)	160	114	69(60.53%)
JM1_r	10,878	2102 (19.32%)	97(1%)	4(4.12%)	74	17	3(17.65%)
PC1_r	1107	73(6.59%)	203(18%)	44(21.67%)	320	320	109(34.06%)

3.2.2 Prediction from Requirements Metrics

CM1_r dataset

CM1_r dataset describes software artifacts of a NASA spacecraft instrument. CM1_r has 160 requirements, but only 114 of them have associated program modules identified. Among these 114 requirements, 69 (60.53%) of them are related to modules which contain at least one fault. This is a very significant percentage. On the other hand, only the modules with identified requirements (109 of them out of 505) were used in the training/testing datasets.

As described in the Experimental Design section, we first developed models that predict defective modules using the requirements metrics only. The performance of different machine learning algorithms is depicted in the ROC curve shown in Figure 3.2. No model appears to provide particularly useful information for project managers, which can probably be attributed to the fact that more than 60% of the requirements are related to defective modules.

JM1_r dataset

JM1 metrics represent a realtime ground system that uses simulations to generate flight predictions. JM1_r has 74 requirements available (see Table 3.3). Only 17 of them are related to program modules. Three of these 17 requirements are associated to defective modules (17.65%). Figure 3.3 compares the performance of different models in an ROC curve. Although the performance of most models fails to provide useful information, the models built using Logistic

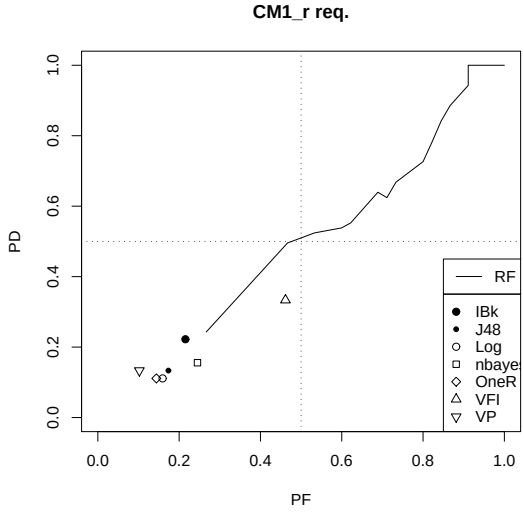

Figure 3.2. CM1_r prediction using requirements metrics only.

and VFI algorithms appear to do better than the others. Although JM1 is a very large project ($> 10,000$ modules overall), JM1_r is very small as the information that relates requirements and program modules is largely missing.

PC1_r dataset

PC1 project refers to an Earth orbiting satellite software system. PC1_r has 320 available requirements (Table 3.3) and, in contrast to CM1_r and JM1_r, all of them are associated with program modules. 109 (34.06%) requirements, or at least some aspect of these requirements, are implemented by modules which contain one or more faults. On the other hand, all these requirements point to only 203 out of 1107 modules indicating, again, missing information. Consistent with our experimental design, we developed PC1_r models using the 320 requirements and 203 modules.

The performance of PC1_r is shown in Figure 3.4. Unlike in the other two experiments, the useful portion of the ROC curve is in the cost-adverse region with low false alarm rate ($PF < 0.5$), but coupled with the low probability of detection ($PD < 0.5$). If the data represented a

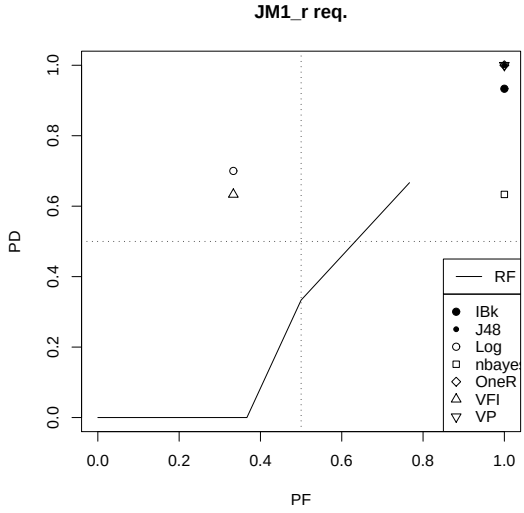

Figure 3.3. JM1_r prediction using requirements metrics only.

stable development environment in which this type of a model was built from an earlier project, the fact that the V&V team could start to develop information about modules in which faults can be expected seems very encouraging. However, given that these are the first-of-the-kind, preliminary experiments with fault prediction models build from requirement metrics, we can only cautiously suggest that our results warrant further research in this area.

3.2.3 Prediction from Static module metrics

As discussed, an extensive body of work describes fault prediction from module static module metrics. Therefore, the purpose of this section is to establish the baseline performance to be used for performance comparison throughout this proposal.

CM1 and PC1 projects contain 43 attributes. We removed 6 attributes that do not impact binary classification: $module\_id$; two fault counting attributes: $error\_count$ and $error\_density$; and the three attributes that have the same value for every module: $global\_data\_complexity$, $global\_data\_density$, and $pathological\_complexity$. We use the remaining 37 static module attributes in CM1_r and PC_r. JM1 dataset comes with only 24 static module attributes. We

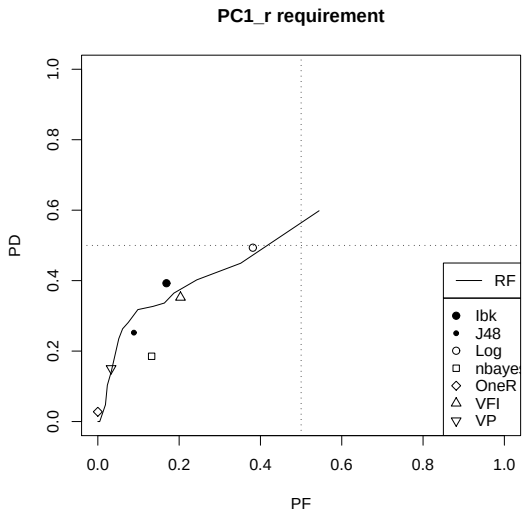

Figure 3.4. PC1_r prediction using requirements metrics only.

removed the module identifier and two error related attributes and used remaining 21 module metrics for the classification.

The predicted class is always the presence of faults in a module (fault-free or faulty). A detailed description of module attributes can be found in [33, 62]. Several recent studies describe the fault prediction results for the NASA MDP datasets [17, 33, 62]. These studies provide insights into the effects of feature (attribute) selection to model performance, as well as the variations caused by the use of a wide spectrum of machine learning algorithms. The observation has been that the choice of learning methods have a much stronger impact on performance than feature selection [62].

Although CM1, JM1 and PC1 have static module metrics data available for all modules (505, $10,878$ and $1,107$, respectively), to make performance comparisons fair, we only use modules that explicitly correspond to requirements (109, 97, 203, respectively). Therefore, the performance results embodied by the ROC curves in this chapter are not the same as reported in related literature [17, 33, 62], but similar.

Figures 3.5, 3.6 and 3.7 depict the performance of fault prediction models based on module

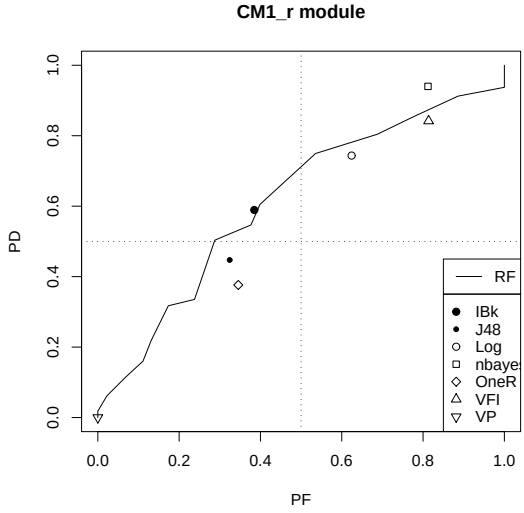

Figure 3.5. CM1_r using module metrics.

metrics. Machine learning algorithms provide models in the different regions of the ROC space. While the performance on neither of the project datasets appears impressive, consistent with the trends observed in related studies, JM1_r dataset proves extremely difficult. Among those who use NASA MDP datasets, JM_1 has been suspected of containing noisy (inaccurate) information. But while its large size usually makes it challenging for fault prediction, in our experiment JM1_r contains only 97 modules, with 4 of them being defective. Therefore it comes as no surprise that most machine learning algorithms (all except the random forest) generate theories that classify all the modules as fault-free.

The machine learning algorithms which typically perform better than others are random forests, naive bayes, and IBk, although the usefulness of their predictions in the context of software engineering projects appears rather limited.

3.2.4 Combining Requirement and Module Metrics

We combine requirement and module metrics using the inner-join method described earlier. Recall that there are 109, 97 and 203 modules associated to requirements in CM1, JM1

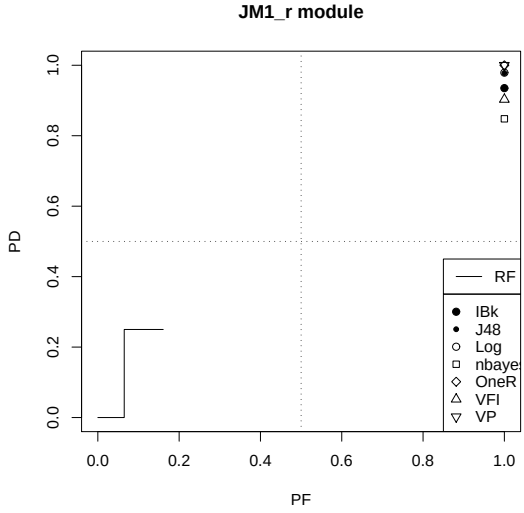

Figure 3.6. JM1_r using module metrics.

and PC1 datasets, respectively. When many-to-many relationship exists between modules and requirements, the inner join creates multiple entries in the resulting table, one for each unique Requirement_ID and Module_ID pair. Consequently, the datasets used in this experiment are larger. CM1_RM has 266 records, while PC1_RM has 477 records. JM1_RM maintained the same number of records, 97 because each module is related to a single requirement. The number of defective modules in each dataset remains the same as in the previous experiments; 147 faulty records for CM1_RM, only 4 for JM1_RM, and 111 for PC1_RM, but multiple fault records may exist for a single module.

Following the preparation of data set, we ran the 5 way cross-validation. First, we divide a dataset into five bins. Then we use four bins of data to train classifiers and build models, a standard experimental procedure. But, the dataset may contain multiple entries for the same module, if it is related to more than one requirement. To avoid counting the classification of the same module more than once, in the fifth bin – the testing bin, we must ensure that it only contains unique modules. In other words, we eliminate multiple entries of the same module and randomly keep one entry for each module. Finally, comparing the ground truth with the classification outcomes, we ascertain PD and PF and generate an ROC curve.

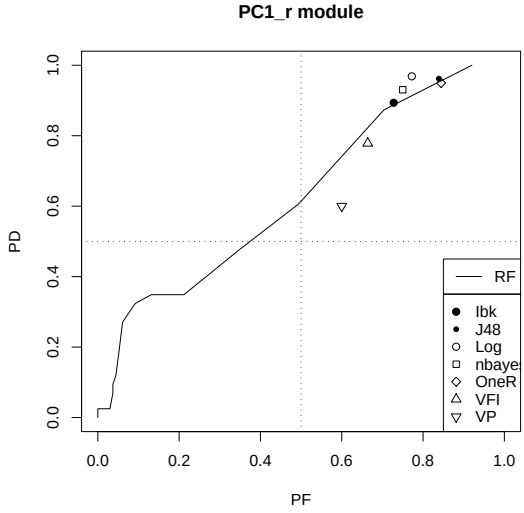

Figure 3.7. PC1_r using module metrics.

The prediction results for CM1_RM are shown in Figure 3.8 and for PC1_RM in Figure 3.9. The presence of additional requirement metrics attributes in JM1_RM did not improve the prediction model which used module metrics only. Both JM1_r and JM1_RM have the same number of entries, 97, which explains the lack of performance gains.

3.3 Discussion

Upon seeing the results of the experiments that combine requirements and static module metrics, we were pleasantly surprised. A nearly 70% improvement of PD at $PF = 0.1$ observed in CM1_r dataset, for example, is very significant. We have been building models for fault prediction in NASA MDP datasets for several years. A recent publication [62] reports that the best model performance based on module metrics only is $(PD = 71\%, PF = 27\%)$ for CM1 dataset and is $(PD = 48\%, PF = 17\%)$ for PC1. Our own experiments with module metric models reported in this chapter reflect the use of subsets of CM1, JM1 and PC1 project data and are somewhat different. But the performance improvement gained by adding requirements metrics has surpassed all our expectations. Figures 3.10, 3.11 and 3.12 depict

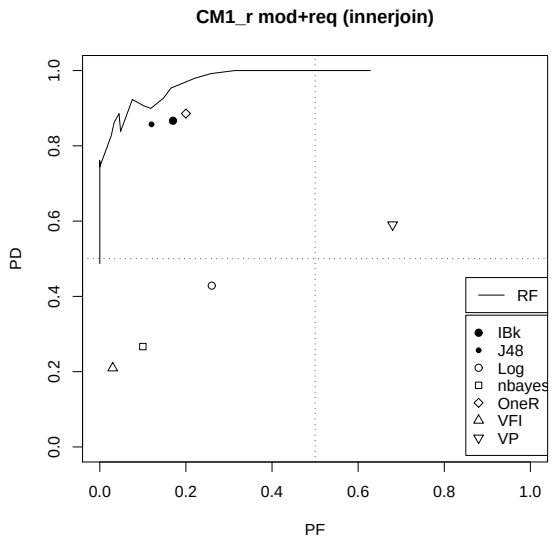

Figure 3.8. CM1_RM model uses requirements and module metrics.

ROC curves for the three datasets, CM1, JM1 and PC1, respectively. Depicted models have been developed using the random forests algorithm. Random forests have demonstrated the most consistent performance in all our experiments and we believe it is appropriate to use them for performance comparison. Each of these three figures contains three lines, depicting the performance of requirements-only based model, the module-only based model and, finally, the model that combines requirements and module metrics.

These results demand a careful analysis. Our first observation is specific and it relates to the inner workings of machine learning algorithms we used for modeling. After the inner-join, the datasets have more records. In other words, the datasets became oversampled. Oversampling is known to be an effective method in signal processing. Recent studies show that tree-based classifiers do not increase the performance as the result of oversampling in the training data [24, 25]. Random forests is a tree based ensemble-forming algorithm. So our results appear to contradict the results of Drummond and Holte in [24, 25]. Why random forests perform so well on the inner-join data? This phenomenon certainly is worth exploring in the future. However, other learners we used such as J48 and IBk, also improve their performance when requirements and module metrics are combined. Therefore, the observed performance improvement cannot

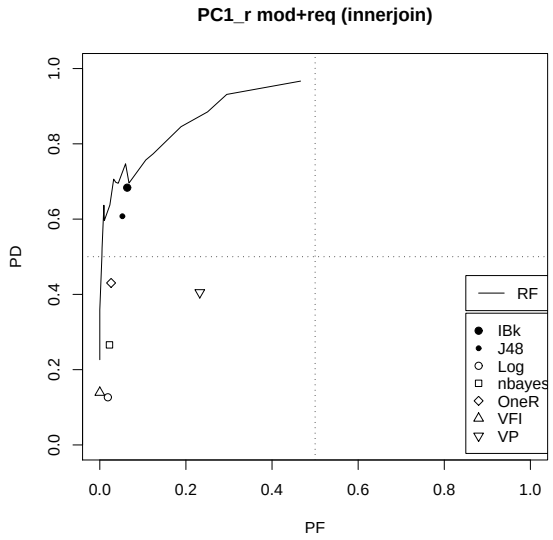

Figure 3.9. PC1_RM model uses requirements and module metrics.

be explained through better understanding of the algorithms behind specific machine learners only.

The obvious speculation here is that the training/test datasets after the inner-join represent real world software engineering situations better. Our experiments provide initial evidence that combining metrics that describe different yet related software artifacts may significantly increase the effectiveness of fault prediction models. Although each metric, regardless whether it describes requirement or module features, appears highly abstract and seemingly unrelated to software faults, when combined, they support the development of what appear to be superior fault prediction models.

We need to add a few caveats. NASA MDP is a continually growing dataset of software engineering data. The data we were able to obtain for the three projects do not appear to be highly accurate. As we mentioned, there are many requirements with no corresponding modules and, even more concerning, many modules with no corresponding requirements. We believe to have alleviated this problem by limiting our experiments to only include modules which have identified links to requirements. But as a result, we have models designed for a subset of project

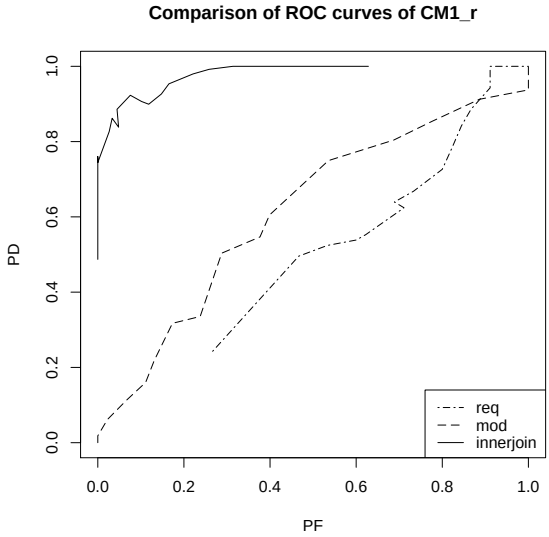

Figure 3.10. ROC curves for CM1 project.

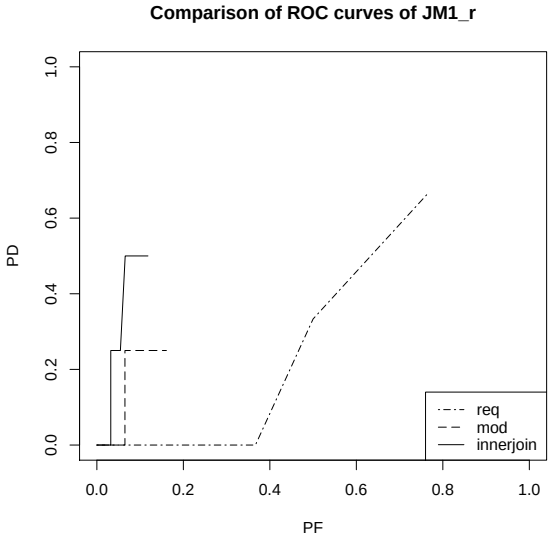

Figure 3.11. ROC curves for JM1 project.

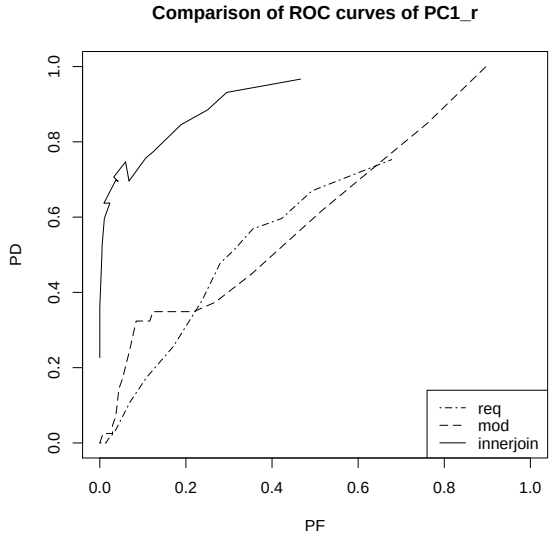

Figure 3.12. ROC curves for PC1 project.

data. These subsets have a higher proportion of faulty entries than the the datasets representing projects they come from. Machine learning algorithms generally perform better on smaller datasets. They also perform better on data sets where the size difference between majority and minority classes is insignificant. Therefore, our results are encouraging, but they may be overly optimistic.

3.4 Summary

In this chapter, we describe fault prediction models built using metrics extracted from unstructured textual requirements, the module (/product) metrics, and the fused requirement and module metrics. The models were developed and evaluated using metrics data from three NASA MDP projects: CM1, JM1, and PC1. Model comparison shows that requirement metrics can be highly useful for fault prediction. Although the requirement metrics do not predict faults well by themselves, they significantly improve the performance of the prediction models that combine requirement and module metrics together using the inner-join method. Our experiments

suggest that the early lifecycle metrics can play an important role in project management, either by pointing to the need for increased quality monitoring during the development or by using the models to assign verification and validation activities.

Lastly, we address the questions Q1 and Q4:

- Q1: Are requirement metrics good enough to build fault prediction models?
  Answer: **No.** The requirement metrics extracted from textual description alone are not sufficient to predict software fault proneness.
- Q4: Do fault prediction models built from a combination of requirement, design, and code metrics provide better performance than models built from any metrics subset?
  Answer: The combination of requirement metrics and module metrics are surprisingly good at fault proneness prediction. Combination models are much better than the models from requirement and module metrics alone.

Chapter 4

Incremental Development of Software Fault Prediction Models

In this chapter, we investigate the following problems:

- Q2: Are design metrics good enough to be fault predictors?
- Q3: Are code metrics good fault-prone predictors?
- Q4:Do fault prediction models built from a combination of requirement, design, and code metrics provide better performance than models built from any metrics subset?
- Q5: How large is a training subset needed to build meaningful fault prediction models?
- Q6: How large is a training subset needed to achieve the best performance fault prediction model?

This chapter is organized as follows. First, we begin with the experimental design. Second, we show the experimental results. Finally, we discuss the results obtained in this chapter.

4.1 Experimental Design

The metrics shown in Table 2.4 have been extracted using McCabe IQ 7.1, a reverse engineering tool that derives software quality metrics from code, visualize flowgraphs and generate

report documents [1]. It is important to note that McCabe IQ reverse engineers design metrics from code flowgraphs, rather than from design documentation. It is not unusual to analyze software design quality from design artifacts reengineered from the code [5, 6, 15, 16, 81, 82]. However, we recognize that this fact is one of the validity threats for our experiment.

The critical issue to be asked is whether the attributes listed in Table 2.4 as design metrics can be obtained before coding begins. We believe all of these measures can be obtained from detailed design documents, PSPEC descriptions for example, as long as such descriptions can be used to build call-graphs and control-flow-graphs. A similar argument has been made numerous times in the software engineering literature. According to Fenton and Pfleeger, McCabe metrics represent the structure of software product [31]. Fenton and Ohlsson [29] further state:"(McCabe) complexity metrics is the rather misleading term used to describe a class of measures that can be extracted directly from source code. Occasionally and more beneficially, (McCabe) complexity metrics can be extracted before code is produced, such as when the detailed designs are represented in a graphical language". Ohlsson and Alberg extracted design metrics from Formal Design Language (FDL) graphs [72]. Please note that these are the same metrics we designate as design metrics.

The experiments conducted in this chapter use 14 data sets listed in Table 2.3 including all 13 data sets from MDP and $ar4$ from PROMISE data repository. $ar4$ data set has 29 attributes including 9 design metrics, 18 code metrics, and 2 metrics classified as $other$.

4.1.1 Design of Experiments

Our goal is to investigate development of fault prediction models throughout the design and implementation phases of the life cycle yields models with improved performance. There are two interesting aspects of incremental model development:

1. Utilization of the increasing number of modules as training data, and
2. Timely (sequential) utilization of design and code metrics.

To reach these goals, we will compare the performance of models derived from:

- different percentages of data for model development (training): 10%, 25%, 50%, 75%, and 90%. In each experiment, the reminder of the data set will be used for model evaluation.
- different metric group: $design$, $code$, and all

The data sets we used do not include development schedule information. Therefore, we cannot faithfully emulate the actual "arrival" schedule of modules and their metrics. However, we will deploy cross-validation, the statistical practice of randomly partitioning a sample of data into two subsets:training and testing ten times. Reporting the median result from the ten experiments should minimize the impact a development sequence could have on the prediction results. The predicted variable is $DEFECT$, that is, the models predict whether a module is likely to contain fault(s).

Cross-validation is the statistical practice of randomized partitioning of a data sample into two subsets, one for model development (training) and the other for model testing. 10 way cross validation implies that each experiment is repeated 10 times, using different training and testing subsets. As the size of training subset varies, for example, between 10%, and 90%, in our experiments the corresponding testing subset includes all the remaining data, i.e., 90%, down to 10% of the data set. All experiments undergo 10 repetitions, with no adjustments based on sample sizes. Due to 10-way cross validation, comparing a large number of ROC curves becomes difficult. For this reason, we visualize the ten corresponding values of AUC in a Box-plot diagram.

We will use 5 algorithms for software quality prediction: random forest, bagging, logistic regression, boosting, and NaiveBayes. These algorithms have consistently been recommended to practitioners and provide adequate performance [39–43]. More importantly, the classifiers are implemented in publicly available machine learning toolkit Weka [87]. The use of multiple machine learning algorithms allows us to compare the results with the recent study conducted by Lessman et al. [53], in which they compare the performance of two dozen classification algorithms on MDP data sets. We use the default parameters in all the classifiers except in random forest, in which we follow the recommendation of algorithm's creator L. Breiman [14] and use 500 tree ensemble (rather than Weka default of 10 trees, which optimizes run-time).

In total, we conducted 10,500 experiments utilizing *1*4 data sets, 5 different sizes of training subset, 3 metrics groups, 10 cross validation runs, all of these repeated using 5 classification algorithms.

4.2 Experimental Results

4.2.1 Increasing the size of training set

One of the questions that repeatedly surface in discussions about fault prediction modeling deals with the amount of data needed to build reasonably accurate models. The question is not critical when models from earlier releases are used in the quality assurance of the new version or product release. Presumably, such systems are part of the product line and fault prediction models from earlier releases are adequate for the new release [74]. However, when an organization develops one-of-kind system or the first system release with no substantial history (and limited reuse), the amount of data needed to develop the models is the real issue in practice. This is certainly the case for most of the projects described in the NASA MDP repository. We will not directly address the "minimal" data set requirement for model development. Rather, following the idea of incremental model development, we would like to know the rate of model improvement as additional modules and their metrics descriptors become available.

For these reasons, we compare the performance of models generated using five training subset, containing 10%, 25%, 50%, 75%, and 90% of project modules. Although these proportions represent different sizes of training data for different projects, they match realistic milestones in the development life cycle.

We will test the following statistical hypotheses:
H_{10}:*The size of the model's training set has no influence on model performance.*
vs.
$H_{1\alpha}$: *Some (at least two) models developed using different training subset sizes result in significantly different performance.*

For this experiment, we utilize 14 data sets, 5 different training subsets, 3 groups of metrics,and 5 different machine learners, 10 times cross validation runs. Due to the sheer volume of information, Demsar's statistical analysis procedure [23] will provide acceptable summarization of results. The results are shown separately for models that utilize $design$ metrics only, $code$ metrics only, and all metrics.

The general trends to be reported are not surprising:

- 10% training subset results in the weakest performance.
- In most cases, the "best" performance can be expected from models which use 90% of data for training. Unfortunately, these models are the least useful as only 10% of the modules are left for fault prediction.
- Models increase their performance as the size of the training data set grows. However, these increases are rather minimal and statistical significance must be considered.

A more detailed analysis of the experimental results, including their statistical significance and practical implications follows.

Design metrics models

We test the null hypothesis first on fault prediction models that use only $design$ metrics.

Figure 2.6 shows the test result of 14 data sets measured by AUC using Demsar test procedure. The rank is represented on a horizontal scale from 1 to 5: the higher the rank is, the worse the performance of the model. CD represents the critical difference value for this statistical test: if the difference between two ranks is greater than the value of CD, they are statistically different, otherwise, they are not. To account for the statistical significance, we enclose the five models from each project into two pairs of brackets. Each bracket pair encloses the distance equivalent to the value of $CD = 2.728$. The round brackets enclose models which from a performance cluster with the worst performing model (typically the model which trains from the 10% subset). We will call this cluster *the lower rank cluster*. The square brackets enclose the performance cluster which includes the best performing model (typically inferred from the 90% subset). This will be our *higher rank cluster*.

Let us look closely into the performance of the five models built from $cm1$ in Figure 2.6. For the increasing sizes of training subsets, the corresponding ranks are 4.8, 3.6, 3.4, 2.2, and 1. These ranks form two performance clusters. Models built from 25%, 50%, and 75% subsets are in the intersection between the lower and higher performance rank clusters. The fault prediction models for projects $jm1$ and $kc1$ exhibit no statistically significant differences regardless of the size of the training subset. Therefore, they are included in a single performance rank cluster. The models for all other projects form two performance rank clusters. It is also interesting to note that in project $kc4$, the best performing model is developed using 75% of modules for training. In smaller projects (kc4 contains only 125 modules), training from 90% may result in over-fitting.

There are never more than two performance clusters. More detailed result comparisons in Table 4.1 and in Figure 4.1 will describe only 10%, 50%, and 90% training subsets.

Table 4.1. Median(m) and variance(v) of 10%, 50%, and 90% training subset models from $design$ metrics, measured by AUC

data	$m_{10\%}$	$v_{10\%}$	$m_{50\%}$	$v_{50\%}$	$m_{90\%}$	$v_{90\%}$	$\frac{m_{50\%}}{m_{10\%}} - 1$	$\frac{m_{90\%}}{m_{10\%}} - 1$	$\frac{m_{90\%}}{m_{50\%}} - 1$
cm1	0.54	0.0035	0.57	0.002	0.60	0.0133	5.56%	11.11%	5.26%
jm1	0.66	9e-04	0.67	8e-04	0.66	0.0013	1.52%	0	-1.49%
kc1	0.72	0.0027	0.73	9e-04	0.73	0.0028	1.39%	1.39%	0
kc3	0.61	0.0194	0.74	0.0121	0.77	0.0170	21.31%	26.23%	4.05%
kc4	0.75	0.0109	0.78	0.0017	0.76	0.0175	4.00%	1.33%	-2.56%
mc1	0.53	0.0054	0.62	0.0091	0.68	0.0214	16.98%	28.30%	9.68%
mc2	0.60	0.0087	0.63	0.0035	0.63	0.0271	5.00%	5.00%	0
mw1	0.66	0.0163	0.76	0.0038	0.78	0.0282	15.15%	18.18%	2.63%
pc1	0.57	0.0067	0.66	0.0028	0.69	0.0118	15.79%	21.05%	4.55%
pc2	0.50	0.0269	0.67	0.0174	0.75	0.0503	34.00%	50.00%	11.94%
pc3	0.65	0.0033	0.73	8e-04	0.73	0.0041	12.31%	12.31%	0
pc4	0.73	0.0029	0.78	0.001	0.78	0.0036	6.85%	6.85%	0
pc5	0.94	9e-04	0.95	1e-04	0.95	3e-04	1.06%	1.06%	0
ar4	0.62	0.0185	0.70	0.0085	0.72	0.0803	12.9%	16.13%	2.86%

Table 4.1 shows median and variance values for 10%, 50%, and 90% of data as training subsets. The median value is rounded to two digits, variance to 4 digits. The last three columns show the percentage increase in model performance. If the performance increases, the value is positive. Table 4.1 provides a closer look into the actual performance differences between models.

Figure 4.1 shows boxplot diagrams for 10%, 50%, and 90% of data as training subsets on design metrics which give a more intuitive view of comparison of these models.

Taken together from Table 4.1, Figure 4.1 and Figure 4.1, we can see that the gains in fault prediction vary from project to project and they are generally difficult to anticipate. Clearly, the significance results indicate that building only one or two models over the development life cycle, given that only design metrics are utilized, should be sufficient.

Code metrics models

We repeated the same statistical analysis procedure for the models derived from code metrics. From Figure 4.2, we can see that:

1. For 8 data sets, lower cluster includes models developed from 10% to 50% of modules.
2. For 6 data sets, lower cluster includes models developed from 10% to 75% of modules.
3. For 5 data sets, the higher cluster includes models developed from 50% to 90%.
4. For 9 data sets, the higher cluster includes models developed from 25% to 90%.

Table 4.2 shows median and variance values of the AUC built from 10%, 50%, and 90% of data as training subsets. Figure 4.3 shows the corresponding boxplot diagrams. We note that performance increases due to the growing size of training samples in $code$ metrics based fault prediction models are more modest than in case of models built from $design$ metrics. However, the magnitude of performance improvement varies between projects and its difficult to anticipate.

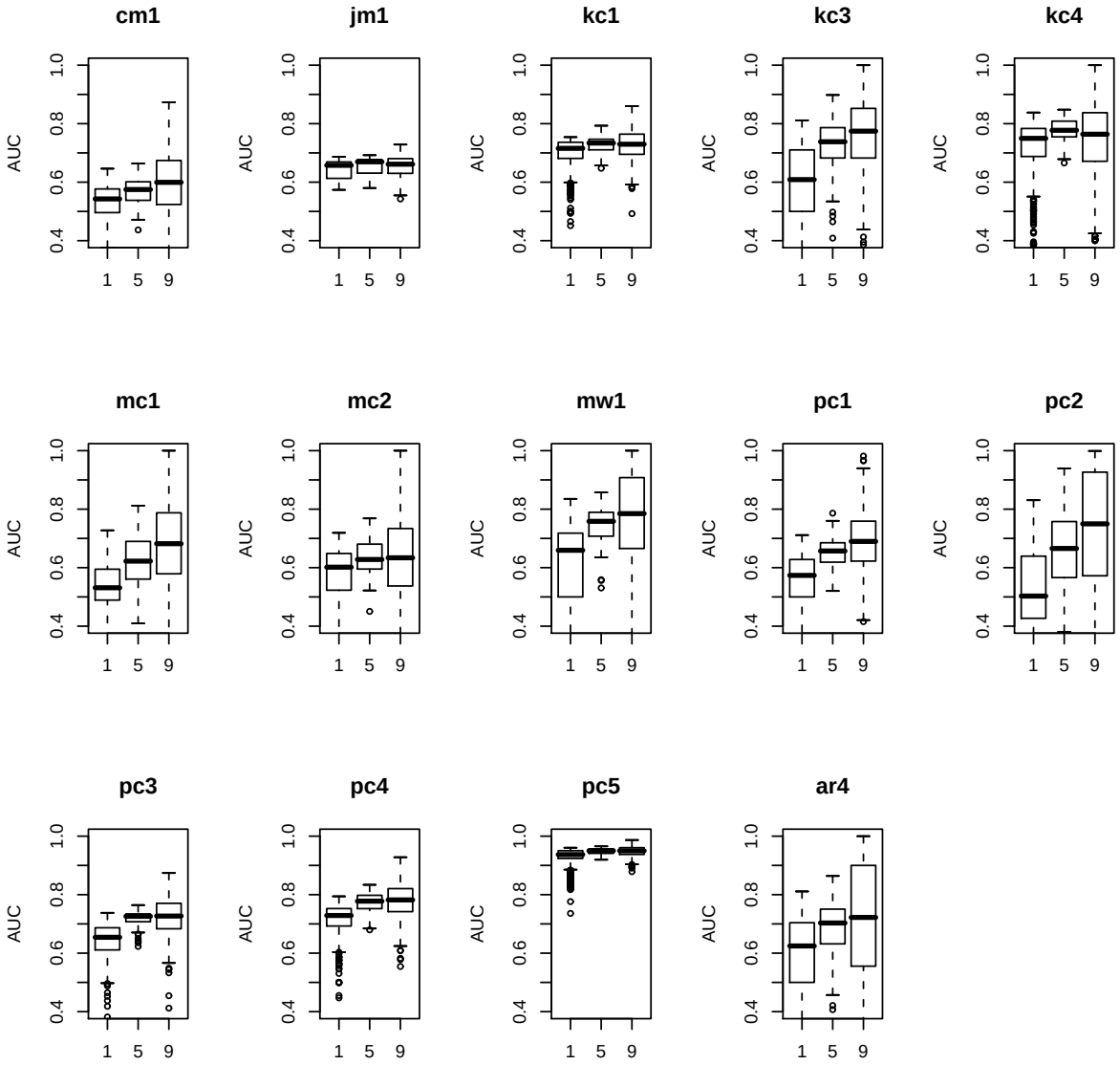

Figure 4.1. Boxplot diagrams of using 10%, 50%, and 90% data as training subset on $design$ metrics, measured by AUC. In x-axis, "1" stands for 10%; "5" stands for 50%; "9" stands for 90%.

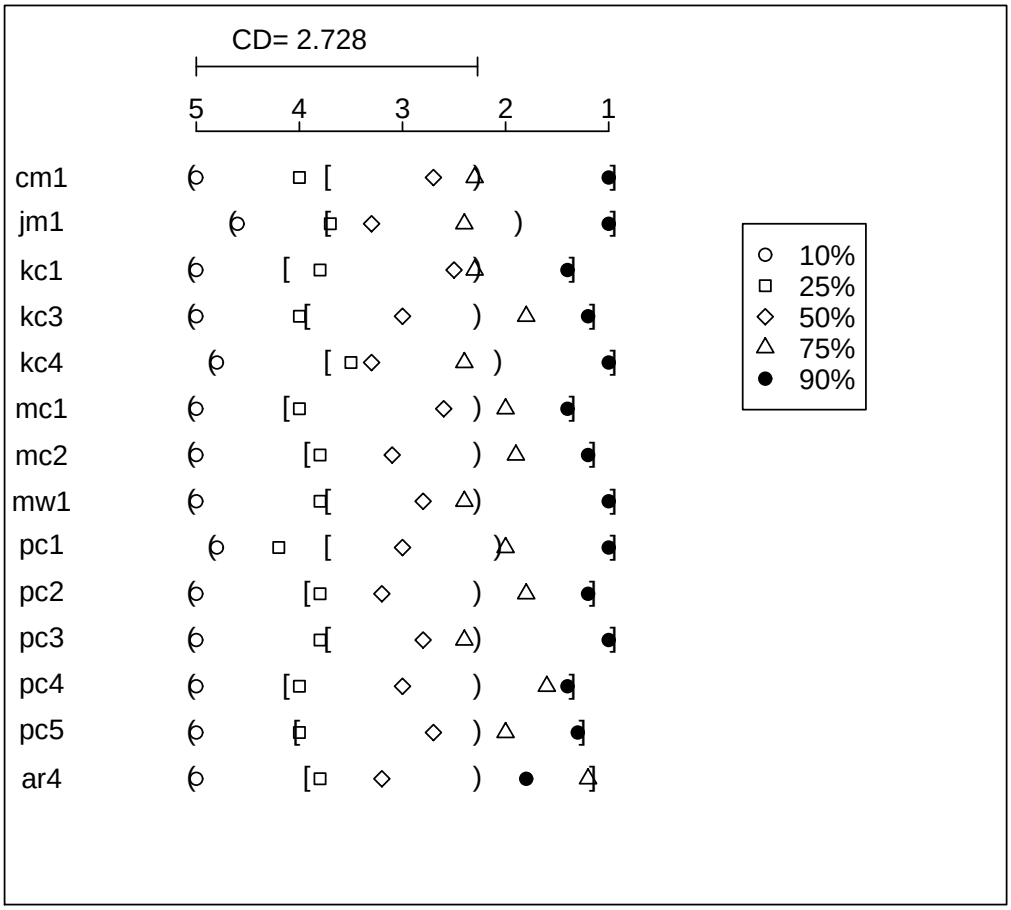

Figure 4.2. The Friedman test on $code$ metrics models evaluated using AUC.

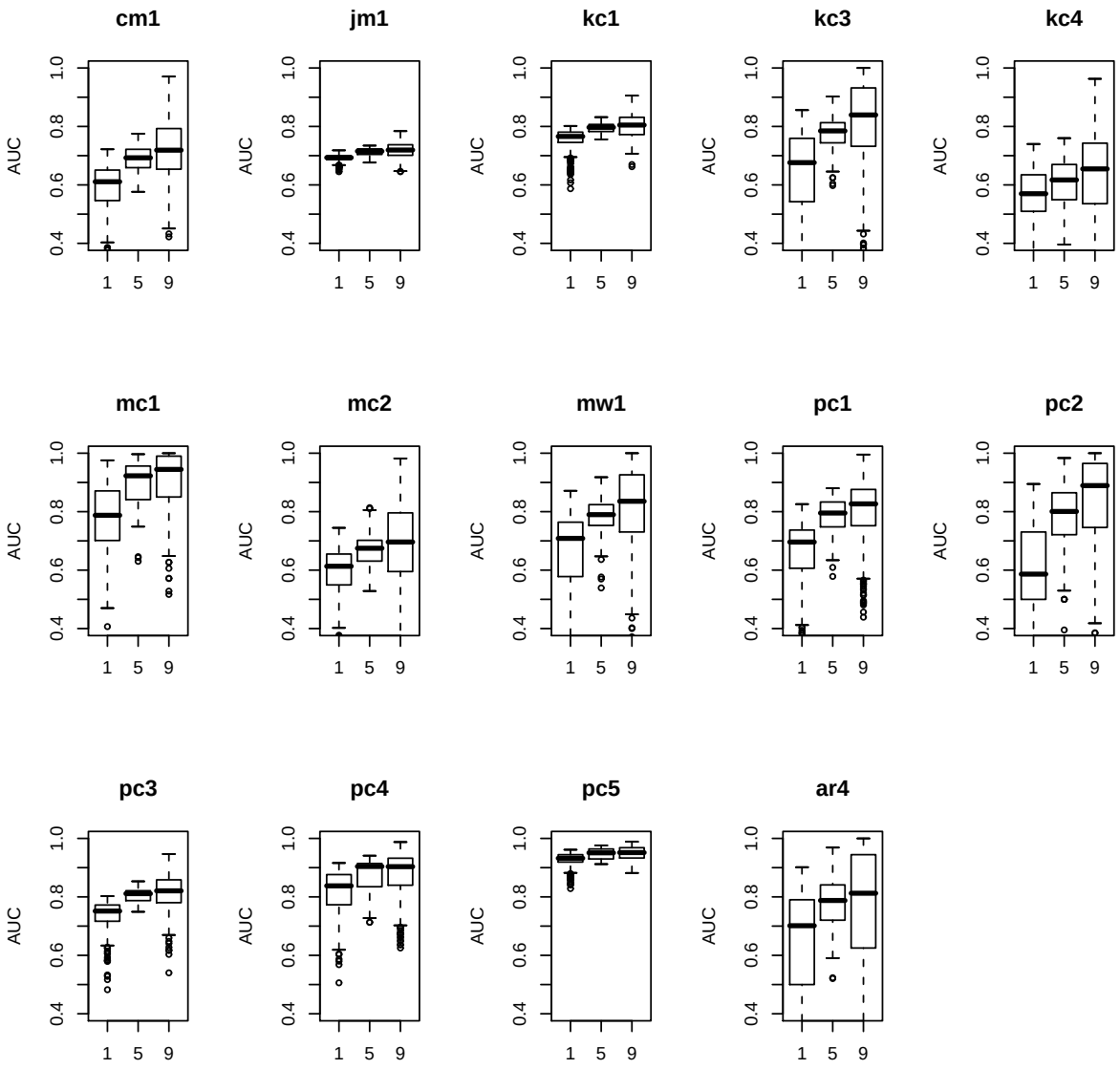

Figure 4.3. Boxplot diagrams of fault prediction models built from 10%, 50%, and 90% of data using $code$ metrics, measured by AUC. In x-axis, "1" stands for 10%; "5" stands for 50%; "9" stands for 90%.

Table 4.2. Median(m) and variance(v) for models trained from 10%, 50%, and 90% of modules using $code$ metrics, measured by AUC.

data	$m_{10\%}$	$v_{10\%}$	$m_{50\%}$	$v_{50\%}$	$m_{90\%}$	$v_{90\%}$	$\frac{m_{50\%}}{m_{10\%}} - 1$	$\frac{m_{90\%}}{m_{10\%}} - 1$	$\frac{m_{90\%}}{m_{50\%}} - 1$
cm1	0.61	0.0054	0.69	0.0019	0.72	0.0112	13.11%	18.03%	4.35%
jm1	0.69	1e-04	0.72	2e-04	0.72	7e-04	4.35%	4.35%	0
kc1	0.77	0.0011	0.80	3e-04	0.80	0.0017	3.90%	3.90%	0
kc3	0.68	0.0212	0.78	0.0041	0.84	0.0273	14.71%	23.53%	7.69%
kc4	0.57	0.0088	0.62	0.0082	0.65	0.0240	8.77%	14.04%	4.84%
mc1	0.79	0.0150	0.92	0.0065	0.94	0.0092	16.46%	18.99%	2.17%
mc2	0.61	0.0066	0.67	0.0032	0.70	0.0224	9.84%	14.75%	4.48%
mw1	0.71	0.0160	0.79	0.0047	0.84	0.0281	11.27%	18.31%	6.33%
pc1	0.70	0.0103	0.80	0.0037	0.83	0.0113	14.29%	18.57%	3.75%
pc2	0.59	0.0341	0.80	0.0139	0.89	0.0338	35.59%	50.85%	11.25%
pc3	0.75	0.0025	0.81	5e-04	0.82	0.0036	8.00%	9.33%	1.23%
pc4	0.84	0.0046	0.90	0.0036	0.90	0.0055	7.14%	7.14%	0
pc5	0.93	5e-04	0.95	3e-04	0.95	5e-04	2.15%	2.15%	0
ar4	0.70	0.0293	0.79	0.0089	0.81	0.0853	12.86%	15.71%	2.53%

All metrics

All metrics refers to the entire set of module attributes available for each fault prediction data set. These attributes include design and code metrics, as well as some software measures that combine them. The results of the ranking analysis for the incremental learning experiment with all metrics, using AUC for performance evaluation, are shown in Figure 4.4. A quick summary follows:

1. Regardless of the evaluation technique, jm1 does not show statistically significant difference for any training methods.

2. There are 10 data sets for which the lower cluster includes 10% to 50% subsets of available modules: cm1, kc1, kc4, mc1, mc2 pc1, pc2, pc4, pc5, ar4.

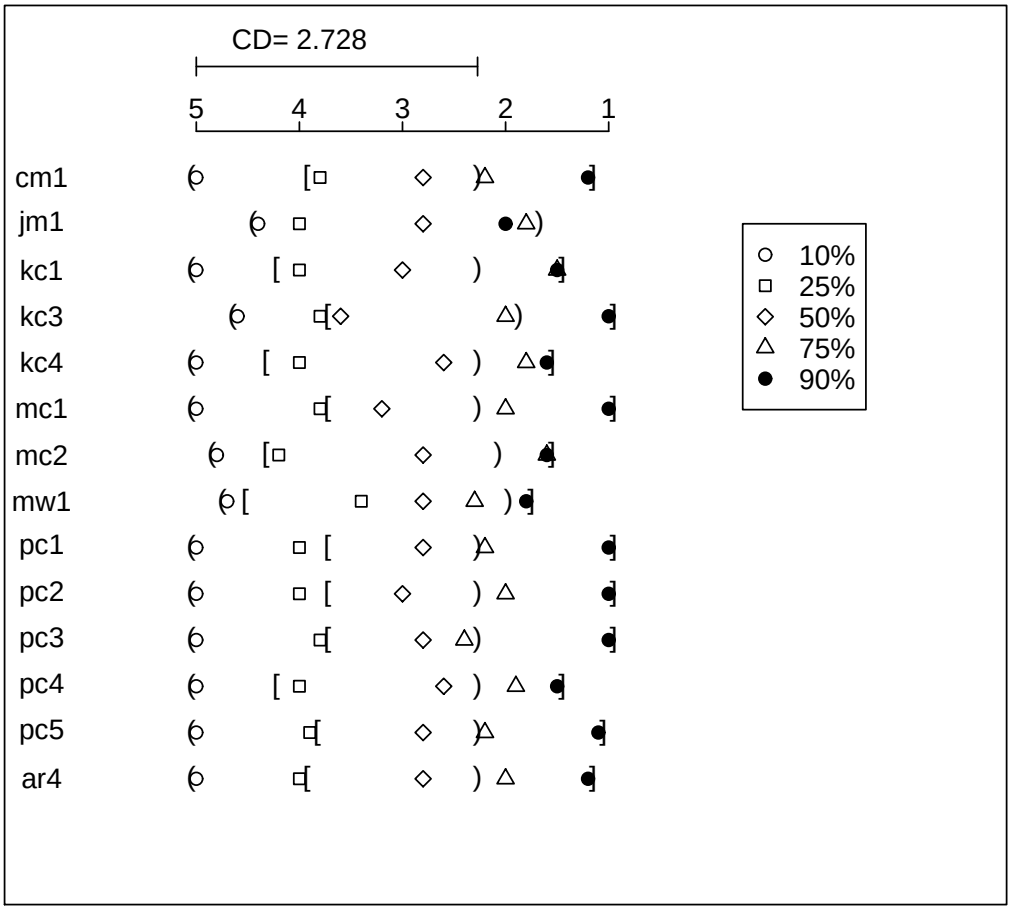

Figure 4.4. The Friedman test on all metrics using AUC.

3. There are 3 data sets for which the lower cluster includes models trained from 10% to 75%: kc3, mw1, and pc3.

4. There are 7 data sets for which the higher cluster includes models trained from 50% to 90% subsets: kc3, mc1, pc1, pc2, pc3, pc5, and ar4.

5. There are 6 data sets for which the higher performance cluster is from 25% to 90%: cm1, kc1, kc4, mc2, mw1, and pc4.

Table 4.3. Median(m) and variance(v) of models built from 10%, 50%, and 90% of module using all metrics, measured by AUC

data	$m_{10\%}$	$v_{10\%}$	$m_{50\%}$	$v_{50\%}$	$m_{90\%}$	$v_{90\%}$	$\frac{m_{50\%}}{m_{10\%}} - 1$	$\frac{m_{90\%}}{m_{10\%}} - 1$	$\frac{m_{90\%}}{m_{50\%}} - 1$
cm1	0.64	0.0047	0.72	0.0018	0.75	0.0084	12.50%	17.19%	4.17%
jm1	0.69	1.00E-04	0.72	3.00E-04	0.72	9.00E-04	4.35%	4.35%	0
kc1	0.76	0.0018	0.80	2.00E-04	0.80	0.0018	5.26%	5.26%	0
kc3	0.65	0.0177	0.76	0.0131	0.79	0.0263	16.92%	21.54%	3.95%
kc4	0.74	0.0104	0.81	0.0025	0.81	0.0132	9.46%	9.46%	0
mc1	0.80	0.0176	0.95	0.0058	0.97	0.0069	18.75%	21.25%	2.11%
mc2	0.62	0.0063	0.69	0.0042	0.70	0.0174	11.29%	12.90%	1.45%
mw1	0.70	0.0182	0.78	0.0049	0.84	0.0234	11.43%	20.00%	7.69%
pc1	0.70	0.0077	0.79	0.0028	0.83	0.0074	12.86%	18.57%	5.06%
pc2	0.59	0.0313	0.83	0.0229	0.88	0.0254	40.68%	49.15%	6.02%
pc3	0.74	0.0048	0.81	0.0011	0.82	0.0034	9.46%	10.81%	1.23%
pc4	0.85	0.003	0.91	0.0013	0.92	0.0024	7.06%	8.24%	1.10%
pc5	0.94	0.0012	0.96	2.00E-04	0.96	3.00E-04	2.13%	2.13%	0
ar4	0.70	0.0275	0.78	0.0078	0.83	0.0781	11.43%	18.57%	6.41%

Table 4.3 shows median and variance values of AUC for models built from 10%, 50%, and 90% of data as training subsets, as well as their comparison. Figure 4.5 shows the corresponding boxplot diagrams. An interesting observation inferred is that in case of comprehensive metrics attributes, fault prediction models never degrade when the size of the training set grows. This is similar to $code$ metrics models, but different from models which use $design$ information only.

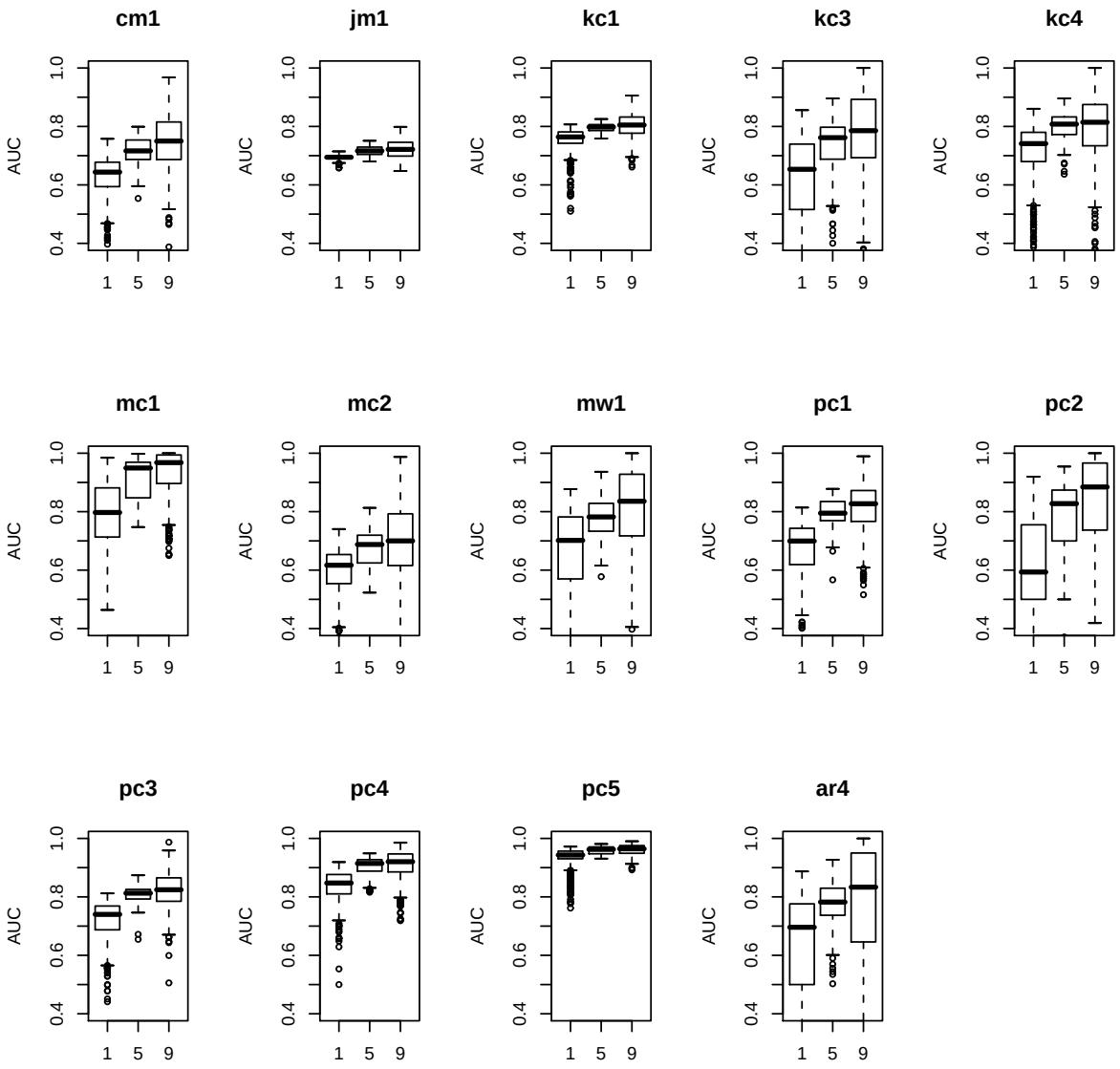

Figure 4.5. Box-plot diagrams of fault prediction models built from 10%, 50%, and 90% of data using all metrics, measured by AUC. In x-axis, "1" stands for 10%; "5" stands for 50%; "9" stands for 90%.

The analysis of all the experiments leads to the following perspective. Regardless of the type of metrics used for model development ($design$, $code$, and all), fault prediction models should be built as early as an initial set of modules becomes available. Using 10% of project modules for model development results in models that, when statistical significance is taken into account, are outperformed only by models built from 90% of the modules. Larger training set generally helps improve model performance, but the cost of incremental model rebuilding should be compared with the performance benefits. When justified, model rebuilding does not need to be a continual or a frequent activity. Rather, models could be built early and, possibly, updated at the mid point of the development.

4.2.2 Comparison of $design$, $code$, and all metrics models

The increase in size of available data for model definition is only one aspect in the incremental development of fault prediction models. The other opportunity comes from the fact that design artifacts and their metrics are typically available before the modules are implemented and code metrics can be computed. In rapid prototyping or agile processes, a few modules will be developed, implemented and tested in the first few process cycles. Their fault proneness status can be used to build models predicting the quality of design or code artifacts. Therefore, the question of comparing the performance of fault prediction models built from different types of metrics is important.

To guide statistical analysis comparing design, code, and all metrics based models, we define the following hypotheses:
H_{20}: *There is no difference in the performance of fault prediction models developed using the three different metrics groups.*
vs.
$H_{2\alpha}$: *Fault prediction models developed from (at least) one of the three metrics groups offer significantly different performance.*

The lesson learned in the previous section is that fault prediction models do not need frequent updates. For this reason, we decided it will be sufficient to compare $design$, $code$, and all models built from 10%, and 50% of the modules only.

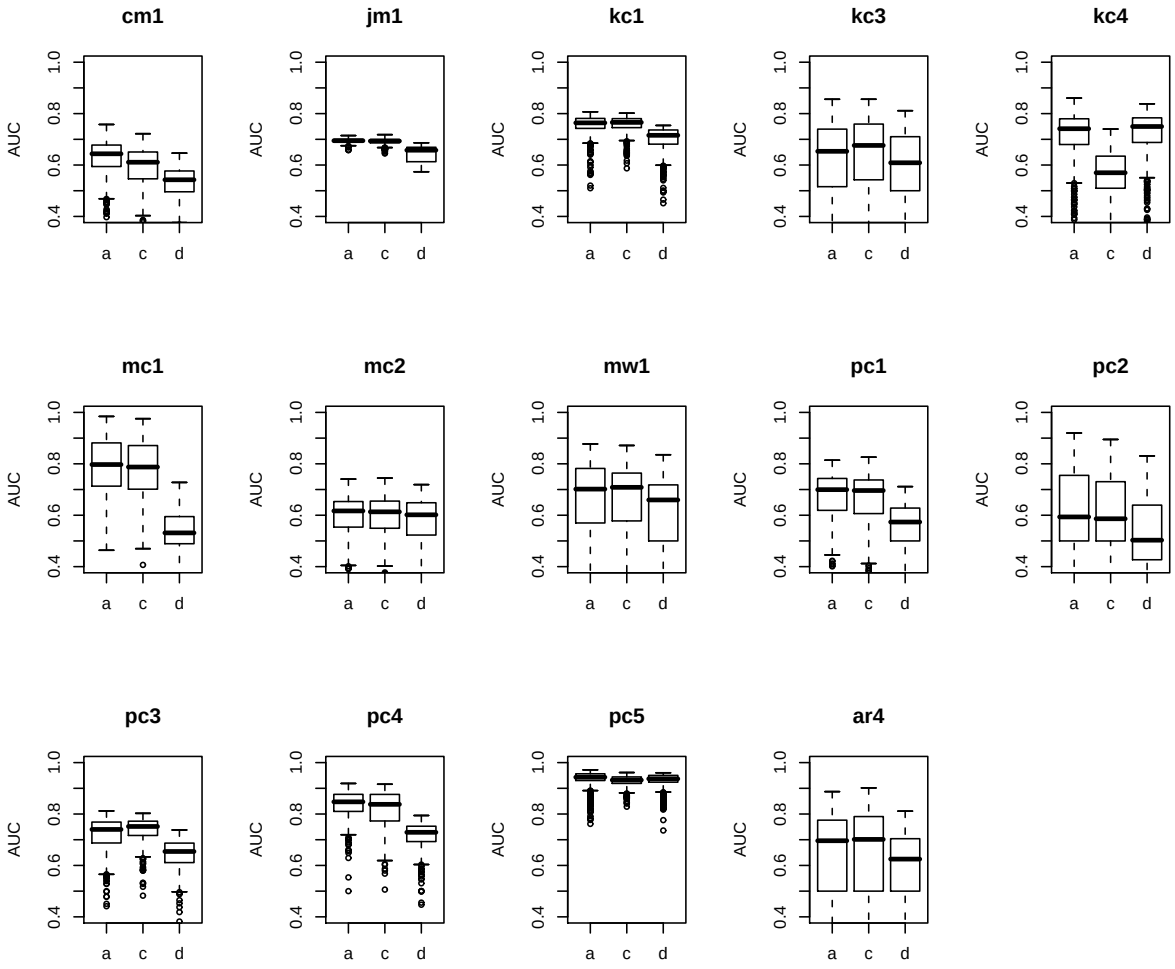

Figure 4.6. Boxplots comparisons of the performance of models built from $design$(d), $code$(c), and all(a) metrics using 10% data as training subset.

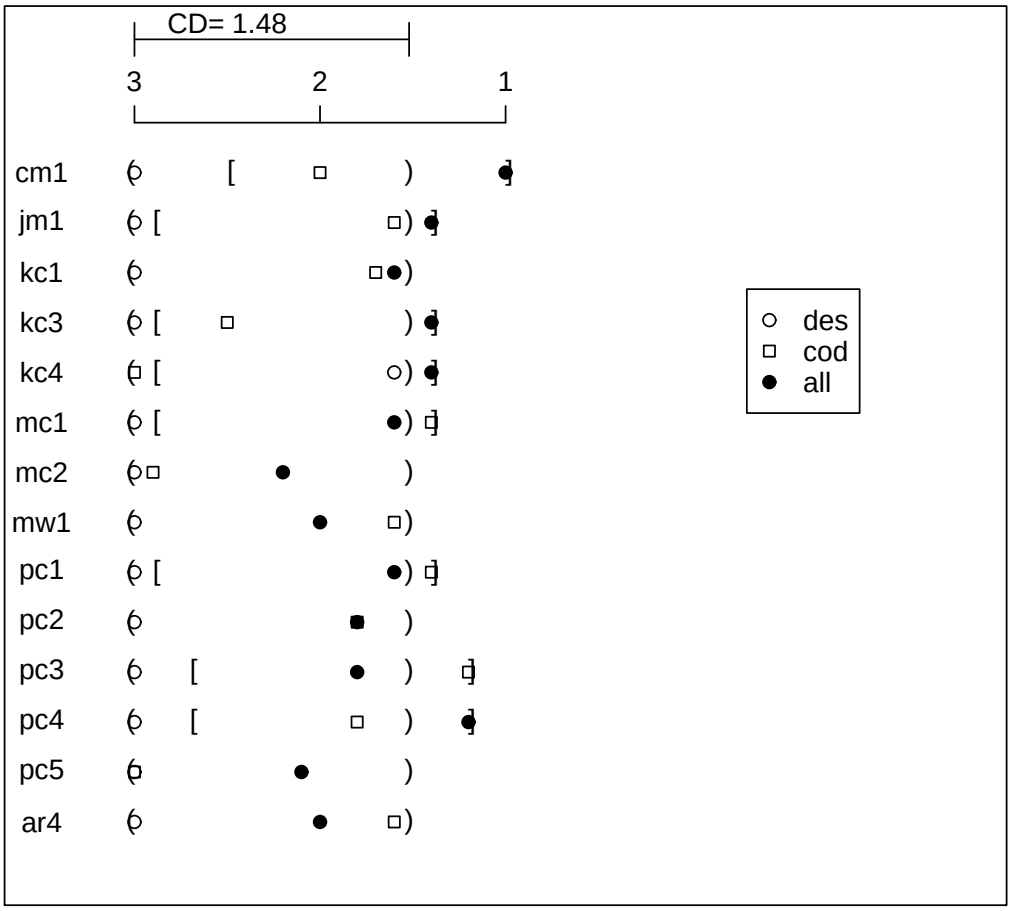

Figure 4.7. Statistical performance ranks of $design$, $code$, and all models built using 10% data for training.

Figure 4.7 depicts test result for experiments with all the 14 data sets. The diagrams compare the performance of models which use design, code, and all metrics built from 10% subsets. The figure indicate that the majority of models built from all metrics outperform those built from $code$ metrics, which in turn outperform $design$ metrics models. The statistical significance of these results will be tested following Demsar's procedure [23]. It is worth mentioning here that the reported performance reflects models which use all five classifiers. While there are differences between classification algorithms, reporting median AUC across all of them (following 10-way cross validation of each) minimizes the impact of the classifiers and emphasizes the inherent properties of metrics data sets. We study the impact of classification algorithms in the next section, but if Lessmann's result [53] are correct, models developed using different algorithms are not likely to have statistically significant performance differences.

Following the notation introduced earlier, Figure 4.7 introduces performance clusters based on the value of the Critical Difference, which for this experiment assumes value $CD = 1.48$. Enclosure of two or more models within either round brackets (lower performance cluster) or square brackets (higher performance cluster) indicate that they do not exhibit statistically distinguishable performance. We summarize the findings below.

- The performance of $design$ metrics models is ranked the lowest in 13 out of 14 data sets. The exception is project $kc4$ in which code metrics model was the weakest one. In $pc5$, the rank of $design$ and $code$ mdoels overlap.
- The performance of all metrics models is ranked the best in 9 out of 14 data sets.
- The performance of $code$ metrics models is ranked the best at 5 out of 14 data sets.
- The rank distance between $design$ and $code$ is typically greater than the distance between $code$ and all models.

From the statistical test point of view:

1. In 6 data sets, $design$, $code$, and all models demonstrate no significant difference.

2. In 5 data sets, $design$ $code$, and all fault prediction models form two perofrmance clusters: $design$ and $code$ models form the lower cluster (inside a pair of round brackets), $code$ and all form the higer cluster (inside a pair of square brackets).

3. In 3 data sets, the lower cluster includes $design$ and all models, while all and $code$ form the higher cluster.

We repeated the same analysis using models built from 50% subsets of data sets. The summary of the findings about models built from 50% data subsets is very similar to those we had about models built from 10% subsets.

More interesting observations emerge from the diagram in Figure 4.9. Using 50% of data for model development seems to stabilize performance trends. For example, except in $kc4$, in all other data sets $design$ metrics models offer the inferior performance. Even more interestingly, models built from all metrics outperform other models in all data sets. Please note that this was not the case with models built from 10% of data, where in 5 projects code models outperformed those drawing from all metrics as attributes.

Further statistical analysis of models built from the 50% subsets reveal that:

- Only one data set, $mw1$, offers $design$, $code$, and all models with statistically indistinguishable performance.
- In the remaining 13 data sets,the three models form two clusters. Except in $kc4$, all and $code$ models form the higher performance cluster, In $kc4$, all and $design$ models form the higher performance cluster.

We summarize the findings emerging from experimentation with $design$, $code$, and all metrics models built from 10%, 50% subsets as follows:

1. Although $code$ metrics models typically outperform $design$ metrics models, the difference in their performance measured by AUC index is not statistically significant.

2. Whenever possible, fault prediction models should be developed using a combination of $design$ and $code$ metrics. all metrics models typically outperform $design$ and $code$ metrics models. The difference in performance between all metrics models and $design$ metrics models is typically statistically significant.

3. Larger model training data sets, in this case 50% subsets, stabilize the expected model performance. More specifically, models built from 50% subsets almost uniformly offer

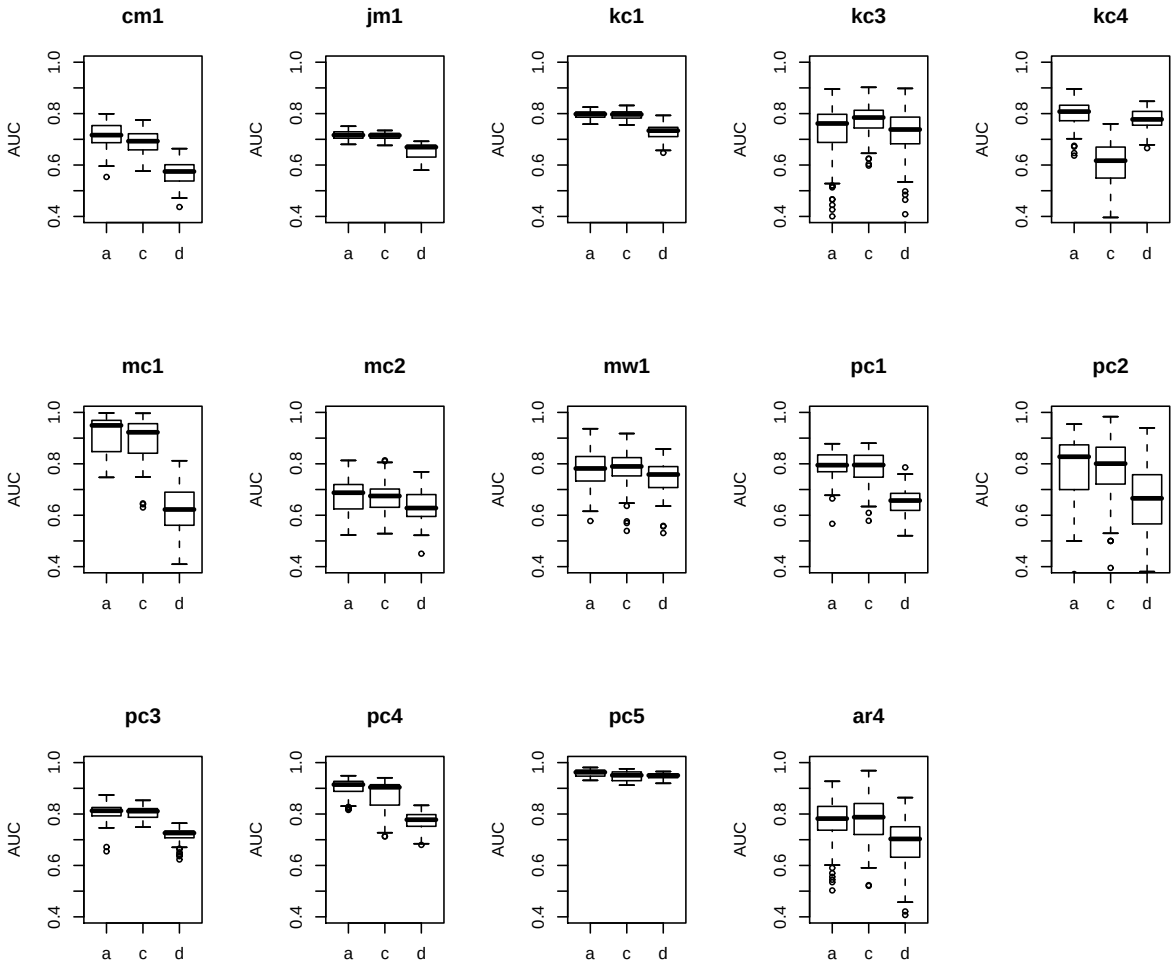

Figure 4.8. Boxplots comparison of the performance of all(a), $code$(c), and $design$(d) metrics using 50% of data for training.

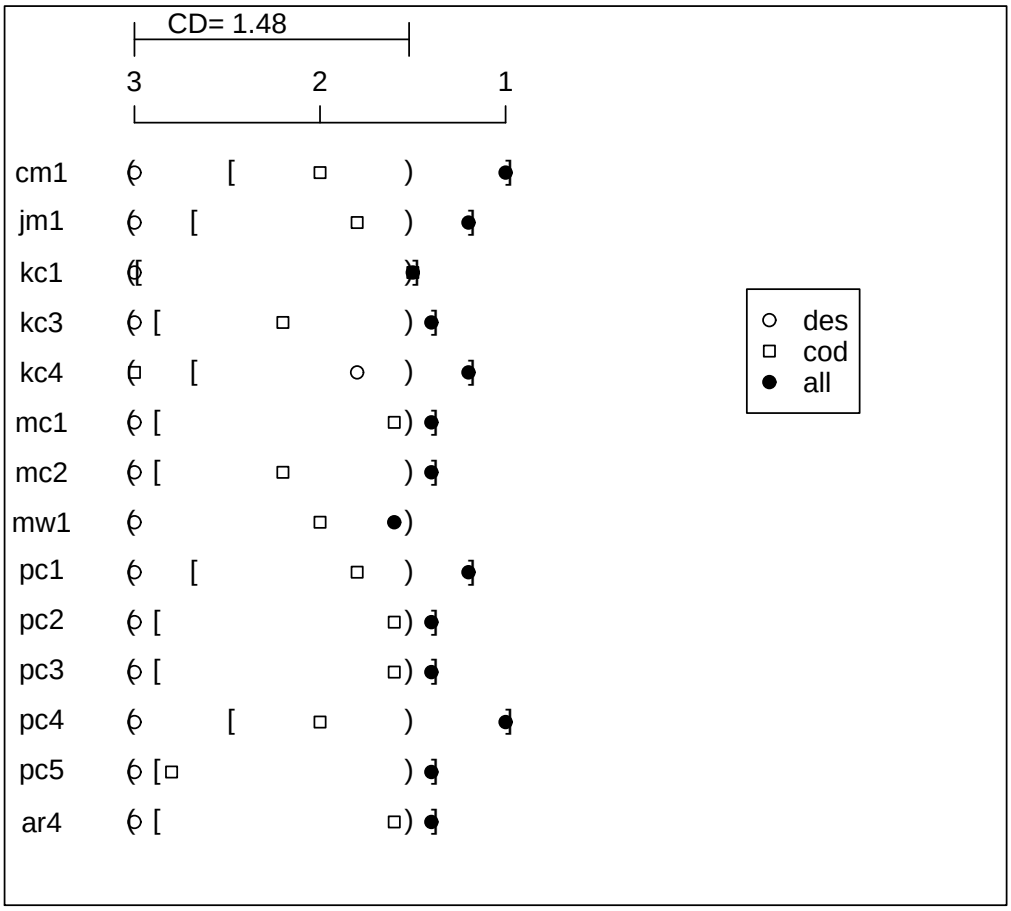

Figure 4.9. Statistical performance ranks of $design$, $code$, and all models built using 50% of data for training.

two rank performance clusters (design and code vs. code and all). In almost half of the models built from 10% data subsets, the performance of $design$, $code$, and all forms one rank performance cluster, i.e., their performance is statistically indistinguishable.

An interesting question arises regarding $kc4$ data set, in which design metrics outperform code metrics. While all other projects use C, C++, or Java, kc4 relies on the scripting language Perl. The use of Perl is an obvious project anomaly (in the context of the data sets included in this dissertation) and a possible cause for much higher proportion of faulty modules in $kc4$ (48%). While we can not offer a more detailed analysis of the causes of anomalous manifestations in fault prediction models for this data set, we are inclined to suggest a more thorough research consideration be given to software quality for projects which use scripting language like Perl.

4.2.3 Comparison of models developed using different classifiers

In the experiments reported so far, our results reflect median model performances achieved using all five classification algorithms. As we mentioned before, by collapsing the performance of all the classifiers and models into a single performance index (AUC), we intended to emphasize the inherent properties of metrics data sets. In this section, we reveal the performance impact of different classification algorithms.

A recent study by Lessmann et al. [53] indicates that most classification algorithms do not offer models which exhibit statistically significant performance differences. Sixteen out of 19 algorithms included in Lessmann's study belong to a "higher" rank cluster. In addition to the 10 data sets included in their study,we use four additional project data sets: mc1, mc2, pc5, and ar4. Out of five classifiers in our experiments, three (rf, log, and nb) were reported by Lessmann to achieve statistically indistinguishable performance. Further, Lessmann reports results only from all metrics models using $\frac{2}{3}$ of the data for training. We find some very interesting observations by experimenting with the three types of metric sets (design, code, and all) and by increasing the size of the training subset (10% to 90%). An additional difference between the two studies reflects our opinion that software quality practitioners are most likely to use off-the-shelf classifiers with their default parameter values. We follow this principle and report

median results from 10-way cross validation. Lessmann et al. use grid-search approach to find an optimal combination of algorithmic hyper-parameters which maximize the performance of each classifier and report the mean values of AUC.

Our statistical hypotheses for this experiment can be formulated as follows:
H_{30}: *Fault prediction models developed from the same data sets using five different classification algorithms do not result in statistically different performance.*
vs.
$H_{3\alpha}$:*At least two classifiers provide models with significantly different performance.*

Figure 4.10 the outcome of out experiments analyzed through Demsar's procedure. The 15 experiments (lines in Figure 4.10) reflect the three metrics groups and the five sizes of training subsets. These results can be summarized as follows:

- Regardless of the classification algorithm, for all sizes of training subsets, models developed from $design$ metrics have significantly indistinguishable performance.
- For all sizes of training subsets, models developed from all metrics are grouped in two performance rank clusters; Random forest (rf) models are consistently in the higher rank cluster.
- $Code$ metrics models fall in between. When trained on 10% to 50% of the data, they result in two rank clusters. When trained on larger subsets, all models fall into the single performance rank clusters.

These are very interesting insights. For all and $code$ metrics models, we can infer that the choice of classification algorithm in mature data sets (those where we train from larger subsets), consistent with Lessmann's results, matters less than when the models are built early. When the samples come earlier in the development life cycle from a smaller number of completed modules (10% or 25%), the choice of the classification algorithm matter more.

To gain better insights into these experiments, in Table 4.4, 4.5, and 4.6, we preview median and variance values of models developed for each group of metrics set: $design$, $code$, and all. The top median AUC for each project is marked in bold. We decided to report these values from experiments in which models are trained using 50% of data as training subset.

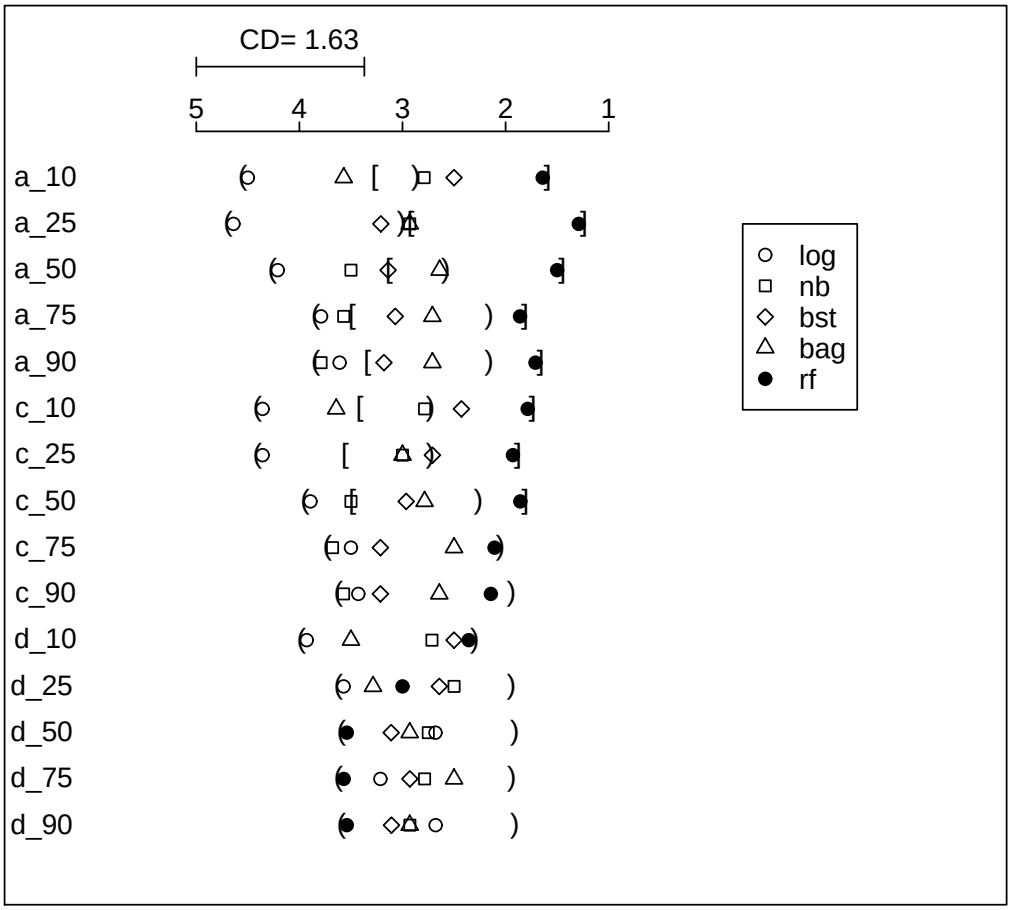

Figure 4.10. Comparison of 5 classification algorithms over different sizes of training subset and three metric groups. Label "a_10" (or "d_50"), for example, stands for all ($design$) metrics and 10%(50%) training subset. The reported results reflect performance ranks over all 14 data sets.

Table 4.4. Median and variance of AUC on $design$ metrics of using 50% data as the training subset

data	bag		bst		log		nb		rf	
	median	var	median	var	median	var	median	var	median	var
cm1	**0.59**	0.0023	0.57	0.0026	0.57	0.0017	**0.59**	9e-04	0.54	0.0018
jm1	0.67	0	0.67	0	**0.68**	0	0.63	4e-04	0.63	1e-04
kc1	0.73	5e-04	**0.74**	4e-04	**0.74**	5e-04	**0.74**	3e-04	0.68	4e-04
kc3	0.71	0.0167	0.73	0.0108	0.7	0.0151	**0.79**	0.004	0.74	0.0073
kc4	**0.79**	0.0011	0.76	0.0018	0.77	0.002	**0.79**	9e-04	0.75	0.0017
mc1	0.65	0.0067	0.6	0.0035	0.63	0.002	0.52	0.0034	**0.74**	0.0023
mc2	0.65	0.0022	0.62	0.0029	0.59	0.0027	**0.68**	0.0027	0.62	0.0026
mw1	0.76	0.0028	0.77	0.0019	0.75	0.0063	**0.79**	0.0016	0.71	0.0028
pc1	**0.68**	0.0023	0.66	0.0012	0.62	0.0021	0.63	0.004	**0.68**	0.0024
pc2	0.6	0.0091	0.71	0.0035	0.61	0.0202	**0.79**	0.0041	0.57	0.0094
pc3	**0.73**	4e-04	0.72	3e-04	**0.73**	4e-04	0.68	0.0016	**0.73**	4e-04
pc4	0.79	4e-04	0.79	3e-04	**0.81**	2e-04	0.75	0.0011	0.74	4e-04
pc5	**0.96**	1e-04	**0.96**	0	0.94	1e-04	0.94	1e-04	0.95	1e-04
ar4	0.7	0.01	0.68	0.0071	0.64	0.0108	**0.76**	0.0051	0.72	0.0041

Models built from 50% subsets are almost always statistically similar to those developed from larger and smaller training subsets.

From performance ranks in Figure 4.10, we find that random forest classifier achieves the highest average rank for all and $code$ metrics models. Of course, this does not mean that random forest perform the best for each data set and training subset size. Table 4.5 and 4.6 reveal almost half of code metrics models and a quarter of all metrics models feature a different classifier as the best one. This is not surprising and practitioners should be prepared to try out different classification algorithms. Given that applying off-the-shelf classifiers with (mostly) default parameter values is not costly or difficult, the exercise may be worth the effort. However, looking back on Figure 4.10, in many cases, the differences will be at the margin of statistical significance.

Table 4.5. Median and variance of AUC on $code$ metrics of using 50% data as the training subset

data	bag		bst		log		nb		rf	
	median	var	median	var	median	var	median	var	median	var
cm1	0.71	0.0018	0.66	0.0014	0.69	9e-04	0.67	0.0013	**0.72**	9e-04
jm1	**0.72**	0	0.71	1e-04	0.71	0	0.69	0	**0.72**	0
kc1	**0.8**	3e-04	0.79	3e-04	0.79	3e-04	0.79	2e-04	**0.8**	2e-04
kc3	0.77	0.0037	0.76	0.0037	0.76	0.0071	**0.82**	0.0022	0.79	0.0014
kc4	**0.67**	0.0021	0.66	0.0011	0.53	0.0026	0.55	0.0105	0.63	0.0039
mc1	0.94	0.0046	0.95	1e-04	0.88	0.0017	0.8	0.0046	**0.96**	0.0012
mc2	0.63	0.0038	0.64	0.0024	**0.69**	0.0046	**0.69**	0.0011	0.67	0.002
mw1	0.78	0.0051	0.8	0.0022	0.75	0.0089	**0.82**	0.002	0.79	0.0024
pc1	0.81	0.0017	0.79	0.0011	0.8	0.0015	0.7	0.0036	**0.84**	0.001
pc2	0.73	0.0226	**0.87**	0.0064	0.72	0.0093	0.85	0.0044	0.81	0.007
pc3	0.81	3e-04	0.81	3e-04	0.81	5e-04	0.78	5e-04	**0.82**	4e-04
pc4	0.91	1e-04	0.91	1e-04	0.84	4e-04	0.77	0.0012	**0.92**	1e-04
pc5	0.96	0	0.95	0	0.93	1e-04	0.93	0	**0.97**	0
ar4	0.78	0.0079	**0.82**	0.0069	0.67	0.0087	0.81	0.0054	0.8	0.0052

Table 4.4, 4.5, and 4.6 also reveal that AUC indices across the five classifier models (within each row) are similar to each other. For example, in Table 4.6, all classifiers perform very well on pc5 data set and quite poorly on mc2 data set. It is evident that the performance of a fault prediction model is determined more by the characteritics of a data set, rather than the differences between the classification algorithms. Similar variances represent further evidence.

It is also interesting to note that random forest classifier does not appear to rank well on $design$ metrics models, although its results belong to the same rank cluster with the other four algorithms. The number of attributes representing $design$ metrics is lower than the number of attributes in $code$ and especially, all models. It is known that the strength of random forest classifier are the data sets with a large number of attributes and a large number of instances [14].

Table 4.6. Median and variance of AUC on all metrics of using 50% data as the training subset

data	bag		bst		log		nb		rf	
	median	var	median	var	median	var	median	var	median	var
cm1	0.73	0.001	0.71	0.0019	0.70	0.0024	0.70	0.0011	**0.76**	8e-04
jm1	0.73	0	0.71	1e-04	0.71	0	0.69	0	**0.74**	0
kc1	0.8	1e-04	0.79	3e-04	0.79	2e-04	0.79	1e-04	**0.81**	1e-04
kc3	0.72	0.0052	0.76	0.0027	0.53	0.0166	**0.80**	0.0023	0.78	0.0015
kc4	**0.82**	0.0014	0.80	0.0024	0.81	0.0036	0.78	0.0025	**0.82**	0.0014
mc1	0.96	0.0048	0.95	2e-04	0.94	0.0039	0.80	0.0019	**0.98**	8e-04
mc2	0.67	0.0032	0.65	0.0047	0.64	0.0049	**0.72**	0.0014	0.70	0.0031
mw1	0.77	0.0048	0.79	0.0047	0.72	0.0063	**0.82**	0.0019	0.79	0.003
pc1	0.82	0.0013	0.80	9e-04	0.76	0.0047	0.77	7e-04	**0.85**	5e-04
pc2	0.74	0.0347	0.85	0.0048	0.65	0.0185	0.87	0.004	**0.83**	0.0071
pc3	0.81	3e-04	0.81	3e-04	0.82	7e-04	0.78	0.0017	**0.83**	3e-04
pc4	0.92	2e-04	0.92	1e-04	0.90	3e-04	0.84	3e-04	**0.94**	1e-04
pc5	**0.97**	0	0.96	0	0.95	1e-04	0.94	0	**0.97**	0
ar4	0.78	0.0036	**0.84**	0.0058	0.68	0.0067	0.80	0.0054	0.81	0.0035

4.3 Discussion

The experiments reported in this chapter have been motivated by the hypothesis that combining software metrics from different stages in the development benefits the accuracy of fault prediction models. This motivation includes the utilization of different metrics types, those available from software design or code, as well as the proactive use of measurements from software modules for the development of fault prediction models when they become available during project development. Publicly available NASA MDP and Promise repositories offer a substantial number of data sets, 14 of which we have used in the experiments. The large number of project data sets and a rigorous statistical test procedures we applied offer strong evidence in support of our conclusions, presented below.

The starting position for our analysis of the relative strengths of design and code metrics in building fault prediction models comes from Zhao et al. [91]. They claimed that design and code models perform comparatively well and that little improvement can be achieved if design and code metrics are jointly used in fault prediction. Our results from 14 data sets of NASA MDP show that although there is no significant statistical difference between design and code metrics, but the combination of design and code metrics usually outperform design metrics statistically. However, given the early available of design metrics,

The impact the size of the fault data set used for training has on model performance is also significant and interesting. One of the basic questions regarding the practicality of fault prediction is: When does the project have sufficient amount of data to build a model? Before we describe our recommendations, it is necessary to mention that in many organizations fault information from early (or earlier) product releases has been successfully used for fault prediction for the new release [71,82]. In organizations which practice software product lines or those where upgrades form the majority of project releases, data sufficiency is not a major problem. But there are many other organizations and projects which develop oneof- a-kind systems. Thirteen out of 14 data sets we analyzed come from such environments. These organizations must rely on the metrics from reused modules and those delivered and tested early in the development life cycle for building fault prediction models.

We believe the results reported here have been obtained following a valid experimental methodology, using publicly available data sets. As any other experimental study we are aware of potential validity threats too. For example, we mentioned that the design metrics used in the experiments have been reengineered from the code. While in principle similar metrics can (and have been) extracted from design documentation, it is likely that the metrics used in our experiments reflect the code more faithfully than the metrics collected at the design stage would. Had design models demonstrated better fault prediction performance than code models, one could argue that any reduction in the code-level details from attributes would have a tendency to improve performance. But, design models in our experiments do not perform as well as code models. Therefore, it seems logical that if design metrics do not reflect code structure as close as they do in our data sets, this would likely deteriorate the performance even further. Without additional research, we cannot offer further assurances.

It is also worth repeating here that the data sets we analyzed do not contain information

about when in the project development life time modules became available. To advocate incremental model development, we made the assumption that random selection of data subset used for training (repeated 10 times in each experiment) represents a valid sample of modules as if they became available before the modules we used for model evaluation. We are aware that software modules that become available early might suffer from quality deficiencies which projects reduce as their processes mature. If fault introduction is reduced over the life time of the project, our results about the suitability of fault prediction models built from smaller data subsets may be overly optimistic. Also, this might imply that updating models more frequently during the projects life time is warranted. Unfortunately, the data sets we use do not allow us to study this problem further.

Lastly, we did not investigate the impact of feature selection on model performance. Feature selection algorithms minimize the number of attributes used in model development based on some measure of their information content. Our experience with MDP datasets indicates that a smaller number of attributes (metrics), typically a dozen or less, could offer models that perform almost as well as the models which use the entire set of attributes [62]. This could have an impact on the effort invested in metrics collection. But, we have never been able to develop a model from a reduced set of attributes which outperforms models developed from comprehensive attribute sets. For this reason, we believe that feature selection is not likely to impact the validity of our results.

4.4 Summary

Our results confirm that the performance of design and code models, measured through the area under the ROC curve (AUC), is typically statistically indistinguishable. In other words, although code metrics based models outperform design metrics models, the performance margin is not statistically significant. On the other hand, the performance of models built from all metrics (which include design and code metrics) typically outperform design models by a statistically significant margin. Our experiments, therefore, offer support for utilizing a combination of design and code metrics in building fault prediction models. However, if design metrics are available earlier, design models should not be discarded as they offer meaningful fault prediction performance.

Most results from our experiments which test the impact of the training data size on the fault prediction performance are not surprising. When derived from a larger data set, model performance improves. The interesting aspect of our results comes from statistical hypothesis testing. In simple terms, the performance margin between models derived from 50% data subsets and those derived from just 10% is not statistically significant. Further, models built from 50% data subsets and 90% data subsets typically belong the the same performance cluster too (but models built from 10% and 90% do not). The implication of this result is, we believe, very positive. Models developed from data sets, presumably early in the project life time, offer fault prediction capability comparable with models that can only be developed much later. Therefore, while updating the fault prediction model is a good idea, it does not have to be practiced often. This conclusion offers the real chance to optimize the cost of fault prediction model development. Fault prediction models, in turn, optimize the cost of verification and validation activities.

We believe the results from both experiments support the general hypothesis: fault detectors can be improved by increasing the information content of the training set. Therefore continuing to explore the effects of combining the attributes from multiple phases of the development life cycle, process and business related attributes, appears to be the most promising research direction [40, 69, 85].

In the third group of experiments, we examined the impact of the selection of the classification algorithm in fault prediction modeling. In the recent paper, Lessmann [53] offers convincing arguments that most classification algorithms offer statistically the same performance (i.e., their performance differences are not significant). One limitation of Lessmann's analysis is the uniform use of $\frac{2}{3}$ of the data for model training. In our experiments we varied the size of the training subset and the type of metrics used for training. We observed that statistically significant differences between classification algorithms do occur when models are developed from smaller training subsets. Further, significant differences are more likely to occur when training from all and code metrics than from design metrics. Consequently, we recommend experimentation with several classification algorithms, recommended in the literature (for example [53]) for fault prediction modeling.

Combined, the outcomes of our experiments provide a good guidance for an incremental process for software fault prediction modeling. Models can be built early, from design metrics

and/or from relatively small subsets of available data. Updating such early models is recommended. The frequency of such updates can be optimized as model performance gains justify intermittent (sporadic) upgrades, rather than recurring ones.

We conclude this chapter by submitting our findings with the questions posed in the dissertation:

- Q2: Are design metrics good enough to be fault-proneness predictors?
  Answer: Yes. Design metrics are useful fault-prone predictors.
- Q3: Are code metrics good fault-prone predictors?
  Answer: Yes.
- Q4: Do fault prediction models built from a combination of requirement, design, and code metrics provide better performance than models built from any metrics subset?
  Answer: The performance of the combination models is usually better than design or code metrics alone.
- Q5: How large is a training subset needed for building meaningful fault prediction models?
  Answer: 10% subset data can be used to train models, this allow software fault prediction early in the project life cycle.
- Q6: How large is a training subset needed in order to achieve the best performance fault prediction models?
  Answer: Using 50% data as training subset will achieve the best performance models statistically.

Chapter 5

Cost-specific Fault Prediction Models

This chapter will investigate the following problems:

- Q7: How can we incorporate different misclassification costs into fault prediction models?
- Q8: How can we achieve the lowest misclassification cost for a given data set?
- Q9: How can fault prediction models guide different risk level projects from misclassification cost perspective?
- Q10: How can we evaluate individual module's specific misclassification cost in fault prediction models?

In this chapter, we first introduce the cost curve, a novel evaluation technique to evaluate fault prediction models. With the introduction of cost curves, we are able to answer Q7 and Q8. Then, we will tackle problems Q9 and Q10 through the experiments. Finally, we will discuss our results.

5.1 Cost Models

While the techniques for model development which explicitly account for misclassification cost differential have not been studied in software fault prediction modeling, there have been

attempts to include cost factors into model evaluation. *F-measure*, for example, offers a technique to account for the cost factor [39] when comparing different models. Khoshgoftaar and Allen [49] proposed the use of prior probabilities of misclassification to select classifiers which offer the most appropriate performance. In [50] they compare the return-on-investment in a large legacy telecommunication system when $V\&V$ activities are applied to software modules selected by a quality model vs. at random. Cost has been considered in test case selection for regression testing too [28].

There is a steady trend in fault prediction modeling literature recommending model evaluation with lift charts [39], sometimes called Alberg diagrams [72, 74]. Lift is a measure of the effectiveness of a classifier in the detection faulty modules. It calculates the ratio of correctly identified faulty modules with and without the predictive model. Lift chart is especially useful when the project has resources to apply verification activities to a limited number of modules. Cost effectiveness measure described by Arisholm *et al.* [7] can account for the nonuniform cost of module-level $V\&V$.

5.1.1 Cost Curve

Cost curve, proposed by Drummond and Holte [26], is a visual tool that allows us to describe classifier's performance based on the cost of misclassification. The x-axis represents probability cost function, denoted PC(+). Its y-axis represents normalized expected misclassification cost. It indicates the difference between the maximum and the minimum cost of misclassifying faulty modules.

Let us denote the faulty module with a "+" and a fault-free module as "-". $C(+|-)$ denotes the cost of incorrectly predicting a fault-free module as faulty. $C(-|+)$ represents the cost of misclassifying a faulty module as being fault-free. p(+) and p(-) are the probabilities of a software module being faulty or fault free when the project is deployed. Because we are uncertain about the value of p(+) and p(-), thus, we plot all kinds of possible values of p(+) (implicitly p(-)) in cost curve. These probabilities can become known only after the deployment of the model, since the proportion of faulty modules in model's training and test sets are approximations of the proportion of faulty modules the model will encounter during its field use. Cost curves support visualization of model's performance across all possible values of p(+) and p(-),

offering performance predictions for various deployment environments. Equations 5.1 and 5.2 present the formulae for computing the values that appear on x-axis and y-axis:

$$x - axis = \frac{p(+) * C(-|+)}{p(+) * C(-|+) + p(-) * C(+|-)} \tag{5.1}$$

$$y - axis = (1 - PD - PF) * PC(+) + PF \tag{5.2}$$

Typical regions of a cost curve are shown on Figure 5.1. The diagonal line connecting point (0,0) to (1,1) stands for a trivial classifier(TC) that always classifies all the modules to fault-free. And the points in the area above this line indicate the performance of a classifier is worse than the trivial classifier. The other diagonal line, connecting point (0,1) to (1,0), stands for another trivial classifier that always classifies all the modules as faulty modules. And the points above this line are cases where the performance of a classifier is worse than this trivial classifier. The horizontal line connecting point (0,1) to (1,1) stands for the extreme situation when the model misclassifies all the modules. The x-axis line represents for the ideal model that correctly classifies all the modules. From this typical cost curve, we find that to draw the y-axis in the range of 0 to 0.5 is enough as the lower half of the curve is the most interesting part because we are not interested in a classifier which perform worse than trivial classifier. As the figures we present in the following, we restrict y-axis from 0 to 0.5.

An ROC curve connects a set of calculated (PF, PD) pairs. Cost curves are generated by drawing a straight line connecting points (0,PF) to (1,1-PD), corresponding to (PF, PD) point in the ROC curve. After drawing all straight lines that have the counterpart points in the ROC curve, the lower envelope of cost curve is formed by connecting all the intersection points from left to right. Figure 5.2 shows an example of a cost curve resulting from the application of Logistic classification model in project KC4. The lower envelope of the cost curve corresponds to the convex hull in the ROC curve. ROC curves and cost curves are closely related: a point in the ROC curve corresponds to a line in the cost curve.

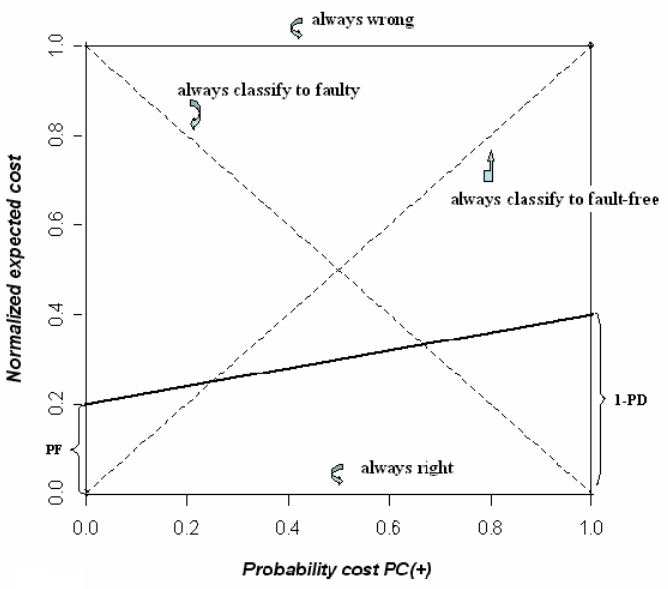

Figure 5.1. Typical regions in cost curve.

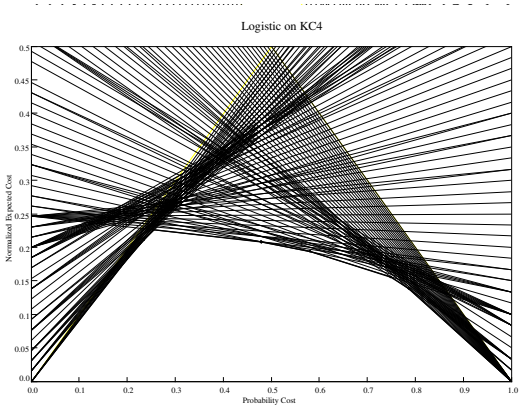

Figure 5.2. A cost curve of logistic classifier on KC4 project data.

The Meaning of Classification Result

Assume the ratio of misclassifying a fault-free module to that of misclassifying a faulty module is φ, that is, $\varphi = C(+|-):C(-|+)$. Equation 5.1 can be rewritten in term of φ as:

$$x - axis = PC(+) = \frac{p(+)}{p(+) + p(-) * \varphi} \tag{5.3}$$

In Equation 5.3, p(+) is the percentage of faulty module in the project, and p(-) = 1- p(+). Thus, if we are interested in a particular value of φ, we can calculate the corresponding PC(+) value easily according to Equation 5.3. In the mean time, if we know PC(+) value, the corresponding φ can be derived directly. For instance, say, in KC4 data, from Table 2.3, we can see that the proportion of faulty modules in KC4 is 48%. Therefore, $p(+) = 0.48$ and $p(-) = 1 - 0.48 = 0.52$. (1) In the case of $\varphi = 1$, that is, the cost of misclassifying a faulty module and the cost of misclassifying a fault-free module are the same, $C(+|-)$ = $C(-|+)$, according to Equation 5.3, we get that $PC(+) = 0.48$. (2) In the case of $\varphi = 5$, that is, the misclassification of a fault-free module cost 5 times more than that of the misclassification of a faulty one, we have $C(+|-) : C(-|+)$ = 5. Taking this to Equation 5.3, we have $PC(+) = 0.156$. (3) In the case of $\varphi = \frac{1}{5}$, that is, $C(+|-) : C(-|+) = \frac{1}{5}$, we have $PC(+) = 0.82$.

Figure 5.3 shows cost curves of 5 classifiers (bagging, boosting, logistic, naiveBayes, and random forest) on KC4 data. The places where $\varphi = 100, 5, \frac{1}{5}, \frac{1}{100}$ are indicated by vertical dashed lines of corresponding points at 0.009, 0.156, 0.82, and 0.99 respectively. If we are interested in a particular point (or a particular misclassification ratio),i.e., $PC(+) = 0.48$ ($\varphi = 1$), we should choose a classifier that offers the minimal misclassification cost along the corresponding vertical line ($PC(+) = 0.48$). From Figure 5.3,we can see that when $PC(+) = 0.48$, the best model which has the minimal misclassification cost is boosting. When $PC(+) < 0.48$, the misclassification cost of fault-free modules is greater than that of misclassifying a faulty module ($\varphi > 1$). The region $PC(+) < 0.48$ stands for models which are adequate for "cost-adverse" projects in software engineering. The cost region of $PC(+) > 0.48$ represents models which are adequate for "risk adverse" projects, those where misclassifying a faulty module is significantly more consequential ($0 < \varphi < 1$). Cost curves open significant opportunities for cost-based software quality management. Misclassification costs can dictate the preference for

the model and model parameters which are the most appropriate for the given project.

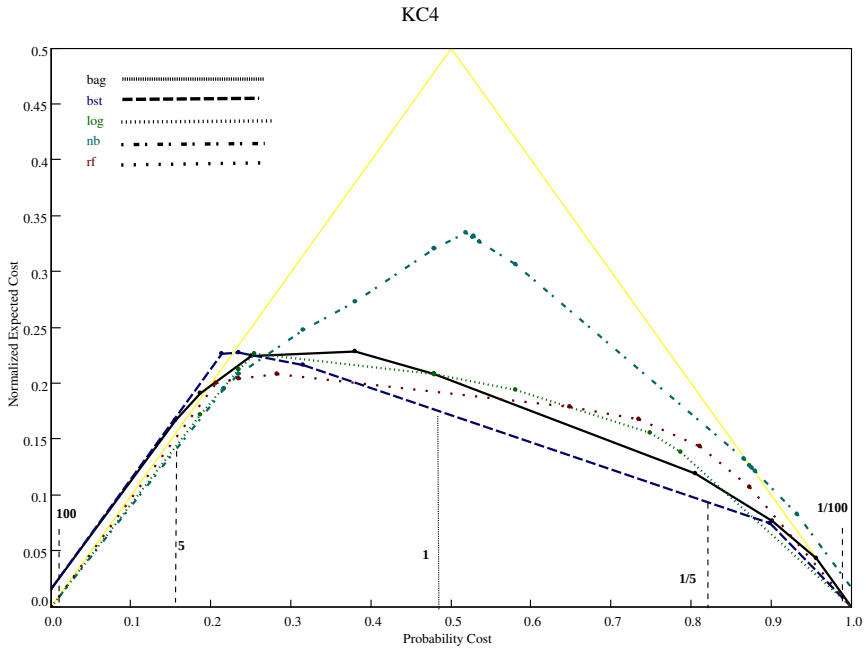

Figure 5.3. Cost curve of 5 classifier on KC4.

The goal of cost curve is to minimize the misclassification cost, that is, minimize the lower envelope area. The smaller the area under the lower envelope boundary is, the lower the misclassification cost would be and, hence, the better the performance of the classifier. However, it is likely that a single model will not be equally good in all cost regions. If a project evolves to an understanding that the misclassification cost differential is not the same one used in the past, the project may decide to change the quality model it uses even though the underlying metric distributions in the dataset remain the same. Cost curves support this type of decisions. For instance, looking back to Figure 5.3, in the interval (0.22, 0.385), random forest has the best performance; in the interval (0.385,0.88), boosting outperforms the other learners; finally,in the range from 0.88 close to 1.0, logistic model turns out to have the minimal misclassification.

In Figure 5.3, when PC(+) is from 0 to 0.2, bagging and boosting perform worse than trivial classifier which simply classifies every module as faulty-free. When PC(+) is from 0 to 0.05, the

other three learners logistic, naiveBayes, and random forest overlap with the trivial classifier, it means their performance are similar to that of the trivial classifier. Let's look at the other end where PC(+) close to 1. After $PC(+) > 0.87$, naiveBayes performs worse than the trivial classifier which classifies each module as faulty one. After $PC(+) > 0.95$, bagging overlaps with the trivial classifier too. We can see that at the both ends of x-axis where PC(+) is either close to 0 or 1, the trivial classifier is not a bad classifier with allowable performance. Which may means in some extreme situations, extreme cost adverse project and risk adverse project, the V&V activity of check nothing (extreme cost adverse) and check everything (extreme risk adverse) would be meaningful.

Cost curves have the ability to reveal the differences among classifiers from perspectives that are not obvious in other commonly used curves such as Receiver Operating Characteristic (ROC). Figure 5.4 demonstrates their differences using MC2 dataset as an example. The cost curve, shown in panel (b) of Figure 5.4 provides a clear answer to the comparison of two classifiers: Logistic classifier is better than random forest (rf) because its lower envelope is always below that of random forest. Deriving this conclusion from ROC curves would be difficult.

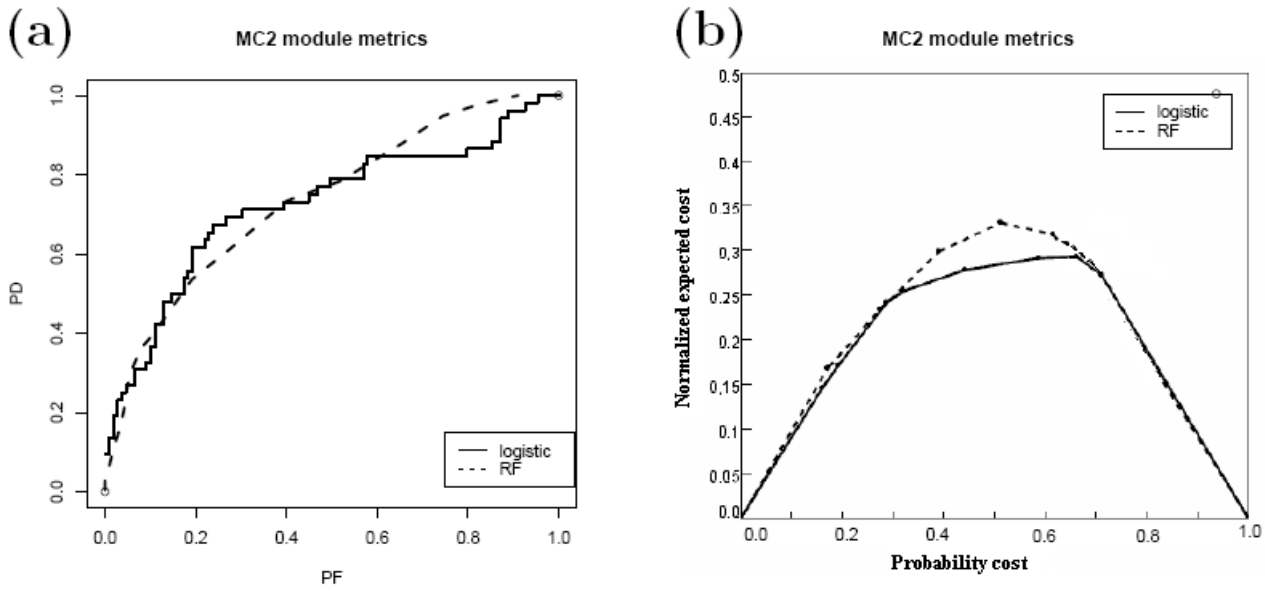

Figure 5.4. (a) A ROC curve (b) Corresponding cost curve of MC2.

Statistical Significance in Cost Curve

The measure of classifier's performance is derived from a confusion matrix. In cost curve, as well as in other performance analysis graphs, a confidence interval is generated from a series of confusion matrices. We illustrate the use of confidence bounds on cost curves, following the procedure outlined by Drummond and Holte in [26]. The procedure is based on the resampling of typically 500 confusion matrices using the bootstrap method. The 95% confidence interval for a specific $PC(+)$ value, for example, is obtained by eliminating the highest and the lowest 2.5% of the normalized expected cost values.

In some ranges of the curve, the performance between two classifiers may have significant difference, but in the other ranges, it may not. Figure 5.5 shows an example of the difference in performance between two classifiers, IB1 and j48 on PC1 dataset. The shaded area in Figure 5.5 represents the 95% confidence interval. There are three lines inside the shaded area. The middle line in the shaded area represents the difference between the means of the two classifiers IB1 and j48. The other two lines along the edges of shaded area represent the 95% confidence interval of the difference between the mean of two classifiers. If the confidence interval contains zero line, then there is no significant difference between the two classifiers; Otherwise, there is. The larger the distance of the confidence interval from the zero line, the more significant the difference between the two classifiers. Referring to Figure 5.5, when PC(+) is in the range of (0.0, 0.1), the confidence interval is above the zero, indicating IB1 outperforms j48. In the range (0.27, 0.59), the confidence interval is below the zero line, indicating that j48 performs better than IB1. Between (0.1, 0.27) and (0.59, 0.64), the confidence interval contains the zero line indicating no significant difference between the two classifiers. When $PC(+) > 0.64$, the performance of the two classifiers is the same. Given that the proportion of faulty modules in project PC1 is 7%, at $PC(+) = 0.07$ the misclassification costs for faulty and fault-free classes is the same, $C(+|-)$ = $C(-|+)$. When $PC(+) < 0.07$ the misclassification of fault free modules outweighs the misclassification of faulty modules. The opposite is true for $0.07 < PC(+)$. Cost curve analysis shows that (1) j48 is a bad choice when $PC(+) < 0.07$ (worse even than the trivial classifier); (2) j48 outperforms IB1 in $(0.27, 0.59)$ region; (3) in other regions of the cost curve, IB1 and j48 perform similarly.

The confidence band in cost curve reveals aspects which cannot be inferred from the sta-

tistical analysis of AUC in ROC curves. When misclassification costs are known or can be guessed, cost curves and their statistical analysis provide the most meaningful guidance for model selection.

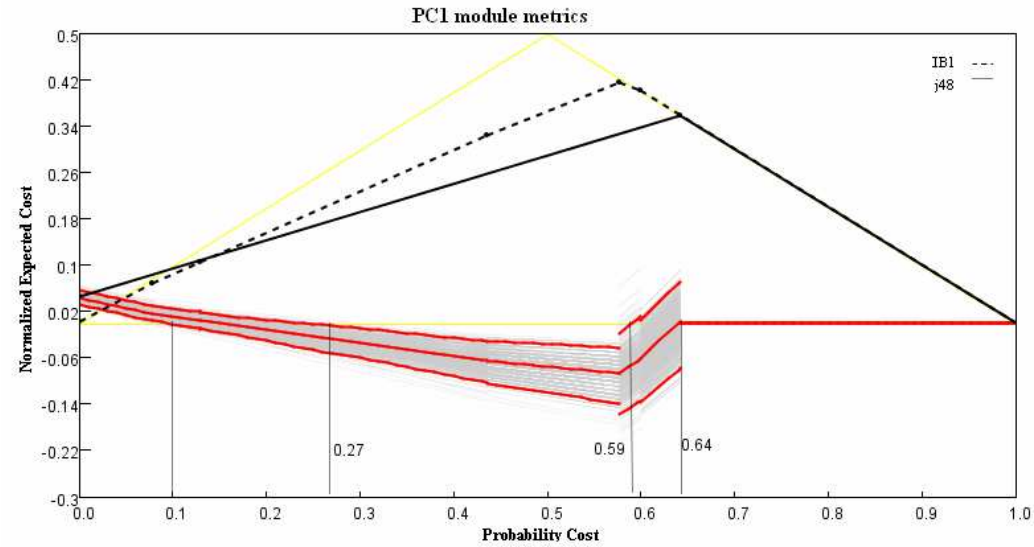

Figure 5.5. The 95% cost curve confidence interval comparing IB1 and j48 on PC1 dataset.

5.2 Experimental Design

Given the problems stated in the dissertation, we carried out our experiments in the following steps.

- Compare the "best" performance classifier to the trivial classifier (TC) to justify the benefit of using software fault prediction models.
- Compare the "best" and the "worst" performance classifiers for those "worst" classifiers are not worse than the trivial classifier (TC).
- Investigate which classifier will generate the lowest misclassification cost for a module with specific risk level.

In these experiments, we used the five classifiers: random forest(rf), boosting(bst), logistic(log), naiveBayes(nb), and bagging(bag), and all 16 data sets from the NASA MDP repository [2] and

PROMISE (3 datasets) [13] which are shown in Table 2.3.

5.3 Comparison of the "Best" Classifier Against the Trivial Classifier

In practice, it is difficult to determine the exact value of misclassification ratio $\varphi = C(+|-) : C(-|+)$. But it should not be difficult to estimate project's approximate risk level in terms of a range of probability misclassification costs. For this chapter, we divide projects into three groups based on their misclassification risk levels. We discussed this simplistic grouping with developers involved in projects we analyze and they found it reasonable. A project exhibits a low risk level if its misclassification ratio $\varphi = C(+|-) : C(-|+)$ is in range of (100,5). In simple terms, misclassifying a fault free module as fault module is highly undesirable, 5 to 100 times more than misclassifying a faulty modules as fault free. These are clearly cost adverse projects, trying to optimize $V\&V$ activity and speed up product delivery. A project is assigned a $high$ risk level if the ultimate goal of analysis is to identify as many fault prone modules as possible. Such risk averse projects typically allocate a substantial portion of their budgets to verification activities. Their range of misclassification cost ratio $\varphi = C(+|-) : C(-|+)$ is $(\frac{1}{5}, \frac{1}{100})$. $Medium$ risk projects are the remaining ones, for which the misclassification cost ratio φ is in the interval $(5, \frac{1}{5})$.

The risk levels for 16 data sets used here have been determined in discussion with software quality engineers at NASA, and the metrics collection experts at the white goods manufacturer shown in Table 5.1. Seven NASA control and navigation software projects are categorized as having $high$ risk level. The three data sets from the white-goods manufacturer are classified as low risk, mostly due to strong time-to-market pressures. Two additional NASA data sets, KC4 and MC2, are also assigned a low risk level: KC4 is a ground-based subscription server and MC2 is a video guidance system. The remaining four projects are assigned $medium$ risk level. These data sets are associated with storage management for ground data, a zero gravity experiment, and a spacecraft instrument system. The misclassification cost ratio (φ) are assigned accordingly. Since we know the ground truth about fault prone modules in all data sets, assigning probability parameters $p(+)$ and $p(-)$ for our experiments is simple. From Equation 5.3, we obtain probability costs $PC(+)$ and calculate the range of interest for each project, implied by its risk classification and probability parameters. Specific values of these parameters are shown

in Table 5.1.

5.3.1 Analysis

Having determined risk levels for all the projects in the above section, we want to understand the impact of cost curve modeling in evaluating fault prediction models. More precisely, we want to know how well the models perform when misclassification cost is taken into account. Can cost modeling be used to argue for or against software fault prediction modeling in the context of a specific project?

We developed cost curves describing the performance of fault prediction models from the five classifiers on each of the 16 data sets listed in Table 2.3. Figure 5.6 shows 15 cost curves except KC4 (KC4 is shown as an example in Figure 5.3). To increase readability of these diagrams, we limit the displayed range of the probability cost (x-axis) to include only the region of interest inferred from the project's risk level: Low–$(100,\ 5)$; medium–$(5,\ \frac{1}{5})$; or high – $(\frac{1}{5},\ \frac{1}{100})$. The lower and upper bound of each region are marked with thin solid vertical lines. Most cost curves also show the operational point where $\varphi = C(+|-) = C(-|+) = 1$.

A quick look at the curves in Figure 5.6 reveals two general trends: (1) when $PC(+)$ is close to 0 or close to 1, the performance of fault prediction models becomes very similar to trivial classifiers; (2) the most significant performance benefit of fault prediction modeling is usually achieved in the middle region of the cost curve, surrounding $PC(+) = 0.5$. This observations imply that fault prediction modeling has limited impact in cases of extremely risk averse or cost averse projects. On the extreme risk averse side, we would like the model to identify all the faulty modules. On the extreme cost averse side, we do not want to misclassify any fault free module as fault prone. These goals are, in fact, achieved by the two trivial classifiers which assign all the modules in a single class! In binary classification problems, near perfect module assignment to one class results in high misclassification rate in the other class. High misclassification cost differential between the two classes, therefore, leads to trivial solutions.

Table 5.1 shows each project's risk level, the corresponding probability cost range of interest and the classification algorithm that achieves the best performance inside the probability cost range. The table also shows the probability cost range in which the best model is significantly

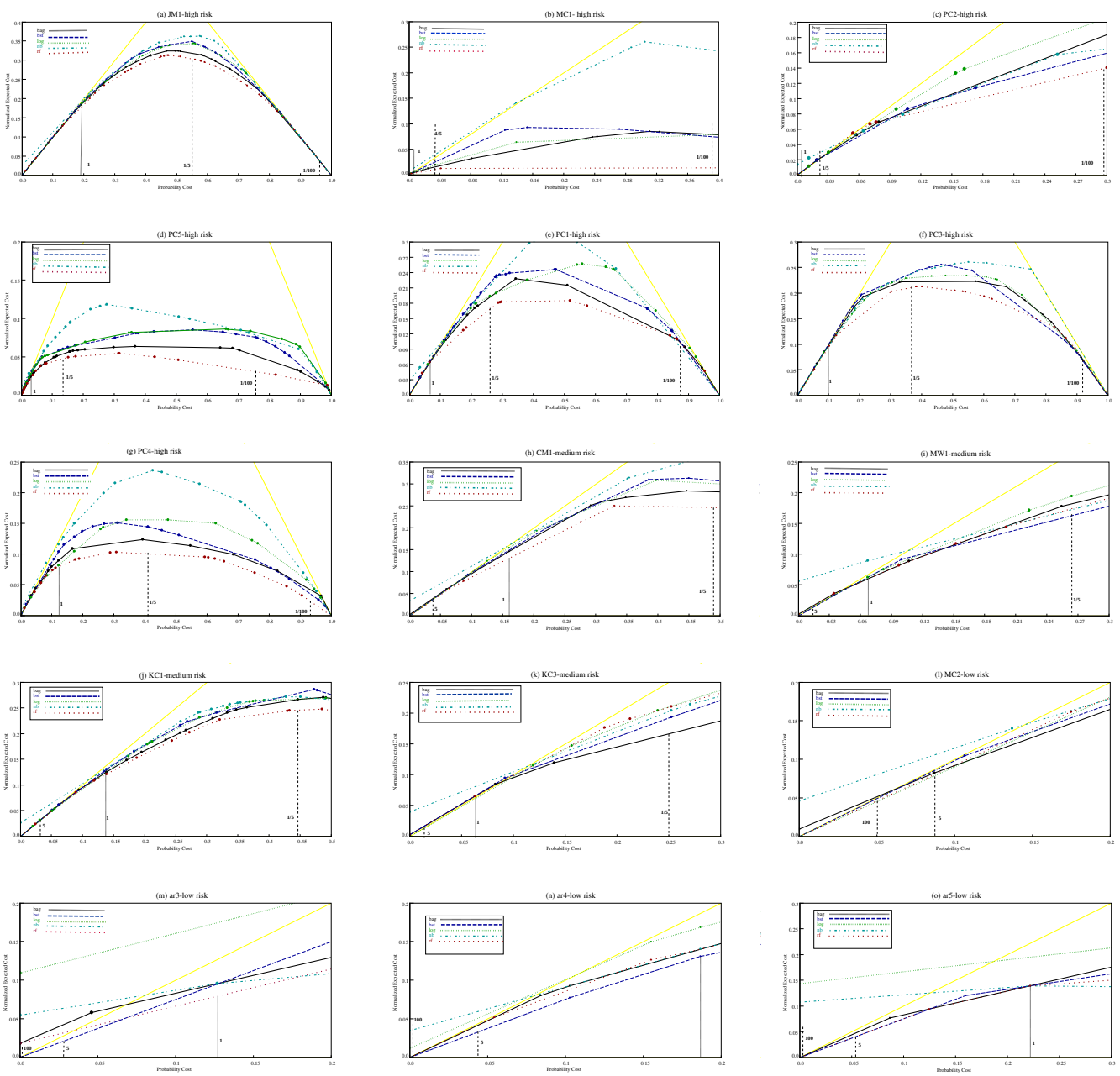

Figure 5.6. Cost curves of projects.

Table 5.1. Probability cost of interested and significant ranges.

Data	**risk**	**probability cost range of interested**	**best classifier**	**significant range** and $\%\frac{significantRange}{interestedRange}$		**significant** φ **range**
JM1	high	(0.545, 0.960)	rf	(0.545, 0.72)	42.17%	$(\frac{1}{5},\ \frac{1}{10})$
MC1	high	(0.031, 0.391)	rf	entire	100%	$(\frac{1}{5},\ \frac{1}{100})$
PC2	high	(0.021, 0.297)	rf	(0.15, 0.297)	53.26%	$(\frac{1}{40},\ \frac{1}{100})$
PC5	high	(0.134, 0.756)	rf	entire	100%	$(\frac{1}{5},\ \frac{1}{100})$
PC1	high	(0.261, 0.876)	rf	(0.261, 0.80)	87.64%	$(\frac{1}{5},\ \frac{1}{50})$
PC3	high	(0.368, 0.921)	rf	(0.368, 0.82)	81.74%	$(\frac{1}{5},\ \frac{1}{40})$
PC4	high	(0.411, 0.933)	rf	entire	100%	$(\frac{1}{5},\ \frac{1}{100})$
CM1	med.	(0.037, 0.489)	rf	(0.11, 0.489)	83.85%	$(1.5,\ \frac{1}{5})$
MW1	med.	(0.014, 0.264)	bst	(0.13, 0.264)	53.6%	$(\frac{1}{2},\ \frac{1}{5})$
KC1	med.	(0.031, 0.447)	rf	(0.14, 0.447)	73.8%	$(1,\ \frac{1}{5})$
KC3	med.	(0.013, 0.252)	bag(nb)	(0.18, 0.252)	30.13%	$(\frac{1}{3.3},\ \frac{1}{5})$
KC4	low	(0.009, 0.156)	nb	(0.10, 0.156)	38.10%	$(8,\ 5)$
MC2	low	(0.005, 0.087)	rf,log	no	0	
ar3	low	(0.001, 0.028)	bst	no	0	
ar4	low	(0.002, 0.044)	bst	no	0	
ar5	low	(0.003, 0.054)	bst	no	0	

better (using the 95% confidence interval) than the trivial classification, the size of statistically significant interval with respect to the probability cost range of interest and the corresponding range of misclassification cost ratio φ in which the model outperforms the trivial classification. The last column offers the project-specific misclassification cost interval in which software fault prediction models offer benefits. If the project's actual misclassification cost ratio is not within this interval, using models to select fault prone modules would not justify the effort.

From these experiments, it became obvious that we need to understand whether fault prediction models are any better than the trivial classifier within specific misclassification cost ranges. For this purpose, we applied a rigorous statistical significance analysis procedure. This analysis indicates that although nb, boosting and bagging outperform other algorithms for low and medium risk projects, their classification results are in many cases not significantly better than the trivial classification. The most benefit we can achieve is in $high$ risk projects where most classifiers have better performance than the trivial classifier.

The data sets described as high risk were derived from mission and safety critical development projects. Figures 5.6 (a)-(g) depict the cost curves of high risk projects. Although we noted that highly imbalanced misclassification cost may favor the trivial classification, the high risk data sets we analyzed all indicate that fault prediction models outperform the trivial classifiers. However, statistical significance does not necessarily span over the entire range of the misclassification cost ratio $(\frac{1}{5}, \frac{1}{100})$. For example, for JM1 in Figure 5.6(a), the model generated using random forest algorithm performs significantly better than the trivial classifier in the probability cost region $(0.545,\ 0.72)$, which corresponds to the misclassification cost ratio range $\varphi = (\frac{1}{5}, \frac{1}{10})$. Similar observations can be made for projects PC1 and PC3: when $PC(+) < 0.80(\varphi \simeq \frac{1}{50})$ or $PC(+) < 0.82$ $(\varphi \simeq \frac{1}{40})$, respectively, best models perform better than the trivial classifier. The same is not true for higher misclassification cost ratios, $\varphi > \frac{1}{50}$ or $\varphi > \frac{1}{40}$, respectively. In PC2 data set, the statistically significant performance difference between the best classifier and the trivial classifier lays in the region of the highest misclassification cost ratio, $\varphi = (\frac{1}{40},\ \frac{1}{100})$. With the remaining three high risk projects, MC1, PC4 and PC5, the best model outperforms the trivial classifier in the entire range of misclassification cost ratio. It is also interesting to observe that in all high risk projects, models developed using random forest algorithm outperform other machine learning algorithms used in our experiments.

The cost curves of medium risk projects CM1, MW1, KC1, and KC3, where $\varphi = (5,\ \frac{1}{5})$,

are shown in Figures 5.6(h)-(k). Interestingly, fault prediction models do not outperform trivial classification throughout the range of interest. Cost curve diagrams indicate that medium risk resides in the low range of the probability cost, i.e., $PC(+) < 0.5$. This is not a region of the most beneficial performance impact. Bagging, boosting and random forest are the classification algorithms which seem to perform the best for medium risk projects.

The cost curves of low risk projects MC2, ar3, ar4 and ar5 are depicted in Figures 5.6(l)-(o) while KC4 is shown in Figure 5.3. In these projects, the trivial classification is as good for selecting fault prone modules as any of the models we experimented with. In other words, if the goal of software quality modeling is to avoid wasting tight $V\&V$ budgets on verification of fault-free modules, the best way to achieve this is by not classifying any module as fault prone.

Explaining this observation is not difficult. In most projects the proportion of fault prone modules is small, causing a severe class size imbalance between faulty and fault-free modules. For example, in MC1, only 0.64% of the modules are fault prone and 99.36% are fault free. In our study, KC4 has a large percentage of faulty modules, 48%, but it is also the smallest project (only 125 modules) and the only one that uses a scripting language (Perl). In MC1, identifying 61 faulty amongst $9,644$ modules is a difficult task for any classification algorithm. What makes the effort worthy is the high cost premium for fault detection. If the premium is not there, given what we know about this data set, no fault prediction model justifies the investment. Looking at the shape of cost curves, the most significant benefits for software assurance are achieved in the mid range of the probability cost axis, surrounding $PC(+) = 0.5$. Due to class size imbalance, the classification performance for neutral misclassification cost ratio is at the value of $PC(+)$ equal to the (low) percentage of faulty modules in the data set. For this simple reason, most economical opportunity to apply fault prediction modeling lays to the right of the neutral misclassification cost point. This is the region of interest for high and some of the medium risk projects.

We are further intrigued by the observation that some modeling algorithms are consistently better for $high$ risk projects, while others have advantages for $medium$ or low risk projects. For example, the random forest algorithm is mostly the "best" learner which consistently performs better than the other classifiers in $high$ risk projects. This result indicates that future research may need to consider fault prediction models developed specifically for the type of software under the development. We strongly recommend using several classifiers to evaluate

software fault prediction models simultaneously, in order to choose a better classifier or to use a combination of several classifiers.

5.4 Comparison of the "Best" and the "Worst" Classifiers

We also compare the performance of the "best" and the "worst" classifiers in data sets. For those low risk projects of KC4, MC2, ar3, ar4, and ar5 data, the "worst" classifiers are all outside the range of the trivial classifier (TC) (worse than TC). Therefore, comparing the "best" classifier vs. the trivial classifier is enough for low risk projects. Thus, we compare the performance of the "best" and the "worst" classifiers in the other 11 high and medium risk data sets.

Table 5.2. Comparison of the "best" vs. the "worst" performance classifiers.

Data	risk	probability cost range of interested	best classifier	worst classifier	significant difference between best and worst	percentage interval with significant difference
JM1	high	(0.545, 0.960)	rf	nb	(0.545, 0.72)	42.17%
MC1	high	(0.031, 0.391)	rf	nb	entire	100%
PC2	high	(0.021, 0.297)	rf	log	(0.15, 0.297)	53.26%
PC5	high	(0.134, 0.756)	rf	nb	entire	100%
PC1	high	(0.261, 0.876)	rf	nb	(0.261, 0.7867)	85%
PC3	high	(0.368, 0.921)	rf	nb	(0.40, 0.82)	75.95%
PC4	high	(0.411, 0.933)	rf	nb	entire	100%
CM1	med.	(0.037, 0.489)	rf	nb	none	0
MW1	med.	(0.014, 0.264)	bst	log	none	0
KC1	med.	(0.031, 0.447)	rf	nb,bst	(0.22, 0.447)	54.57%
KC3	med.	(0.013, 0.252)	bag	nb+rf	(0.18, 0.252)	30.13%

Table 5.2 shows the comparison results. Column 2-4 of Table 5.2 have already been shown in Table 5.1, we add them here for easy reading. Column 5 shows the "worst" performance classifier. Column 6 shows the statistically significant difference between the "best" and the "worst" classifiers in 95% confidence interval (CI). And the percentage of significant difference to the whole interested range of each project is shown in the last column.

Let's compare Table 5.2 to Table 5.1. For three medium risk projects, the ranges of significant difference in Table 5.2 are reduced greatly comparing to the ranges from Table 5.1. CM1 and MW1 projects do not demonstrate any significant difference between the "best" and the "worst" classifiers. The percentage range of significant difference in KC1 is reduced from 73.80% to 54.57%. KC3 remains the same range.

Among 7 high risk projects, five projects remain the same ranges in Table 5.2 and Table 5.1. The significantly different ranges of PC1 and PC3 projects are slightly reduced from 87.64%, 81.74% to 85.00% and 75.95% respectively.

For $high$ risk projects, the performance of the "best" and the "worst" classifier is still different in wide range of interested. Thus, for $high$ risk projects, it is not only meaningful to compare the performance of the "best" classifier against TC, but also the performance of the "best" and the "worst" classifier.

5.5 Incorporating Module Priority

The goal of incorporating module specific risks is to investigate whether cost curves provide module-level guidance to a project. More specifically, we would like to investigate whether module specific risks in cost curves provide better guidance than regarding all modules having the same risk (priority, or severity).

For demonstrating the use of cost curves purpose, we would like to find a criterion into which modules can be divided several groups. In MDP data sets, five data sets have priority information for faulty modules: CM1, KC3, MC1, PC1, and PC4. Each fault in MDP data sets has priority information. Priority "1" is the highest priority which means that the fault to be resolved is critical. Priority "3" is the lowest priority which may imply that the fixing of the fault can be delayed. Priority "2" is in-between them.

We associate a module with the priority of its faults. For a module with more than two faults, the highest priority is used. All fault-free modules do not have any priority information; we use the nearest neighborhood method, an unsupervised learning method, to assigned their priorities. We admit this priority-assigned method might not reflect a module's reality risk level. In real field use, for a project manager, it should not be difficult to decide a module's approximate risk range.

After all modules are assigned a priority, we will examine different module priority ranges, different thresholds for these ranges, testing statistically significant differences inside them. And most importantly, we investigate whether the method of module specific priority provides

any better guidance than unique priority for all modules.

5.5.1 Nearest neighbor method

Three distance methods are used to calculate the nearest neighbor for each module without priority. These three distance methods are: Manhattan distance, Euclidean distance, and Canberra distance.

Given module x and y, each with n metrics, $x = (x_1, x_2, ..., x_n)$, $y = (y_1, y_2, ..., y_n)$, the formula of these three distances are as follows.

$$Euclidean = \sum_{i=1}^{n}(x_i - y_i)^2 \tag{5.4}$$

$$Manhattan = \sum_{i=1}^{n}|x_i - y_i| \tag{5.5}$$

$$Canberra = \sum_{i=1}^{n}\frac{|x_i - y_i|}{|x_i + y_i|} \tag{5.6}$$

We use the following procedures to assign priority to a module without priority.

- calculate distance from a module without priority, m_i, to each module with known priority;
- select a module which is closest to m_i in distance, and assign its priority to m_i;
- for three different methods of calculating the distance, the above procedure is repeated.

Three different methods generate different nearest neighbors for each module without priority. We compare the similarity and difference of these three methods in every module. Table 5.3 compares the similarity of three different distance methods. From the table, we can see that Euclidean method is very close to Manhattan method with more than 91% similarity. However, Canberra method is quite different from Euclidean and Manhattan method although in some

Table 5.3. Comparison the similarity of three different distance methods.

dataset	Euclidean-Manhattan	Euclidean-Canberra	Manhattan-Canberra
mc1	1.00	0.98	0.98
pc1	0.98	0.49	0.51
pc4	0.91	0.46	0.46
cm1	0.90	0.58	0.62
kc3	1.00	0.92	0.92

Table 5.4. Priority distribution of five projects.

dataset	priority1	priority2	priority3
mc1	0	9282	184
pc1	308	500	299
pc4	487	469	502
cm1	281	130	94
kc3	439	13	6

data sets the similarity is above 90%. We decide to use the popular Euclidean distance method to assign priority for our further experiments.

Using Euclidean distance, we have the priority distribution as shown in Table 5.4. From the table, we find that MC1 project does not have modules of priority 1. However, the majority of KC3 project are modules with priority 1. These observations are discrepant to previous assumption of risk levels for MC1 and KC3 projects. Thus, we rejudge MC1 as $medium$ risk project and KC3 as $high$ risk project.

We repeat the aforementioned cost curve analysis to MC1 and KC3 projects. Cost curves with readjusted risk level of MC1 and KC3 projects are shown in Figure 5.7.

We compare the performance of the "best" classifier against the trivial classifier and show results in Table 5.5. We compare the performance of the "best" classifier against the "worst" classifier and the result is shown in Table 5.6.

A person might find that, for MC1 project, the significant range of comparison for the "best" vs. the "worst" classifier is wider than the "best" vs. the trivial classifier (TC). The reason is

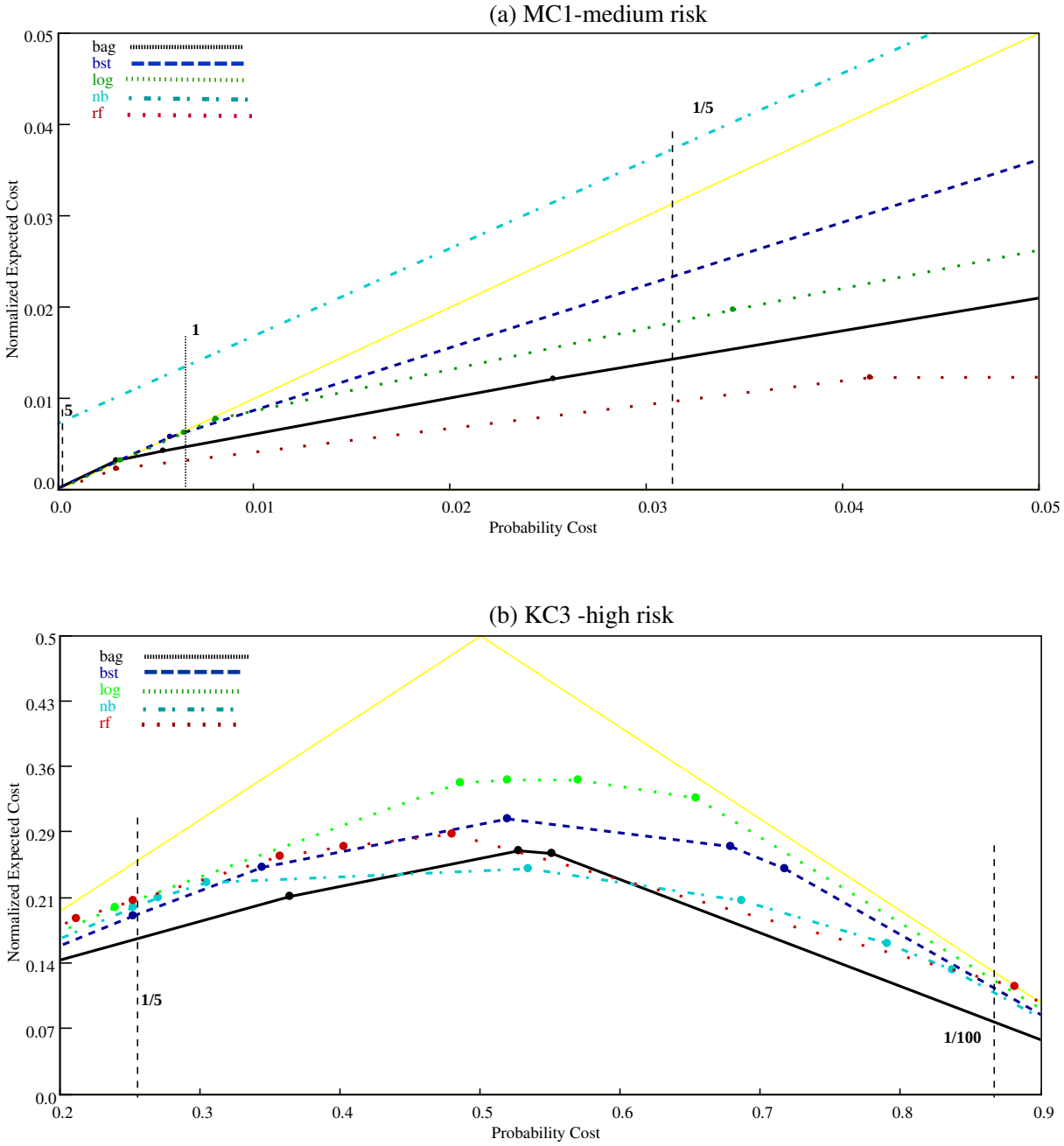

Figure 5.7. Cost curves of MC1 and KC3 projects.

Table 5.5. Probability cost of interested and significant ranges of MC1 and KC3.

Data	risk	probability cost range of interested	best classifier	significant range and $\%\frac{significantRange}{interestedRange}$		significant φ range
MC1	med.	(0.0013, 0.031)	rf	(0.0030, 0.0312)	94.3%	$(2.14, \frac{1}{5})$
KC3	high	(0.252, 0.87)	bag(nb)	entire	100%	$(\frac{1}{5}, \frac{1}{100})$

Table 5.6. Comparison of the "best" vs. the "worst" performance classifiers on MC1 and KC3.

Data	risk	probability cost range of interested	best classifier	worst classifier	significant difference between best and worst	percentage interval with significant difference
MC1	med.	(0.0013, 0.0312)	rf	nb	entire	100%
KC3	high	(0.013, 0.252)	bag(nb)	log	(0.55, 0.87)	51.61%

that in some ranges of cost curve, NaiveBayes (nb) is outside the range of the trivial classifier (in the other word, nb is worse than the trivial classifier).

5.5.2 Priority incorporated evaluation

We incorporate the module specific priority information into the cost curve.

- First, we investigate whether any classifier performs statistically "better" than the others in module specific priority ranges.
- Then, we show whether module specific priority ranges provide any "better" guidance than unique module priority.

For $high$ risk projects, we assume the following misclassification ranges for each module, according to its priorities: (1) priority 1: $\frac{1}{50} < \varphi < \frac{1}{100}$; (2) priority 2: $\frac{1}{25} < \varphi < \frac{1}{50}$; (3) priority 3: $\frac{1}{5} < \varphi < \frac{1}{25}$. For $medium$ risk projects, we assume three misclassification cost ranges for each module as follows: (1) priority 1: $\frac{1}{2} < \varphi < \frac{1}{5}$; (2)priority 2: $1 < \varphi < \frac{1}{2}$; (3) priority 3: $5 < \varphi < 1$.

Table 5.7. Priority ranges and significant differences for five projects.

dataset	priority3	diff?	best	priority2	diff?	best	priority1	diff?	best
φ (med.)	$(5,\ 1)$			$(1,\ \frac{1}{2})$			$(\frac{1}{2},\ \frac{1}{5})$		
cm1	(0.04 , 0.16)	no		(0.16 , 0.28)	no		(0.28 , 0.49)	no	
mc1	(0.0013 , 0.006)	yes	rf	(0.006 , 0.0128)	yes	rf	(0.0128 , 0.0312)	yes	rf
φ (high)	$(\frac{1}{5},\ \frac{1}{25})$			$(\frac{1}{25},\ \frac{1}{50})$			$(\frac{1}{50},\ \frac{1}{100})$		
kc3	(0.25 , 0.63)	yes	bag(nb)	(0.63 , 0.77)	yes	bag	(0.77 , 0.87)	yes	bag
pc1	(0.26 , 0.64)	yes	rf	(0.64 , 0.78)	yes	rf(bag)	(0.78 , 0.88)	yes	rf(bag,bst)
pc4	(0.41 , 0.78)	yes	rf(bag,bst))	(0.78 , 0.87)	yes	rf(bag,bst)	(0.87 , 0.93)	yes	rf(bag,bst)

Using these assumptions, we first examine which classifier performs "better" than others. Table 5.7 shows the priority ranges, whether there is significant difference, and the "best" classifier for that priority range for each project. Then we figure out which kinds of thresholds is more suitable for a classifier to achieve the lowest misclassification cost.

From Table 5.7, we can observe that, for $medium$ risk project, CM1, in all three priority ranges, five classifiers have the same performance statistically. For project MC1, the worst classifier, nb, is worse than the trivial classifier in the whole medium risk range from $(0.0013,\ 0.0312)$. Hence, there is statistical significance difference between the "best" (rf) and the "worst" classifier (nb). It is meaningless to further study a classifier which is worse than the trivial classifier. And the comparison of the "best" classifier (rf) against the trivial classifier has been shown in Table 5.1.

For $high$ risk projects, it is meaningful to study which classifier performs the "best". Three projects, KC3, PC1, and PC4, are $high$ risk projects. The thresholds and the corresponding performance indices for them are listed in Table 5.8, and the figure with thresholds is shown in Figure 5.8.

For KC3, bagging has the best performance. It performs better than all other four classifiers except when PC is from $(0.45,\ 0.60)$ where nb slightly outperforms it without demonstrating statistical significance. For this reason, we only introduce thresholds for the bagging classifier instead of naiveBayes. In KC3 project, 439 modules belongs to priority 1. From Table 5.7, we can see that in priority 1, bagging has the "best" performance when $\frac{1}{50} < \varphi < \frac{1}{100}$ with PC from 0.63 to 0.87. In this range, bagging has the threshold of 0.030586, where $recall = 1$, $precision = 0.095$, $FN = 0$, $FP = 276$, and $PF = 0.64$. This is clearly the case where the misclassification of faulty modules as fault free is undesired. Classifier is designed to have high $recall$ with high false alarm rate and low $precision$. For 13 modules in priority 2 of

Table 5.8. Performance thresholds for three priorities in KC3, PC1, and PC4 high risk projects.

dataset	threshold	recall	PF	precision	FN	FP	priority	PC(x-axis)	φ	NEC(y-axis)
KC3	**0.143**	**0.52**	**0.063**	**0.36**	14	27	3	(0.25, 0.36)	$(\frac{1}{5}, \frac{1}{8})$	$(0.166, 0.216)$
	0.10933	0.59	0.103	0.28	12	44	3	(0.36, 0.53)	$(\frac{1}{8}, \frac{1}{17})$	$(0.216, 0.266)$
	0.046753	0.79	0.34	0.14	6	144	3	(0.53, 0.55)	$(\frac{1}{17}, \frac{1}{18})$	$(0.266, 0.261)$
	0.030586	**1**	**0.64**	**0.095**	0	276	3,2 and 1	(0.55, 0.87)	$(\frac{1}{18}, \frac{1}{100})$	$(0.261, 0.93)$
PC1	0.243794	0.48	0.045	0.43	38	47	3	(0.26, 0.29)	$(\frac{1}{5}, \frac{1}{6})$	$(0.170, 0.180)$
	0.089	**0.81**	**0.179**	**0.242**	14	185	3	(0.29, 0.52)	$(\frac{1}{6}, \frac{1}{15})$	$(0.180, 0.186)$
	0.0426	0.90	0.28	0.18	7	292	3 and 2	(0.52, 0.57)	$(\frac{1}{15}, \frac{1}{19})$	$(0.186, 0.175)$
	0.036	**0.92**	**0.30**	**0.18**	6	312	2 and 1	(0.57, 0.84)	$(\frac{1}{19}, \frac{1}{75})$	$(0.175, 0.118)$
	0.023667	0.93	0.37	0.15	5	386	1	(0.84, 0.87)	$(\frac{1}{75}, \frac{1}{95})$	$(0.118, 0.110)$
	0.000094	0.99	0.73	0.087	1	755	1	(0.87, 0.88)	$(\frac{1}{95}, \frac{1}{100})$	$(0.110, 0.098)$
PC4	**0.216167**	**0.92**	**0.113**	**0.53**	15	144	3	(0.41, 0.59)	$(\frac{1}{5}, \frac{1}{10})$	$(0.103, 0.097)$
	0.154	0.95	0.165	0.44	11	186	3	(0.59, 0.64)	$(\frac{1}{10}, \frac{1}{13})$	$(0.097, 0.091)$
	0.178	0.94	0.145	0.47	9	211	3	(0.64, 0.67)	$(\frac{1}{13}, \frac{1}{15})$	$(0.091, 0.088)$
	0.133	0.96	0.18	0.42	7	233	3	(0.67, 0.71)	$(\frac{1}{15}, \frac{1}{18})$	$(0.088, 0.079)$
	0.092	**0.98**	**0.23**	**0.37**	3	297	3 and 2	(0.71, 0.86)	$(\frac{1}{18}, \frac{1}{44})$	$(0.079, 0.048)$
	0.056	0.99	0.30	0.32	1	383	2 and 1	(0.86, 0.91)	$(\frac{1}{44}, \frac{1}{72})$	$(0.048, 0.032)$
	0.032	1	0.36	0.28	0	466	1	(0.91, 0.93)	$(\frac{1}{72}, \frac{1}{100})$	$(0.032, 0.024)$

KC3 project, the same threshold, 0.030586, for priority 1 is held. For the remaining 6 modules which have priority 3, there are four thresholds achieving the "best" performance in different PC ranges. Observations from PC1 and PC4 are similar to KC3 project.

Examining Table 5.8 closely, we can observe that with the increasing of priority level (from priority 3 to priority 1), FN (misclassifying faulty modules as fault free) decreases and FP (misclassifying fault-free modules as faulty) increases. This means that with the increase of misclassification costs of faulty modules, in order to achieve the overall lowest misclassification cost, a classifier tends to increase FP and decrease FN, resulting in the increasing $recall$ and PF, and the decreasing $precision$.

We also test the statistically significant performance on these thresholds in 95% CI. We test the significance of two thresholds in pair. No one single threshold has the same statistical performance within three priority ranges. However, two thresholds are able to account for all the statistical performances inside all three priority ranges. These two thresholds are bold in Table 5.8 and are shown in Figure 5.8 by two straight lines.

We use the middle point of each priority range to demonstrate the usage of module specific costs. We calculate the overall Normalized Expected Cost (NEC), y-axis, when modules have their specific priority range and when the same priority is applied to all the modules.

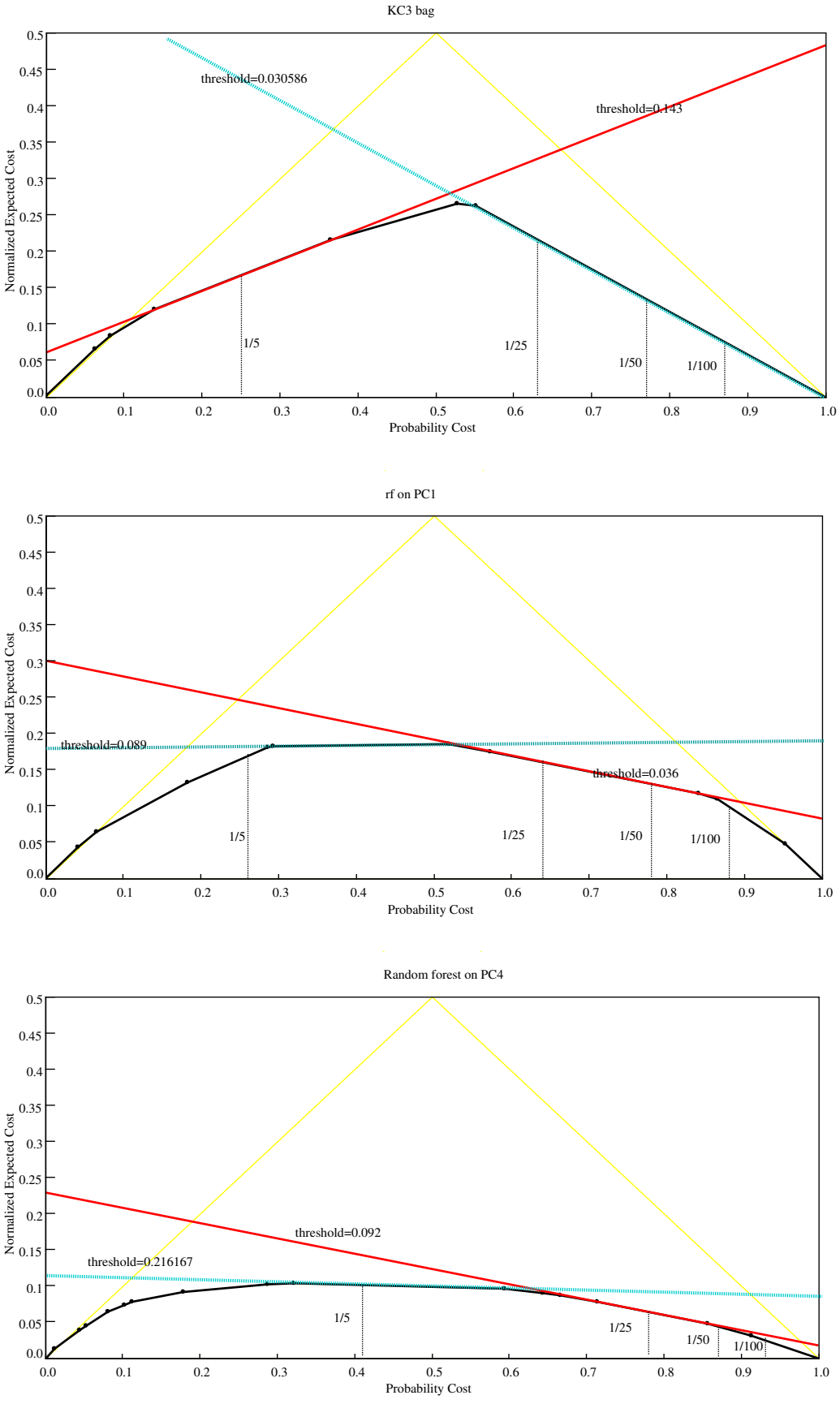

Figure 5.8. Cost curves with lowest misclassification costs for KC3, PC1, and PC4.

In KC3 project, there are 6 modules with priority 3; 13 modules with priority 2 and 439 modules with priority 1. The total number of modules in KC3 is 458. We use the middle points to represent the specific priority range. In KC3, the middle point of priority 3 is PC(+)=0.44 ($\varphi = \frac{1}{12}$) with NEC=0.2387; the middle point of priority 2 is PC(+)=0.7 ($\varphi = \frac{1}{35}$) with NEC=0.1767; the middle point of priority 1 is PC(+)=0.82 ($\varphi = \frac{1}{68}$) with NEC=0.1042. The overall NEC for KC3 is calculated as a weight sum:
$\frac{439}{458} * 0.1042 + \frac{13}{458} * 0.1767 + \frac{6}{458} * 0.2387 = 0.1080199$.
If we use the same priority for all modules, the middle point of KC3 high risk range is $PC(+) = 0.56$ ($\varphi = \frac{1}{19}$) with NEC = 0.2568 . Thus, using module specific priority, the overall Normalized Expected Cost (NEC=0.1080199) would be lower than using the same priority for all the modules (NEC = 0.2568).

In PC1 data, there are 299 modules with priority 3; 500 modules have priority 2; and 308 modules have priority 1. The middle points of priority 3, 2, and 1 are PC(+)=0.45 ($\varphi = \frac{1}{12}$, NEC=0.1843), PC(+)=0.71 ($\varphi = \frac{1}{35}$, NEC=0.15), and PC(+)=0.83 ($\varphi = \frac{1}{69}$, NEC=0.1193) respectively. Hence, the overall NEC for PC1 is:
$\frac{308}{1107} * 0.1193 + \frac{500}{1107} * 0.15 + \frac{299}{1107} * 0.1843 = 0.1507228$.
If all the modules have the same priority, using the middle point of $PC = 0.57$ ($\varphi = \frac{1}{19}$) with NEC=0.18 . Thus, similar to KC3 project, using module specific priority achieve lower overall Normalized Expected Cost.

In PC4 data, there are 502 modules with priority 3, 469 modules with priority 2 and 487 modules with priority 1. The middle points of priority 3, 2, and 1 are PC(+)=0.585 ($\varphi = \frac{1}{10}$, NEC=0.098), PC(+)=0.825 ($\varphi = \frac{1}{34}$, NEC=0.0543), and PC(+)=0.9 ($\varphi = \frac{1}{65}$, NEC=0.0382) respectively. Hence, the overall NEC for PC1 is:
$\frac{487}{1456} * 0.0382 + \frac{469}{1456} * 0.0543 + \frac{502}{1456} * 0.098 = 0.06405639$.
Comparing the middle point of PC=0.67 (NEC=0.0955, with $\varphi = \frac{1}{15}$) when having the same priority for all modules, we find that using module specific priority achieve lower overall Normalized Expected Cost.

The comparison of NEC with and without module specific priority is summarized in Table 5.9. Column 2 is weighted NEC with module specific priority. Column 3 is NEC with uniform priority. All three high risk projects with module specific priority yield lower overall NEC than using the middle point of uniform priority. This is the benefit of using module spe-

cific priority. Column 4-6 list NEC if all modules are applied an uniform priority of priority 1, priority 2, or priority 3 separately. These last three columns are shown for comparison purpose. For KC3 data, the value of weighted overall NEC is in between of priority 1 to priority 2. For PC1 and PC4 data, their weighted overall NEC is in between of priority 2 and priority 3.

Table 5.9. Comparison of Normalized Expected Cost (NEC) with and without module specific priority

data	NEC with module specific priority	NEC with uniform priority			
		middle point	priority 1	priority 2	priority 3
KC3	0.1080199	0.2568	0.1042	0.1767	0.2387
PC1	0.1507228	0.18	0.1193	0.15	0.1843
PC4	0.06405639	0.0955	0.0382	0.0543	0.098

From the practical view point, if a project manager can determine the priority of a module, then it is possible to choose a classifier and the corresponding threshold(s) for the module. Furthermore, modules with different priority reflect more faithfully reflect the field software. A project manager can put more $V\&V$ resource on important modules, which finally leads to better quality. Thus, we are able to give guidance for a module with its specific risk level using cost curves.

5.6 Summary

In this chapter, we used cost curve to analyze software project's specific needs and module's specific needs. Cost curve is a complementary model evaluation technique. It offers a different viewpoint by integrating the cost of module misclassification into performance evaluation. The utility of fault prediction models varies with the misclassification cost: (1) in low risk projects, building software fault prediction models may not be justified; (2) in $medium$ risk projects the utility of a model is typically limited to a subset of misclassification cost ratio range; (3) the most benefit is drawn by $high$ risk projects. We are aware that determining a misclassification cost ratio is not an exact science. It arguably needs additional consideration by the software assurance research community. We discussed cost curve methodology with

software quality professionals; they did not appear to have a difficulty assessing approximate cost of misclassifying fault free modules as fault prone or the other way around. Importantly, misclassification cost assessment is project-specific. Factors that influence the cost are the size of the $V\&V$ budgets (i.e., how many modules can be analyzed for faults), schedule pressures, software configuration items criticality, failure risks and severity, and various nonfunctional requirements including reliability, availability, etc. However, it remains to be investigated whether the specific misclassification cost ratio ranges we used to characterize high, medium and low risk projects are the most appropriate generalizations which should be reused in future studies.

Incorporating priority into the cost curve evaluation, we found that modules in high risk project are possible to have their own specific "best" classification threshold(s). Using the cost curve evaluation, a project manager is able to build module specific models with lowest misclassification by estimating a module's priority or severity.

Based on our experience with cost curves, we strongly recommend their inclusion in the evaluation of software fault prediction models. Results of our experiments indicate that high risk projects draw the most significant benefit from fault prediction models. And the modules in high risk projects are able to have their own specific module-based classification by using cost curve.

We conclude by submitting answers to the problems stated in this dissertation:

- Q7: How can we incorporate misclassification costs in fault prediction models?
  Answer: Cost curves are able to account for different misclassification costs in fault prediction models.
- Q8: How can we achieve the lowest misclassification cost for a given data set?
  Answer: By connecting the lowest envelope of a cost curve formed by using several classifiers, we can obtain the lowest misclassification cost for a data set.
- Q9: Q9: How can fault prediction models guide different risk level projects from misclassification cost perspective?
  Answer: Cost curves are able to address different data set's risk level (based on their priority or severity). High risk projects have the most benefit by fault prediction modeling among three risk level projects. Low risk projects seldom have justified benefits by using fault prediction models.

- Q10: How can we evaluate individual module's specific misclassification cost in fault prediction models?
  Answer: For an individual module with a specific priority (or severity) characteristic, a cost curve is able to provide module specific guidance.

Chapter 6

Conclusion and Future Work

In this dissertation, we investigate how metrics from the early lifecycle of software project, such as, requirement metrics and design metrics, can be used to predict faulty modules. Requirement metrics from textual description do not predict faults well by themselves, but they can greatly improve the performance of the prediction models when they are combined with software module metrics (design metrics and code metrics). Design metrics, another group of metrics from software early lifecycle, are good predictors for software faulty modules. Thus, we conclude that metrics from the early software lifecycle are useful and should be used.

Along with the investigation of the benefit of early lifecycle metrics, we also investigate the effectiveness of incremental development of fault prediction models from design metrics and code metrics. We compare the performance of models built from increasing volume of training subset of design, code, and their combination metrics from as small as 10% to as large as 90% using Demsar's statistical non-parametric testing methods. These experiments are a simulation of software evolvement from the early design phase to the late code development stage. We find that models built from a small training subset (as small as 10%) of design metrics are still useful to predict software faulty modules. As the process of adding more data from design metrics and code metrics, the performance of models are increased gradually. Projects which follow iterative development processes, such as rapid prototyping or agile development, offer the opportunity for the incremental development of fault prediction models.

When a software project evolves into testing phase, it is necessary to analyze the misclassification costs of a project and even the misclassification cost for modules in a project. Misclassification cost assessment is not only project-specific, but also module-specific: different modules in the same project could have different importance. The cost curve provides this flexibility. It incorporates different misclassification costs to a fault prediction model and provides guidance for different projects' needs. Throughout the analyzing 16 different data sets from publicly available software engineering data repositories, we find that cost curves offer a different viewpoint by integrating the cost of module misclassification into performance evaluation which is unable to be provided by any other kinds of state-of-the-art evaluation techniques. Our experiments integrate the cost of project-specific and module-specific misclassifications into performance evaluation. The results from our experiments strongly encourage to use cost curves to evaluate software fault prediction models.

In summary, we conclude by submitting our finding with the questions posed in the dissertation:

- Q1 and Q4: Although the requirement metrics do not predict faults well by themselves, they significantly improve the performance of the prediction models that combine requirement and module metrics together using the inner-join method.
- Q2: Models built from $design$ metrics achieve meaningful performance. In principle, design metrics of reused modules can be utilized to evaluate the designs of new modules to identify problem areas possibly even before implementation.
- Q3: Models built from $code$ metrics are usually better than $design$ metrics in ranks although they do not demonstrate any statistical significant difference.
- Q4: Models which combine design and code attribute metrics typically outperform $design$ metric based models by a statistically significant margin. Therefore, we conclude that combining design and code metrics when modeling fault prediction is desirable. The type of metrics used for modeling impacts model performance more than the selection of the classification algorithm.
- Q5: Fault prediction models from relatively small training subsets of project modules (10%, for example) achieve meaningful performance.
- Q6: These early models do not need to be updated frequently. Models built from 50% of the modules almost always belong to the best performance rank cluster, i.e, their performance is statistically indistinguishable from the models that could be built later in the

project development cycle.

- Q7: Cost curve evaluation technique is able to incorporate different misclassification costs into fault prediction models.
- Q8: Cost curve can provide project manager the lowest misclassification cost for a given data set when multiple learners are applied to fault prediction models.
- Q9: Cost curve is able to address different data set's risk level (based on their priority, severity or any other kinds of estimation) and provide useful advice in fault prediction modeling. High risk projects have the most benefit by fault prediction modeling among three risk level projects. Low risk projects seldom have justified benefits by using fault prediction models.
- Q10: For an individual module with specific priority (severity, priority or other kinds of estimation) characteristic, cost curve is able to provide module specific guidance.

An additional lesson we learned from our experiments is that we should use multiple classification algorithms to build fault prediction models. On one hand, when models are built from smaller subsets of available modules, different classifiers are likely to offer statistically meaningful performance differences. On the other hand, multiple learners on fault prediction models when using cost curve are able to provide the lowest misclassification cost for a data set.

The outcomes of our experiments provide a good guidance for an incremental process for software fault prediction modeling. Models can be built early, from design metrics and/or from relatively small subsets of available fault data. Updating such early models is recommended, the frequency of such updates can be optimized as model performance gains justify intermittent (sporadic)upgrades, rather than recurring ones.

Based on our experience with cost curves, we strongly recommend their inclusion in the evaluation of software fault prediction models. Results of our experiments indicate that high risk projects draw the most significant benefit from fault prediction models. And each module in high risk projects is able to have their own specific module-based classification by using cost curves.

6.1 Future Works

There are some aspects we do not touch in this dissertation, one important and debating problem is the relationship between faults and failures [30]. A fault in a software may be concealed and never occur to cause system failures. On the other hand, a serious fault would probably account for the majority of running failures. It would be interesting to incorporate different costs for different faults in a system (or across different systems) further to investigate the degree of severity failures they might cause.

From Chapter 2 related work, we learned that social organizational information, such as developer information [85], the number of messages (emails) passing among developers [55], and organizational structure complexity metrics [69], is effective predictor for faults. And we also note that the performance of a data miner is changed by different business contexts. This may imply that all kinds of artifacts from the software development lifecycle would potentially be effective predictors for faults. Thus, new (or better) metrics are still called for. In addition, the practice of combining new sets of metrics (i.e, organizational metrics) to existing sets of metrics (i.e., module metrics) would be worthy to be explored.

Because of data limitation, we only have partial requirement metrics for three data sets; it would be very interesting to compare (and/or combine) requirement metrics, design metrics, and code metrics from software projects in large scale. To do this, the biggest problem is data, that is, extracting requirement metrics data and design metrics data. Code metrics data is easier to obtain relatively. In order to have requirement metrics data, software projects need to follow strictly software documentary process or by using more costly formal specification requirement. On the other hand, extracting design metrics from UML diagrams would be interesting. UML diagrams, as a recommended standard software project design tools, are popular in software engineering and are practiced by many organizations. Nevertheless, extraction metrics from requirement and design phase is a challenging work.

In the current literature of software fault prediction, most reported work is supervised learning. The only unsupervised learning work is conducted by Zhong *et al.* [92, 93] who combined unsupervised learning and human experts to predict faults. The possible reason for few reported unsupervised learning is that unsupervised learning usually yields undesirable perfor-

mance which cannot be published in the literature. However, as we all know, unsupervised learning potentially would have more benefits than supervised learning when a new software project does not have any historical data. Furthermore, unsupervised learning will provide guidance earlier than supervised learning. Thus, more effort, including new metrics and/or better clustering algorithms, is called for in unsupervised learning.

In our experiments, we did not formally examine variance of software quality models. Isaksson et al. [37] compare variances of error rate for three different sample sizes of 20, 100, and 1000 and find that smaller sample sizes data (20 and 100 in their case) have large variance. Software projects have their own characteristic, as introduced in [39]. We would like to further investigate variance in fault-prone prediction models, including: (1) What kinds of measurements are more trustworthy in software fault prediction models in term of variance, i.e, precision, recall, or AUC of ROC? (2) How large a data set should be in order to achieve reliable performance? (3) What kinds of practical methods can compensate larger variance for small data?

Bibliography

[1] Do-178b and mccabe iq. available in `http://www.mccabe.com/iq_research_whitepapers.htm`.

[2] Metric data program. NASA Independent Verification and Validation facility, Available from `http://MDP.ivv.nasa.gov`.

[3] Promise data repository. available `http://promisedata.org/repository`.

[4] F. Akiyama. An example of software system debugging. *Information Processing*, 71:353–379, 1971.

[5] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering tracebility links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[6] G. Antoniol, G. Casazza, M. Penta, and R. Fiutem. Object-oriented design patterns recovery. *Journal of Systems and Software*, 59(2):181–196, 2001.

[7] E. Arisholm, L. Briand, and M. Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. In *proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE 2007),Sweden*, pages 215–224, 2007.

[8] D. Azar, D. Precup, S. Bouktif, B. Kegl, and H. Sahraoui. Combining and adapting software quality predictive models by genetic algorithms. In *17th IEEE International Conference on Automated Software Engineering*, pages 285–288, 2002.

[9] V. Basili, L. Briand, and W. Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22(10):751–761, Oct 1996.

[10] V. R. Basili and B. T. Perricone. Software errors and complexity: an empirical investigation. *Commun. ACM*, 27(1):42–52, 1984.

[11] V. R. Basili, M. V. Zelkowitz, D. I. Sjoberg, P. Johnson, and A. J. Cowling. Protocols in the use of empirical software engineering artifacts. *Empirical Softw. Engg.*, 12(1):107–119, 2007.

[12] B. Boehm and P. Papaccio. Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, 14:1462–1477, Oct. 1988.

[13] G. Boetticher, T. Menzies, and T. Ostrand. Promise repository of empirical software engineering data, 2007. available in `url{http://promisedata.org/}`.

[14] L. Breiman. Random forests. *Machine Learning*, 45:5–32, 2001.

[15] L. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of uml sequence diagrams for distributed java software. *IEEE Transactions on Software Engineering*, 32(9):642–663, 2006.

[16] G. CanforaHarman and M. D. Penta. New frontiers of reverse engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 326–341, Washington, DC, USA, 2007. IEEE Computer Society.

[17] V. U. Challagulla, F. B. Bastani, and I.-L. Yen. A unified framework for defect data analysis using the mbr technique. In *Proc. of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06)*, pages 39–46, 2006.

[18] V. U. Challagulla, F. B. Bastani, I.-L. Yen, and R.A.Paul. Empirical assessment of machine learning based software defect prediction techniques. pages 263–270, 2005.

[19] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Trans. Softw. Eng.*, 24(8):629–639, 1998.

[20] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[21] W. J. Conover. *Practical Nonparametric Statistics*. John Wiley and Sons, Inc., 1999.

[22] J. Davis and M. Goadrich. The relationship between precision-recall and roc curves. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 233–240, New York, NY, USA, 2006. ACM.

[23] J. Demsar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7, 2006.

[24] C. Drummond and R. C. Holte. C4.5, class imbalance, and cost sensitivity: why under-sampling beats over-sampling. In *Workshop on Learning from Imbalanced Datasets II, Washington DC*, July 2003.

[25] C. Drummond and R. C. Holte. Severe class imbalance: Why better algorithms aren't the answer? In *Proc. of the 16th European Conference of Machine Learning,Porto,Portugal*, Oct. 2005.

[26] C. Drummond and R. C. Holte. Cost curves: An improved method for visualizing classifier performance. *Machine Learning*, 65(1):95–130, 2006.

[27] K. El-Emam, S. Benlarbi, N. Goel, and S. Rai. Comparing case-based reasoning classifiers for predicting high-risk software components. *Journal of Systems and Software*, 55(3):301–320, 2001.

[28] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *In Proceedings of the 23rd International Conference on Software Engineering*, pages 329–338, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[29] N. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8):797–814, August 2000.

[30] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999. Available from `http://citeseer.nj.nec.com/fenton99critique.html`.

[31] N. E. Fenton and S. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. PWS Publishing Company,International Thompson Press, 1997.

[32] A. Ferdinand. A theory of system complexity. *International Journal of General Systems*, 1:19–33, 1974.

[33] L. Guo, Y. Ma, B. Cukic, and H. Singh. Robust prediction of fault-proneness by random forests. In *Proc. of the 15th International Symposium on Software Relaibility Engineering ISSRE'04*, pages 417–428, 2004.

[34] M. H. Halstead. *Elements of Software Science*. Elsevier, North-Holland, 1975.

[35] L. Hatton. Reexamining the fault density-component size connection. *IEEE Softw.*, 14(2):89–97, 1997.

[36] T. Illes-Seifert and B. Paech. Exploring the relationship of history characteristics and defect count: an empirical study. In *DEFECTS*, pages 11–15, 2008.

[37] A. Isaksson, M. Wallman, H. Göransson, and M. G. Gustafsson. Cross-validation and bootstrapping are unreliable in small sample classification. *Pattern Recognition Letters*, 29(14):1960–1965, October 2008.

[38] T. Javed, M. e Maqsood, and Q. S. Durrani. A study to investigate the impact of requirements instability on software defects. *SIGSOFT Softw. Eng. Notes*, 29(3):1–7, 2004.

[39] Y. Jiang, B. Cukic, and Y. Ma. Techniques for evaluating fault prediction models. *Empirical Softw. Engg.*, 13(5):561–595, 2008.

[40] Y. Jiang, B. Cukic, and T. Menzies. Fault prediction using early lifecycle data. pages 237–246. Software Reliability, The 18th IEEE International Symposium on, 11 2007.

[41] 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*, pages 16–20, New York, NY, USA, 2008. ACM.

[42] Y. Jiang, B. Cukic, and T. Menzies. Cost curve evaluation of fault prediction models. *Software Reliability Engineering,19th International Symposium on*, pages 197–206, Nov. 2008.

[43] Y. Jiang, B. Cukic, T. Menzies, and N. Bartlow. Comparing design and code metrics for software quality prediction. In *Proceedings of the 4th international workshop on Predictor models in software engineering*, pages 11–18, New York, NY, USA, 2008. ACM.

[44] Y. Jiang, J. Lin, B. Cukic, and T. Menzies. Variance analysis in fault prediction models. *submitted to 20th International Symposium on Software Reliability Engineering*, 2009.

[45] T. Khoshgoftaar. An application of zero-inflated poisson regression for software fault prediction. In *Proceedings of the 12th International Symposium on Software Reliability Engineering, Hong Kong*, pages 66–73, Nov 2001.

[46] T. Khoshgoftaar and D. Lanning. A neural network approach for early detection of program modules having high risk in the maintenance phase. *Journal of Systems and Software*, 29(1):85–91, 1995.

[47] T. Khoshgoftaar and N. Seliya. Tree-based software quality estimation models for fault prediction. In *METRICS 2002, the Eighth IEEE Symposium on Software Metrics. IEEE Computer Society*, pages 203–214, 2002.

[48] T. Khoshgoftaar and N. Seliya. Analogy-based practical classification rules for software quality estimation. *Empirical Software Eng. J.*, 8(4):325C350, Dec. 2003.

[49] T. M. Khoshgoftaar and E. B. Allen. Classification of fault-prone software modules: Prior probabilities,costs, and model evaluation. *Empirical Softw. Engg.*, 3(3):275–298, 1998.

[50] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Cost-benefit analysis of software quality models. *Software Quality Control*, 9(1):9–30, 2001.

[51] S. Kim, T. Zimmermann, E. J. W. Jr., and A. Zeller. Predicting faults from cached history. In *29th International Conference on Software Engineering,ICSE'07*, pages 489 – 498, May 2007.

[52] M. Kubat, R. Holte, and S. Matwin. Machine learning for the detection of oil spills in satellite radar images. *Machine Learning*, 30(2-3):195–215, 1998.

[53] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Trans. Softw. Eng.*, 34(4):485–496, 2008.

[54] D. Lewis and W. Gale. A sequential algorithm for training text classifiers. In *the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 3–12, 1994.

[55] P. L. Li, J. Herbsleb, and M. Shaw. Finding predictors of field defects for open source software systems in commonly available data sources: A case study of openbsd. In *METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium*, page 32, Washington, DC, USA, 2005. IEEE Computer Society.

[56] C. X. Ling and C. Li. Data mining for direct marketing: Problems and solutions. In *Proc. of the 4th Intern. Conf. on Knowledge Discovery and Data Mining, New York*, pages 73–79, 1998.

[57] Y. Ma and B. Cukic. Adequate and precise evaluation of predictive models in software engineering studies. In *Proceedings of the PROMISE workshop*, 2007.

[58] R. Mahanti and J. Antony. Confluence of six sigma, simulation and software development. *Managerial Auditing Journal*, 20(7):739–762, 2005.

[59] Y. Malaiya and J. Denton. Requirement volatility and defect density. In *Proc. International Symposium on Software Reliability Engineering*, pages 285–294, Nov. 1999.

[60] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

[61] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald. Problems with precision. *IEEE Transactions on Software Engineering*, 33(9):637–640, Sept. 2007.

[62] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, January 2007.

[63] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang. Implications of ceiling effects in defect predictors. In *Proceedings of the 4th international workshop on Predictor models in software engineering*, New York, NY, USA, 2008. ACM.

[64] M. Mertik, M. Lenic, G. Stiglic, and P. Kokol. Estimating software quality with advanced data mining techniques. *Software Engineering Advances, International Conference on*, pages 19–19, Oct. 2006.

[65] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *ICSE 2005, St. Louis*, 2005.

[66] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *27th International Conference on Software Engineering, ICSE'05*, pages 284–292, May 2005.

[67] N. Nagappan, T. Ball, and B. Murphy. Using historical data and product metrics for early estimation of software failures. In *Proc. ISSRE , Raleigh, NC*, pages 62–71, 2006.

[68] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 452–461, New York, NY, USA, 2006. ACM Press.

[69] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: an empirical case study. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 521–530, New York, NY, USA, 2008. ACM.

[70] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *ACM conference on Computer and Communication Security*, Alexandria, Virginia, USA, 2007.

[71] U. of Waikato. Weka software package. The University of Waikato, available `http://www.cs.waikato.ac.nz/ml/weka/`.

[72] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22(12):886–894, 1996.

[73] N. Ohlsson, A. C. Eriksson, and H. M. E. Early risk-management by identification of fault-prone modules. *Empirical Software Engineering*, 2(2):166–173, 1997.

[74] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[75] R. Ramakrishnan. *Database Management Systems*. WCB/McGraw-Hill, 1997.

[76] A. Schröter, T. Zimmermann, and A. Zeller. Predicting component failures at design time. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering*, pages 18–27, New York, NY, USA, 2006. ACM Press.

[77] V. Y. Shen, T.-J. Yu, S. M. Thebaut, and L. R. Paulsen. Identifying error-prone software an empirical study. *IEEE Trans. Softw. Eng.*, 11(4):317–324, 1985.

[78] F. Shull, B. Basili, B. Boehm, A. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz. What we have learned about fighting defects. In *Proceedings of 8th international software metrics symposium,Ottawa, Canada*, pages 249–258, 2002.

[79] S. Siegel. *Nonparametric Satistics*. New York: McGraw- Hill Book Company, Inc., 1956.

[80] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Softw. Eng.*, 29(4):297–310, 2003.

[81] T. Systa. *Static and dynamic reverse engineering techniques for Java software systems*. PhD thesis, 2000.

[82] P. Tonella and A. Potrich. *Reverse engineering of object oriented code*. Springer-Verlag, Berlin, Heidelberg, New York, 2005.

[83] B. Turhan and A. Bener. A multivariate analysis of static code attributes for defect prediction. In *QSIC '07: Proceedings of the Seventh International Conference on Quality Software*, pages 231–237, Washington, DC, USA, 2007. IEEE Computer Society.

[84] T. Vuk, M.and Curk. Roc curve, lift chart and calibration plot. *Metodoloski zvezki*, 3:89–108, 2006.

[85] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Using developer information as a factor for fault prediction. In *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, page 8, Washington, DC, USA, 2007. IEEE Computer Society.

[86] W. Wilson, L. Rosenberg, and L. Hyatt. Automated analysis of requirement specifications. In *ICSE '97*, pages 161–171, May 1997.

[87] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, October 2005.

[88] Z. Xu, X. Zheng, and P. Guo. Empirically validating software metrics for risk prediction based on intelligent methods. In *International Conference of Intelligent Systems Design and Applications (ISDA)*, pages 1049–1054, 2006.

[89] W. Youden. Index for rating diagnostic tests. *Cancer*, 3:32–35, 1950.

[90] W. Yousef, R. Wagner, and M. Loew. Comparison of non-parametric methods for assessing classifier performance in terms of roc parameters. In *in Proceedings of Applied Imagery Pattern Recognition Workshop*, volume 33(13-15), pages 190–195, 2004.

[91] M. Zhao, C. Wohlin, N. Ohlsson, and M. Xie. A comparison between software design and code metrics for the prediction of software fault content. *Information and Software Technology*, 40(14):801–809, 1998.

[92] S. Zhong, T. Khoshgoftaar, and N. Seliya. Analyzing software measurement data with clustering techniques. *Intelligent Systems, IEEE*, 19(2):20–27, Mar-Apr 2004.

[93] S. Zhong, T. Khoshgoftaar, and N. Seliya. Unsupervised learning for expert-based software quality estimation. *High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on*, pages 149–155, March 2004.

[94] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *PROMISE'07: International Workshop on ICSE Workshops 2007*, pages 9–9, May 2007.