IMPROVING THE PERFORMANCE OF SOFTWARE DEFECT PREDICTORS WITH INTERNAL AND EXTERNAL INFORMATION SOURCES

by

Burak Turhan

B.S., in Computer Engineering, Boğaziçi University, 2002

M.S., in Computer Engineering, Boğaziçi University, 2004

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

Graduate Program in
Boğaziçi University
2008

ACKNOWLEDGEMENTS

My thanks to my advisor Assist. Prof. Ayşe Bener are not template forced; they are most sincere. I will not even try to explain her contributions not only to my research but also to my view of life. I am grateful to Prof. Lale Akarun and Assoc. Prof. Necati Aras for their insightful comments, feedbacks and guidance during the process. I thank Prof. Oğuz Tosun and Prof. Volkan Atalay for kindly accepting to be in my defense jury. Further, I owe great thanks to Assoc. Prof. Tim Menzies, who has considerable contributions to the research to be described in this dissertation. This work would not have been completed without partial supports of Boğaziçi University Research Fund (BAP 06HA104), Turkish Scientific Research Council (TÜBİTAK EEEAG 108E014) and industrial supporters: Arçelik, Turkcell, Bilmed and Infina. Although we had no relation, I somehow cannot help thanking Dharma Initiative for their motivational support. By the way, I managed to complete this work despite all the counter efforts of Bedaş A.Ş. Who can eliminate the endless support provided by friends? Ali Haydar Özer, Onur Dikmen, Itır Karaç, Oya Aran, Berk Gökberk, Albert Ali Salah, Aydın Ulaş, Rabun Koşar, Koray Balcı, Olcay Taner Yıldız and all of you in or from the department, whose names I cannot fit in FBE's page limits, have my sincere gratitudes. Technical contributions and friendship of all previous and current members of Softlab made this research possible: Onur Kutlubay, Ataç Deniz Oral, Yomi Kastro, Ayşe Tosun, and the list goes on. I also thank everyone, who resides at most two levels away from me in the family tree. Nothing I will say can appreciate Kamel Karamel's efforts for the completion of this dissertation. He shared his home with me along these years. He was kind enough to piss everywhere, but not on my laptop. Although we had some troubles once in a while, he has always been passionate later on. As the 11^{th} custodian of the secrets of the great being *Sipsi*, I kneel down in the presence of him. And... Yes, I did not forget you. Oh Junior! Where are thou? My wife hates her name to appear in documents like this (I suspect that she may be some kind of under cover black-op agent). Therefore, with due respect, *this dissertation is dedicated to my lovely wife* (See? I did not reveal your name...)

ABSTRACT

IMPROVING THE PERFORMANCE OF SOFTWARE DEFECT PREDICTORS WITH INTERNAL AND EXTERNAL INFORMATION SOURCES

In this dissertation, we make an analysis of software defect prediction problem from a data mining perspective, where software characteristics are represented with static code features and defect predictors are learned from historical defect logs. We observe that straightforward applications of data mining methods for constructing defect predictors have reached a performance limit due to the limited information content in static code features. Therefore, we aim at increasing the information content in data without introducing new features, since collecting these may either be expensive or not possible in all contexts. We feed data mining methods with richer data in terms of information content. For this purpose, we propose the following methods: 1) relaxing the assumptions of data miners, 2) using project data from multiple companies, 3) modeling the interactions of software modules. For the first method, we use naive Bayes data miner and remove its i) independence and ii) equal importance of features assumptions. Then we compare the performance of defect predictors learned from local and remote data. Finally, we introduce call graph technique to model the interactions of modules. Our results on public industrial data show that: 1) relaxing the assumptions of naive Bayes may increase defect prediction performance significantly, 2) predictors learned from remote data have great capability of detecting defects at the cost of high false alarms, however this cost can be removed with the proposed filtering method 3) proposed way of modeling interactions may decrease the false alarm rates significantly. Our techniques provide guidelines for 1) employing defect prediction using remote information sources when local data are not available, 2) increasing prediction performances using local information sources.

ÖZET

DAHİLİ VE HARİCİ BİLGİ KAYNAKLARI İLE YAZILIM HATA TAHMİNİ PERFORMANSININ İYİLEŞTİRİLMESİ

Bu tezde, yazılım hata tahmini probleminin, yazılım karakteristiklerinin statik kod ölçütleriyle temsil edildiği ve hata tahmin modellerinin geçmiş hata kayıtlarından öğrenildiği, veri madenciliği perspektifinden analizi yapılmıştır. Hata tahmin modelleri oluşturmak için uygulanan veri madenciliği metodlarının, statik kod ölçütlerindeki kısıtlı bilgi içeriğinden dolayı üst performans limitlerine ulaştığı gözlemlenmiştir. Bu sebeple, yeni ölçütler kullanmadan, verideki bilgi içeriğinin arttırılması hedeflenmiştir. Çünkü yeni ölçütlerin toplanması ya maliyetli olmaktadır ya da her durumda mümkün olmamaktadır. Veri madenciliği metodları bilgi içeriği açısından zengin veriler ile beslenmiştir. Bu amaçla 1) veri madenciliği metodlarının varsayımları, 2) birden çok şirketin proje verilerinin kullanılması, 3) yazılım modülleri arasındaki ilişkilerin modellenmesi analizleri gerçekleştirilmiştir. İlk analizde, naive Bayes metodunun ölçütlerin i) bağımsızlığı ve ii) eşit öneme sahip oldukları varsayımları ortadan kaldırılmıştır. Daha sonra yerel ve yabancı veriyle öğrenilen hata tahmin modelleri karşılaştırılmıştır. Son olarak, modül ilişkilerini modellemek için çağrı grafikleri analizi yapılmıştır. Kamuya açık endüstriyel veriler üzerinde yapılan analiz sonucunda: 1) naive Bayes varsayımlarının ortadan kaldırılmasının hata tahmini performansını arttırabildiği, 2) yabancı verilerle öğrenilen hata tahmini modellerinin hata yakalama kapasitelerinin -fazla yanlış alarm maliyetiyle- çok yüksek olduğu; ancak bu maliyetin önerilen süzme tekniğiyle ortadan kaldırılabildiği, 3) önerilen ilişki modeli ile yanlış alarmların azaltılabildiği gözlemlenmiştir. Yapılan analizler 1) yerel veri olmadığı durumlarda yabancı veriyle hata tahmini yapabilmek, 2) yerel verilerle tahmin performansını arttırmak açısından yol gösterici ilkeler sağlamaktadır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

AI	Artificial Intelligence
BAL	Balance
CC	Cross Company
CE	Cross Entropy
CGBR	Call Graph Based Ranking
CMMI	Capability Maturity Model Integrated
CVS	Concurrent Versions System
EP	Exponential Probability
GR	Gain Ratio
IEEE	Institute of Electrical and Electronics Engineers
IG	Information Gain
IID	Independent and Identically Distributed
ISO	International Standards Organization
KL	Kullback Leibler Divergence
LOC	Line of Code
LD	Linear Discriminant
LP	Log Probability
MWU	Mann-Whitney U Test
NB	Naive Bayes
NN	Nearest Neighbor
OR	Odds Ratio
PC	Personal Computer
PCA	Principal Component Analysis
PD	Probability of Detection
PF	Probability of False Alarm
PROMISE	Predictor Models in Software Engineering
QD	Quadratic Discriminant
SE	Software Engineering
SOFTLAB	Software Engineering Research Laboratory

SVN	Subversion
V & V	Verification and Validation
WC	Within Company
WNB	Weighted Naive Bayes

1. INTRODUCTION

Software products play a part in our lives covering a wide spectrum including PC applications, mobile applications, home appliances, telecommunication infrastructure, automated production lines, mission critical space explorations, nuclear powerplant control and so on. This was not the case in the midst of the 20^{th} century, when software applications were being developed by certain parties for specific tasks. As the complexity of software systems and the interactions between increasing number of developers have grown, the need for an engineering discipline has emerged to solve common problems of the domain: completing projects on time, within budget with minimum errors.

Software projects are inherently difficult to control. A landmark is the 1995 Standish group report that described a $250 billion dollar American software industry where 31% of projects were canceled and 53% of projects incurred costs exceeding 189% of the original estimate [6]. In 2004 version of the report (see Figure 1.1), the percentage of failed projects dropped to 18%, yet the rate of successful projects was only 29%. These reports are important in the sense that they reflect a snapshot of the greatest problems of software industry.

Considering the criteria of Figure 1.1, a software project needs to have two important properties in order to be classified as succeeded. The first one is that it should be completed within schedule and budget. Most of the software projects are regarded as challenged using this property. To satisfy the schedule constraints, every step of the software project life-cycle should be carefully planned and employed. Moving on to the next step requires the most possible confidence on the quality of the preceding level. The budget constraint depends on both satisfying the schedule constraint and the quality of the final product. Unfortunately, most software companies do not take maintenance costs into account and try to minimize the expenses considering only the period up to the release of the software product.

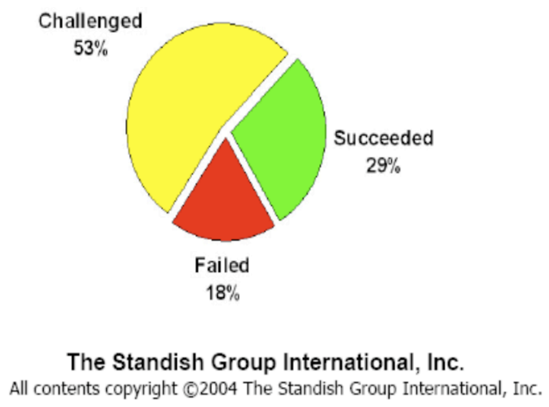


Figure 1.1. Distribution of failed, challenged and succeeded software projects in 2004. From [1].

The second important property of a successful software project is that the final product should satisfy the customer expectations by conforming to their requirements. Clearly, this is not enough for a software company to permanently hold a share in a competitive market place. Additional functionality and originality should also be involved. However, producing software with better and more complex functionality does not always yield successful projects in terms of customer satisfaction due to quality problems.

1.1. Software Quality, Testing and Defect Prediction

The key element that affects the success of a software project is the concept of quality [7]. Ensuring to have quality standards at each step of software development lifecycle inherently includes the satisfaction of customer expectations, schedule constraints and the budget constraints, and brings along success.

Like every other product or service providing industry, there are two main concerns of the software industry:

1. to present the highest quality products and services,
2. to achieve these at the lowest costs.

These concerns have drawn much attention to software quality. Software quality deals with establishing models of software development (i.e. waterfall, spiral and agile methods), standardization of software development processes and improving these (i.e. CMMI, ISO, IEEE Standards) [8, 9, 10, 11]. Empirical software quality research makes use of quantitative and qualitative features referred to as software metrics. Several metrics that are assumed to represent the characteristics of software are proposed [12, 13, 14]. Metrics enable measuring and quantifying the properties of software systems.

Quality of software is often measured by the number of defects in the final product. Minimizing the number of defects -maximizing software quality- requires a thorough testing of the software in question. On the other hand, testing phase requires approximately 50% of the whole project schedule [15, 16]. This means testing is the most expensive, time and resource consuming phase of the software development lifecycle. An effective test strategy should consider minimizing the number of defects while using resources efficiently. Therefore, effective testing leads to a significant decrease in project costs and schedules.

In 2002 IEEE Metric Panel, a group of noted researchers have agreed that [17] fixing defects in a software product after being delivered to the customer is up to 100 times more expensive than finding and fixing it during the requirements and design phases [18, 17]. They have also argued that up to 50% of effort is spent on avoidable work, 80% of which comes from a small number of defects (i.e. $\tilde{2}0\%$) in the system.

In this sense, defect prediction models are helpful tools for guiding software testing. The aim of defect prediction is to give an idea about the testing priorities, so that either exhaustive testing is prevented or larger number of defects are detected in shorter times. Accurate estimates of defective modules may yield decreases in testing times and project managers may benefit from defect predictors in terms of allocating the testing resources effectively [19]. Correctly identifying defective modules not only yields shorter test phases but also enables earlier release of products, which is very important in competitive markets. Shorter test phases also decrease project costs in terms of man hours spent on the project.

1.2. AI in Software Engineering

In many real world problems, there are lots of random factors affecting the outcomes of a decision making process. It is usually impossible to consider all these factors and their possible interactions. Under such uncertainty, AI methods are helpful tools for making generalizations of past experiences in order to produce solutions for the previously unseen instances of the problem [20]. These past experiences are extracted from available data, which represent the characteristics of the problem.

Fundamental concepts of AI are commonly used in other disciplines for problem solving. A hardly exhaustive list, where methods and tools inspired from machine learning and data mining community are used, includes: image processing, face recognition, robotics, multi agent systems, medicine, finance [21, 22, 23, 24, 25, 26]. Many data mining applications deal with large amounts of data and their challenge is to reduce this large search spaces. On the other hand, there exist domains with very limited amount of available data. In this case, the challenge becomes making generalizations from limited amounts of data.

In this context, software engineering is a domain with many random factors and relatively limited data. Nevertheless, in software domain, remarkably effective predictors for software products have been generated using data mining methods. The success of these models seems unlikely considering all the factors involved in software development. For example, organizations can work in different domains, have different processes, and define or measure defects and other aspects of their product and process in different ways. Furthermore, most organizations do not precisely define their processes, products, measurements, etc. Nevertheless, it is true that very simple models suffice for generating approximately correct predictions for software development time [27], and the location of software defects [28].

One candidate explanation for the strange predictability in software development is that despite all seemingly random factors influencing software construction the net result follows very tight statistical patterns. Other researchers have argued for similar results [29, 30, 31, 32, 33]. In this dissertation we also observe such patterns.

Building software defect predictors via data mining is also an inductive generalization over past experience. According to Mitchell's classic model of data mining [20], any inductive generalization explores a space of possible theories. All data miners hit a performance ceiling effect when they cannot find additional information that better relates software metrics with defect occurrence. What we observe from recent results is that current research paradigm, which relied on relatively straightforward application of machine learning tools, has reached its limits. To overcome these limits, researchers use combinations of metric features from different artifacts of software, which we call *information sources*, in order to enrich the information content in the search space. However, these features from different sources come at a considerable collection cost and are not available in all cases. Another way to avoid these limits is to use domain knowledge. In this dissertation we combine the most basic type of these features, i.e. source code measurements, with domain knowledge and we propose novel ways of increasing the information content using these information sources. We especially aim at startup or small companies that have scarce of no historical defect data. Using domain knowledge, we show that data miners for defect prediction can easily be constructed with limited or no data.

1.3. Contributions

The contributions of this dissertation are relevant for both academia and practice. They will be discussed in detail in Chapter 6. Nevertheless, we provide a summary of contributions below:

- *Evaluation of model assumptions on software defect data*: Prior to this dissertation, all applications of machine learning models for constructing defect predictors were, in general, straightforward use of algorithms. No study was performed to extend these applications by investigating the model assumptions. We use naive Bayes as a baseline method and investigate the validity of its assumptions on software defect data.

- *Constructing defect predictors by combining data from multiple companies:* Prior to this dissertation, no study was performed to investigate the relative merits of using cross company or within company data for constructing defect predictors. We make an extensive analysis to determine the conditions for using remote data from other companies and find that it may be preferred in certain domains.
- *Empirical evaluation of the common belief that local data is better for constructing defect predictors:* We compare performances of defect predictors learned from cross company and within company data. In favor of the common belief, we empirically show that using within company data is better for defect prediction problem.
- *Empirical evaluation of the common belief that the time required to collect local data takes too long:* We employ an incremental learning approach and compare defect predictors learned from different amounts of local data. Contrary to the common belief, we empirically show that the required local data for constructing defect predictors can be easily and quickly collected within a few person-months.
- *A methodology that allows companies to perform defect prediction without collecting local defect logs:* We use a simple filtering of cross company (CC) data to obtain a more homogeneous subset of CC data that reflects local coding characteristics. Our analysis suggests that in the absence of local data, companies can benefit from cross company data with: high probability of detecting defective methods, affordable false alarm rates, a minor investment, which is to use an automated tool to collect local static code measurements.
- *Modeling the interactions between software modules rather than relying on simple counts of interactions:*. We propose to use module interactions in a way that has not been employed in defect prediction literature and practice. We use PageRank algorithm to assign ranks to software modules, where modules correspond to web pages and interactions correspond to hyperlinks. Our proposed method decreases the false positive rate of defect predictors significantly, while detecting the same number of defects with standard approach.
- *Donation of new industrial datasets to public repositories:* We have collected data from Turkish industry and made it publicly available for the use of other researchers and practitioners. This is an important contribution in the field of

software quality where industrial defect data are difficult to obtain.

- *Prest: An open source software metric collection and defect analysis tool:* With contributions of other SOFTLAB researchers, we have developed an open source tool, named Prest, to collect and analyze data from our industrial partners. Though there are commercial products for the same purpose, Prest is the only free tool with a capability of extracting more than 40 static code features from C, C++, Java and Jsp codes. It also has an analysis module currently including naive Bayes and decision tree data miners.

1.4. Organization

This dissertation is organized as follows: In Chapter 2, we provide the necessary background on software metrics and defect prediction. We discuss the related work together with the motivational background of the study and present our research questions in Chapter 3. Chapter 4 describes the data mining methods, data used in our analysis and the measures for assessing performance in detail. A set of six experiments that are designed to answer the research questions are carried on in Chapter 5. In Chapter 6, we conclude our research, summarize the results of the experiments, give answers to the research questions and point future research directions.

2. BACKGROUND

This chapter is reserved for the necessary background to describe the approaches used to solve the defect prediction problem. We will first describe software metrics, which provide the characteristics of software projects in qualitative and quantitative forms. Then, we describe the software defect prediction problem and give details on the commonly used approaches.

2.1. Background on Software Metrics

As in any machine learning problem, software defect prediction models require a set of features (i.e. independent variables) to characterize the problem and to give an estimation on the defect proneness of the system (i.e. dependent variable). In software quality, these attributes are referred to as *software metrics*. Metrics are the attributes that represent software; they are the raw data for software domain. An effective management of any software development process requires monitoring and analysis of software metrics. Sequential measurements of quality attributes of software and also processes can provide an effective foundation for initiating and managing process improvement activities [34].

Considering the software defect prediction problem, defect predictors have been successfully learned from product and process metrics. While product metrics are derived from the software product itself, process metrics are derived from the processes that yield the product. Although we only use product metrics in this dissertation, we will provide brief information about process metrics for the sake of completeness.

In the following sections we give details about types of software metrics relevant to this dissertation.

2.1.1. Static Code Features

Static code features are the most basic type of software metrics that are directly extracted from source code. They give indications about the size and the complexity of the implemented code. As a quality indicator, static code features can only be used at later stages of software development (i.e. implementation), since they require source code to be available. However, analysis of static code metrics is crucial for ensuring the quality of later stages, i.e. testing and maintenance, which are the most resource consuming stages of software development lifecycle [15, 16].

Static code features can be organized into three main categories and each category of metrics uses a different perspective to estimate the complexity of the code. The first category is based on simple line counts of source code and referred to as *line of code (LOC)* metrics. Other two categories are the McCabe [12] and Halstead [13] metrics, which are more complex than LOC metrics. They give estimates about the code complexity based on program flow and readability of the code, respectively. We leave a critical discussion on static code features to Section 3.2 and describe these metrics in detail in the next section.

<ins>2.1.1.1. Line of Code (LOC) Metrics.</ins> LOC metrics are the simplest measures that can be extracted from source code. These include, but are not limited to:

- Total Lines of Code: It is a simple count metric, where one line equals one count. It is assumed that longer codes indicates more complexity, hence are more defect-prone.
- Blank lines of code: It is a count of blank lines in source code. It can be used as a measure of compliance to programming standards.
- Lines of Commented Code: It is a count of code comments and can be useful for estimating the maintainability and readability of source code. Comments usually exist in block forms before the actual implementation of methods, providing a generic description.

Table 2.1. Sample Source Code

```
void main()
{       //This is a sample code


        int a, b, c;


        a = 2; b = 5;                                                //M1


        //Find the sum and display c if greater than zero
        c = sum(a, b);                                               //M2
        if c  >  0                                                   //M3
                printf(%d\n, c);                                     //M4
        return;                                                      //M5
}

int sum(int a, int b)
{       // Returns the sum of two numbers
        int c;


        c = a + b;
        return c;
}
```

- Line of Code and Comment: It is a count of code lines that include both executable statements and comments. This kind of comments allows developers to understand what that line is supposed to do, rather than providing a generic description of the functionality as in block comments.
- Line of Executable Code: It is a count of the actual code statements that are executable (i.e. total lines of code after the blank and commented lines are ignored)

For the main method of the sample code given in Table 2.1, there are a total of 13 lines of code with three blank lines. Seven lines include comments, five of which have both code and comments. Executable code line count is six.

<u>2.1.1.2. McCabe Metrics.</u> Introduced in 1976 by Thomas McCabe, the idea behind McCabe metrics is to capture the structural complexity level of a code [12]. The assumption is that it is more likely for the number of defects to increase as the source code

gets more complex. McCabe metrics include cyclomatic complexity, design complexity and essential complexity. McCabe argued that the complexity of pathways *between* module symbols are more insightful than just a count of the symbols [12].

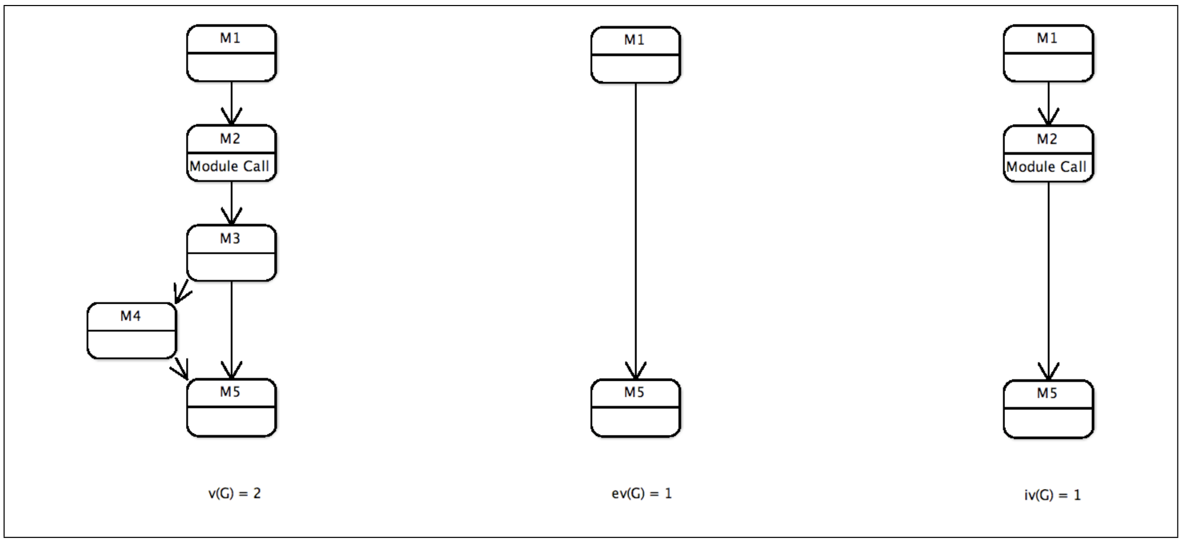


Figure 2.1. Flowgraphs and McCabe metrics for the main method of sample code.

In order to calculate McCabe metrics, flow graphs should be generated. A flow graph is a directed graph G where each node corresponds to a program statement, and each arc indicates the flow of control from one statement to another. Then the *cyclomatic complexity* of a module is

$$v(G) = e - n + 2 \tag{2.1}$$

where G is a program's flow graph, e is the number of arcs in the flow graph, and n is the number of nodes in the flow graph [35].

Cyclomatic complexity measures the number of linearly independent paths of a program execution. For example, if there is no branching in a module (i.e. no conditional statements such as if, while), then the corresponding cyclomatic complexity is 1. Because in this case $e = n - 1$ and $v(G) = -1 + 2 = 1$. Further, if there is only one branching, then there are two possible paths that the program can execute, hence its cyclomatic complexity is 2.

Although, high cyclomatic complexity by itself does not necessarily mean that a module has high risk, there is a certain consensus among researchers and practitioners on the thresholds of this metric. For example, NASA Independent Validation and Verification (IV&V) activities use this metric for prioritizing the test order [36]. In general, a module with cyclomatic complexity greater than 10 is considered as a high risk module.

Other types of McCabe metrics are derived from cyclomatic complexity.

- The *essential complexity*, ($ev(G)$) of a module is the cyclomatic complexity of the reduced program flow graph where the structured paths are removed. Therefore, it is a measure of unstructured constructs in a module.
- The *design complexity* ($iv(G)$) of a module is the cyclomatic complexity of the reduced program flow graph, where the paths that do not include a call to another module are removed. Therefore, it is a measure of module's interactions with other modules.

Figure 2.1 shows an example of Table 2.1 main method's flowgraphs and corresponding McCabe metrics.

<u>2.1.1.3. Halstead Metrics.</u> Halstead metrics were derived by Maurice Halstead in 1977, who argued that the harder the code to read, the more defect prone the modules are [13]. While McCabe metrics are measures of structural complexity, Halstead metrics are measures of lexical complexity. Halstead metrics include four basic metrics, which are the:

- N_1: total number of operators,
- N_2: total number of operands,
- n_1: unique number of operators ($min(n_1) = 2$),
- n_2: unique number of operands ($min(n_2) : number of module parameters$).

These basic Halstead metrics are then used to compute the derived Halstead features:

- Length: $N = N_1 + N_2$
- Vocabulary: $C = n_1 + n_2$
- Volume: $V = N * \log C$
- Level: $L = \frac{2}{n_1} * \frac{n_2}{N_2}$
- Difficulty: $D = 1/L$
- Content: $I = L * V$
- Effort: $E = V * D$
- ProgramTime (seconds): $T = E/18$

For the example in Table 2.1, these metrics are: $N_1 = 4$, $N_2 = 8$, $n_1 = 2$, $n_2 = 3$, $N = 12$, $C = 5$, $V = 12 \log 5$, $L = \frac{3}{8}$, $D = \frac{8}{3}$, $I = \frac{9}{2} \log 5$, $E = 32 \log 5$, $T = \frac{16}{9} \log 5$,

2.1.2. Other Types of Metrics

2.1.2.1. Object Oriented Design Metrics. After the object-oriented software development paradigm became popular, a lot of research has been dedicated to finding appropriate metrics especially for object-oriented design [37], [14], [38]. The idea is again to capture the complexity level of a software by inspecting within module and between module relations. Naturally, these metrics only apply to software developed with an object oriented language. These metrics are explained below.

- WMC (Weighted Methods per Class): In a simple framework where all methods in a class are considered to be equally complex, this metric is equal to the number of methods in a class.
- DIT (Depth of Inheritance Tree): This metric is defined as the maximum depth of inheritance graph of each class.
- NOC (Number of Children of a Class): This metric is equal to the number of direct descendants of a class for each class.
- CBO (Coupling between Object Classes): A class is considered to be coupled to another class if it uses methods and/or instance variables of that other class.

CBO counts the number of couplings between classes.

- RFC (Response for a Class): This metric quantifies the number of methods that can be executed for a message received by an object of a class.
- LCOM (Lack of Cohesion in Methods): This metric is equal to the number of pairs of methods of a class that share instance variables, subtracted from pairs of methods of the same class that do not share instance variables. This value is taken to be zero whenever the above operation results in nonnegative values.

2.1.2.2. Code churn/ History Metrics. Code churn metrics basically represent the change in source code during the implementation phase. Large scale software development typically involves the use of versioning systems such as Subversion (SVN)[1] and Concurrent Versions System (CVS).[2] In versioning systems, developers share a common repository of source code, so that they can work on different parts (in some cases on the same parts) of the code simultaneously. Considering the difficulty of communication and coordination issues, an increased risk factor is introduced with the number of different developers working on the same code. Furthermore, editing an existing code, by itself, introduces risk factors by adding, removing or modifying functionalities.

Code churn metrics are helpful for assessing the risk factor by measuring the degree of change in source code. Versioning systems usually provide a comparison tool between different versions of the committed code. Simply, these tools provide metrics such as:

- added lines of code
- deleted lines of code
- modified lines of code
- age of the code
- status of the code (i.e. new, changed, unchanged)
- number of changes made on the code(i.e. commit count)
- number of distinct developers who worked on the code

[1]`http://subversion.tigris.org`
[2]`http://www.cvshome.org`

- whether the code is modified by a developer, who has not created it
- whether a developer is working on that code for the first time

2.1.2.3. Developer Metrics. Additional information regarding the developers may also be helpful for determining the risk factor. Developers, who commit their code to the repository with unique ID's, may not necessarily have the same overall experience in software development or in the development of a specific software. While subjective measures of experience, developers' experiences are usually represented with their educational background, the number of years in profession or whether they have been involved in the development of a specific kind of software (i.e. the application domain: embedded, mission critical, pervasive, web application etc.). It is expected that an "experienced" developer introduces relatively less defects to the code than an "inexperienced" one.

2.1.2.4. Call Graphs. Module interactions play an important role in defect proneness of a given code [39]. Call graphs (or dependency graphs) can be used in tracing the software code module by module. Specifically, each node in the call graph represents a module and each edge (a, b) indicates that module a *calls* module b. Call graphs can be used to better understand the program behavior, i.e. the control flow between modules. Another advantage is locating procedures that are rarely or frequently called.

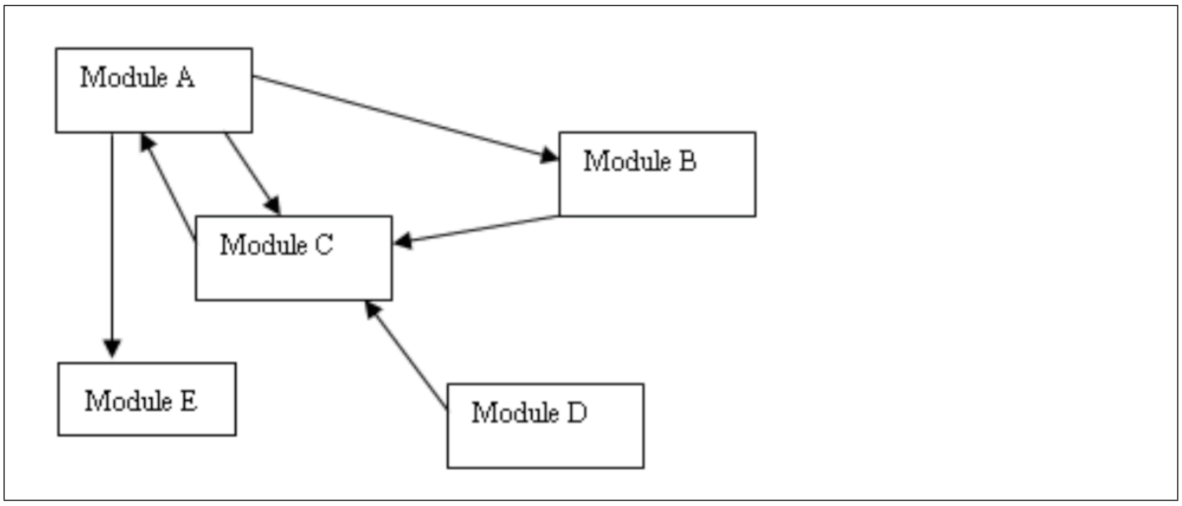


Figure 2.2. A sample architecture showing module interaction.

Table 2.2. Call graph matrix for the sample.

Caller/ Calee	A	B	C	D	E
A	0	1	1	0	1
B	0	0	1	0	0
C	1	0	0	0	0
D	0	0	1	0	0
E	0	0	0	0	0

An $N \times N$ matrix is constructed for building module level binary call graphs, where N is the number of modules. In this matrix, rows contain the information whether a module calls the others or not, which are represented by 1 and 0 respectively. Columns contain how many times a module is called by other modules. Table 2.2 shows an example call graph matrix for the sample module interactions in Figure 2.2.

In Table 2.2, module A calls three other modules, which are modules B, C and E. The first row in Table 2.2 shows the modules that are called by module A. These calls are represented with 1. In addition, if we look at the first column in Table 2.2, we observe that module A is only called from module C. Also module D is dead code unless it is a main routine, since it is not called from any other modules. Module C is the most critical one, since it is commonly used by other modules and any defect in module C can propogate into other modules.

2.2. Background on Defect Prediction

2.2.1. Defects

Before explaining defect prediction, we should first define what we are trying to predict: 'defect'. Unfortunately, the perception of what a defect is varies in different contexts. For example, bug databases of open source projects contain many user reported entries that request for bug corrections, however their quality is questionable [40]. These requests are frequently considered as feature requests or design decision manifestations by the developers of those systems. That means what one stakeholder considers as a defect may not necessarily be perceived as the same by other stakeholders.

The severity of defects may also cause different interpretations. Let us consider two cases where there are problems in the core functions of a billing system (i.e. miscalculations) and in the user interface of the system (i.e. wrong background color of a screen). In one point of view, both should be considered as defects since both are nonconformance to the requirements. On the other hand, the quality engineer of the system is likely to demand the detection of the defect in the core function rather than the user interface problem.

Accordingly, Nagappan & Ball make a contextual classification of software systems: zero-defect tolerant (i.e. safety and mission critical systems), low defect tolerant (i.e. operating systems) and high-defect tolerant (non-critical user applications) [41, 42]. In defect prediction perspective, a defect is defined by the context of the software system, considering what practitioners want to predict.

2.2.2. Defect Predictors

A defect predictor is a tool or method that guides testing activities. According to Brooks [43], half the cost of software development is in unit and systems testing. Harold and Tahat also confirm that testing phase requires approximately 50% of the whole project schedule [15, 16]. Therefore, the main challenge is the testing phase and practitioners seek predictors that indicate where the defects might exist *before* they start testing. This allows them to efficiently allocate their scarce resources. Defect predictors are used to make an ordering of modules to be inspected by verification and validation teams:

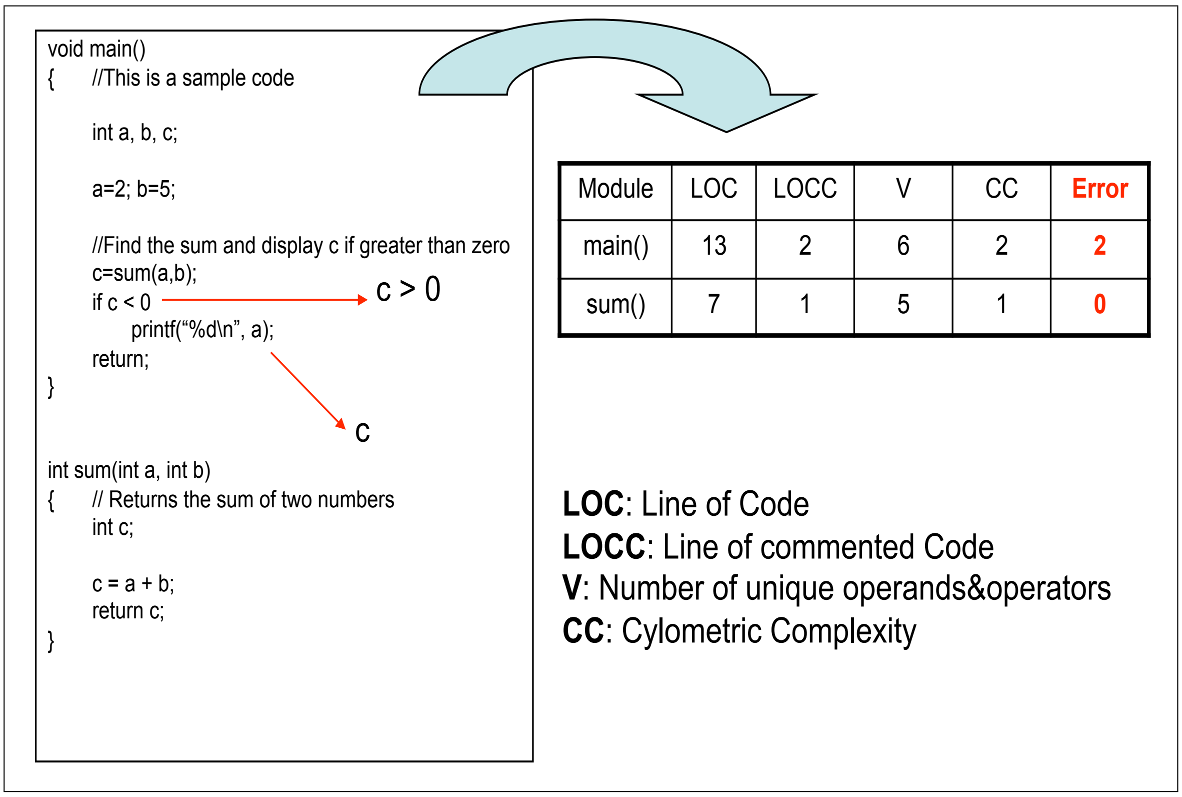
- In the case where *there are insufficient resources to inspect all code* (which is a very common situation in industrial developments), defect predictors can be used to increase the chances that the inspected code will have defects.

- In the case where *all the code is to be inspected*, but that inspection process will take weeks to months to complete, defect predictors can be used to increase the chances that defective modules will be inspected earlier. This is useful since it gives the development team earlier notification of what modules require rework, hence giving them more time to complete that rework prior to delivery.

2.2.2.1. Types of Defect Predictors. Application of defect predictors may vary depending on the context. One view is to focus on whether a module is defective or not and not to consider the number of defects in a module. The conceptual basis for this view is the idea that a potentially defective module has to be inspected no matter how many defects reside in it. This kind of application of defect predictors is suitable for domains such as mission critical, health-care, embedded and telecommunication infrastructure. Any defects in the final products of these domains may cause safety related issues (i.e. mission critical, health-care), yield great costs to update the software (embedded) or interrupt the availability of service (i.e. telecommunication infrastructure).

Other application type of defect predictors deals with estimating the number of defects at parts of software. This provides a test order for testing personnel to inspect as many defects as possible in the shortest amount of time.

The first application type handles defect prediction as a classification problem, whereas the second type handles it as a regression problem. For both types, the granularity level of predictions may vary depending on the availability of data: predictions can be made in module, class, file, binary file, package, component levels. However, classification type is usually preferred in fine-grained systems (i.e. module, class, file), whereas regression type is used in other systems (i.e. file, package, component) [44]. In this dissertation, we employ the first application type (i.e. classification) on the module level for reasons to become clear in Section 3.1 and Section 3.2, also considering the available data we use in our analysis (See Section 4.2).



```
void main()
{      //This is a sample code

       int a, b, c;

       a=2; b=5;

       //Find the sum and display c if greater than zero
       c=sum(a,b);
       if c < 0
              printf("%d\n", a);
       return;
}

int sum(int a, int b)
{      // Returns the sum of two numbers
       int c;

       c = a + b;
       return c;
}
```

Module	LOC	LOCC	V	CC	Error
main()	13	2	6	2	2
sum()	7	1	5	1	0

LOC: Line of Code
LOCC: Line of commented Code
V: Number of unique operands&operators
CC: Cylometric Complexity

Figure 2.3. A sample transformation from source code to quantitative defect prediction data.

2.2.2.2. Defect Prediction as a Classification Problem. Software defect prediction can be viewed as a supervised binary classification problem. Software modules are represented with software metrics, and are labelled as either defective or non-defective. To learn defect predictors, data tables of historical examples are formed where one column has a boolean value for "defects detected" (i.e. dependent variable) and the other columns describe software characteristics in terms of software metrics (i.e. independent variables).

An example is provided in Figure 2.3. In this example, two errors are introduced in the if and $printf$ statements of Table 2.1 on purpose. The sample code is supposed to display variable c if it is greater than zero. However, implemented code displays variable a if c is less than zero. Each row in the right hand side table of Figure 2.3 stores data from one"module" (i.e. main and sum). In this example we show only four metrics for simplicity. Construction of such data tables, except the "Error" column, can be easily and quickly done by metric extraction tools [45]. The"Error" column or the class label should be matched to corresponding modules after an examination of defect logs. Our experiences show that it is difficult (sometimes impossible) to do this matching, since defect logs are hard to obtain and understand, or do not exist.

After these data tables are constructed, the data mining task is to find combinations of features that predict for the value in the defects column. Once such combinations are found, managers can use them to determine where to best focus their testing effort. Better yet, if they have already focused their testing effort on the most critical portions of the system, the detectors can guide them towards modules that need the most attention. This type of data miners do not predict the total number of defects, just the number of modules containing more than zero defects.

3. RELATED WORK AND DISCUSSIONS

Body of research on defect prediction can be considered to aim at two main goals. These goals are not mutually exclusive, yet there is a gray area between them that makes it difficult to categorize individual studies in either one. These goals are:

- to understand the characteristics of defects and their relation with software metrics
- to build better defect predictors in practice.

Research aiming at the former goal inherently includes the latter, however research aiming at the latter one is not necessarily involved with the former one.

3.1. Defect Prediction Studies

Considering the first goal, Fenton and Ohlson made an extensive analysis of defects in two releases of a commercial product using product metrics [46]. They investigated: the defect distribution on software modules, the relation between pre-release and post-release defects, the effect of size and complexity on defect proneness and finally the stability of defect density among releases. They report that most of the defects reside in a small number of modules (i.e. Pareto principle [47]). The same fault distribution has been observed by Ostrand, Weyuker and Bell in very large scale telecommunication projects from AT&T [29, 30, 31, 32]. Furthermore, the defect trends in Eclipse also follow similar patterns, where they are explained better by a Weibull distribution [48].

Fenton and Ohlson found that pre-release defects are not good predictors of post-release defects. They also report that they have found no evidence regarding the relation between defects and the size/ complexity of the modules and they observe similar defect densities in the two versions. Their study is replicated by Andersson and Runeson in order to verify/ refute Fenton and Ohlson's findings in three products of a different domain [49]. While Andersson and Runeson's results conversely found support that pre-release and post-release defects are correlated, their replication confirmed the results of Fenton and Ohlson on other issues.

Koru and Tian, in their analysis of two IBM and four Nortel Networks products, also report that most defect prone modules are not necessarily the ones with the highest complexity measures [50]. On the other hand, Koru and Liu's further research on open source Mozilla project strongly argues that smaller modules are more defect-prone than larger modules, whereas Fenton and Ohlson find limited support for this hypothesis [51]. Further, in their analysis of a NASA project, Koru and Liu reports that averaging metrics at a higher granularity level (i.e. class level rather than module level) yields better prediction results [52]. However, this observation is based on a single project due to limited data availability. A discussion of granularity levels will be detailed later on this section.

Ratzinger et.al. investigates the relation between refactorings and defects in open source software [53]. They conclude that refactorings and defects have inverse correlation, that is refactored modules are less defect prone. This is somehow expected, considering that refactoring is a technique for decreasing the complexity of software modules and defects are likely to be introduced as the code gets more complex.

Ohlsson and Alberg developed tools for extracting design metrics from design documents of Ericsson telephone switches and their predictor detected 47% of the defects in 20% of all modules [54]. The importance of this model lies in its applicability before the implementation phase. Considering object oriented design metrics introduced by Chidamber and Kemerer [14], different studies performed experiments to validate their use [55, 56, 57, 58]. Subramanyam and Krishnan found them useful cautioning that the predictive power of these metrics vary in different programming languages [56]. However, El Emam et. al. argued that when the effect of class size is also considered, only 4 of these metrics are related with defects and just two of them are useful for building predictors [57].

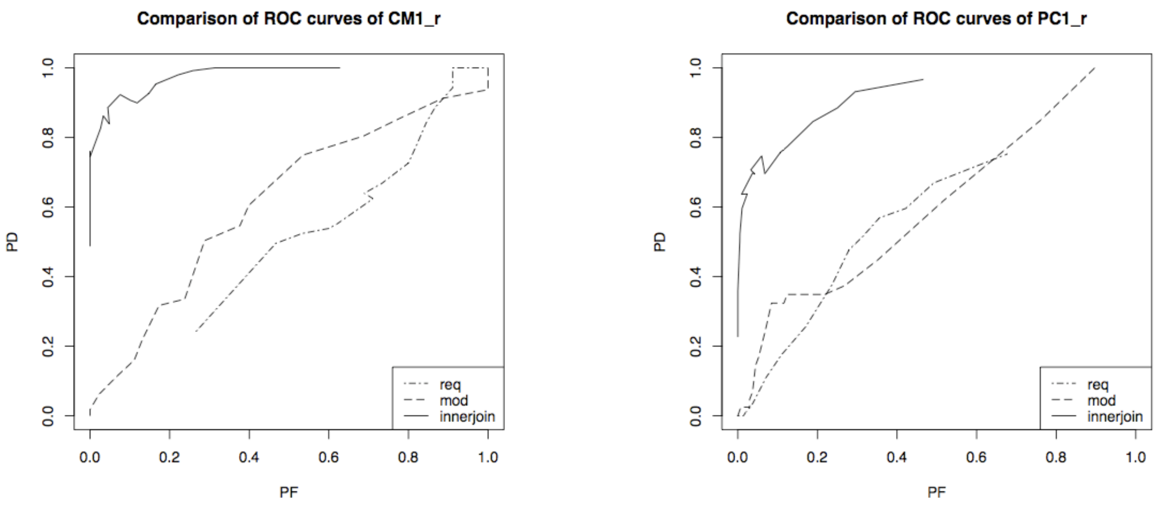


Figure 3.1. Defect predictor performances with requirement metrics, static code features and their combinations. From [2].

Further, Jiang et al. compared the predictor performances that are learned from design metrics, static code features and both on 13 NASA projects, concluding that combination of these metrics are able to predict more accurately than their individual use [59]. Same conclusion is also achieved by Zhao et.al in their analysis of a real time telecommunication system [60].

Jiang et.al. have further explored combining static code measures with other measures that a particular domain may contain. For example, at ISSRE 2007 [2], they reported experiments on NASA projects, where static code measures were combined with requirement metrics that were extracted from requirement documents with a text miner. They report that how a remarkable improvement in learner performance was achieved by applying *combinations* of requirements and code features (see Figure 3.1).

Fenton et.al use Bayseian Belief Networks to model the causal relationships between the defects and a set of qualitative and quantitative features [47, 61]. They report high accuracies (e.g. $R^2 = 0.9311$), however their data collection procedure is challenging. They perform surveys to collect data and to build the causal model, which turns out to be company specific and requires a lot work to reproduce in other companies. For example, another study that constructs Bayesian Belief Networks, merges both software and hardware development of embedded system [62]. They mainly reconstruct Bayesian Network of Fenton et al. and modify it by using embedded software development lifecycle. The results of embedded and general model indicate that it is necessary to use an embedded model for practical use.

Brook's argument on the effects of organizational factors on software quality is empirically investigated by Nagappan et.al. on Microsoft Windows Vista product [43, 63]. Along with complexity metrics, they propose to use a set of features that reflects the organizational complexity that leads to the software product. Their results imply that organizational factors have better predictive power than complexity metrics in case the organization have at least 30 engineers with a minimum three levels of organizational hierarchy. [63, 64, 65].

In another study, Nagappan et.al. also analyze the relation between code churn metrics and post-release defects in another Microsoft software [41]. They conclude that relative code churns are good predictors for post-release defects. In further studies they investigate many sources of information such as static code features, code churn, dependency graphs, developers experience, organizational structure, assertions and their combinations [64, 66]. For example Zimmermann and Nagappan report a 10% increase in detection performance when metrics derived from dependency graphs are used rather than static code features. Further, Kudrjavets et.al. report an inverse correlation between assertions in code and defect density [67].

Graves et. al. argue that change history metrics are better predictors of defects than their size [68]. Supporting this claim, Arisholm and Briand report that code history metrics are paramount for better prediction performance especially for systems that are under continuous development [69]. They compare both approaches in a Java legacy system using step-wise logistic regression, and show that the testing efforts significantly decreases history data are used while pure code metrics does not improve the testing effort. A cost sensitive analysis is performed by Moser et.al., who also concluded in favor of change metrics compared to static code features. However, they also report the detection performance increases when both type of metrics are used in combination [44]. Their proposed model is to use change metrics at first and then to include static code features to improve performance if needed.

Ostrand, Weyuker and Bell occasionally investigate multi-release AT&T software using static code metrics, history metrics, developer metrics and their combinations [29, 30, 31, 32, 70]. Their goal is to identify and prioritize the top 20% of all files containing the most number of defects. Their negative binomial model is able to identify 80% of the defects in 20% of files. Their observations are similar to other reported results: combining static code metrics with history data improves the prediction performance significantly. However, they recently report that adding information about individual developers metrics does not improve the performance of their negative binomial model significantly [71]. Mockus and Weiss perform a similar analysis on a large telecommunication system using a different set of developer metrics and reach to the conclusion that more experienced developers introduce less defects[72]. Weyuker et.al. also investigates recursive partitioning in defect data, reporting up to 85% defects detected in 20% of the files [70].

Considering history metrics, our experiences in constructing *file* level defect predictors are inline with these research [73]. We have used file level aggregated[3] static code metrics of a large (i.e.   750.000 LOC), multi version telecommunication infrastructure software for defect prediction. As shown in Table 3.1 the detection rates are above 80%, yet the false alarms are also above 50%, which makes this detector impractical in that form. However, most of these false alarms are associated with files that are not changed for a long time. If history metrics were to be used, these false alarms were to be reduced with the prior knowledge that the corresponding files are defect-free and have not been changed. On the other hand, using this prior knowledge would not affect the detection rates. Nevertheless, static code features are useful at the module level as to be shown in Chapter 5 and should better be used together with different types of features in other granularity levels. Another reason for the high false alarms of Table 3.1 is that aggregated measurements lose information by summarizing. Since

[3]static code metrics from modules are aggregated by the minimum, maximum, sum and average measurements of modules in a file.

Table 3.1. File level predictors for four versions of a multi release telecommunication software using static code features only.

	v.2.32	v.2.33	v.2.34	v.2.35
Detection Rate	80%	80%	67%	100%
False Alarm Rate	59%	64%	44%	35%

Table 3.1 projects are very large scale and data miners need larger amounts of data to learn effective predictors, summarizing information is not effective, at least in the beginning. However, once enough data are collected, static code features by themselves can yield better performances. This can be observed in the 100% detection and 35% false alarm rate for the last version(i.e. *v*.2.35) where data from all previous three versions are used to construct that predictor.

Researchers on defect prediction have used various machine learning techniques such as linear regression, discriminant analysis, decision trees, neural networks and naive Bayes. Munson and Khoshgoftaar investigate linear regression models and discriminant analysis to conclude the performance of the latter is better [74]. Bullard et.al employ a rule based classification model in a telecommunication system and report that their model produces lower false positives, which are considered as high cost classification errors [75]. Khoshgoftaar and Gao propose a multi-strategy classifier for embedded software, which cascades a rule based approach with two case-based learning schemes for the same purpose [76]. A cascading classifiers approach is also performed by Tosun et.al., where they report decreased testing efforts on embedded software. Specialized prediction models for embedded systems are also investigated by Khoshghoftaar et al., where they built a classification and regression tree for predicting high risk software modules in telecommunications system software [77]. They also investigate genetic programming approaches to optimize multiple objectives for minimizing the false positives while maximizing the number of detected defects. They present the applicability of their model on a real life industrial software.

Nagappan et.al. also uses linear regression analysis with the STREW metric suite [78]. This suite of metrics are extracted from the testing process and are used to estimate the post-release defects. They validate their approach on industrial, open source and student projects and find strong correlations between the proposed metric suite and post-release defects. On open source software, Denaro and Pezze analyzed Apache using logistic regression with static code features and their 80% prediction performance pointed 50% of the modules to be inspected [79].

Nevertheless, In January 2007, Menzies et.al. published a study [28] that defined a repeatable experiment in learning defect predictors. The intent of that work was to offer a benchmark in defect prediction that other researchers could repeat/ improve/ refute. Surprisingly, very simple Bayes classifiers (with a simple logarithm pre-processor for the numerics) outperformed the other studied methods. They have later tried to find better data mining algorithms for defect prediction. The experiments that have found no additional statistically significant improvement from the application of the further data mining methods include: logistic regression, average one-dependence estimators [80], under- or over-sampling [81], random forests [82], RIPPER [83], J48 [84], OneR [85], Bagging [86] and Boosting [87].

Lessmann et. al. also investigated this issue and in a very recent paper in IEEE TSE, he reported no statistical difference between the results of 19 learners, including naive Bayes, on the same datasets [3]. Figure 3.2 shows that the simple Bayesian method discussed above ties in first place along with 15 other methods.

In summary, a general overview of defect prediction literature suggests the following:

- Different sets of software metrics are used to characterize the problem. However, there is not an agreement on a universal metric set. Defect predictors are constructed with whatever data are available. Combining different sets of metrics, i.e. combining different *information sources* improves the performance of defect predictors.
- Similarly, straightforward applications of data mining methods are usually used to approach the problem. However, there is not a consensus on which method is

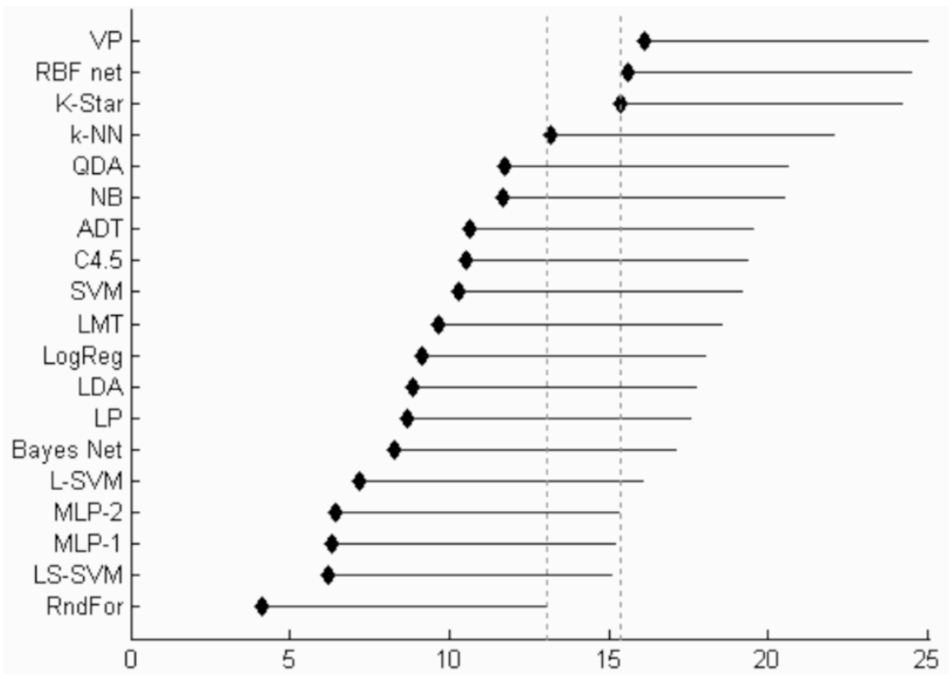


Figure 3.2. All the methods whose top ranks are 4 to 12 are statistically insignificantly different. From [3].

better. Latest results suggest that the selection of the method is not so important.

- All previous studies assume the existence of local defect data, and construct defect predictors accordingly. There are no studies investigating the problem in case of no defect data.
- There is not an analysis regarding the minimum amount of data required for constructing defect predictors.
- All studies assess the defect proneness of modules independently. Although some research include inter-module measurements, none of them models these interactions.

Considering these observations, we focus on increasing the information content in metric data. In doing so, we use the most basic type of metric, i.e. static code features, and we show how their information content can be increased in a novel way. Further, our proposed method does not require local defect data to be available. We also investigate the minimum amount of data required for constructing local defect predictors, when local defect data are available. In order to explore these issues in detail, we will first discuss the value of static code features, then the value of additional information sources for increasing the information content in data. We will provide further evidence regarding that performance increase in defect predictors based on static code features will not come from application of different algorithms due to their limited information content. Before stating our research questions, we will explore using data from multiple companies.

3.2. On the value of Static Code Features

In this dissertation, we focus on defect predictors based on static code features only. Therefore, we provide a discussion in this section for the justification of our approach.

In theory, defect predictors based on static code features are not useful. As Fenton suggests, *the same* functionality can be achieved using *different* programming language constructs resulting in *different* static measurements for that module [88]. Fenton uses this example to argue the uselessness of static code attributes. Further, Fenton & Pfleeger note that the main McCabe's attribute (cyclomatic complexity, or $v(g)$) is highly correlated with lines of code [35]. Shepperd & Ince repeated that result, commenting that [89].

> for a large class of software it (cyclomatic complexity) is no more than a proxy for, and in many cases outperformed by, lines of code

There are, indeed, many reasons to doubt the value of static code attributes for defect prediction. In their discussion on the difficulties of software process research, Basili and Shull state that

> It is clear that there are many sources of variation between one development context to another.

Accordingly, descriptions of software modules *only* in terms of static code attributes can overlook some important aspects of software including the type of application domain;

the skill level of the individual programmers involved in system development; contractor development practices; the variation in measurement practices; and the validation of the measurements and instruments used to collect the data. For this reason some researchers augment or replace static code features with other information sources such as the history of past faults or changes to code or number of developers who have worked on the code [68].

However, in practice, static code features are quite effective. If the above criticisms are correct then we would expect that, in general, the performance of a predictor learned by a data miner should be very poor. More specifically, the supposedly better static code attributes such as Halstead and Mccabe should perform no better than just simple thresholds on lines of code. However, many studies [90, 28, 91, 92, 93, 94, 95, 96, 97, 2, 98, 99], report performance results much higher than known industrial averages for manual defect detection [100, 101]. Bhat and Nagappan also acknowledges that static code features are useful as early indicators of software quality based on their analysis of Microsoft Windows code base [102]. The defect predictors learned from static code attributes perform surprisingly well.

Menzies et.al. compare their results against standard binary prediction results from the UC Irvine machine learning repository of standard test sets for data miners [103]. Their performance measures turn out to be very close to the standard results on this repository which, they say, is noteworthy in two ways:

> It is unexpected. If static code attributes capture so little about source code (as argued by Shepherd, Ince, Fenton and Pfleeger), then we would expect lower probabilities of detection and much higher false alarm rates.

- Their results are better than currently used industrial methods such as the 60% detection rates reported at the 2002 IEEE Metrics panel or the median detection rates varying from 21% to 50% reported by Raffo[28]

Considering the pros and cons, we study defect predictors learned from static code attributes since they are *useful*, *easy to use*, and *widely-used.*

Useful: Defect predictors are considered useful, if they provide a prediction performance that is comparable or better than manual reviews [44]. Further, defect predictors based on static code featuras are considered as static analysis, since they do not require the execution of code. Zheng et.al. compare automated static analysis (ASA) with manual inspections on Nortel software and conclude that [104]

> our results indicate that ASA is an economical complement to other verification and validation techniques.

Easy to use: Employing defect predictors in practice should not take too much time for data collection and constructing the models themselves [44]. An advantage of static code features is that they can be quickly and automatically collected from the source code, even if no other information is available. Static code attributes like lines of code and the McCabe/Halstead attributes can be automatically and cheaply collected, even for very large systems [42]. By contrast, other methods such as manual code reviews are labor-intensive; e.g. 8 to 20 LOC/minute can be inspected and this effort repeats for all members of the review team, which can be as large as four or six [105]. Furthermore, other features (e.g. number of developers, the software development practices used to develop the code) may be unavailable or hard to characterize.

Widely used: Many researchers use static features to guide software quality predictions (see [13, 12, 106, 91, 107, 108, 109, 42, 110, 111, 112, 113, 114, 97, 94, 93, 115, 116, 117, 118]). Verification and validation (V&V) textbooks ([119]) advise using static code features to select modules worthy of manual inspections. Menzies reports[4] that during his studies on-site at the NASA software Independent Verification and Validation facility, he has witnessed several large government software contractors that won't review software modules *unless* tools like McCabe predict that they are fault prone. He also acknowledges of two briefings he has attended with NASA and European Space Agency test engineers, who both cited McCabe's $v(g) \geq 10$ as one of their triggers for modules requiring closer inspections. Further, major software related companies including NASA, Microsoft, AT&T, IBM, HP, Nortel Networks use static code features

[4]Personal communication

as a factor in guiding their software quality efforts.

We take great care not to demand defect prediction should *always* be based either on static code features or other types of features. The "best" set of features may change from project to project [28, 120]. More pragmatically, we note that different projects use different tools and build software. Here, we identify a particular type of feature (i.e. static code features) which we can collect from multiple projects. With that data in hand, we will explore our research questions. For future work, we would repeat this analysis with other kinds of features, if and when they become available from multiple projects.

3.3. On Additional Information Sources

A closer look in the studies that are explained in Section 3.1 reveals an important observation. All studies include some basic features and augment the data with other information sources such as code churn, dependency graphs, developer information, requirement metrics when possible. Moreover, they report improved results whenever multiple information sources are combined.

The attempts to combine multiple information sources should not necessarily interpreted as "always augment some kind of feature with some other kind of feature". This is impractical, since not all domains allow data mining specialists access to source code and the processes that produced them. Furthermore, different projects have access to different sets of features which may be useful for predicting where defects hide; for example:

- An open source software (OSS) or agile development process may have no access to the detailed requirement documents used to build model by Jiang et.al. [2];
- Public project data from NASA sub-contractors report neither the personnel information used by Nagappan et.al. [121] nor the code churn history used by Ostrand et.al. [32]

However, it is not surprising to observe that combining multiple sources of information increases the performance of data miners. Providing additional information to data miners is expected to increase their performance. Results from data mining literature are analogous to these observations in defect prediction literature. For example, in the JMLR special issue on feature selection, Guyon and Elisseeff provide simple examples showing that the information content of data can be significantly increased when features are used together rather than individually [122].

In the context of this dissertation, we would re-express these examples as:

- Using features from different information sources, e.g. requirements and source code, can significantly increase the information content of SE data.
- Software Engineering domain should make use of domain specific knowledge (when possible) to differentiate from general data mining tasks.

Our goal in this dissertation is also to increase the information content in data, yet in a previously unexplored way. Before detailing our approach, in the rest of this section, we provide more evidence from literature and perform a small experiment, both suggesting that improvements will not come from application of different algorithms.

3.4. To Improve the Algorithms or to Improve Data?

As documented in Section 3.1, the current state of the art in learning defect predictors is curiously static. Better results have not been forthcoming, despite the application of supposedly better data miners. Considering all the failed efforts to improve the benchmark results by Menzies et. al. [28], these failed attempts to improve defect prediction performance require an explanation. One explanation is that exploring better algorithms may not be productive. Reported results in defect prediction literature suggest that further progress in learning defect predictors may not come from application of different algorithms.

After a careful study of 19 data miners for learning defect predictors, Lessmann et.al. [3] conclude

> the importance of the classification model is less than generally assumed and that practitioners are free to choose from a broad set of candidate models when building defect predictors.

If different types of algorithms are not useful for increasing the performance of data miners, perhaps it is time to better understand the training data. Further, we need to improve the information content of the training data that are fed to these algorithms. Accordingly, in this dissertation, instead of searching for better *data miners* we search for better *training data*.

In this context, Menzies et.al. further report over and under sampling experiments in order to check whether the performance of defect predictors can be increased by sampling methods due to their possible bias from the unbalanced nature of defect data. Over- and under-sampling [81, 123] are examples of more controlled sub-sampling methods than random sampling. Both methods might be useful in data sets with highly unbalanced class frequencies, which is usually the case for defect datasets (see Table 4.2). Regardless of which sub-sampling method is used, the result is a data set with an equal number of target and non-target classes.

That study again used a simple Bayes classifier, since it was useful in their prior experiment [28], plus a C4.5-like decision tree learner, J4.8 [124], since that was used in prior under- and over- sampling experiments [81].

The results are given in Table 3.2 and they are consistent with certain prior results. The simple Naive Bayes they recommended previously [28] performed as well as anything else. Seemingly cleverer learning schemes did not outperform simple Bayesian classifiers. Just like their previous results, throwing away data (i.e. under-sampling) does not degrade the performance of the learner. In fact, in the case of J48, throwing away data improved the median performance from around 40% to over 70%. Under-sampling beats over-sampling for both J48 and Naive Bayes. This result is consistent with Drummond & Holte's sub-sampling experiments [81] and the sub-sampling clas-

Table 3.2. Over-sampling, under-sampling and no sampling results. From [5]

treatment	0	25%	50%	75%	100%
NB/ none	21.9	67.7	74.6	81.9	100.0
NB/ under	19.9	67.1	74.1	81.6	100.0
J48/ under	21.6	64.8	73.6	82.6	100.0
NB/ over	17.5	42.0	62.5	72.2	100.0
J48/ over	0.0	29.3	45.6	56.2	100.0
J48/ none	0.0	29.3	42.3	54.5	100.0

sification tree experiments of Kamei et.al. [125].

However, Table 3.2 has also some new results. We can observe that $NB/none$ is one of the topped-ranked methods. That is, sub-sampling decision tree learning does not outperform Naive Bayes. $NB/none$ ties with $NB/under$. That is, while sub-sampling offers no improvement over un-sampled Bayesian learning, under-sampling does not harm classifier performance. This last point is the most significant. It means effective detectors can be learned from a very small sample of the available data. Further, Seliya and Khoshgoftaar performed experiments for defect prediction with limited data where they learned defect predictors with a subset of available data. In that study they used 800 defect free and 200 defective modules for building predictors and reported comparable results to using all data [126]. To explore this issue, we have performed a small experiment on the same project data where we focus on even a smaller scale. In order to determine the lower-limit on the number of cases that are required to build a stable defect predictor, we employed an extreme under-sampling policy, which we call *micro-sampling*. Given N defective modules in a data set,

$$M \in \{25, 50, 75, .\} \leq N$$

defective modules were selected at random. Another M non-defective modules were selected, at random. The combined $2M$ data set was then passed to a 10*10-way cross validation.

Formally, under-sampling is a micro-sampling where $M = N$. Micro-sampling explores training sets of size up to N, standard under-sampling just explores once data set of size $2N$.

For this study we used Naive Bayes since, in all the above work, it seems to be doing as well as anything else. Our results indicate that detectors learned from small M instances do as well as detectors learned from any other number of instances. For eight data sets, micro-sampling at $M = 25$ did just as well as any larger sample size. For one data set, best results were seen at $M = 75$. However , in many cases $\frac{8}{11}$, $M = 25$ did just as well as anything else. For three data sets best results were seen at $M = \{200, 575, 1025\}$. However, for these three data sets $M = 25$ did as well as any larger value.

In summary, the number of cases that must be reviewed in order to arrive at the performance ceiling of defect predictors is very small: as low as 50 randomly selected modules (25 defective and 25 non-defective).

3.4.1. Implications of Ceiling Effects

Considering the sampling experiments of Menzies et.al. and the outcomes of our small experiment in the previous section, we argue that data mining methods have hit a "performance ceiling"; i.e. some inherent upper bound on the amount of information offered by static code features when identifying modules which contain faults.

We have been motivated to find an explanation for this ceiling effect. We claim that static code features have "limited information content"; i.e. their information can be quickly and completely discovered by even simple learners. Specifically, we documented how *throwing training data away* does not affect the performance of defect predictors. The results are quite surprising. In experiments with different sampling strategies, we observe that *much of the training data can be discarded without losing effectiveness in defect prediction.* This leads to the following notion:

Hypothesis: Static code features have *limited information content.*

which is supported by these three observations:

- Observation 1: The information from static code features can be quickly and completely discovered by even simple learners.
- Observation 2: More complex learners do not find new information.
- Observation 3: Further progress in learning defect predictors does not come from better algorithms, but from improving the information content of the training data.

Different from previous line of research, our goal in this dissertation is to show that the performance of defect predictors can be increased by improving the information content in data by previously unexplored kinds of information sources, which do not require collection of new features:

- Explore the assumptions of data miners to check whether these assumptions are valid for software defect data.
- Increase the information content of data by combining project data from multiple companies. This combined data may yield more information content since they span a large class of software projects developed at different sites for a wide range of applications.
- Model the interactions between modules, rather than assuming their independence on defect proneness.

The second issue deserves more discussion due to its practical impacts. It suggests that defect predictor can be constructed locally *without* using local data. Furthermore, it does not require collection of additional features in order to increase the information content in data. In the next section, we discuss the merits of using within and cross company data.

3.5. Within- vs. Cross- Company Data

What we propose is to increase the information content in data by using cross company (CC) data for building within company (WC) defect predictors. Intuitively, it could be argued that there is no issue in comparing WC vs. CC studies, because of the common belief that local data is always better than imported data. Nevertheless, it is important to question the value of WC vs. CC studies.

Prior to this dissertation, there was no test of this intuition in the domain of *defect prediction*. However, in the domain of *effort estimation*, it turns out that this intuition has only mixed support:

- Mendes et al. [127] found within-company data performed much better than cross-company data for predicting estimation effort of web-based projects. They only recommend using cross-company data in the special case when that "data are obtained using rigorous quality control procedures".
- A similar conclusion was reached by Abrahamsson et al. who discussed learning effort predictors in the context of an agile development process [128]. They strongly advocate the use of WC-data.

However, other studies are not so clear in their conclusions:

- MacDonell & Shepperd tried to find trends in a set of papers relating to project management and effort estimation [129]. However, the papers studied by MacDonell & Shepperd used a wide range of data sets, so these authors found it hard to offer a definitive combined conclusion.
- In other work, after a review of numerous case studies, Kitchenham et al. [130] concluded that the value of CC vs. WC data for effort estimation is unclear:

  > some organizations would benefit from using models derived from cross-company benchmarking databases but others would not [130].

- Premraj and Zimmermann suspects that the reason for the contradictory results are due to heterogeneity in data. Therefore, they build business specific cost models to have homogeneity in data. They compare within company, cross company and business specific cost models and report that although cross company models perform slightly worse, neither model is significantly better than others [131].

One possible explanation for these contradictory results is that effort estimation requires the collection of project data, some of which has ambiguous definitions. For example, one of the features of the COCOMO-family [132] of effort predictors is "applications experience" (aexp). According to one on-line source[5] , this feature is defined as follows: "the project team's equivalent level of experience with *this type* of application". No guidance is offered regarding how to characterize "this type of application". Hence, there is some degree of ambiguity in this definition. We conjecture that the ambiguity of the effort estimation features is one reason for the variance in the results reported by MacDonell & Shepperd and Kitchenham et al. [129, 130].

Static code features that we use in this study, on the other hand, are not so ambiguous. Simple toolkits can be used to collect these features in a rapid, automatic, and uniform manner across multiple projects. Therefore, in theory, conclusions reached from these features should be less ambiguous than those reached from effort estimation features.

3.6. Research Questions

In this section we pose our above discussions in terms of research questions. We state six research questions, and in the rest of this dissertation, we will perform several experiments to look for empirical evidence for the answers to these questions.

3.6.1. How can we improve the information content of local data without introducing new information sources?

Assumptions of certain algorithms may prohibit the use of inherently available information in defect data. Specifically, we analyze the Naive Bayes' assumptions to

[5]http://sunset.usc.edu/research/COCOMOII/expert_cocomo/drivers.html

check whether they are valid for software data. The first assumption we analyze is the independence of features considering that many static code features are shown to be correlated. The other assumption under question is the equal importance of attributes, that is all static code features have the same impact on defectiveness.

3.6.2. How can companies construct local defect predictors with remote data?

Our goal is to identify the conditions under which cross-company data may be preferred to within-company for the purposes of learning defect predictors. In the related experiment, we compare the performance of defect predictors learned from WC data to those learned from CC data.

3.6.3. How can companies filter remote data for local tuning?

CC data includes information from many diverse projects and are heterogeneous compared to WC data. The goal of the related experiment is to select a subset of the available CC data that is similar to WC data (however, with more information content) and to investigate the effect of data homogeneity on the defect prediction performance. We apply a simple nearest neighbor (NN) filtering to CC data for constructing a locally tuned repository. We use the Euclidean distance between static code features of WC and CC data for measuring similarity and determining neighbors.

3.6.4. How much local data do organizations need for constructing a defect prediction model?

A common belief is that collecting local data that are required to build defect prediction models takes too much time and effort. Kitchenham *et al.* [130] argue that organizations use cross-company data since within-company data can be so hard to collect:

- The time required to collect enough data on past projects from within a company may be prohibitive.

- Collecting within-company data may take so long that technologies change and older projects do not represent current practice.

In the related experiment, we will check whether this belief is valid or not for defect data. We employ an incremental learning approach to WC data in order determine the number of samples in local repositories for building defect prediction models.

3.6.5. Can our results be generalized?

We initially use only NASA datasets in our CC and WC experiments to answer the above questions. In order to check the external validity of our results, this experiment will replicate all three experiments on data from a company that has no ties with NASA: specifically, a Turkish company writing software controllers for Turkish whitegoods.

3.6.6. How can we improve the information content in static code features with more local resources?

In order to answer this question, we propose to use an approach that is shown to be useful in other disciplines including search engines and reputation of multi-agent systems [133, 21]. We apply PageRank algorithm to module dependency graphs in order to model the inter-module structure of software systems in addition to their intra-module structure explained by static code features.

4. METHODOLOGY

This chapter explains in detail: the algorithms for constructing defect predictors, the data used in the experiments and the measures used to evaluate the performance of defect predictors.

4.1. Methods Used in the Study

This section explains predictor models used for defect prediction. The Naive Bayes classifier is taken as a baseline, since it is shown to acquire best results obtained so far [10]. We remove the assumptions of the Naive Bayes classifier one at a time and construct the linear, quadratic discriminants and the weighted Naive Bayes. The first assumption in Naive Bayes is that the features of data sample are independent, thus it employs the univariate normal distribution. We use a multivariate normal distribution to model the correlations among features. In the next section, univariate and multivariate normal distributions are briefly explained. We then use a weighting scheme for analyzing the second assumption (i.e. equal importance of attributes).

4.1.1. Univariate vs. Multivariate Normal Distribution

In univariate normal distribution, $x \sim N(\mu, \sigma^2)$, x is said to be normal distributed with mean μ and standard deviation σ with the probability distribution function is defined as:

$$p(x) = \frac{1}{\sqrt{(2\Pi)}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{4.1}$$

The term inside the exponentiation in Equation 4.1 is a distance function where the distance of a data sample x to the sample mean μ is measured in terms of standard deviations σ. This ensures to scale the distances of different features in the case that feature value cardinalities are different. This measure does not consider the correlations among features.

In the multivariate case, $\vec{x}$ is a d dimensional vector that is normal distributed, i.e. $\vec{x} \sim N_d(\vec{\mu}, \Sigma)$, and the pdf of a multivariate normal distribution is defined as:

$$p(x) = \frac{1}{(2\Pi)^{d/2}\Sigma^{1/2}} \exp\left(-\frac{1}{2}(\vec{x}-\vec{\mu})^T\Sigma^{-1}(\vec{x}-\vec{\mu})\right) \tag{4.2}$$

where Σ is the covaraince matrix and $\vec{\mu}$ is the mean vector. The term inside the exponentiation in Equation 4.2 is again a distance function and it is called the *Mahalanobis distance* [134]. In this case, the distance to the mean vector is normalized by the covariance matrix and the correlations of features are also considered. This results in less contribution of highly correlated features and features with high variance.

Since software data attributes are highly correlated, a multivariate model would be more appropriate than the univariate model. Besides multivariate normal distribution is analytically simple, tractable and robust to departures from normality [134]. However, using a multivarite model increases the number of parameters to estimate. In the univariate case only two parameters, μ and σ are estimated. But in the multivariate case, d parameters for $\vec{\mu}$ and $d \times d$ parameters for Σ need to be estimated.

4.1.2. Multivariate Classification

In software defect prediction, one aims to seperate classes C_0 and C_1 where samples in C_0 are non defective and samples in C_1 are defective. In this binary classification problem, it is sufficient to find one discriminant that seperates instances from the two distinct classes. Combining the multivariate normal distribution and the Bayes rule, then using different assumptions results in different discriminants with different complexity levels.

Bayes theorem states that the posterior distribution of a sample is proportional to the prior distribution and the likelihood of the given sample. More formally:

$$P(C_i|x) = \frac{P(x|C_i)P(C_i)}{P(x)} \tag{4.3}$$

Equation 4.3 is read as:

The probability of a given data instance x to belong to class C_i is equal to the multiplication of the likelihood that x is coming from the distribution that generates C_i and the probability of observing C_i's in the whole sample, normalized by the evidence.

Evidence is given by

$$P(x) = \sum_i P(x|C_i)P(C_i) \tag{4.4}$$

and it is a normalization constant for all classes, thus it can be safely discarded. Then Equation 4.3 becomes:

$$P(C_i|x) = P(x|C_i)P(C_i) \tag{4.5}$$

In a classification problem we compute the posterior probabilities $P(C_i|x)$ for each class and choose the one with the highest posterior. This is equaivalent to defining a discriminant function $g_i(x)$ for class C_i derived from Equation 4.5 by taking the logarithm for convenience.

$$g_i(x) = \log P(x|C_i) + \log P(C_i) \tag{4.6}$$

In order to achieve a discriminant value, one needs to compute the prior and likelihood terms. Prior proability $P(C_i)$ can be estimated from the sample by counting. The critical issue is to choose a suitable distribution for the likelihood term $P(x|C_i)$.

This is where the multivariate normal discussion takes place. In this study likelihood term is modelled by the multivariate normal distribution.

Computing discriminant values for each class and assigning the instance to the class with the highest value is equivalent to using Bayes Theorem for choosing the class with the highest posterior probability. For the binary classification case, it is sufficient to construct a single discriminant by $g(x) = g_0(x) - g_1(x)$. Using discriminant point of view, we will explain different classifiers (more complex to simpler ones) in the following sections.

4.1.3. Quadratic Discriminant

4.1.3.1. Assumptions.

1. Data sample is i.i.d. (independent and identically distributed)
2. Each class is formed by a single group, i.e. unimodal.
3. Each class has distinct Σ_i and $\vec{\mu_i}$

4.1.3.2. Derivation. Combining Equation 4.2 and 4.6 we get,

$$g_i(\vec{x}) = -\frac{d}{2}\log 2\pi - \frac{1}{2}\log|\Sigma_i| - \frac{1}{2}(\vec{x}-\vec{\mu_i})^T \vec{\Sigma_i}^{-1}(\vec{x}-\vec{\mu_i}) + \log P(C_i) \quad (4.7)$$

First term can be safely dropped because it is a constant term in all discriminants. Then Σ_i, μ_i and $P(C_i)$ are replaced with their maximum likelihood estimates S_i, m_i and $\hat{P}(C_i)$respectively.

$$g_i(\vec{x}) = -\frac{1}{2}\log|S_i| - \frac{1}{2}(\vec{x}-\vec{m_i})^T \vec{S_i}^{-1}(\vec{x}-\vec{m_i}) + \log\hat{P}(C_i) \quad (4.8)$$

Rearranging terms we obtain

$$g_i(\vec{x}) = -\frac{1}{2}\log|S_i| - \frac{1}{2}(\vec{x}^T S_i^{-1}\vec{x} - 2\vec{x}^T S_i^{-1}\vec{m_i} + \vec{m_i}^T S_i^{-1}\vec{m_i}) + \log\hat{P}(C_i) \quad (4.9)$$

and by defining new variables W_i, w_i and w_{i0}, the quadratic discriminant is obtained.

$$g_i(\vec{x}) = \vec{x^T} W_i \vec{x} + w_i^T \vec{x} \vec{+} w_{i0} \tag{4.10}$$

where

$$\begin{aligned} W_i &= -\frac{1}{2} S_i^{-1} && (4.11) \\ w_i &= S_i^{-1} \vec{m_i} && (4.12) \\ w_{i0} &= -\frac{1}{2} \vec{m_i^T} S_i^{-1} m_i - \frac{1}{2} \log |S_i| + \log \hat{P}(C_i) && (4.13) \end{aligned}$$

We classify an instance $\vec{x}$ as C_i such that $i = argmax_k(g_k(\vec{x}))$

Quadratic model considers the correlation of the features differently for each class. In case of K classes, the number of parameters to estimate is $K \times (d \times d)$ for covariance estimates and $(K \times d)$ for mean estimates. Also K prior probability estimations are needed.

4.1.4. Linear Discriminant

4.1.4.1. Assumptions.

1. Data sample is i.i.d.
2. Each class is formed by a single group, i.e. unimodal.
3. Each class has a common Σ and distinct $\vec{\mu_i}$

4.1.4.2. Derivation. Assumption 3 states that classes share a common covariance matrix. The estimator is calculated as

$$S = \sum_i \hat{P}(C_i) S_i \tag{4.14}$$

Placing this term in Equation 4.9 we get

$$g_i(\vec{x}) = -\frac{1}{2}\log|S| - \frac{1}{2}(\vec{x}^T S^{-1}\vec{x} - 2\vec{x}^T S^{-1}\vec{m}_i + \vec{m}_i{}^T S^{-1}\vec{m}_i) + \log\hat{P}(C_i) \tag{4.15}$$

Please note that now the first term of the equation and the first term in the parantheses become common in all discriminant and can be safely dropped and it reduces to

$$g_i(\vec{x}) = -\frac{1}{2}(-2\vec{x}^T S^{-1}\vec{m}_i + \vec{m}_i{}^T S^{-1}\vec{m}_i) + \log\hat{P}(C_i) \tag{4.16}$$

which is now a linear discriminant in the form of

$$g_i(\vec{x}) = \vec{w}_i{}^T\vec{x} + w_{i0} \tag{4.17}$$

where

$$\begin{aligned} w_i &= S_i^{-1}\vec{m}_i && (4.18)\\ w_{i0} &= -\frac{1}{2}\vec{m}_i{}^T S_i^{-1}\vec{m}_i + \log\hat{P}(C_i) && (4.19) \end{aligned}$$

We classify an instance $\vec{x}$ as C_i such that $i = argmax_k(g_k(\vec{x}))$

This model considers the correlation of the features but assumes the variances and correlation of features are the same for both classes. The number of parameters to estimate for covariance matrix is now independent of K. For covariance estimates $(d \times d)$, for mean estimates $(K \times d)$ and for priors K parameters should be estimated.

4.1.5. Naive Bayes

4.1.5.1. Assumptions.

1. Data sample is i.i.d.

2. Each class is formed by a single group, i.e. unimodal.
3. Each class has a common Σ with off-diagonal entries equal to 0, and distinct $\vec{\mu_i}$

4.1.5.2. Derivation. Assumption 3 states the independence of features by using a diagonal covariance matrix. Then the model reduces to a univariate model given in Equation 4.20.

$$g_i(x) = -\frac{1}{2}\sum_{j=1}^{d}\left(\frac{x_j^t - m_{ij}}{s_j}\right)^2 + \log(\hat{P}(C_i)) \tag{4.20}$$

This model does not take the correlation of the features into account and measures the deviation from the mean in terms of standart deviations. This results in the classes to be aligned to axes and variance in each axis may be different. For Naive Bayes, d covariance, $(K \times d)$ mean and K prior parameters should be estimated.

Bayesian classifiers offer a relationship between fragments of evidence E_i, a prior probability for a class $P(H)$, and a posteriori probability $P(H|E)$. Please note that the likelihood $P(H|E)$ is approximated by the product term due to the i.i.d. assumption:

$$P(H|E) = \left(\prod_i P(E_i|H)\right)\frac{P(H)}{P(E)} \tag{4.21}$$

For example, in our data sets, there are two hypotheses: modules are either defective or not; i.e. $H \in \{defective, nonDefective\}$. Also, if a particular module has $numberofSymbols = 27$ and $LOC = 40$ and was previously classified as "defective" then

$$\begin{aligned} E_1 &: numberOfSymbols = 27 \\ E_2 &: LOC = 40 \\ H &: defective \end{aligned}$$

When building defect predictors, the posterior probability of each class ("defective" or "defect-free") is calculated, given the features extracted from a module. So, if a data set has 100 modules and 25 of them are faulty, then:

$$P(defective) = 0.25$$

When testing new data, a module is assigned to the class with the higher probability, calculated from Equation 4.3.

For numeric features, a feature's mean μ and standard deviation σ is used in a Gaussian probability function [124]:

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

All the static code features of Table 4.3 are numeric and are highly skewed. Therefore, we replace all numerics with a "log-filter", i.e. N with $log(N)$. This spreads out skewed curves more evenly across the space from the minimum to maximum values (to avoid numerical errors with $ln(0)$, all numbers under 0.000001 are replaced with $ln(0.000001)$). This "spreading" can significantly improve the effectiveness of data mining, since the distribution of log-filtered feature values fits better to the normal distribution assumption [28].

4.1.6. Weighted Naive Bayes

Standard Naive Bayes derivation can be obtained by placing a special form of multivariate normal distribution, as the likelihood estimate in the Bayes theorem. By special form we mean that the off-diagonal elements of the covariance matrix estimate are assumed to be zero, i.e. the attributes are independent. In this case, the multivariate distribution can be written as the sum of univariate normal distributions of each attribute (See Equation 4.20). While assuming the independence of attributes, a weighting term can be introduced to reflect the relative importances of attributes. Then Weighted Naive Bayes can be written as in Equation 4.22 [135, 136].

Table 4.1. List of heuristics used in this study.

Heuristic	Equation
PCA	See text
IG	$IG(x,A) = Entropy(x) - \sum_{a \in A} \frac{\lvert x=a \rvert}{\lvert x \rvert} Entropy(x=a)$
GR	$GR(x,A) = \frac{IG(x,A)}{SplitInfo(x,A)}$, where $SplitInfo(x,A) = -\sum_{a \in A} \frac{\lvert x=a \rvert}{\lvert x \rvert} \log \frac{\lvert x=a \rvert}{\lvert x \rvert}$
KL	$D_{KL}(x,A) = \sum_{a \in A} p(x=a \mid pos) \log(x=a \mid neg)$
OR	$OR(x,A) = \log \sum_{a \in A} \frac{p(x=a \mid pos)(1-p(x=a \mid neg))}{(1-p(x=a \mid pos))p(x=a \mid neg)}$
LP	$LP(x,A) = \log \sum_{a \in A} \frac{p(x=a \mid pos)}{p(x=a \mid neg)}$
EP	$EP(x,A) = \exp\left(\sum_{a \in A} p(x=a \mid pos) - p(x=a \mid neg)\right)$
CE	$CE(x,A) = \sum_{a \in A} p(x=a \mid pos) \log p(x=a \mid neg)$

$$P(C_i|x) = -\frac{1}{2}\sum_{j=1}^{d} w_j \left(\frac{x_j^t - m_{ij}}{s_j}\right)^2 + \log(\hat{P}(C_i)) \tag{4.22}$$

Now that we have introduced another parameter, the weight term w_j, we should find a way of estimating it accurately. For this purpose we use eight heuristics, which are explained in the next section.

4.1.7. Attribute Weight Estimation

We use eight heuristics mostly derived from attribute ranking techniques in order to estimate weights for the static code features. In all heuristics we compute the rank values for the features and then derive weights by normalizing over the sum of all rank values (See Equation 4.23). Thus, all weights are scaled to lie in the [0, 1] interval. A complete list of heuristics used in this research is given in Table 4.1.

$$w_j = \frac{Rank(j)}{\sum_i Rank(i)} \tag{4.23}$$

First heuristics is based on the Principal Component Analysis (PCA), which projects the data points onto orthogonal principal axes such that the variance in each axis is maximized. We do not directly use PCA for dimensionality reduction. Rather, we claim that attributes with higher contributions for determining principal components should have higher weights in the prediction method. In our proposed heuristic, we use k eigenvalue and eigenvector pairs that correspond to the 95% of the proportion of the variance explained. Eigenvalues are written as $\lambda_1, \lambda_2, .., \lambda_k$ and eigenvectors are written as e_{id} where $i = 1..k, d = 1..D$ and D is the number of attributes. Then the weight of attribute d is estimated as a weighted sum of the corresponding eigenvector elements as given in Equation 4.24.

$$w_d = \frac{\sum \lambda_i e_{id}}{\sum \lambda_i} \tag{4.24}$$

Among proposed heuristics, GainRatio (GR) and InfoGain (IG) are mainly used in decision tree construction to determine the attributes that best splits the data [84]. Zhang and Sheng use the GainRatio heuristic for attribute weight assignment [135]. InfoGain is used in other studies for subset selection by ranking attributes [137, 28]. Our goal is to convert these ranking estimates into attribute weights. For this purpose we also evaluate OddsRatio (OR), LogProb (LP), ExpProb (EP), CrossEntropy (CE) and Kullback-Leibler (KL) Divergence.

In defect prediction context, these heuristics correspond to the following: Given an attribute A,

- KL measures the similarity between the distributions of defective and nondefective modules. The more different the distributions are, the higher weight attribute A has.
- OddsRatio measures whether defective modules are more likely to occur than the nondefective modules.

- LogProb is the logarithm of the ratio of probability of a module being defective over probability of a module being nondefective.
- ExpProb is the exponentiation of the difference of probability of a module being defective and probability of a module being nondefective.
- CrossEntropy is the average number of bits needed to differentiate between the defective and nondefective module distributions.

Assigning weights with these heuristics takes linear time. On the other hand, ranking the attributes with these methods and then searching for an optimal subset requires both an exhaustive search in the attribute space and the evaluation of performance with each candidate subset. We expect to observe that the attributes that are discarded by the subset selection methods would have relatively small weights than the selected attributes.

4.1.8. Call graph based ranking (CGBR) framework

For this framework, we are inspired from the web searching algorithms. For example, Google use the PageRank algorithm that is developed by Page and Brin [133]. The PageRank algorithm computes the most relevant results of a search by ranking web pages. We have adopted this ranking methodology to software modules.

In web page ranking, the PageRank theory holds that an imaginary surfer, who is randomly clicking on links, will continue surfing with a certain probability. This probability, at any step, that the person will continue is defined as a damping factor d. The damping factor is in the range $0 < d < 1$, and it is usually set to 0.85 [4, 6]. In our case, we do not necessarily take the damping factor as a constant, rather we dynamically calculate it for each project. The web surfer analogy corresponds to calling other software modules in our context. Therefore, we define the damping factor, d, of software with N modules, as the ratio of actual module calls to the all possible module calls.

In CGBR framework, the software modules are analogous to web pages, and call interactions are analogous to the hyperlinks between the web pages. We assign equal initial values (i.e. 1) to all modules and iteratively calculate module ranks. We name the module rank results as the Call Graph Based Ranking (CGBR) values. The formula for calculating CGBR values is given in Equation 4.1.8.

$$CGBR(A) = (1-d) + d * \sum_i \frac{CGBR(T_i)}{C(T_i)} \tag{4.25}$$

where $CGBR(A)$ is the call graph based rank of module A, $CGBR(T_i)$ is the call graph based rank of module T_i which calls for module A, $C(T_i)$ is the number of outbound calls of module T_i and d is the damping factor. Usually after 20 iterations the CGBR values converge [133, 39].

4.2. Data Description

Software engineering is a poor domain for obtaining datasets compared to other data mining domains. There are several reasons for this such as the confidentiality of information. Another reason is that software companies prefer allocating their resources on continuous development and, ironically, trying to locate and fix defects, rather than collecting and analyzing metrics. Even when researchers obtain necessary data, most of the time they cannot share the data publicly and therefore, these studies can not be replicated.

As the application of machine learning methods in software studies increased, efforts for forming a public data repository of software resulted in the PROMISE Software Engineering Repository [138]. NASA Metrics Data Program (MDP) is another repository that offers software data [36]. PROMISE contains a subset of data available at NASA. Repository datasets are mainly categorized as defect prediction, cost estimation, text mining applications. Most of the defect prediction data are supplied by NASA and includes McCabe, Halstead, line of code metrics which will be detailed in the following section.

As a contribution of this dissertation, we have collected similar datasets from

Turkish software industry, to make similar analysis and donated them to PROMISE repository for the use of other researchers.

The experiments of this dissertation use the projects of Table 4.2, which are downloaded from the PROMISE repository[6] . The static code features of our projects are shown in Table 4.3. These features are divided into lines of code features, Halstead features, and McCabe features. Shared features between projects are given in Table 4.4 and are to be used in our CC, WC experiments.

These projects are from software developed in different geographical locations across North America (NASA) and Turkey (SOFTLAB). Within a system, the sub-systems shared some common code base but did not pass personnel or code between sub-systems. While NASA and SOFTLAB are one single *source* of data, there are several projects within each source. For example, NASA is really an umbrella organization used to co-ordinate and fund a large and diverse set of projects [139]:

- The NASA data were collected from across the United States over a period of five years from numerous NASA contractors working at different geographical centers.
- These projects represent a wide array of projects, including satellite instrumentation, ground control systems and partial flight control modules (i.e. Attitude Control).
- The data sets also represent a wide range of code reuse: some of the projects are 100% new, and some are modifications to previously deployed code.

Nevertheless, using our connections with the Turkish software industry, we collected new data sets in the format of Table 4.3 from a Turkish white-goods manufacturer. The SOFTLAB datasets ({ar3, ar4, ar5}) in Table 4.2 are the controller software for:

- A washing machine;
- A dishwasher;

[6] `http://promisedata.org/repository`

- And a refrigerator.

We have used a tool, named Prest, in order to collect these metrics from industry. Prest[7] is an open source metric extraction and defect analysis tool, which is developed by SOFTLAB researchers in the context of this research. Prest was developed as a free alternative to commercial counterparts. Prest can extract static code features from C, C++, Java and Jsp files and generates corresponding call graphs. SOFTLAB data collected with Prest are manually matched with defect reports with the help of corresponding project developers and managers.

In summary, seven datasets are from NASA projects developed at different sites by different teams, hence we treat each of them as if they were from seven different companies. Remaining three datasets are from a Turkish company collected from software for domestic home appliances. Therefore, we use ten projects from eight different companies.

Table 4.2. Data from ten software projects.

			(# modules)				.
source	project	language	examples	features	loc	%defective	
NASA	pc1	C++	1,109	21	25,924	6	94.
NASA	kc1	C++	845	21	42,965	15	45.
NASA	kc2	C++	522	21	19,259	20	49.
NASA	cm1	C++	498	21	14,763	9	83.
NASA	kc3	JAVA	458	39	7749	9	38.
NASA	mw1	C++	403	37	8341	7	69.
SOFTLAB	ar4	C	107	29	9196	18	69.
SOFTLAB	ar3	C	63	29	5624	12	70.
NASA	mc2	C++	61	39	6134	32	29.
SOFTLAB	ar5	C	36	29	2732	22	22.
			4,102				

4.3. Performance Evaluation

We have used probability of detection (pd or recall) and probability of false alarm (pf) as the performance measures [28]. Formal definitions for these performance criteria are given in Equation 4.26 respectively and they are derived from the confusion matrix

[7]`http://svn.cmpe.boun.edu.tr/svn/softlab/prest/trunk/Executable/PrestTool.rar`

Table 4.3. Static code features available in Table 4.2 projects.

#	Feature	pc1	kc1	kc2	cm1	kc3	mw1	ar4	ar3	mc2	ar5
1	*branchcount*	X	X	X	X	X	X	X	X	X	X
2	*codeandcommentloc*	X	X	X	X	X	X	X	X	X	X
3	*commentloc*	X	X	X	X	X	X	X	X	X	X
4	*cyclomaticcomplexity*	X	X	X	X	X	X	X	X	X	X
5	*designcomplexity*	X	X	X	X	X	X	X	X	X	X
6	*halsteaddifficulty*	X	X	X	X	X	X	X	X	X	X
7	*halsteadeffort*	X	X	X	X	X	X	X	X	X	X
8	*halsteaderror*	X	X	X	X	X	X	X	X	X	X
9	*halsteadlength*	X	X	X	X	X	X	X	X	X	X
10	*halsteadtime*	X	X	X	X	X	X	X	X	X	X
11	*halsteadvolume*	X	X	X	X	X	X	X	X	X	X
12	*totaloperands*	X	X	X	X	X	X	X	X	X	X
13	*totaloperators*	X	X	X	X	X	X	X	X	X	X
14	*uniqueoperands*	X	X	X	X	X	X	X	X	X	X
15	*uniqueoperators*	X	X	X	X	X	X	X	X	X	X
16	*executableloc*	X	X	X	X	X	X	X	X	X	X
17	*totalloc*	X	X	X	X	X	X	X	X	X	X
18	*halsteadcontent*	X	X	X	X	X	X			X	
19	*essentialcomplexity*	X	X	X	X	X	X			X	
20	*halsteadvocabulary*	X	X	X	X			X	X		X
21	*blankloc*	X	X	X	X	X	X	X	X	X	X
22	*callpairs*					X	X	X	X	X	X
23	*conditioncount*					X	X	X	X	X	X
24	*cyclomaticdensity*					X	X	X	X	X	X
25	*decisioncount*					X	X	X	X	X	X
26	*decisiondensity*					X	X	X	X	X	X
27	*halsteadlevel*					X	X	X	X	X	X
28	*multipleconditioncount*					X	X	X	X	X	X
29	*designdensity*					X	X	X	X	X	X
30	*normcyclomaticcomplexity*					X	X	X	X	X	X
31	*formalparameters*					X	X	X	X	X	X
32	*modifiedconditioncount*					X	X			X	
33	*maintenanceseverity*					X	X			X	
34	*edgecount*					X	X			X	
35	*nodecount*					X	X			X	
36	*essentialdensity*					X	X			X	
37	*globaldatacomplexity*					X				X	
38	*globaldatadensity*					X				X	
39	*percentcomment*					X	X			X	
40	*numberoflines*					X	X			X	
	Total	21	21	21	21	39	37	29	29	39	29

Table 4.4. Static code features shared by NASA and SOFTLAB projects in Table 4.2 projects.

#	Feature	NASA shared	All Shared
1	*branchcount*	X	X
2	*codeandcommentloc*	X	X
3	*commentloc*	X	X
4	*cyclomaticcomplexity*	X	X
5	*designcomplexity*	X	X
6	*halsteaddifficulty*	X	X
7	*halsteadeffort*	X	X
8	*halsteaderror*	X	X
9	*halsteadlength*	X	X
10	*halsteadtime*	X	X
11	*halsteadvolume*	X	X
12	*totaloperands*	X	X
13	*totaloperators*	X	X
14	*uniqueoperands*	X	X
15	*uniqueoperators*	X	X
16	*executableloc*	X	X
17	*totalloc*	X	X
18	*halsteadcontent*	X	
19	*essentialcomplexity*	X	
	Total	19	17

Table 4.5. Confusion Matrix

	Estimated	
Real	Defective	Nondefective
Defective	A	C
Nondefective	B	D

given in Table 4.5. pd is a measure of accuracy for correctly detecting the defective modules. Therefore, higher pd's are desired. pf is a measure for false alarms and it is an error measure for incorrectly detecting the nondefective modules. pf is desired to have low values. Since we need to optimize two parameters, pd and pf, a third performance measure called balance is used to choose the optimal (pd, pf) pairs. Balance is defined as the normalized Euclidean distance from the desired point (0,1) to (pd, pf) in a ROC curve [28].

$$\begin{aligned} pd &= (A)/(A+C) \\ pf &= (B)/(B+D) \end{aligned} \tag{4.26}$$

Zhang and Zhang argue that using (pd,pf) performance mesures in imbalanced classification problems is not practical due to low precisions [140]. On the contrary, Menzies et.al argue that precision has an unstable nature and it can be misleading to determine the better predictor [28]. They also give examples of low precision predictors that are successfully used (i.e. using a web search engine, a user may find the relevant web page in the 3^{rd} page). Therefore, we do not use precision as a performance measure.

In addition, we would like to point out that balance performance measure should be used carefully for determining the best among a set of predictors. Since it is a distance measure, i.e. the distance of (pd,pf) to the optimal point (0,1), a specific balance value defines a quarter-circle on the ROC graph with radius of (1 - bal) and with the origin (0,1) (See Figure 4.1). So, predictors with different (pd,pf) values can have the same balance value. This does not necessarily show that all predictors with

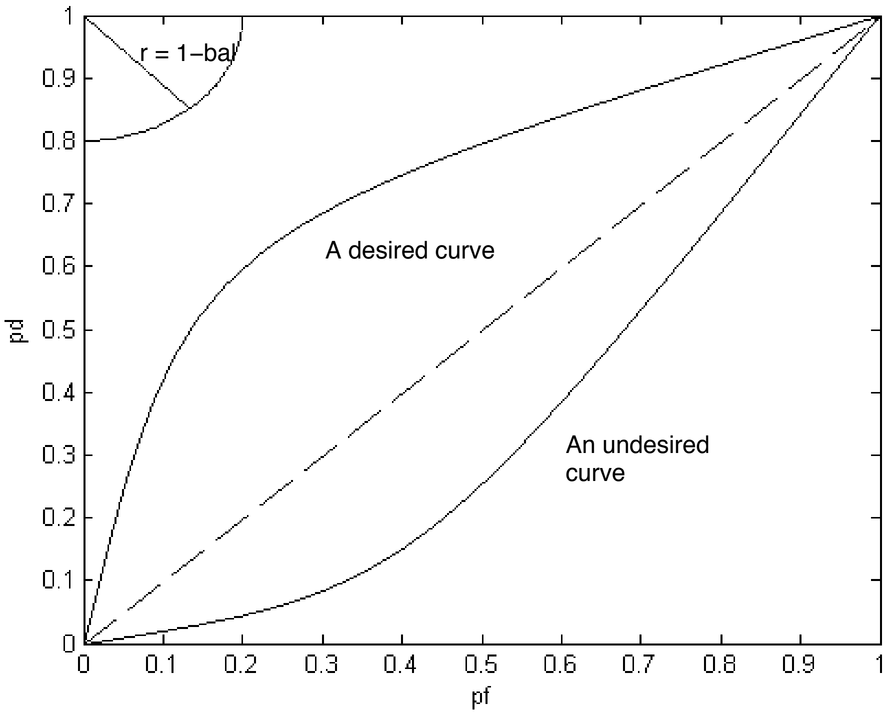


Figure 4.1. Balance defines a set of points in the ROC curve.

the same balance value have the same practical usage. As mentioned above, domain specific requirements may lead us to choose a predictor with a high pd rank although it may also have a high pf rank.

Our results are visualized using either *boxplots* or *quartile charts*. These two visualization methods summarize the results similarly and we use the more convenient one in reporting our results. To generate quartile charts, the performance measures for a population are sorted to show the median and the lower and upper quartile of numbers. The following example (from [28]):

$$\{\overbrace{4,7,15}^{q1},20,31,\overbrace{40}^{median},52,64,\overbrace{70,81,90}^{q4}\}$$

looks like as follows:

0% — ● | —— 100%

In quartile charts, the upper and lower quartiles are marked with black lines; the median is marked with a black dot; and vertical bars are added to mark the 50% percentile value.

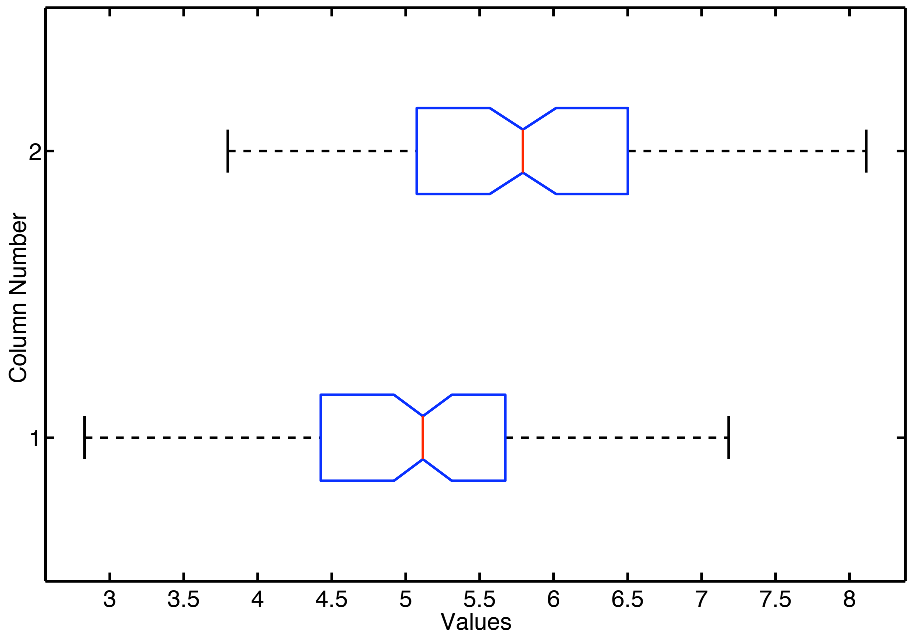


Figure 4.2. Sample boxplot.

Figure 4.2 shows boxplots for two populations with normal distribution parameters $\mu_1 = 5, \mu_2 = 6$ and $\sigma_1^2 = \sigma_2^2 = 1$. In Figure 4.2, the leftmost and the rightmost lines in the boxes correspond to the 25% and 75% quartiles and the line in the middle of the box is the median. The notches around the median correspond to the 95% confidence interval of the median. It is a sign of skewness if the medians are not centered between 25% and 75% quartiles. The dashed lines in the boxplots indicate 1.5 times of the interquartile range, i.e. the distance between the 25% and 75% quartiles. Data points outside these lines are considered as outliers.

We use *t-test* and *Mann-Whitney U test* [141] to test for statistical difference between results.

Mann-Whitney U (MWU) test assesses whether the distribution of two samples are the same. MWU sorts performance measure inside two populations to compare. These sorted samples are then ranked and pairwise compared using the number of wins

(W), losses (L) and ties(T). U statistics are calculated for both populations, using the W, L, T counts and the smaller one is used to test for significance. Such non-parametric tests are recommended in data mining since many of the performance distributions are non-Gaussian [142]. However, in order to compare our results with previous research, where t-tests are used, we also use t-tests for assessing significance. $t-test$s assume the Student's $t-distribution$ of populations and test their difference using the t statistic. Therefore, t-test is a parametric test whereas MWU is a non-parametric one. In all statistical significance tests, we use $\alpha = 0.05$.

5. EXPERIMENTS AND RESULTS

5.1. Experiment I: Analysis of Bayesian Assumptions

The amount of research for relaxing the assumptions of Naive Bayes has significantly increased in recent years. These research focus on modifications to break the conditional independence assumption and weighting attributes [143, 144, 135, 136]. Especially the weighting studies reported results that are generally 'not worse' than the standard Naive Bayes, while preserving the simplicity of the model.

This section attempts to tackle the assumptions of Naive Bayes in defect prediction context. First, we analyze the "independence of attributes" assumption. In order to overcome this, we incorporate multivariate approaches rather than univariate ones. Univariate approaches assume the independence of features whereas multivariate approaches take the relations between features into consideration. Obviously univariate models are simpler than multivariate models. While it is good practice to start modeling with simple models, the problem at hand should also be investigated by using more complex models. Then it should be validated by measuring performance whether using more complex models is worth the extra complexity introduced in the modeling. This section performs experiments with both simple and complex models and compares their performances.

We then analyze the other assumption of Naive Bayes, which is the "equal importance of attributes". We use an attribute weighting scheme to overcome this assumption. Attribute weighting has been explored to some extent for other problems such as software cost estimation. Auer et. al. employ attribute weigthing for analogy based cost estimation [145]. However, they assign *random* weights to project features and search for the optimal weigths. Similarly, neural network models for defect prediction have inherent attribute weighting. However, neural networks are non-deterministic and complex models that require optimization of the network structure together with many model parameters. Thus they require relatively large number of data samples for building a predictor. In practice, usually a limited amount of data is available. Further, the weights of the neural network model can not be easily interpreted especially in complex networks.

Considering defect prediction, we claim that all static code attributes do not have equal effect on defect prediction and they should be treated accordingly. Our goal is to develop a methodology that permits the use of static code attributes in terms of their relevance to defect prediction. Menzies et.al. state that [28]:

> how the attributes are used to build predictors is much more important than which particular attributes are used

We also focus on *how* rather than *which.* For this purpose, we propose attribute weighting along with several heuristics for determining the degree of importance of static code attributes.

We reproduce the experiments on NASA datasets with Naive Bayes by Menzies et.al in order to construct a baseline for comparison [28]. We also use multivariate methods and weighted Naive Bayes classifier in order to analyze the effect of the abovementioned assumptions.

5.1.1. Design

We have compared the state of the art classifier (log-filter, InfoGain attribute selection, standard Naive Bayes) with the Linear Discriminant, Quadratic Discriminant and weighted Naive Bayes classifiers constructed by our proposed heuristics on NASA datasets. We have also reproduced Menzies et.al.'s experiments on NASA datasets as benchmark results [28]. However, in [28], Menzies et.al. use a different version of projects measurements in their experiments than given in Table 4.2. This version of NASA projects include 38 features and can be accessed online from [36]. Furthermore, some projects used in that study turned out to have measurement errors and PROMISE community does not advise their use in further experiments. Therefore, only for this

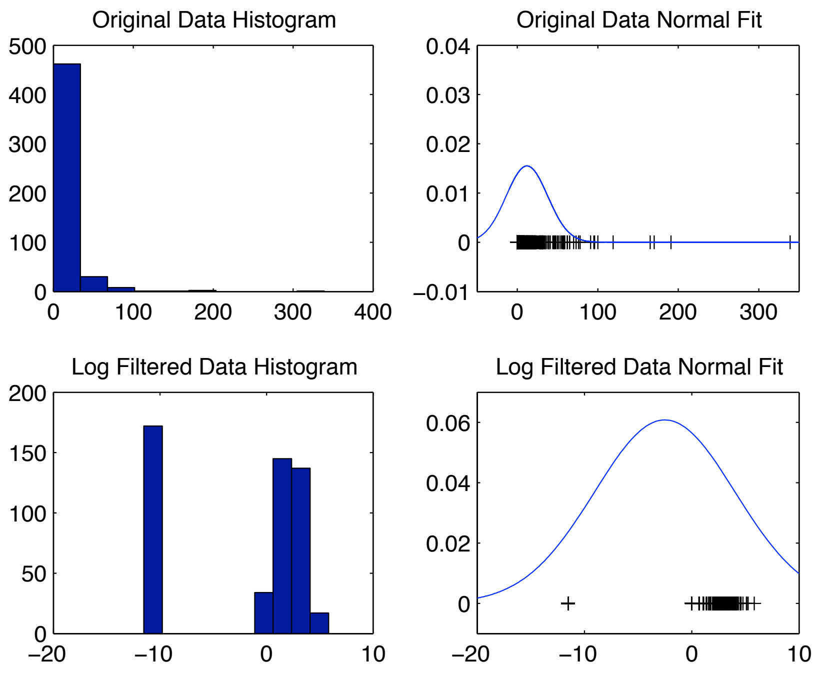


Figure 5.1. Effect of Log-filtering

experiment, we will use the same datasets in [28] to be able to make comparisons. The other experiments of this dissertation will use Table 4.2 projects.

The experimental design follows the framework suggested as a baseline by Menzies et.al. [28]. We have applied log-filtering on the datasets before we trained the predictor [28]. As Figure 5.1 shows, normal distribution fits better on the data points after log-filtering (this is also verified with goodness of fit tests). Since the Naive Bayes predictor assumes the existence of normal distribution, it is expected to see better results when log-filtering is employed. In [28], InfoGain is used for feature selection. Singularity issues occured in our preliminary LD and QD results due to low rank data matrices. Therefore, we use PCA for reducing the dimensionality and removing the co-linearities in data (i.e. 95% proportion of variance explained).

We have used 10-fold cross-validation in all experiments. That is, datasets are divided into 10 bins, 9 bins are used for training and 1 bin is used for testing. Repeating these 10 folds ensures that each bin is used for training and testing while minimizing the sampling bias. Each holdout experiment is also repeated 10 times and in each repetition the datasets are randomized to overcome any ordering effect and to achieve reliable statistics. Overall, we have performed 10x10=100 experiments per heuristic for each dataset and our reported results are the means of these 100 experiments for each dataset. We have applied t-test with $\alpha = 0.05$ in order to determine the statistical significance of mean results. We also include boxplots in order to visualize our results.

5.1.2. Results and Discussions

5.1.2.1. Independence of Attributes Assumption. Results on relaxing the independence assumption (i.e. using LD or QD rather than NB) are tabulated in Table 5.1. Mean results of (*pd*,*pf*) pairs selected by the *bal* measure after 10×10 holdout experiments are given. For 'Subset Selection' columns, the best subset of features obtained by InfoGain is used as reported in [28]. Each predictor's performance on each dataset is given and the best predictor according to the *bal* measure is written in bold face. Cells with a dash (i.e. '-') indicate no result due to singularities.

Subset selection is better in only one out of eight datasets (CM1). In the remaining datasets, best performances are obtained by pre-processing with PCA. As for the predictors, Naive Bayes (NB) wins five times, linear discriminant (LD) wins twice and quaratic disrciminant (QD) wins only once.

Overall performance of our approach numerically improves the best results reported so far. Menzies et al. reported mean((*pd*,*pf*)) = $(71, 25)$ which yields *bal* = 72 [28]. Our results give mean((*pd*,*pf*)) = $(76, 27)$ where *bal* = 74. For replicated experiments, we achieved mean((*pd*,*pf*)) = $(64, 19)$ and *bal* = 71, still below our performance.

Even though NB is the majority winner, it is observed that performances on some datasets are increased by using QD or LD. However, these improvements are not statistically significant. Therefore, we assert that there is no need to increase the complexity of the Naive Bayes by modeling correlations and the independence assumption is valid for software defect data, at least after PCA processing. Similarly, Domingos and Pazzani show theoretically that the independence assumption is a problem in a

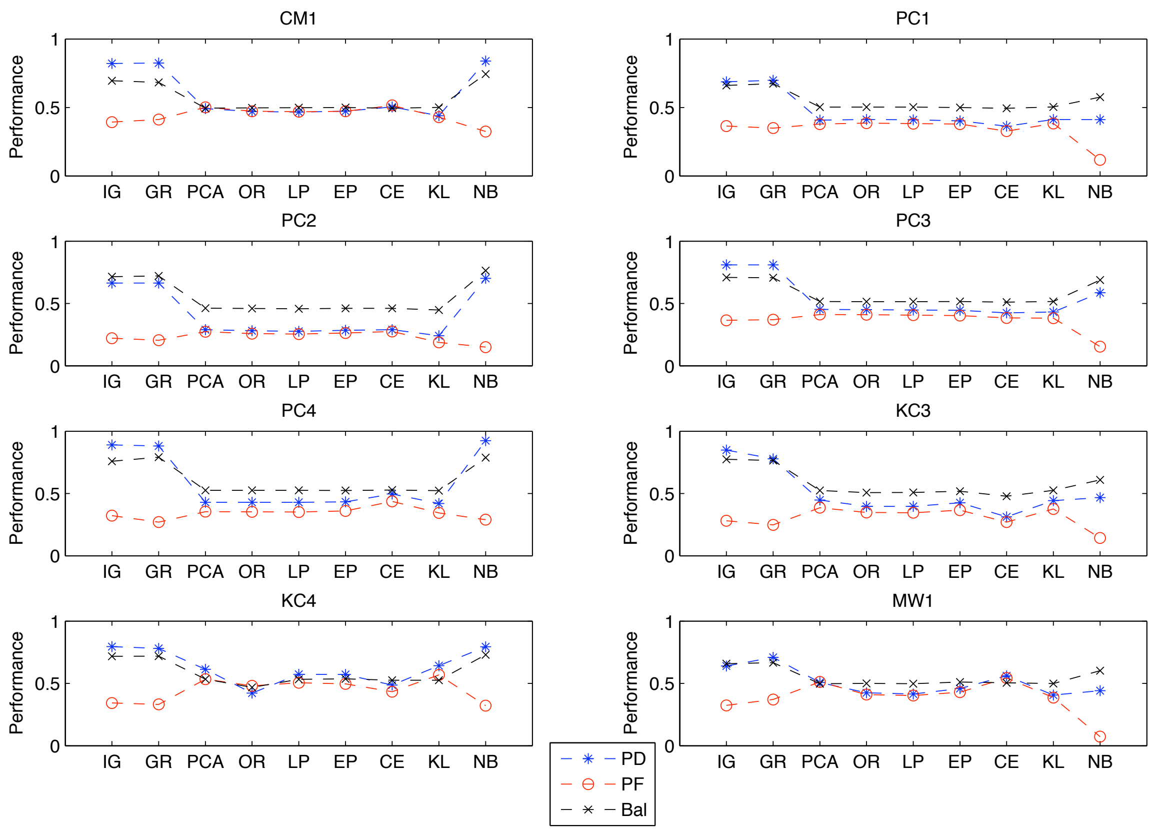


Figure 5.2. Experiment I results of feature weighting heuristics for Nasa projects

vanishingly small percent of cases [146]. This explains the repeated empirical result that, on average, seemingly naive Bayes classifiers perform as well as other seemingly more sophisticated schemes.

<u>5.1.2.2. Equal Importance of Attributes Assumption.</u> (pd, pf, bal) results of 100 experiments on NASA projects are plotted in Figure 5.2 for each heuristic. We observe that Infogain(IG), GainRatio(GR) heuristics and standard Naive Bayes with log-filtering(LNB) outperform other heuristics. These three methods show statistically significant performances than others in all datasets. Thus, we only tabulate these three methods' mean (pd,pf,bal) values in Table 5.2.

In NASA projects, overall evaluation yields five, six and four wins for InfoGain, GainRatio heuristics and LNB respetively after applying pairwise t-tests. These results indicate that our proposed approach yields comparable and in some cases better results than the ones reported on these datasets so far. InfoGain and GainRatio heuristics

Table 5.1. Experiment I results for independence assumption.

		PCA			Subset Selection		
Dataset	Discriminant	pd (%)	pf (%)	bal (%)	pd (%)	pf (%)	bal (%)
CM1	QD	76	35	70	91	51	64
CM1	LD	82	38	71	90	54	61
CM1	**NB**	82	37	71	84	32	**74**
PC1	QD	71	38	66	43	27	56
PC1	LD	66	24	70	-	-	-
PC1	**NB**	68	25	**71**	40	11	57
PC2	QD	78	32	72	68	14	75
PC2	LD	71	12	78	72	15	78
PC2	**NB**	72	13	**78**	72	15	78
PC3	QD	75	37	68	79	47	64
PC3	**LD**	76	31	**72**	-	-	-
PC3	NB	77	32	72	58	15	68
PC4	**QD**	88	20	**83**	98	31	78
PC4	LD	87	23	81	98	31	78
PC4	NB	87	24	81	91	29	79
KC3	QD	79	42	67	74	28	73
KC3	LD	79	25	77	53	19	64
KC3	**NB**	79	25	**77**	47	14	61
KC4	QD	75	29	73	69	38	66
KC4	LD	78	33	72	73	38	67
KC4	**NB**	80	32	**73**	80	32	73
MW1	QD	73	41	65	16	1	41
MW1	**LD**	70	34	**68**	-	-	-
MW1	NB	70	35	67	45	7	61
Best Avg:		*76*	*27*	**74**	*64* (71)	*19* (25)	71 (72)

Table 5.2. Experiment I results for Equal Importance Assumption

Data	IG+WNB (%)			GR+WNB (%)			LNB (%)		
	pd	pf	bal	pd	pf	bal	pd	pf	bal
CM1	82	39	69	82	41	68	83	32	74
PC1	69	36	66	69	35	67	41	12	57
PC2	66	22	72	66	20	72	70	15	76
PC3	81	37	71	81	37	71	59	15	69
PC4	89	32	76	88	27	79	92	29	78
KC3	85	28	77	78	25	76	47	14	61
KC4	80	34	72	78	33	72	79	32	73
MW1	64	32	66	71	37	67	44	07	60
Avg:	77	33	71	77	32	72	64	20	69

achieve higher pd and pf values compared to LNB. We argue that the projects that require high reliability should have higher pd values. Since these datasets have this requirement, InfoGain and GainRatio based heuristics may be preferred over LNB.

Figure 5.3 to Figure 5.10 show the boxplots of 100 balance results for IG, GR and LNB. We observe that the weighting results are more stable than standard Naive Bayes, since in all datasets, the spread of balance values are less than or equal to

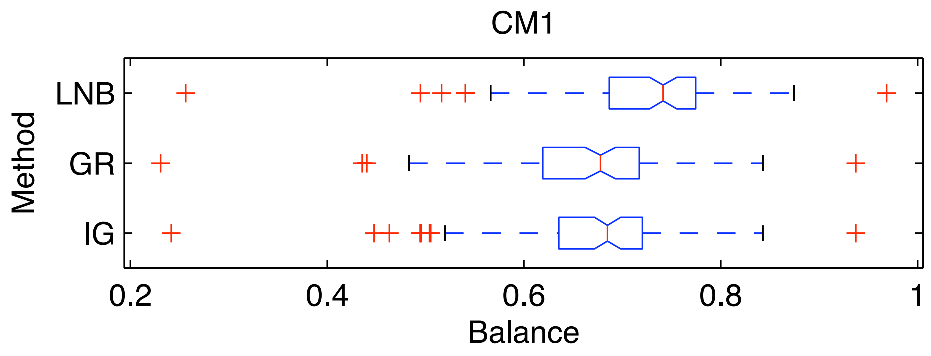


Figure 5.3. Experiment I boxplots for CM1 dataset.

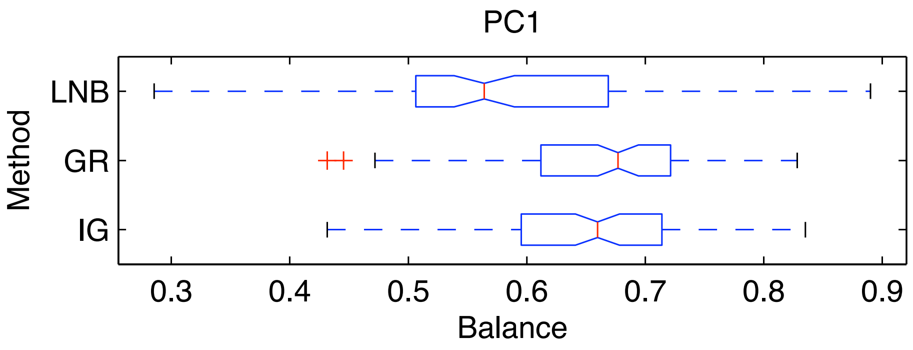


Figure 5.4. Experiment I boxplots for PC1 dataset.

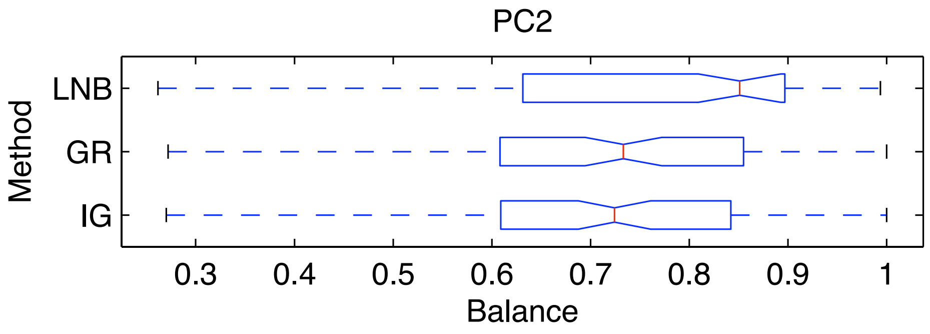


Figure 5.5. Experiment I boxplots for PC2 dataset.

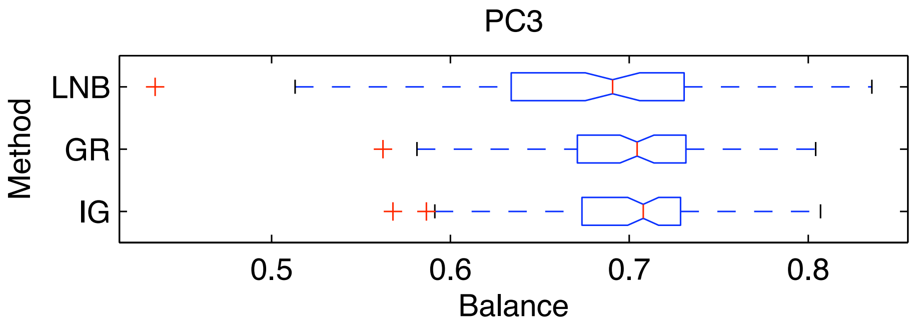


Figure 5.6. Experiment I boxplots for PC3 dataset.

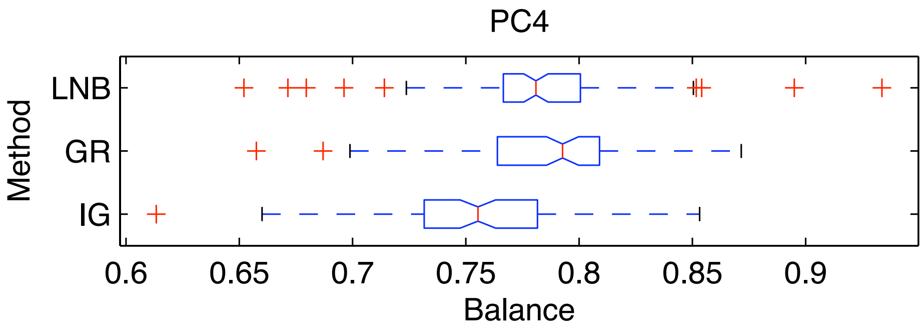


Figure 5.7. Experiment I boxplots for PC4 dataset.

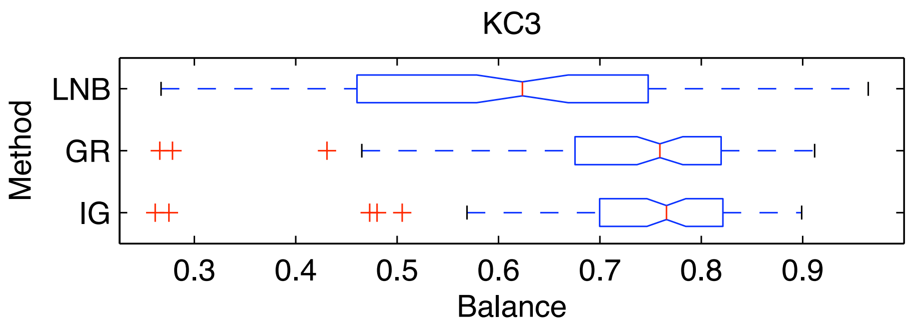


Figure 5.8. Experiment I boxplots for KC3 dataset.

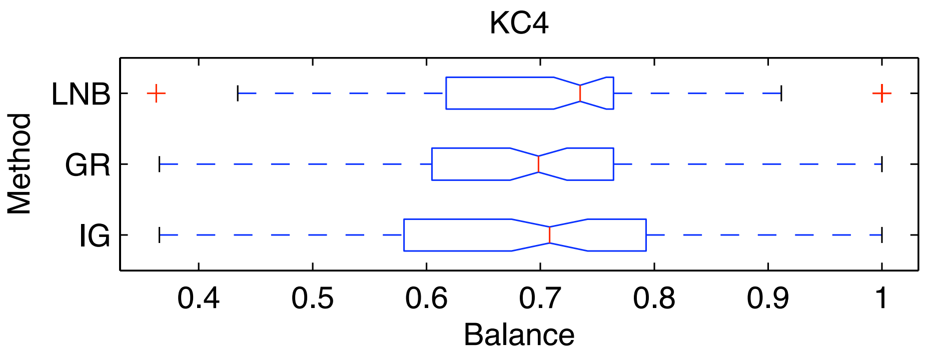


Figure 5.9. Experiment I boxplots for KC4 dataset.

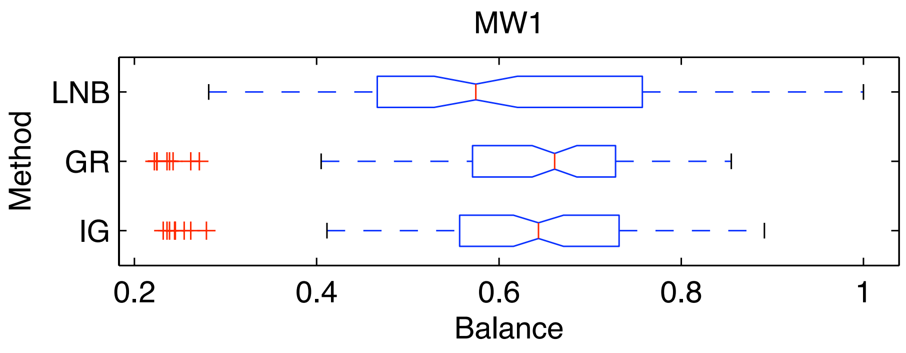


Figure 5.10. Experiment I boxplots for MW1 dataset.

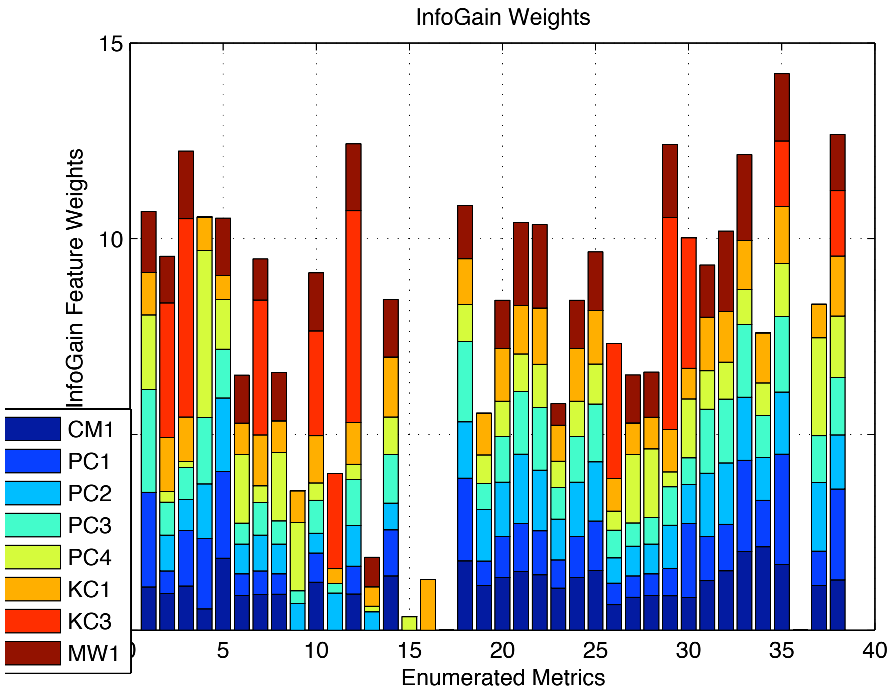


Figure 5.11. Experiment I InfoGain Weights.

that of Naive Bayes. Visually inspecting the quartile charts, we also observe that the statistical significance of the means also applies for the medians of the three methods in most cases.

In Figure 5.11 and Figure 5.12, we have plotted the relative weights of 38 metrics available in the NASA projects. These figures show the cumulative metric weight sums over eight projects. Figure 5.11 shows these values for InfoGain based heuristic and Figure 5.12 plots values for GainRatio based heuristic. Examining these figures, we see that metrics enumerated with 17 and 36 are never used. These metrics are 'Global Data Density' and 'Pathological Complexity'. An analysis of datasets shows that these metrics have a unique value for all modules in all datasets. Thus, they do not have any discriminative power and they are eliminated by the weighting approach. Also metrics enumerated with 15 and 16 ('Parameter Count' and 'Global Data Complexity') are used only in PC4 and KC3 where similar observations are valid. The general trend of weight assignment by both heuristics is similar. Metrics enumerated by 3, 12, 29, 33, 35, 38 in Figure 5.11 and Figure 5.12, which are 'Call Pairs', 'Edge Count', 'Node Count', 'Number of Unique Operands', 'Total Number of Lines' and 'Total Number of Line of

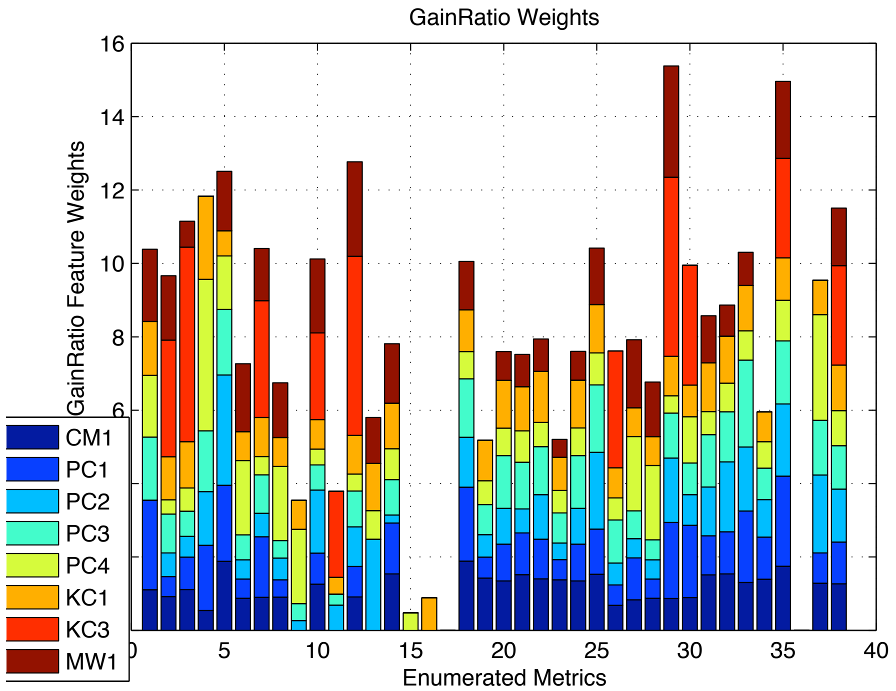


Figure 5.12. Experiment I GainRatio Weights.

Code' respectively, are consistently selected by these heuristics. These attributes are consistent with the set of attributes that Menzies et. al. reported for subsetting [28]. These validate our expectation of observing relatively small weights for attributes that are discarded by subsetting.

In summary, our results indicate that assigning weights to static code attributes may increase the prediction performance significantly, while removing the need for feature subset selection. On the other hand, we observe that the independence assumption of Naive Bayes is valid for software defect data, at least after PCA processing. Therefore, we conclude that using weighted Naive Bayes may produce better results to locate defects.

5.2. Experiment II: Are the defect predictors learned from CC data beneficial for organizations?

Several research in defect prediction focus on building models with available local data (i.e. within company predictors). To employ these models, a company should have a data repository where project metrics and defect information from past projects are stored. However, few companies apply this practice. We suspect that a common reason for not using defect predictors in practice is the lack of local data repositories. Constructing such a repository requires keeping track of project metrics together with related defect information. In the absence of certain infrastructure like automated tools to collect these metrics, manual effort is unavoidable to accomplish this process. Our view is that managers do not take the effort to collect metrics for process improvement due to tight schedule and budget constrains.

On the other hand, there are public data repositories including projects from companies such as NASA [138]. In this context, we investigate whether these public project data can be helpful for other companies for building localized defects predictors, especially for those with limited or no defect data repository.

While there exists numerous research in learning predictors using within company data [69, 29, 30, 31, 32, 28, 98, 99, 147, 148, 149, 2, 150, 97, 96, 114, 94, 93, 92, 91, 28, 90, 42], utilizing cross-company data has not been investigated in defect prediction research. On the other hand, this issue has been throughly analyzed in a closely related area, that is, cost estimation. In their systematic review, Kitchenham et.al. argue that it is not possible to reach a conclusion whether to use within or cross company data for cost estimation purposes [130].

In this dissertation, we turn the attention to *defect prediction* and perform experiments to check if we can reach a conclusion in favor of either CC or WC data. Specifically, we assess the relative merits of cross-company (CC) vs. within-company (WC) data for defect prediction. To the best of our knowledge, this kind of analysis is a novel one in defect prediction literature.

5.2.1. Design

Our first WC-vs-CC experiments repeated the procedure explained in Table 5.3, for all seven NASA projects of Table 4.2. For each project, test sets were built from

10% of the data, selected at random. Defect predictors were then learned from:

- CC data: all data from the other six projects.
- WC data: remaining 90% data of that project;

Most of the Table 4.2 data come from systems written in "C/C++" but at least one of the systems was written in JAVA. For cross-company data, an industrial practitioner may not have access to detailed meta-knowledge (e.g. whether it was developed in "C" or JAVA). They may only be aware that data, from an unknown source, are available for download from a certain url. To replicate that scenario, we will make no use of our meta-knowledge about Table 4.2.

In order to control for *order effects* (where the learned theory is unduly affected by the order of the examples) our procedure was repeated 20 times, randomizing the order of data in each project each time. In all, we ran 280 experiments to compare WC-vs-CC:

$$(2\text{ experiments}) * (20\text{ randomized orderings}) * (7\text{ projects})$$

The project data come from different sources and, hence, have different features. For this experiment, only the features that are common in all NASA projects are used, a total of 19 features. These features are marked in "NASA Shared" column of Table 4.3.

5.2.2. Results and Discussions

Table 5.4 shows the $\{pd, pf\}$ quartile charts for CC vs. WC data averaged over seven NASA projects. The pattern is very clear: CC data dramatically increases *both* the probability of detection and the probability of false alarms. The *pd* results are particularly striking.

For cross-company data:

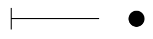
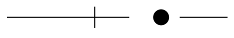
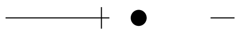
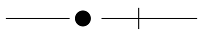
Table 5.3. Experiment II Design

```
DATA = [PC1, KC1, KC2, CM1, KC3, MW1, MC2]
LEARNER = [Naive Bayes]

C_FEATURES <- Find common features IN DATA
FOR EACH data IN DATA
        data = Select C_FEATURES in data
END
REPEAT 20 TIMES
        FOR EACH data in DATA
                CC_TRAIN = DATA - data
                WC_TRAIN = random 90% of data
                TEST = data - WC_TRAIN

                CC_PREDICTOR = Train LEARNER with CC_TRAIN
                WC_PREDICTOR = Train LEARNER with WC_TRAIN
                [cc_pd, cc_pf, cc_bal] = CC_PREDICTOR on TEST
                [wc_pd, wc_pf, wc_bal] = WC_PREDICTOR on TEST
        END
END
```

Table 5.4. Experiment II results averaged over seven NASA projects.

	treatment	min	Q1	median	Q3	max	
pd	CC	50	83	97	100	100	
	WC	17	63	75	82	100	
pf	CC	14	53	64	91	100	
	WC	0	24	29	36	73	

- 50% of the pd values are at or above 97%
- 75% of the pd values are at or above 83%;
- And all the pd values are at or over 50%.

To the best of our knowledge, Table 5.4 are the largest *pd* values ever reported from these data. However, these very high *pd* values come at some considerable cost. In Table 5.4 the median false alarm rate has increased from 29% (with WC) to 64% (with CC) and the maximum pf rate reaches 100%. We should caution that a 100% pf rate means that all defect-free modules are classified as defective, which yields inspection of

Table 5.5. Summary of U-test results (95% confidence): moving from WC to CC.

group	pd $WC \rightarrow CC$	pf $WC \rightarrow CC$	tables	$\|tables\|$
a	increased	increased	CM1 KC1 KC2 MC2 MW1 PC1	6
b	same	same	KC3	1

all these modules unnecessarily and contradicts with the purpose of defect prediction. However, it is not right to assess the general behavior of the CC defect predictors with such an extreme case. We mention this issue in order to clarify that high false alarm rates may be prohibitive in the practical application of defect predictors.

We explain these increases in pd, pf with the *extraneous factors* in CC data. More specifically, using a large training set (e.g. seven projects in Table 4.2) informs not only all the causes of errors, but also of numerous irrelevancies (e.g. using code features gathered from JAVA programs for predicting defects in "C" programs, using code features gathered from different companies and different project domains). Hence, large training sets increase the probability of detection (since there are more known sources of errors) as well as the probability of false alarms (since there are more *extraneous factors* introduced to the analysis). We will test the validity of this claim in the next experiment.

Once a *general result* is defined (e.g. CC data dramatically increases both pf and pd), it is good practice to check for specific exceptions to that pattern. Table 5.5 shows a summary of results when U tests with $\alpha = 0.05$ were applied to test results from each of the seven projects, *in isolation* and Table 5.6 to Table 5.12 show the $\{pd, pf\}$ quartile charts for Experiment #1 for each NASA project:

- Usually ($\frac{6}{7}$), the general pattern (i.e. both (pd, pf) increases in CC models compared to WC models) still holds (see group a).

Table 5.6. Experiment II results for CM1.

Table ($cm1$)

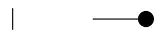
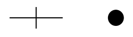
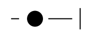
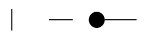
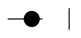
	treatment	min	Q1	median	Q3	max	
pd	CC	80	100	100	100	100	
	WC	40	60	80	100	100	
pf	CC	87	91	96	96	98	
	WC	24	27	33	38	47	

Table 5.7. Experiment II results for KC1.

Table ($kc1$)

	treatment	min	Q1	median	Q3	max	
pd	CC	82	88	94	94	100	
	WC	64	73	82	85	97	
pf	CC	47	49	51	53	57	
	WC	27	34	36	38	40	

- In one case (see group b), there was no difference in the results of the CC-WC data

Overall, the *general pattern* holds in the majority of cases (i.e. $\frac{6}{7}$), which is not a 100% internally consistent, however still a very clear effect.

When practitioners use defect predictors with high false alarm rates (e.g.the 64% reported above), they must allocate a large portion of their debugging budget to the unfruitful exploration of erroneous alarms. Defect predictors with high false alarms can be useful in industrial contexts:

- *When the cost of missing the target is prohibitively expensive.* In mission critical or security applications, the goal of 100% detection may be demanded in all situations, regardless of the cost of chasing false alarms.
- *When only a small fraction the data is returned.* Hayes, Dekhtyar, & Sundaram

Table 5.8. Experiment II results for KC2.

Table ($kc2$)

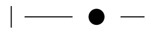
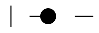
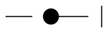
treatment		min	Q1	median	Q3	max	
pd	CC	82	91	91	100	100	
	WC	55	73	82	91	100	
pf	CC	57	62	64	74	81	
	WC	14	24	31	33	45	

Table 5.9. Experiment II results for KC3.

Table ($kc3$)

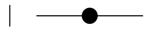
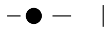
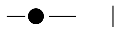
treatment		min	Q1	median	Q3	max	
pd	CC	60	80	80	80	100	
	WC	40	60	80	80	100	
pf	CC	14	19	24	31	38	
	WC	10	17	21	26	36	

Table 5.10. Experiment II results for MC2.

Table ($mc2$)

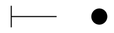
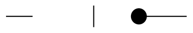
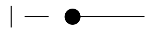
treatment		min	Q1	median	Q3	max	
pd	CC	50	67	83	100	100	
	WC	17	33	67	67	83	
pf	CC	55	64	73	73	100	
	WC	0	27	36	45	73	

Table 5.11. Experiment II results for MW1.

Table (*mw*1)

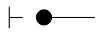
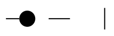
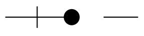
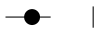
treatment		min	Q1	median	Q3	max	
pd	CC	75	75	100	100	100	
	WC	25	50	75	75	100	
pf	CC	50	55	63	66	82	
	WC	13	18	21	29	37	

Table 5.12. Experiment II results for PC1.

Table (*pc*1)

treatment		min	Q1	median	Q3	max	
pd	CC	88	100	100	100	100	
	WC	38	63	63	75	88	
pf	CC	89	92	93	95	99	
	WC	17	25	27	30	34	

call this fraction *selectivity* and offer an extensive discussion of the merits of this measure [151].

- *When there is little or no cost in checking false alarms.*

Nevertheless, we suspect that most sites do *not* accept false alarm rates as high as 64%. Therefore, the conditions under which the benefits of CC data (high probabilities of detection) outweigh their high costs (high false alarm rates) are quite rare. In summary, for most software applications, very high pf rates like the CC results of Table 5.4 make the predictors impractical to use.

5.3. Experiment III: How can companies filter CC data for local tuning?

The results of Experiment II limits the use of CC data to a limited domain (i.e. mission critical) and may look discouraging at first glance. In that experiment we explained our observations with the *extraneous factors*, that the results are affected by the irrelevant factors in CC data. In this section we hypothesize this claim and test for its validity.

5.3.1. Design

In this experiment, we try to construct more homogeneous defect datasets from CC data. For this purpose we use a simple filtering method (i.e. nearest neighbor (NN)). Our idea behind sampling is to collect similar instances together in order to construct a learning set that is homogeneous with the validation set. More formally, we try to introduce a bias in the training data that reflects the validation set characteristics. We simply use the k-Nearest Neighbor (k-NN) method to measure the similarity between the validation set and the training candidates. The similarity measure is the Euclidean distance between the static code features of validation and training candidate sets. The expected outcome of the sampling part is to obtain a subset of available CC data that shows similar characteristics to the local code culture.

We do not use the class information (i.e. a module is defective or defect-free) while measuring similarity and selecting neighbors. This corresponds to a real life case, where the development of some modules are completed and they are ready for testing: there is no defect related information available, however static code features are collected with automated tools.

The experiment design is given in Table 5.13. We calculate the pairwise Euclidean distances between the validation set and the *candidate* training set (i.e. all CC data). Let N be the number of validation set size. For each validation data, we pick its $k = 10$ nearest neighbors from candidate training set. Then we come up with a total of $10 \times N$ similar data points (i.e. module features). These $10 \times N$ samples may not be unique (i.e. a single data sample can be a nearest neighbor of many data samples in the validation set). Using only unique ones, we form the training set and use a random 90% of it for training a predictor. We repeat the last step 100 times.

5.3.2. Results and Discussions

If the high false alarm rates of Experiment II are due to the extraneous factors in CC data (as we suspect), then we would expect lower pf's in NN results than CC results. Therefore, we define the null hypothesis as:

$$H_0 : NN_{pf} \geq CC_{pf}$$

H_0 is rejected by the U test with $\alpha = 0.05$. Therefore, using NN filtered CC data significantly decreases the false alarms compared to CC data. Yet, we observe that *pd*s have also decreased. However, false alarm rates are more dramatically decreased than detection rates as seen in Table 5.14 and Table 5.15. For example, in $CM1$ project, the median false alarm rate decreases nearly one thirds, from 91% to 33%, whereas the median detection rate slightly decreases from 94% to 82%. In all cases, NN dramatically reduces the high false alarm rates associated with the use of cross-company data. Often that reduction halves the false alarm range. For example, in MW1, the median false alarm rate drops between CC to NN from 68% to 33%.

Table 5.13. Experiment III Design

```
DATA = [PC1, KC1, KC2, CM1, KC3, MW1, MC2]
LEARNER = [Naive Bayes]

C_FEATURES <- Find common features IN DATA
FOR EACH data IN DATA
        data = Select C_FEATURES in data
END
REPEAT 100 TIMES
        FOR EACH data in DATA
                WC_TRAIN = random 90% of data
                TEST = data - WC_TRAIN
                CC_TRAIN = DATA - data

                //NN-Filtering
                FOR EACH test IN TEST
                        dist = L2_DISTANCE{test, CC_TRAIN}
                        NNCC_TRAIN <- 10 Samples in CC_TRAIN with min{dist)
                END
                NNCC_TRAIN = UNIQUE(NNCC_TRAIN)

                NNCC_PREDICTOR = Train LEARNER with NNCC_TRAIN
                CC_PREDICTOR = Train LEARNER with CC_TRAIN
                WC_PREDICTOR = Train LEARNER with WC_TRAIN
                [nncc_pd, nncc_pf, nncc_bal] = NNCC_PREDICTOR on TEST
                [cc_pd, cc_pf, cc_bal] = CC_PREDICTOR on TEST
                [wc_pd, wc_pf, wc_bal] = WC_PREDICTOR on TEST
        END
END
```

Table 5.14. Experiment III pd results where $NN_{pd} \geq WC_{pd}$.

	rank		0	25	50	75	100
CM1	1	CC	98	98	98	98	98
	2	NN	76	82	82	84	84
	3	WC	20	60	80	80	100
MW1	1	CC	90	90	90	90	90
	2	NN	68	68	68	68	71
	2	WC	0	50	50	75	100
PC1	1	CC	99	99	99	99	99
	2	NN	74	77	77	77	78
	3	WC	38	62	62	75	100

Columns 0–100: quartiles

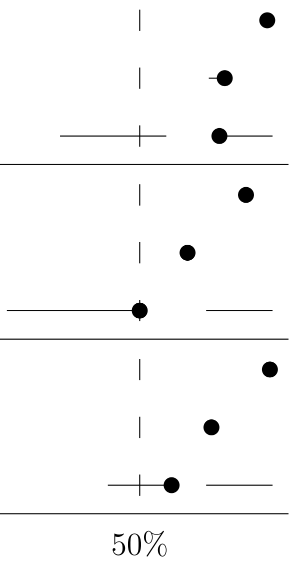


Table 5.15. Experiment III pd results where $NN_{pd} < WC_{pd}$.

	rank		0	25	50	75	100
KC1	1	CC	94	94	94	94	94
	2	WC	64	76	82	85	94
	3	NN	60	64	65	66	69
KC2	1	CC	94	94	94	94	94
	2	WC	45	73	82	91	100
	2	NN	77	78	79	79	80
KC3	1	CC	81	81	81	84	84
	2	WC	20	60	80	100	100
	3	NN	60	63	65	67	70
MC2	1	CC	83	83	83	83	85
	2	WC	17	50	67	83	100
	3	NN	56	56	56	56	58

Columns 0–100: quartiles

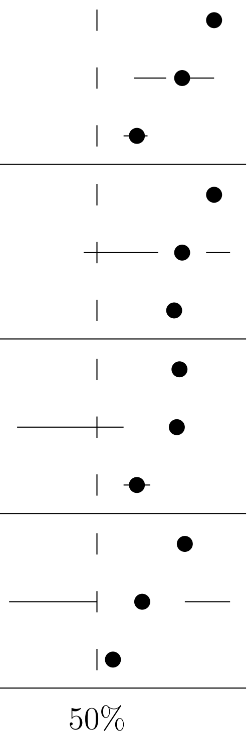


Table 5.16. Experiments III pf results where $NN_{pf} \leq WC_{pf}$.

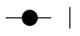
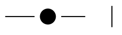
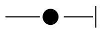
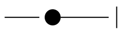
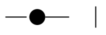
	rank		quartiles					
			0	25	50	75	100	
KC1	1	NN	22	23	24	25	27	
	2	WC	26	32	35	37	43	
	3	CC	59	60	60	60	60	
KC2	1	NN	24	25	25	25	27	
	1	WC	10	21	26	31	40	
	2	CC	67	67	67	67	67	
KC3	1	NN	17	18	18	19	20	
	2	WC	7	17	21	26	31	
	3	CC	26	27	27	27	27	
MC2	1	NN	29	30	31	32	35	
	2	WC	0	27	36	45	73	
	3	CC	71	71	71	71	71	
								50%

Table 5.17. Experiment III pf results where $NN_{pf} > WC_{pf}$.

	rank		quartiles					
			0	25	50	75	100	
CM1	1	WC	16	29	33	38	49	
	2	NN	40	43	44	45	46	
	3	CC	90	91	91	91	93	
MW1	1	WC	8	21	26	29	47	
	2	NN	30	32	33	33	36	
	3	CC	67	68	68	69	70	
PC1	1	WC	16	24	28	31	40	
	2	NN	45	48	48	49	53	
	3	CC	94	94	94	94	94	
								50%

Showing that NN is significantly better than CC, the ideal result would be that NN can be used as an alternative to local WC data. This, in turn, would mean that developers could avoid the tedious and expensive work of local data collection. We now investigate the relation between NN and WC.

If NN outperformed WC then two observations would appear:

- *Observation1:* NN would have *pd* values above or equal to WC's *pd*. The examples displaying $Observation1$ are shown in Table 5.14.
- *Observation2:* NN would have pf values below or equal to WC's pf. The examples displaying $Observation2$ are shown in Table 5.16.

Our results suggest that $Observation1$ and $Observation2$ are somewhat mutually exclusive: As shown in Figures Table 5.14 and Table 5.17, the examples where NN increases the probability of detection are also those where it increases the probability of false alarms. Hence, we cannot recommend NN as a replacement for WC. Nevertheless, if local WC data are unavailable, then we would recommend processing foreign CC data with NN.

In Experiment II, we have used random samples of CC data and observed that the false alarm rates substantially increased compared to the WC models. Our new experiment shows that NN filtering CC data removes the increased false alarm rates. Now we argue that using NN filtering instead of using all available CC data helps choosing training examples that are similar to problem at hand. Thus, the irrelevant information in non-similar examples are avoided. However, this also removes the rich sample base and yields a slight decrease in detection rates. Mann-Whitney tests reveal that NN filtering is significantly better than random sampling CC data.

The NN vs. WC results do not give necessary empirical evidence to make a strong conclusion. Sometimes NN may perform better than WC. A possible reason may be hidden in the processes that implemented those projects. Maybe, a group of new developers were working together for the first time and corresponding WC data included more heterogeneity, which is reduced by NN. May be the development methodology changed during the project, producing a different code culture. However, we do not have access to the internals of these projects that allows a discussion of these observations.

Nevertheless, NN filtering picks training examples that are similar to the examples about which we want to make predictions. Without introducing irrelevant information, NN filtering populates a set that has the same characteristics with the problem. That means, the likelihood of implementing similar software modules and having similar defects in these modules is high. Thus, NN filtering simulates that the selected cross company data are coming from within the company. Considering the *extraneous factors*, NN filtering introduces homogeneity whereas random sampling introduces heterogeneity to cross company data.

Combining these results of Experiment II and III, if a company lacks local data, we would suggest a two-phase approach. In phase one, that organization uses imported CC data filtered via NN. Also, during phase one, the organization should start a data collection program to collect static code features. Phase two commences when there is *enough* local WC data to learn defect predictors. During phase two, the organization would switch to new defect predictors learned from the WC data.

Our next experiment is, therefore, designed to determine the number of examples required to build defect predictors from WC data.

5.4. Experiment IV: How much local data do organizations need for constructing a model?

Our results of Experiment II and III reveals that WC data models are better if data are available. In this section, we will show that defect predictors can be learned from very small samples of WC data.

Table 5.18. Experiment IV Design

```
DATA = [PC1, KC1, KC2, CM1, KC3, MW1, MC2]
LEARNER = [Naive Bayes]

REPEAT 100 TIMES
        FOR EACH data IN DATA
                WC_TRAIN = random 90% of data
                TEST = data - WC_TRAIN
                FOR i IN {100, 200, 300, ...}
                        WC_TRAIN_INCREMENTAL <- Random i Examples from WC_TRAIN
                        WC_INC_PREDICTOR = Train LEARNER with WC_TRAIN_INCREMENTAL
                        [iwc_pd, iwc_pf, iwc_bal] = WC_INC_PREDICTOR on TEST
                END
        END
END
```

5.4.1. Design

An important aspect of the Experiment II and III results is that defect predictors were learned using only a handful of defective modules. For example, consider a 90%/10% train/test split on $pc1$ with 1,109 modules, only 6.94% of which are defective. On average, the training set will only contain $1109 * 0.9 * 6.94/100 = 69$ defective modules. Despite this, $pc1$ yields an adequate median $\{pd, pf\}$ results of $\{63, 27\}\%$.

Experiment IV was, therefore, designed to check how much data are required to learn defect predictors. The design is given in Table 5.18. Experiment IV is essentially the same as Experiment II, but without the cross-company study. Instead, experiment IV takes the seven NASA projects of Table 4.2 and learns predictors using:

- reduced WC data: a randomly selected subset of up to 90% of each project data.

After randomizing the order of the data, training sets were built using just the first 100, 200, 300, ... data samples in the project. After training the defect predictor, its performance is tested on the remaining data samples not used in training.

Experiment II only used the features found in all NASA projects. For this ex-

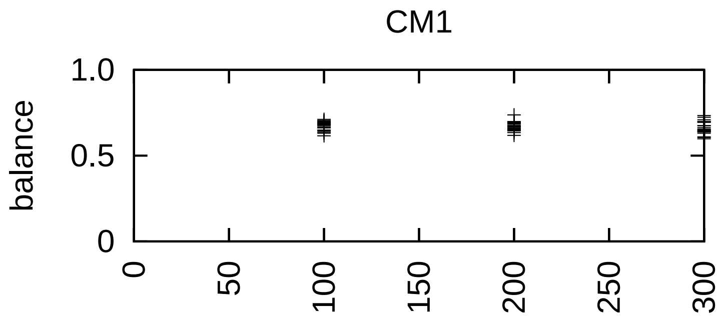


Figure 5.13. CM1 results from experiment IV. Training set size grows in units of 100 examples, moving left to right over the x-axis. .

periment, we imposed no such restrictions and used whatever features were available in each data set.

5.4.2. Results and Discussions

We would like to remind that balance is a combination of $\{pd, pf\}$ that decreases if pd decreases or pf increases. As shown in Figure 5.13 to Figure 5.19, there was very little change in balanced performance after learning from 100, 200, 300 . . . examples. Indeed, it seems that learning from larger training sets had detrimental effects: the more training data, the larger the variance in the performance of the learned predictor. In $kc1$ and $pc1$, as the training set size increases (moving right along the x-axis) the dots showing the balance performance start spreading out.

The Mann-Whitney U test was applied to check the visual trends seen in Figure 5.13 to Figure 5.19. For each project, all results from training sets of size 100, 200, 300, . . . were compared to all other results from the same project. The issue was "how much data are enough?" i.e. what is the *min*imum training set size that never lost to other training set of a larger size. Usually, that min value was quite small:

- In five projects $\{cm1, kc2, kc3, mc2, mw1\}$, $min = 100$;
- In $\{kc1, pc1\}$, $min = \{200, 300\}$ instances, respectively.

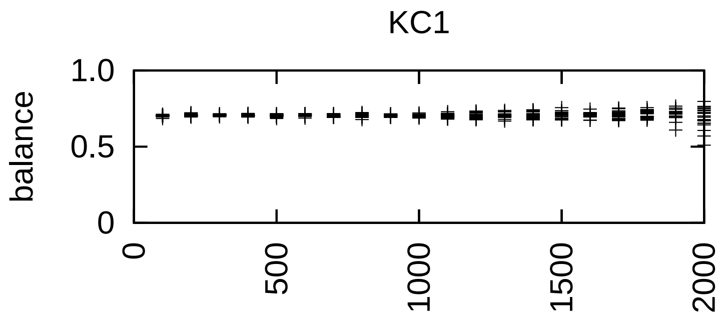


Figure 5.14. KC1 results from experiment IV. Training set size grows in units of 100 examples, moving left to right over the x-axis.

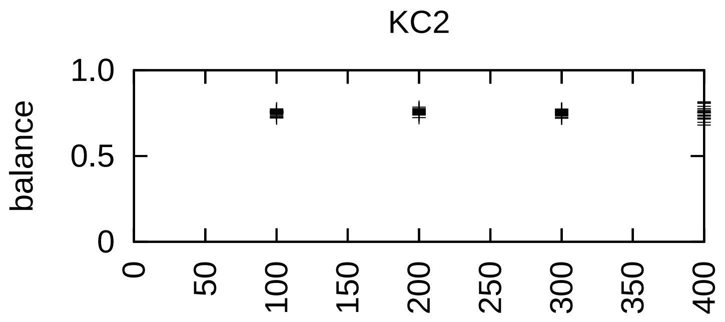


Figure 5.15. KC2 results from experiment IV. Training set size grows in units of 100 examples, moving left to right over the x-axis.

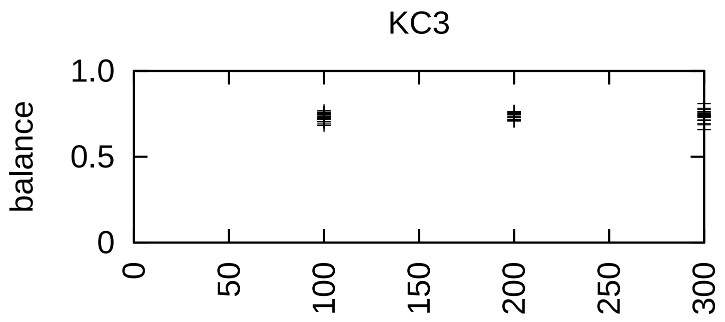


Figure 5.16. KC3 results from experiment IV. Training set size grows in units of 100 examples, moving left to right over the x-axis.

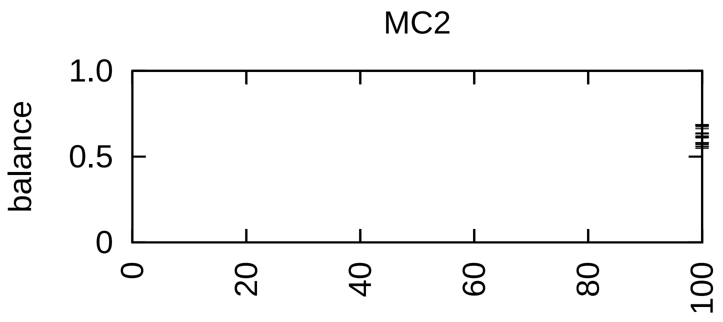


Figure 5.17. MC2 results from experiment IV. Training set size grows in units of 100 examples, moving left to right over the x-axis. The MC2 results only appear at the maximum x-value since MC2 has less than 200 examples.

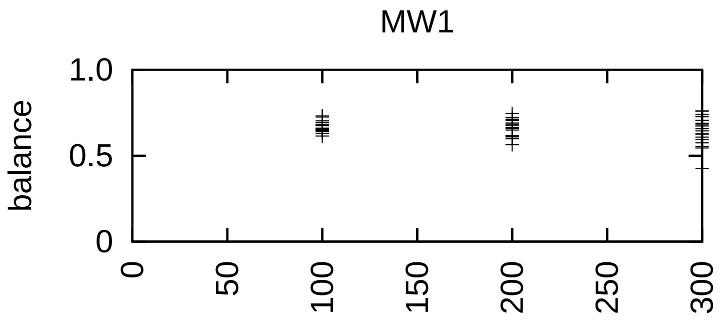


Figure 5.18. MW1 results from experiment IV. Training set size grows in units of 100 examples, moving left to right over the x-axis.

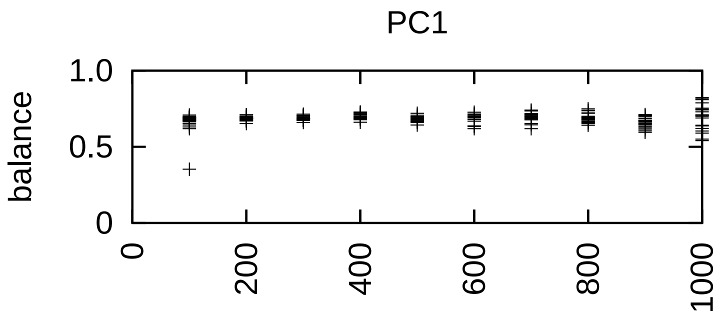


Figure 5.19. PC1 results from experiment IV. Training set size grows in units of 100 examples, moving left to right over the x-axis.

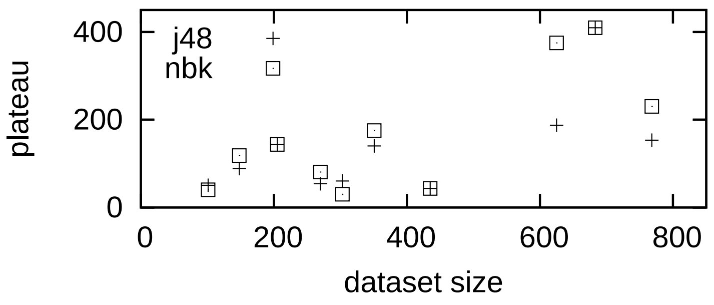


Figure 5.20. Y-axis shows plateau point after learning from UCI data sets that have up to X examples using Naive Bayes and decision tree learners (from [4]).

We explain the experiment IV results as follows. This experiment uses simplistic static code features such as lines of code, number of unique symbols in the module, etc. Such simplistic static code features are hardly a complete characterization of the internals of a function. We would describe such static code features as having *limited information content* [5]. Limited content is soon exhausted by repeated sampling. Hence, such simple features reveal all they can reveal after a small sample.

There is also evidence that the results of Experiment IV (that performance improvements stop after a few hundred examples) have been seen previously in the data mining literature. To the best of our knowledge, this is the first report of this effect in the defect prediction literature:

- In their discussion on how to best handle numeric features, Langley and John offers plots of the accuracy of Naive Bayes classifiers after learning on 10,20,40,..200 examples. In those plots, there is little change in performance after 100 instances [152].
- Orrego [4] applied four data miners (including Naive Bayes) to 20 data sets to find the *plateau point*: i.e. the point after which there was little net change in the performance of the data miner. To find the plateau point, Oreggo used t-tests to compare the results of learning from Y or $Y + \Delta$ examples. If, in a 10-way cross-validation, there was no statistical difference between Y and $Y + \Delta$, the plateau point was set to Y. As shown in Figure 5.20, many of those plateaus

were found at $Y \leq 100$ and most were found at $Y \leq 200$. We observe that these plateau sizes are consistent with the results of Experiment IV.

In the majority case, predictors learned from as little as one hundred examples perform as well as predictors learned from many more examples. This suggests that the effort associated with learning defect predictors from within-company data may not be overly large. For example, the effort required to build and test 100 modules may be as little as 2.4 to 3.7 person months. To generate an effort estimate for these modules, we used the on-line COCOMO [132] effort estimator.[8] In $CM1$ project, the median module size is 17 lines. 100 randomly selected modules would have 1700 LOC. Therefore, estimates were generated assuming 1700 LOC and the required reliability varying from very low to very high.

5.5. Experiment V: Can our results be generalized?

Experiments II to IV were based on NASA projects. In order to search evidence for the external validity of the conclusions of those experiments, we replicate the same experiments on SOFTLAB projects of Table 4.2.

5.5.1. Design

For each SOFTLAB project, we follow the same procedure as in Experiments II and III; i.e. 10% of the rows of each data set are selected at random for constructing test sets and then defect predictors are learned from:

- CC data: all data from seven NASA projects.
- WC data: random 90% data of remaining SOFTLAB projects. In order to reflect the use in practice, we do not use the remaining 90% of the same project for training, we rather use a random 90% of data from other projects. Since SOFTLAB data are collected from a single company, learning a predictor on some projects and to test it on a different one does not violate within company simulation.

[8] `http://sunset.usc.edu/research/COCOMOII/expert_cocomo/expert_cocomo2000.html`

Table 5.19. Experiment V results for the SOFTLAB projects.

Average results on SOFTLAB data

treatment		min	Q1	median	Q3	max	
pd	CC	88	88	95	100	100	| ●
	WC	35	40	88	100	100	– | ●
pf	CC	52	59	65	68	68	|–●
	WC	3	5	29	40	42	- ● - |

Table 5.20. Experiment V results for AR3.

Table (*ar*3)

treatment		min	Q1	median	Q3	max	
pd	CC	88	88	88	88	88	| ●
	WC	88	88	88	88	88	| ●
pf	CC	62	65	65	65	65	| ●
	WC	40	40	40	40	42	● |

- NN filtered CC data: *similar* data from seven NASA projects.

The SOFTLAB projects include 29 static code features, 17 of which are common with the NASA projects (see Table 4.4). In order to simplify the comparison between these new projects and Experiment II and III, we use only these shared attributes in our CC experiments. On the other hand, we use all available features in WC experiments for SOFTLAB projects. Finally, in this experiment, we treated each NASA project as cross company data for SOFTLAB projects.

5.5.2. Results and Discussions

Figure Table 5.19 shows the results:

- The *pd* values for CC data increase compared to WC data with the cost of increased *pf*.

Table 5.21. Experiment V results for AR4.

Table (*ar*4)

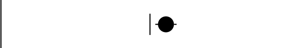
treatment		min	Q1	median	Q3	max	
pd	CC	95	95	95	95	95	
	WC	35	40	40	40	40	
pf	CC	52	55	56	59	60	
	WC	3	3	3	5	5	

Table 5.22. Experiment V results for AR5.

Table (*ar*5)

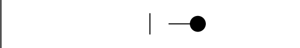
treatment		min	Q1	median	Q3	max	
pd	CC	100	100	100	100	100	
	WC	88	100	100	100	100	
pf	CC	57	68	68	68	68	
	WC	29	29	29	29	29	

Table 5.23. Experiment V pd results for the SOFTLAB projects, where $NN_{pd} \geq WC_{pd}$.

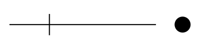
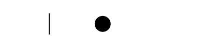
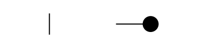
			quartiles					
rank			0	25	50	75	100	
AR4	1	CC	35	90	100	100	100	
	2	NN	65	65	70	70	70	
	3	WC	35	40	40	40	45	
AR3	1	CC	75	88	88	88	88	
	1	NN	88	88	88	88	88	
	1	WC	88	88	88	88	88	
AR5	1	CC	88	100	100	100	100	
	1	NN	100	100	100	100	100	
	1	WC	88	100	100	100	100	
								50%

Table 5.24. Experiment V pf results for the SOFTLAB projects where $NN_{pf} \leq WC_{pf}$.

	rank		quartiles 0	25	50	75	100
AR3	1	NN	38	38	38	38	40
	2	WC	40	40	40	40	42
	3	CC	38	55	56	60	80
AR5	1	NN	21	25	25	25	32
	2	WC	29	29	29	29	29
	3	CC	29	46	50	50	79

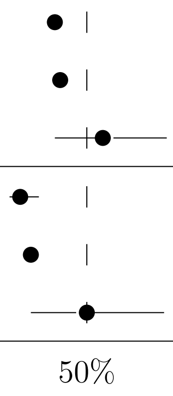
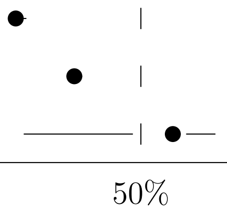


Table 5.25. Experiment V pf results for the SOFTLAB projects, where $NN_{pf} > WC_{pf}$.

	rank		quartiles 0	25	50	75	100
AR4	1	WC	3	3	3	5	7
	2	NN	24	25	25	25	28
	3	CC	6	47	62	67	78



- CC data shifts {Q1, median} of *pf* from {5, 29} to {59, 65}.
- For CC data:
  - 25% of the *pd* values are at 100%.
  - 50% of the *pd* values are above 95%
  - And all the *pd* values are at or over 88%.

These results also provide evidence for the validity of our conclusions for Experiment V. In Experiment V, we conclude that the minimum number of instances for training a defect predictor is around 100 – 200 data samples. Please note that SOFTLAB projects *ar3, ar4* and *ar5* have {63,107, 36} modules respectively, with a total of 206 modules. Thus, WC results in Figure Table 5.19 are achieved using a minimum of

$(63+36)*0.90 = 90$ data samples (i.e. learn a predictor on *(ar3 + ar5)* and test it on *ar4*) and a maximum of $(63+107)*0.90 = 153$ data samples (i.e. learn a predictor on *(ar3 + ar4)* and test it on *ar5*).

Table 5.23 to Table 5.25 shows *Observation1* (i.e. $NN_{pd} \geq WC_{pd}$) and *Observation2* (i.e. $NN_{pf} \leq WC_{pf}$) for SOFTLAB projects. We would like to remind that these observations were mutually exclusive for NASA projects. The pattern is similar in SOFTLAB projects:

- for *ar4* mutual exclusiveness hold: $NN_{pd} \geq WC_{pd}$ and $NN_{pf} > WC_{pf}$
- for *ar3* and *ar5*: $NN_{pf} \leq WC_{pf}$. If the observations were mutually exclusive, we would expect $NN_{pd} < WC_{pd}$. Yet, for *pd* $NN_{pd} \geq WC_{pd}$, however this inequality holds with the equity (see Table 5.23) and $NN_{pd} \not> WC_{pd}$

In summary, the WC, CC and NN patterns found in American NASA rocket software are also observed in software controllers of Turkish domestic appliances. While this is not the definitive proof of the external validity of our results, we find it a very compelling result that is reproducable in different companies.

5.6. Experiment VI: Additional Local Information Sources

In this section, we use static call graph based ranking (CGBR) framework, which can be applied to any local defect prediction model. In this framework, we model both intra module properties and inter module relations. We aim at increasing the information content of static code attributes by including the architectural structure of the code. More precisely, most static code attributes assume the independence of software modules and measure individual module complexities without considering their interactions. In this section we adjust static code attributes by modeling their interconnection schema using call graphs. We calculate call graph based ranking (CGBR) values and assign ranks to each module.

5.6.1. Design

For this experiment, we use SOFTLAB data, since we only have access to source codes of these projects. NASA projects do not contain call graph information. Therefore, we used the available source code to build the static call graphs. In our experiments, we left one of the projects as the test case and built predictors on the remaining two projects. We repeat this for all three SOFTLAB projects. We use a random 90% of the training set for the models and repeat this procedure 20 times in order to overcome ordering effects. Before we train our model, we have applied log-filtering on the datasets and normalized our CGBR values. We classified our CGBR values in 10 bins and give each bin a weight value in 0.1 increments in the [0..1] interval. Then we have adjusted the static code features of each module by multiplying each row in the data table with corresponding weights. We compare the CGBR results with the standard Naive Bayes results on SOFTLAB projects.

5.6.2. Results and Discussions

Figure 5.21 and Figure 5.22 show (pd, pf, bal) results for the original attributes (i.e. without CGBR adjustment) and the dataset with CGBR adjustment respectively. These figures show the boxplots of the (pd, pf) and balance results. Figure 5.21 and Figure 5.22 show that using CGBR adjusted data slightly decreases the median probability of detection, and significantly decreases the median probability of false alarms. Moreover, CGBRs pd results are spread in a narrower interval, from which we can conclude that CGBR produces more stable results than standard approach. Please note that 50% of the results yield detection rates over 88% and false alarms below 18%. We observe that balance also significantly improves.

We have also analysed the cost effectiveness of the model. Table 5.26 shows median (pd, pf) rates of the two models and the required testing effort in terms of LOC [69, 39]. It also includes the estimated LOC for random strategy depending on the pd rates 79% and 77% respectively. In order to detect 79% of the actual defects in the code, the estimated LOC for inspection with random strategy is 13,866 and the

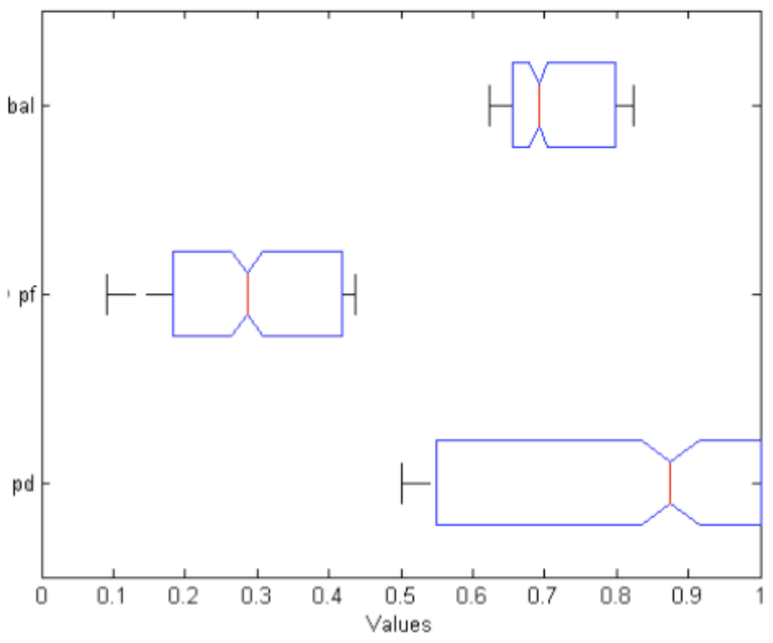


Figure 5.21. Experiment VI standard naive Bayes results on SOFTLAB projects.

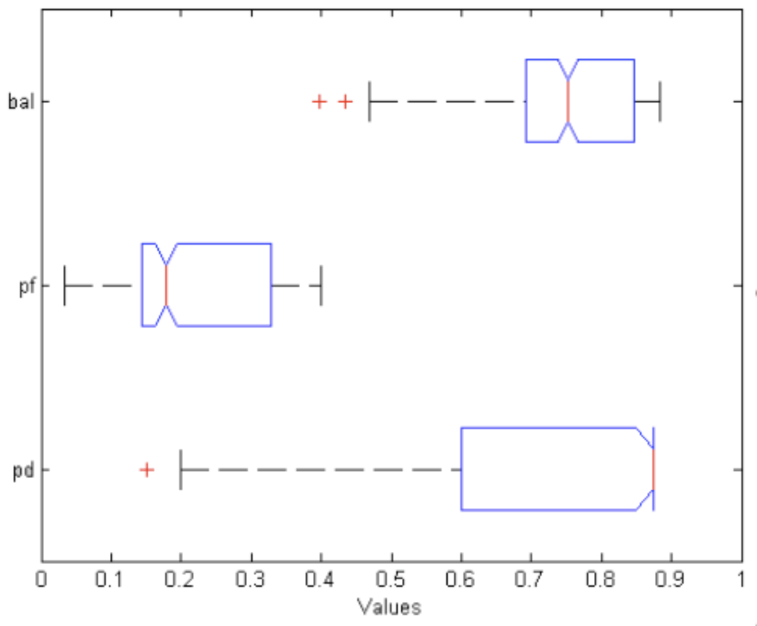


Figure 5.22. Experiment VI CGBR results on SOFTLAB projects.

estimated LOC for original data is 12,513. Therefore, using Nave Bayes predictor with original data achieves an improvement of 10% in testing effort. Conversely for 77% defect detection rate of the CGBR framework, the estimated LOC for random strategy is 13,515 and the required LOC for CGBR adjusted data is 10439. The false alarm rate has also changed from 30% to 20%. Hence CGBR framework decreases the probability

Table 5.26. Cost effectiveness of CGBR.

	Original Data	CGBR
pd	79%	77%
pf	30%	20%
Estimated LOC for inspection	12513	10439
LOC for manual inspection	13866	13515
Improvement on testing effort	10%	23%

Table 5.27. Experiment VI project-wise performance of CGBR.

		AR3	AR4	AR5	Average
CGBR	pd	88	55	88	77
	pf	33	10	18	20
Original	pd	86	50	100	79
	pf	42	18	29	30

of false alarms and testing effort, where the improvement is 23%.

The results for individual projects also validate our findings. Table 5.27 shows the (pd, pf) rates with CGBR adjusted data and with the original data. For all projects we observe the same pattern: pf rates decrease for all projects. For AR5, pf rate is decreased with the cost of missing some defective modules. However, AR5 is a small dataset with only eight defective modules out of a total of 36 projects. Therefore, CGBR adjusted data model misses only a single defective module compared to the original data model.

We explain our observations as follows: Adjusting dataset with CGBR values provide additional information content compared to pure static code attributes. The adjustment incorporates inter module information by using architectural structure, in addition to the intra module information available in static code attributes. Our results show that defect predictors using CGBR framework can detect the same number of defective modules, while yielding significantly lower false alarm rates. On industrial public data, we also show that using CGBR framework can improve testing efforts by

23%.

The benefits of using datasets adjusted with CGBR values are very important and practical. Low pf rates make the predictors practical to use. The reason is that low pf rates keep the amount of code to be manually inspected at a minimum rate. As pf rates increase, the predictors become impractical, even not much better than random testing strategies. Our proposed approach allows decreasing the pf rates, thus it is practical to use.

Please note that CGBR adjusted model locates equivalent number of defects with the standard model, while giving fewer false alarms. Our results show that, if testers were to use the predictions from the CGBR adjusted model, they would spend 13% less testing effort than the original Nave Bayes model and 23% less than random testing.

When we check the correlation between the CGBR values and the defect content of the modules, we observe a negative correlation of -0.149. While this does not represent a strong correlation, it suggests that there are relatively fewer defects when the modules CGBR values are higher. We suspect that if a module is called frequently from other modules, any defect in this module is recognized and corrected in a short time by the developers. On the other hand, the modules which are not called by other modules will be more defective because these types of modules are not used frequently. Thus, developers are not as much aware as they do for frequently used modules. As a result the defects can easily hide in less used modules.

5.7. Threats to Validity

In this section, we discuss the possible threats to the validity of our results. Like other empirical studies, our results are possibly biased by the data and the algorithm we have used.

The reasons for our choice of Naive Bayes as the data miner has been thoroughly discussed in Section 3.1. In summary, the performance of Naive Bayes is identical with at least 14 other methods as demonstrated by Lessmann et.al. [3]. We have used Naive Bayes for its simplicity and high performance. We further explored the validity of its two basic assumptions for software defect data.

Our experiments regarding the assumptions of Naive Bayes is a replication of a benchmark study by Menzies et.al. [28] Therefore, we have used exactly the same datasets which are used in the benchmark study. However, certain data quality issues have been reported in PROMISE repository, risking the construct validity of some NASA projects. Considering these risks, we have not used these potentially flawed datasets in the rest of our experiments. Once again, we have chosen to include those in the first set of experiments, since it was designed to be comparable to the baseline study.

Due to many factors affecting the software process and products, it is hard to claim generality of results in software engineering domain [153]. We have included different project data from different companies to avoid this threat as best as we can. The external validity of generalizing from NASA projects has been discussed by Basili et.al. and Menzies et.al. [7, 28] In summary, NASA uses contractors who are contractually obliged (ISO-9O01) to demonstrate their understanding and usage of current industrial best practices. Nevertheless, in order to test claims of external validity for our cross company experiments, we have followed a procedure, where the SOFTLAB data were initially kept in reserve and these experiments were reproduced on SOFTLAB data.

Concerning our CGBR experiments, we have data shortage, since public data sets do not contain call graph matrices. This is a challenge that we could overcome by using datasets of the company that we had source code access (i.e. SOFTLAB projects). Therefore, this study should be externally validated with other industrial and open source projects of different sizes. Although open source projects allow access to code, our experiences show that matching defects at the module level is difficult.

6. CONCLUSIONS

6.1. Summary of Results

In this dissertation software defect prediction is considered as a data mining problem. We analyzed the validity of Naive Bayes' assumptions (i.e. independence and equal importance of attributes) on software defect data. By relaxing the assumptions of Naive Bayes we have achieved models of different complexities. We have conducted several experiments in order to compare the performances of these models. We have also used several heuristics in order to estimate the weights of attributes based on their relative importance. Our results show that the independence of attributes assumption in Naive Bayes is valid for software defect data, at least after PCA processing. Although more complex models can produce numerically better results, these are not statistically significant. On the other hand, relaxing the other assumption (i.e. equal importance of attributes) produced significantly better results than simple Naive Bayes. The complexity introduced by the weighting term is negligible and the weights can be assigned using several heuristics. Furthermore, the weighting scheme removes the need for feature subset selection by favoring informative attributes. As a conclusion, we advise using weighted Naive Bayes for modelling software defect data.

After our analysis of cross vs. within company data defect predictors, on the contrary to effort estimation literature, we have found clear and unambiguous conclusions for defect prediction. To summarize:

- CC-data dramatically increase the probability of detecting defective modules;
- But CC-data also dramatically increase the false alarm rate.
- NN-filtering CC data avoids the high false alarm rates by removing irrelevancies in CC data;
- Yet WC-data models are still the best and they can be constructed with small amounts of data (i.e. 100 examples).

In the light of these results, we conclude that companies can benefit from raw CC data in extreme cases such as mission critical projects, where the cost of false alarms can be afforded. Further, pruning CC data with NN-filter allows the use of CC data for constructing practical defect predictors in other domains. NN-filtered CC data yields much better results than raw CC data, yet closer but worse results than WC data. Good news is that the best option of using WC data requires the collection of a mere hundred examples from within a company and can be done in a short time (i.e. a few months). We observe the same patterns not only in aerospace software from NASA, but also in software from a completely different company round the globe.

We have also proposed a novel combination of the static code features and architectural structure of software to make predictions about defect content of software modules (i.e. CGBR framework). We model intra module properties by using static code features. Further, we model inter module relations by ranking software modules, where the weights are derived from static call graphs using the well-known PageRank algorithm. This approach yielded around 10% decreases in false alarm rates.

6.2. Theoretical and Methodological Contributions

In this dissertation, we have proposed novel ways for decision making under uncertainty, which has been a very active research area in AI for decades. Our analysis have been carried out in software quality domain where the development of defect prediction models has also attracted the attention of many researchers. The reason for such a significant attention to automated quality predictors lays in their practical importance. Current models are useful, as they allow software project managers to better guide the allocation usually scarce quality assurance resources to artifacts which need them the most. In this dissertation, we have made an performance analysis of defect predictors in different contexts and proposed novel ways to construct and improve them.

We based our analysis on the observation that straightforward applications of data mining methods have hit a performance ceiling. With this observation, we focused our efforts on data related issues rather than method related issues. However, software engineering domain is a data starving domain by its nature. Furthermore, data quality and information content is a critical factor affecting the data mining performance. These issues raised two common data mining challanges:

1. Available data sample should well represent the problem.
2. Solution space should be searched for better quality solutions.

In order to overcome the first challenge, we have increased the number of data samples by using data from multiple companies. This approach yielded better representation. However, it also spanned a wider space than pure local data, which in turn expanded the search space. We used the NN filtering approach for guiding this solution space search for better quality solutions. This can also be perceived as transforming the unsupervised defect prediction problem without labelled within company data into the supervised defect prediction problem of labelled cross company data. For the case, where labeled within company data are available, we improved the data information content with domain specific knowledge such as using call graphs. Further, we fed the data miners with inherently available information in data such as weights and correlations among features. Providing weights improved the quality of the solutions, whereas providing correlations did not since it also increased the problem complexity significantly. Nevertheless, we found better solutions using all approaches, but the correlation modeling.

The theoretical and methodological contributions of this dissertation can be summarized as follows:

1. *Evaluation of Model Assumptions:* Naive Bayes is a simple, yet very effective data miner despite its unrealistic assumptions. The analysis of its assumptions are investigated in data mining literature. In this dissertation, we extend these analysis in software quality domain.
2. *Combining Data From Multiple Sources for Improving Information Content:* In this dissertation, we combine data from multiple companies to improve the rep-

resentation capability of the limited data sample. Then, we carry out a novel comparative analysis of defect predictors learned from CC and WC data. This issue has not been investigated previously. We believe that we fill a certain gap in defect prediction literature.

3. *Removing Noise from Remote Sources:* Combining data from multiple companies improved the information content in data, however it also introduced too much noise that limited the performance of predictions to certain domains. In order to overcome this issue, we have proposed to use NN-filtering technique for removing the noise in data and obtained improved results.
4. *Evaluation of the Validity of Common Beliefs for Defect Prediction:* In favor of the common belief, we have empirically shown that local data are better for constructing defect predictors. Further, on the contrary to the common belief, we have shown that required data can be collected very quickly.
5. *CGBR framework for performance improvement:* We have proposed a framework that significantly improves the performance of defect predictors. Proposed framework is based on call graphs. They have been previously used in other research by carrying out complex social network analysis. However, our proposed method is simple yet effective and its conceptual background has been used effectively in other domains.
6. *Experiment Replication:* Replication studies in empirical research are very important for validating and improving the results of previous research. However, they are rare in software quality research due to data availability. In this dissertation, we have replicated a benchmark experiment, validated its results and also improved them with the proposed methods.
7. *Reproducible Methodology:* Another important aspect of empirical research is reproducibility. We have performed a set of well-defined experiments on publicly available data, which allows other researchers to replicate our research.
8. *Data Collection and Donation:* Software engineering is a data scarce domain. Especially, industrial data are hard to reach and collect. In the context of this dissertation, we have collected industrial data and made them publicly available for the use of other researchers and practitioners.
9. *Open Source Metric Collection and Defect Analysis Tool:* Extracting data from

source codes is a difficult task when carried out manually. Automated tools make this process significantly easier and more reliable. Our developed tool, Prest, is not only a metric collection tool but also a defect analysis tool, which is free for the use of other researchers and practitioners. We hope that Prest will help increasing the number of projects in public software engineering data repositories.

These contributions also have practical implications, which are discussed in the next section.

6.3. Practical Implications

In practice, if a company lacks local data, we would suggest a two-phase approach. In phase one, that organization uses imported CC data filtered via NN. Also, during phase one, the organization should start a data collection program to collect static code attributes. Phase two commences when there is *enough* local WC data to learn defect predictors. During phase two, the organization would switch to new defect predictors learned from the WC data.

Arisholm and Briand have certain concerns on the practical usage of defect predictors [69]. They argue that if $X\%$ of the modules are predicted to be faulty and if those modules contain less than $X\%$ of the defects, then the costs of generating the defect predictor is not worth the effort.

Let us analyze the testing efforts on a sample NASA project (i.e. MW1) from Arisholm and Briand's point of view. For MW1, there are a total of 403 modules with 31 defective and 372 defect-free ones. CC model yields 90% pd and 68% pf, and one should examine 280 modules, which is around a 31% reduction in inspection efforts compared to examining all modules. Yet, we argue that 68% pf rate is quite high and using NN we are able to reduce it to 33% along with 68% pd. This corresponds to examining 144 modules, a reduction of 47% compared to exhaustive testing (and we assume an exhaustive test should examine 274 modules for detecting 68% defects, as Arisholm and Briand suggests).

This analysis can be extended for all projects used in this dissertation. For instance, the company from which SOFTLAB data in Table 4.2 are collected is keen to use our detectors, arguing that they operate in a highly competitive market segment where profit margins are very tight. Therefore, reducing the cost of the product even by 1% can make a major difference both in market share and profits. Their applications are embedded systems where, over the last decade, the software components have taken precedence over the hardware. Hence their problem is a software engineering problem. According to Brooks [43], half the cost of software development is in unit and systems testing. The company also believes that their main challange is the testing phase and they seek predictors that indicate where the defects might exist *before* they start testing. Their expectation from the predictor is not to detect all defects, but to guide them to the problematic modules so that they can detect more defects in shorter times. Hence, any reduction in their testing efforts allows them to efficiently use their scarce resources.

An important issue worth more mentioning is the concern about the time required for setting up a metric program (i.e. in order to collect data for building actual defect predictors). Our incremental WC results suggest that, in the case of defect prediction, this concern may be less than previously believed. Kitchenham *et al.* [130] argue that organizations use cross-company data since within-company data can be so hard to collect:

- The time required to collect enough data on past projects from within a company may be prohibitive.
- Collecting within-company data may take so long that technologies change and older projects do not represent current practice.

In most of our experiments, as few as 100 modules may be enough to learn adequate defect predictors. When so few examples are enough, it is possible that projects can learn local defect predictors that are relevant to their current technology in just a few months.

Further, our experiences with our industry partners show that data collection is not necessarily a major concern. Static code attributes can be automatically and quickly collected with relatively little effort. We have found that when there is high level management commitment, it becomes a relatively simple process. For example, in an extreme case, the three projects of SOFTLAB data were collected in less than a week's time. Neither the static code attributes, nor the mapping of defects to software modules were available when the authors attempted to collect these data. Since these were smaller scale projects, it was sufficient to spend some time with the developers and going through defect reports. Although not all projects have 100 modules individually, the company has a growing repository from several projects and enough data to perform defect prediction.

We also have experience with a large scale telecommunication company, where a long-term metric program for monitoring complex projects (around 750.000 lines of code) requires introducing automated processes. Again with high level management support, it was possible to employ appropriate tool support and these new processes were introduced easily and invisible to the staff. For that project, we have now a growing repository of defects mapped with source code (around 25 defects per month). Note that the software in that project are being developed for more than 10 years and have very low defect rates. We have obtained the first results in the 8^{th} month of a 12 months long project. In summary setting up a metric program for defect prediction can be done more quickly than it is perceived.

6.4. Answers to Research Questions

In this section we provide the answers to the research questions, considering the outcomes of our empirical evaluations.

6.4.1. How can we improve the information content of local data without introducing new information sources?

We showed that relaxing assumptions of certain models may increase their prediction performances. We have used the naive Bayes model in our analysis and our weighting scheme yielded significantly better results than the standard naive Bayes in majority of the projects. Further, we showed that the independence assumption of naive Bayes is valid for defect data, at least after PCA processing, where the only negative results are observed.

6.4.2. How can companies construct local defect predictors with remote data?

Our goal was to identify the conditions under which cross-company data may be preferred to within-company for the purposes of learning defect predictors. Those conditions turned out to be quite extreme; so much that they hold in only a small number of organizations (e.g. organizations would have to tolerate extremely high false alarm rates). Hence, except in very rare cases, we can not recommend the use of unfiltered cross-company data for defect prediction.

6.4.3. How can companies filter remote data for local tuning?

We anticipated that the answer to the previous research question will not be enough to stop the use of cross-company data. Our explanation for the limited applicability of CC data is that it mixes useful information with an excess of extraneous information. We applied a simple nearest neighbor (NN) filtering to CC data for constructing a locally tuned repository and obtained better predictors, which can be used temporarily before collecting local defect data. The performance of these temporary predictors are far better than CC predictors, yet still worse than but much closer to WC predictors.

6.4.4. How much local data do organizations need for constructing a defect prediction model?

A common belief is that collecting local data that are required to build defect prediction models takes too much time and effort [130]. We showed that predictors learned from a mere one hundred examples perform as well as predictors learned from many more examples. That is, defect predictors tuned to the particulars of one company can be learned using very little data, collected in a very small amount of time: two to four person-months.

6.4.5. Can our results be generalized?

We initially used only NASA projects in our experiments to answer the questions related to CC analysis. In order to check the external validity of our results, we replicated all related experiments on data from a company that has no ties with NASA: specifically, a Turkish company writing software controllers for Turkish whitegoods. All the results described above also hold for the Turkish data. While this does not conclusively prove the external validty of our conclusions, it does suggest that these results are not observed in one set of projects by chance.

6.4.6. How can we improve the information content in static code features with more local resources?

Call graphs are easy to generate by using automated tools. We showed that the information provided by call graphs can improve the quality of solutions significantly. However, this was observed in SOFTLAB projects only and could not be evaluated in NASA projects, since they do not include such information. This outcome should be externally validated in further research if and when data are available.

6.5. Future Directions

Standard machine learning algorithms lack the business knowledge which characterizes software projects. To add that business knowledge, we propose to use human-in-the-loop case based reasoning (CBR) tools. We expect that this approach will allow software managers to safely focus on the application of quality assurance techniques of choice, confident that automated quality predictors will raise alerts about the artifacts where quality issues actually exist.

There exist numerous incremental case-based reasoning tools [154, 155, 156] that ask humans to audit a stochastic sample of real-world cases. Insights gained from those sessions are automatically generalized and applied to another random sample. Experts then review the classifications made on the new sample, and offer further refinements. In 1999, Fenton and Neil [47] postulated that such human-machines-based system might outperform systems based on on static code measures (since other features/metrics could be accounted for that cannot currently be addressed using static code metrics).

Case-based reasoning methods require humans to examine and comment on specific cases. This is impractical if learning adequate theories requires examining a very large number of cases. Our results suggest that, for static code measures, it is not necessary to manually inspect thousands of cases. In fact, just a few hundred may suffice. These results raise the possibility that a human-in-the-loop case-based reasoning environment might *perform as well as* automatic methods, despite the automatic methods exploring more examples.

Such an environment might *perform better* than automatic methods. It can be very useful to let experts access and combine features from whatever sources are locally available. Such an "explore whatever" environment is not an automatic black box data miner. Rather, it is a human-in-the-loop case-based reasoning (CBR) environment where humans reflect on the specifics of particular cases, connect to different data sources, and (sometimes) run automatic data miners on combinations or subsets of a variety of types of features.

Another role for human experts in a CBR environment is to instruct the learner how *combinations* of attributes can work together to provide solutions. For example a standard Naive Bayes classifier gives equal weights to all attributes then uses frequency counts to learn the relative importance of each attribute. When, we have assigned unequal attribute weights, we observed that the performance can improve over standard Naive Bayes, and there is also no need for feature subset selection. Though the improvements are not ground-breaking, they provide a hint regarding the value of unequal treatment of information sources. In these figures weights are assigned by the model. These weights provided by models may not be meaningful to humans in the process [47]. However, we argue that weights assigned to different information sources by human experts *with business knowledge* can increase the quality of solutions.

REFERENCES

1. "The Standish Group Report: CHAOS Chronicles, 2004 Third Quarter Research Report", 2004.

2. Jiang, Y., B. Cukic, and T. Menzies, "Fault Prediction using Early Lifecycle Data", *ISSRE'07*, 2007, available from `http://menzies.us/pdf/07issre.pdf`.

3. Lessmann, S., B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings", *accepted for publication IEEE Transactions on Software Engineering*, 2009.

4. Orrego, A., *SAWTOOTH: Learning from Huge Amounts of Data*, Master's thesis, Computer Science, West Virginia University, 2004.

5. Menzies, T., B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang, "Implications of Ceiling Effects in Defect Predictors", *Proceedings of PROMISE 2008 Workshop (ICSE)*, 2008, available from `http://menzies.us/pdf/08ceiling.pdf`.

6. "The Standish Group Report: Chaos", 1995, available from `http://www4.in.tum.de/lehre/vorlesungen/vse/WS2004/1995_Standish_Chaos.pdf`.

7. Basili, V., F. McGarry, R. Pajerski, and M. Zelkowitz, "Lessons Learned from 25 Years of Process Improvement: The Rise and Fall of the NASA Software Engineering Laboratory", *Proceedings of the 24th International Conference on Software Engineering (ICSE) 2002, Orlando, Florida*, 2002, available from `http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/83.88.pdf`.

8. Paulk, M., B. Curtis, M. Chrissis, and C. Weber, "Capability Maturity Model, Version 1.1", *IEEE Software*, Vol. 10, No. 4, pp. 18–27, July 1993, available from `ftp://ftp.sei.cmu.edu/pub/cmm/Misc/cmm.pdf`.

9. IEEE-1012, "IEEE Standard 1012-2004 for Software Verification and Validation", 1998.

10. "ISO/IEC 12207 Standard for Information Technology - Software Lifecycle Process", 1998.

11. Society, I. C., "IEEE Recommended Practice 1278.4-1007 -2004 for Distributed Interactive Simulation- Verification, Validation, and Accreditation", 1997.

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

13. Halstead, M., *Elements of Software Science*, Elsevier, 1977.

14. Chidamber, S. and C. Kemerer, "A metrics suite for object oriented design", *Software Engineering, IEEE Transactions on*, Vol. 20, No. 6, pp. 476 – 493, Jun 1994, `http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=295895&isnumber=7320&punumber=32&k2dockey=295895@ieeejrns`.

15. Harrold, M., "Testing: a roadmap", *Proc. of the Conference on the Future of Software Engineering*, pp. 61–72, ACM Press, NY, 2000.

16. B. V. Tahat, B. K. and A. Bader, "Requirement-Based Automated Black-Box Test Generation", *Proc. 25th Annual Int. Computer Software and Applications Conference*, pp. 489–495, 2001.

17. Shull, F., V. 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", *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, pp. 249–258, 2002.

18. Boehm, B. and V. Basili, "Software Defect Reduction Top 10 list", *IEEE Software*, pp. 135–137, January 2001.

19. Song, Q., M. Shepperd, M. Cartwright, and C. Mair, “Software Defect Association Mining and Defect Correction Effort Prediction”, *IEEE Trans. Softw. Eng.*, Vol. 32, No. 2, pp. 69–82, 2006.

20. Mitchell, T., *Machine Learning*, McGraw-Hill, 1997.

21. Yolum, P. and M. P. Singh, “Emergent Properties of Referral Systems”, *in Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 592–597, 2003.

22. Gokberk, B., *Three Dimensional Face Recognition*, Ph.D. thesis, Department of Computer Engineering, Bogazici University , Istanbul, Turkey, 2007.

23. Adlassnig, K. P. and W. Scheithauer, “Performance evaluation of medical expert systems using ROC curves”, *Computers and Biomedical Research*, Vol. 22, No. 4, pp. 297–313, 1989.

24. Benjamins, V. and W. Jansweijer, “Toward a Competence Theory of Diagnosis”, *IEEE Expert*, Vol. 9, No. 5, pp. 43–53, October 1994.

25. Harmon, D. and D. King, *Expert Systems: Artificial Intelligence in Business*, John Wiley & Sons, 1983.

26. Menzies, T., “An Investigation of the AI and Expert Systems Literature 1980-1984”, *AI Magazine*, Summer 1989.

27. Menzies, T., Z. Chen, J. Hihn, and K. Lum, “Selecting Best Practices for Effort Estimation”, *IEEE Transactions on Software Engineering*, November 2006, available from `http://menzies.us/pdf/06coseekmo.pdf`.

28. Menzies, T., J. Greenwald, and A. Frank, “Data Mining Static Code Attributes to Learn Defect Predictors”, *IEEE Transactions on Software Engineering*, January 2007, available from `http://menzies.us/pdf/06learnPredict.pdf`.

29. Bell, R., T. Ostrand, and E. Weyuker, "Looking for bugs in all the right places", *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, Jul 2006, `http://portal.acm.org/citation.cfm?id=1146238.1146246`.

30. Ostrand, T., E. Weyuker, and R. Bell, "Where the bugs are", *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, Jul 2004, `http://portal.acm.org/citation.cfm?id=1007512.1007524`.

31. Ostrand, T. and E. Weyuker, "The distribution of faults in a large industrial software system", *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, Jul 2002, `http://portal.acm.org/citation.cfm?id=566172.566181`.

32. Ostrand, T., E. Weyuker, and R. Bell, "Automating algorithms for the identification of fault-prone files", *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, Jul 2007, `http://portal.acm.org/citation.cfm?id=1273463.1273493`.

33. Koru, A. and H. Liu, "Modeling the Effect of Size on Defect Proneness for Open-Source Software", *Proceedings of the Third International Workshop on Predictor*, Jan 2007, `http://doi.ieeecomputersociety.org/10.1109/PROMISE.2007.9`.

34. Florac, W. A., R. Park, and A. D. Carleton, *Practical Software Measurement: Measuring for Process Management and Improvement*, CMU/SEI-97-HB-003, Software Engineering Institute, Carnegie Mellon University,, April 1997.

35. Fenton, N. E. and S. Pfleeger, *Software Metrics: A Rigorous & Practical Approach (second edition)*, International Thompson Press, 1995.

36. "Nasa/Wvu IV&V Facility, Metrics Data Program, available from http://mdp.ivv.nasa.gov", Internet; accessed 2007.

37. Turhan, B., A. D. Oral, and A. Bener, "A Short Survey on Software Architecture Evaluation Methods and Object Oriented Metrics", *1. Ulusal Yazilim Mimarisi Konferansi (UYMK'06)*, 2006.

38. Gyimothy, T., R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction", *Software Engineering, IEEE Transactions on*, Vol. 31, No. 10, pp. 897 – 910, Oct 2005, `http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=1542070&isnumber=32934&punumber=32&k2dockey=1542070@ieeejrns`.

39. Turhan, B., G. Kocak, and A. Bener, "Software Defect Prediction Using Call Graph Based Ranking (CGBR) Framework", *to appear in Proceedings of the 34th EUROMICRO Software Engineering and Advanced Applications*, 2008.

40. Bettenburg, N., S. Just, A. Schröter, C. Weiß, R. Premraj, and T. Zimmermann, "Quality of bug reports in Eclipse", *eclipse '07: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, Oct 2007, `http://portal.acm.org/citation.cfm?id=1328279.1328284`.

41. Nagappan, N. and T. Ball, "Use of relative code churn measures to predict system defect density", *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pp. 284 – 292, Apr 2005, `http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=1553571&isnumber=33070&punumber=10409&k2dockey=1553571@ieeecnfs`.

42. Nagappan, N. and T. Ball, "Static analysis tools as early indicators of pre-release defect density", *ICSE*, pp. 580–586, 2005, `http://doi.acm.org/10.1145/1062558`.

43. Brooks, F. P., *The Mythical Man-Month, Anniversary edition*, Addison-Wesley, 1995.

44. Moser, R., W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency

of change metrics and static code attributes for defect prediction", *ICSE '08: Proceedings of the 30th international conference on Software engineering*, May 2008, http://portal.acm.org/citation.cfm?id=1368088.1368114.

45. SOFTLAB, "Prest: Pre-Test Defect Prediction Tool", Available from: http://svn.cmpe.boun.edu.tr/svn/softlab/prest/trunk/Executable/PrestTool.rar.

46. Fenton, N. and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system", *Software Engineering, IEEE Transactions on*, Vol. 26, No. 8, pp. 797–814, 2000.

47. Fenton, N. E. and M. Neil, "A Critique of Software Defect Prediction Models", *IEEE Transactions on Software Engineering*, Vol. 25, No. 5, pp. 675–689, 1999, available from http://citeseer.nj.nec.com/fenton99critique.html.

48. Zhang, H., "On the Distribution of Software Faults", *Software Engineering, IEEE Transactions on*, Vol. 34, No. 2, pp. 301–302, 2008.

49. Andersson, C. and P. Runeson, "A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems", *Software Engineering, IEEE Transactions on*, Vol. 33, No. 5, pp. 273–286, 2007.

50. Koru, A. G. and J. Tian, "An empirical comparison and characterization of high defect and high complexity modules", *JSS*, Jan 2003, http://linkinghub.elsevier.com/retrieve/pii/S0164121202001267.

51. Koru, A. G. and H. Liu, "Identifying and characterizing change-prone classes in two large-scale open-source products", *JSS*, Jan 2007, http://linkinghub.elsevier.com/retrieve/pii/S0164121206001622.

52. Koru, A. and H. Liu, "Building effective defect-prediction models in practice", *IEEE Software*, Jan 2005, http://doi.ieeecomputersociety.org/10.1109/

MS.2005.149.

53. Ratzinger, J., T. Sigmund, and H. Gall, “On the relation of refactorings and software defect prediction”, *MSR ’08: Proceedings of the 2008 international workshop on Mining software repositories*, May 2008, http://portal.acm.org/citation.cfm?id=1370750.1370759.

54. Ohlsson, N. and H. Alberg, “Predicting fault-prone software modules in telephone switches”, *Software Engineering, IEEE Transactions on*, Vol. 22, No. 12, pp. 886 – 894, Dec 1996, http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=553637&isnumber=11972&punumber=32&k2dockey=553637@ieeejrns.

55. Basili, V., L. Briand, and W. Melo, “A validation of object-oriented design metrics as quality indicators”, *Software Engineering, IEEE Transactions on*, Vol. 22, No. 10, pp. 751 – 761, Oct 1996, http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=544352&isnumber=11896&punumber=32&k2dockey=544352@ieeejrns.

56. Subramanyan, R. and M. Krishnan, “Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects”, *IEEE Trans. Software Eng*, pp. 297–310, 2003.

57. Emam, K. E., S. Benlarbi, N. Goel, and S. Rai, “A Validation of Object-Oriented Metrics”, *National Research Council of Canada*, Jan 1999, https://iit-iti.nrc-cnrc.gc.ca/iit-publications-iti/docs/NRC-43607.pdf.

58. Schröter, A., T. Zimmermann, and A. Zeller, “Predicting component failures at design time”, *ISESE ’06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, Sep 2006, http://portal.acm.org/citation.cfm?id=1159733.1159739.

59. Jiang, Y., B. Cuki, T. Menzies, and N. Bartlow, “Comparing design and code

metrics for software quality prediction", *PROMISE '08: Proceedings of the 4th international workshop on Predictor models in software engineering*, May 2008, http://portal.acm.org/citation.cfm?id=1370788.1370793.

60. Zhao, M., 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*, Jan 1998, http://linkinghub.elsevier.com/retrieve/pii/S0950584998000986.

61. Fenton, N., M. Neil, W. Marsh, P. Hearty, L. Radlinski, and P. Krause, "Project Data Incorporating Qualitative Facts for Improved Software Defect Prediction", *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, pp. 2 – 2, Apr 2007, http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=4273258&isnumber=4273254&punumber=4273253&k2dockey=4273258@ieeecnfs.

62. S., A., T. Y., M. O., and K. T., "Constructing a Bayesian Belief Network to Predict Final Quality in Embedded System Development", *IEICE Transactions on Information and Systems*, Vol. E88-D, pp. 1134–1141, 2005.

63. Nagappan, N., B. Murphy, and V. Basili, "The influence of organizational structure on software quality: an empirical case study", *ICSE '08: Proceedings of the 30th international conference on Software engineering*, May 2008, http://portal.acm.org/citation.cfm?id=1368088.1368160.

64. Zimmermann, T. and N. Nagappan, "Predicting Subsystem Failures using Dependency Graph Complexities", *Software Reliability, 2007. ISSRE '07. The 18th IEEE International Symposium on*, pp. 227 – 236, Oct 2007, http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=4402214&isnumber=4402181&punumber=4402180&k2dockey=4402214@ieeecnfs.

65. Nagappan, N. and T. Ball, "Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study", *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pp. 364 – 373, Aug 2007, `http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=4343764&isnumber=4343709&punumber=4343708&k2dockey=4343764@ieeecnfs`.

66. Zimmermann, T. and N. Nagappan, "Predicting defects using network analysis on dependency graphs", *ICSE '08: Proceedings of the 30th international conference on Software engineering*, May 2008, `http://portal.acm.org/citation.cfm?id=1368088.1368161`.

67. Kudrjavets, G., N. Nagappan, and T. Ball, "Assessing the Relationship between Software Assertions and Faults: An Empirical Investigation", *Software Reliability Engineering, 2006. ISSRE '06. 17th International Symposium on*, pp. 204 – 212, Nov 2006, `http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=4021986&isnumber=4021952&punumber=4021951&k2dockey=4021986@ieeecnfs`.

68. Graves, T., A. Karr, J. Marron, and H. Siy, "Predicting fault incidence using software change history", *Software Engineering, IEEE Transactions on*, Vol. 26, No. 7, pp. 653 – 661, Jul 2000, `http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=859533&isnumber=18656&punumber=32&k2dockey=859533@ieeejrns`.

69. Arisholm, E. and L. Briand, "Predicting Fault-prone Components in a Java Legacy System", *5th ACM-IEEE International Symposium on Empirical Software Engineering (ISESE), Rio de Janeiro, Brazil, September 21-22*, 2006, available from `http://simula.no/research/engineering/publications/Arisholm.2006.4`.

70. Weyuker, E., T. Ostrand, and R. Bell, "Comparing negative binomial and recursive partitioning models for fault prediction", *PROMISE '08: Proceedings of*

the 4th international workshop on Predictor models in software engineering, May 2008, `http://portal.acm.org/citation.cfm?id=1370788.1370792`.

71. Weyuker, E. J., T. J. Ostrand, and R. M. Bell, "Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models", *to appear in Empirical Software Engineering Journal*, 2009.

72. Mockus, A. and D. Weiss, "Predicting Risk of Soft- ware Changes", *Bell Labs Technical Journal*, pp. 169–180, April-June 2000.

73. Tosun, A., B. Turhan, and A. Bener, "Direct and Indirect Effects of Software Defect Predictors on Development Lifecycle: An Industrial Case Study", *to appear in Proceedings of the 19th Interntational Symposium on Software Reliability Engineering*, 2008.

74. J., M. and K. T.M, "The Detection of Fault-Prone Programs", *EEE Transactions on Software Engineering*, Vol. 18, pp. 423–433, 1992.

75. Bullard, L., T. Khoshgoftaar, and K. Gao;, "An application of a rule-based model in software quality classification", *Machine Learning and Applications, 2007. ICMLA 2007. Sixth International Conference on*, pp. 204 – 210, Nov 2007, `http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=4457232&isnumber=4457184&punumber=4457183&k2dockey=4457232@ieeecnfs`.

76. T.M., K. and G. K., "Assessment of a Multi- Strategy Classifier for an Embedded Software System", *Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence*, pp. 651–658, 2006.

77. Khoshgoftaar, T. and E. Allen, "Predicting Fault-Prone Software Modules in Embedded Systems with Classification Trees", *HASE*, Jan 1999, `http://portal.acm.org/citation.cfm?id=645433.652890`.

78. Nagappan, N., “Toward a software testing and reliability early warning metric suite”, *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pp. 60 – 62, Apr 2004, `http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=1317422&isnumber=29176&punumber=9201&k2dockey=1317422@ieeecnfs`.

79. G. Denaro, M., Pezze, “An Empirical Evaluation of Fault-Proneness Models”, *Proceedings of International Conference on Software Engineering*, pp. 241–251, 2002.

80. G.I.Webb, J. Boughton, and Z. Wang, “Not So Naive Bayes: Aggregating One-Dependence Estimators”, *Machine Learning*, Vol. 58, No. 1, pp. 5–24, 2005, available from `http://www.csse.monash.edu.au/~webb/Files/WebbBoughtonWang05.pdf`.

81. Drummond, C. and R. C. Holte, “C4.5, class imbalance, and cost sensitivity: why under-sampling beats over-sampling”, *Workshop on Learning from Imbalanced Datasets II*, 2003.

82. Breiman, L., “Random Forests”, *Machine Learning*, pp. 5–32, October 2001.

83. Cohen, W., “Fast effective rule induction”, *ICML’95*, pp. 115–123, 1995, available on-line from `http://www.cs.cmu.edu/~wcohen/postscript/ml-95-ripper.ps`.

84. Quinlan, R., *C4.5: Programs for Machine Learning*, Morgan Kaufman, 1992, iSBN: 1558602380.

85. Holte, R., “Very Simple Classification Rules Perform Well on Most Commonly Used Datasets”, *Machine Learning*, Vol. 11, p. 63, 1993.

86. Brieman, L., “Bagging predictors”, *Machine Learning*, Vol. 24, No. 2, pp. 123–140, 1996.

87. Freund, Y. and R. Schapire, “A Decision-Theoretic Generalization of On-Line

Learning and an Application to Boosting", *JCSS: Journal of Computer and System Sciences*, Vol. 55, 1997.

88. Fenton, N. E. and S. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*, International Thompson Press, 1997.

89. Shepperd, M. and D. Ince, "A Critique of Three Metrics", *The Journal of Systems and Software*, Vol. 26, No. 3, pp. 197–210, September 1994.

90. Menzies, T., A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with Precision", *IEEE Transactions on Software Engineering*, September 2007, `http://menzies.us/pdf/07precision.pdf`.

91. Menzies, T., J. DiStefano, A. Orrego, and R. Chapman, "Assessing Predictors of Software Defects", *Proceedings, workshop on Predictive Software Models, Chicago*, 2004, available from `http://menzies.us/pdf/04psm.pdf`.

92. Menzies, T., J. S. D. Stefano, C. Cunanan, and R. M. Chapman, "Mining Repositories to Assist in Project Planning and Resource Allocation", *International Workshop on Mining Software Repositories*, 2004, available from `http://menzies.us/pdf/04msrdefects.pdf`.

93. Menzies, T. and J. S. D. Stefano, "How Good is Your Blind Spot Sampling Policy?", *2004 IEEE Conference on High Assurance Software Engineering*, 2003, available from `http://menzies.us/pdf/03blind.pdf`.

94. Menzies, T., J. D. Stefano, and M. Chapman, "Learning Early Lifecycle IVV Quality Indicators", *IEEE Metrics '03*, 2003, available from `http://menzies.us/pdf/03early.pdf`.

95. Stefano, J. D. and T. Menzies, "Machine Learning for Software Engineering: Case Studies in Software Reuse", *Proceedings, IEEE Tools with AI, 2002*, 2002, available from `http://menzies.us/pdf/02reusetai.pdf`.

96. Menzies, T., R. Lutz, and C. Mikulski, "Better Analysis of Defect Data at NASA", *SEKE03*, 2003, available from `http://menzies.us/pdf/03superodc.pdf`.

97. Menzies, T., J. S. DiStefeno, M. Chapman, and K. Mcgill, "Metrics that Matter", *27th NASA SEL workshop on Software Engineering*, 2002, available from `http://menzies.us/pdf/02metrics.pdf`.

98. Turhan, B. and A. Bener, "Software Defect Prediction: Heuristics for Weighted Naive Bayes", *Proceedings of the 2nd International Conference on Software and Data Technologies (ICSOFT'07)*, pp. 244–249, 2007.

99. Turhan, B. and A. Bener, "A Multivariate Analysis of Static Code Attributes for Defect Prediction", *Quality Software, 2007. QSIC '07. Seventh International Conference on*, pp. 231 – 237, Sep 2007, `http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=4385500&isnumber=4385459&punumber=4385458&k2dockey=4385500@ieeecnfs`.

100. Fagan, M., "Advances in software inspections", *IEEE Trans. on Software Engineering*, pp. 744–751, July 1986.

101. Shull, F., V. B. ad B. Boehm, A. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What We Have Learned About Fighting Defects", *Proceedings of 8th International Software Metrics Symposium, Ottawa, Canada*, pp. 249–258, 2002, available from `http://fc-md.umd.edu/fcmd/Papers/shull_defects.ps`.

102. Bhat, T. and N. Nagappan, "Building Scalable Failure-proneness Models Using Complexity Metrics for Large Scale Software Systems", *Software Engineering Conference, 2006. APSEC 2006. 13th Asia Pacific*, pp. 361 – 366, Dec 2006, `http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=4137438&isnumber=4137387&punumber=4137386&k2dockey=4137438@ieeecnfs`.

103. Blake, C. and C. Merz, "UCI Repository of machine learning databases", 1998, uRL: http://www.ics.uci.edu/~mlearn/MLRepository.html.

104. Zheng, J., L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk, "On the value of static analysis for fault detection in software", *Software Engineering, IEEE Transactions on*, Vol. 32, No. 4, pp. 240 – 253, Apr 2006, http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=1628970&isnumber=34170&punumber=32&k2dockey=1628970@ieeejrns.

105. Menzies, T., D. Raffo, S. on Setamanit, Y. Hu, and S. Tootoonian, "Model-based Tests of Truisms", *Proceedings of IEEE ASE 2002*, 2002, available from http://menzies.us/pdf/02truisms.pdf.

106. Chapman, M. and D. Solomon, "The Relationship of Cyclomatic Complexity, Essential Complexity and Error Rates", 2002, proceedings of the NASA Software Assurance Symposium, Coolfont Resort and Conference Center in Berkley Springs, West Virginia. Available from http://www.ivv.nasa.gov/business/research/osmasas/conclusion2002/Mike_Chapman_The_Relationship_of_Cyclomatic_Complexity_Essential_Complexity_and_Error_Rates.ppt.

107. "Polyspace Verifier®", 2005, \url{http://www.di.ens.fr/~cousot/projects/DAEDALUS/synthetic_summary/POLYSPACE/polyspace-daedalus.htm}.

108. Hall, G. and J. Munson, "Software evolution: code delta and code churn", *Journal of Systems and Software*, pp. 111 – 118, 2000.

109. Nikora, A. and J. Munson, "Developing Fault Predictors for Evolving Software Systems", *Ninth International Software Metrics Symposium (METRICS'03)*, 2003.

110. Khoshgoftaar, T., "An Application of Zero-Inflated Poisson Regression for Software Fault Prediction", *Proceedings of the 12th International Symposium on Soft-*

ware Reliability Engineering, Hong Kong, pp. 66–73, Nov 2001.

111. Khoshgoftaar, T. and N. Seliya, "Comparative Assessment of Software Quality Classification Techniques: An Empirical Case Study", *Empirical Software Engineering*, Vol. 9, No. 3, pp. 229–257, 2004.

112. Tang, W. and T. M. Khoshgoftaar, "Noise Identification with the k-Means Algorithm", *ICTAI*, pp. 373–378, 2004, `http://doi.ieeecomputersociety.org/10.1109/ICTAI.2004.93`.

113. Khoshgoftaar, T. M. and N. Seliya, "Fault Prediction Modeling for Software Quality Estimation: Comparing Commonly Used Techniques", *Empirical Software Engineering*, Vol. 8, No. 3, pp. 255–283, 2003, `http://dx.doi.org/10.1023/A:1024424811345`.

114. Menzies, T., J. D. Stefano, K. Ammar, K. McGill, P. Callis, R. Chapman, and D. J, "When Can We Test Less?", *IEEE Metrics'03*, 2003, available from `http://menzies.us/pdf/03metrics.pdf`.

115. Porter, A. and R. Selby, "Empirically Guided Software Development Using Metric-Based Classification Trees", *IEEE Software*, pp. 46–54, March 1990.

116. Tian, J. and M. Zelkowitz, "Complexity Measure Evaluation and Selection", *IEEE Transaction on Software Engineering*, Vol. 21, No. 8, pp. 641–649, August 1995.

117. Khoshgoftaar, T. and E. Allen, "Model Software Quality with Classification Trees", Pham, H. (editor), *Recent Advances in Reliability and Quality Engineering*, pp. 247–270, World Scientific, 2001.

118. Srinivasan, K. and D. Fisher, "Machine Learning Approaches to Estimating Software Development Effort", *IEEE Trans. Soft. Eng.*, pp. 126–137, February 1995.

119. Rakitin, S., *Software Verification and Validation for Practitioners and Managers, Second Edition*, Artech House, 2001.

120. Nagappan, N., T. Ball, and A. Zeller, “Mining metrics to predict component failures”, *ICSE ’06: Proceedings of the 28th international conference on Software engineering*, May 2006, `http://portal.acm.org/citation.cfm?id=1134285.1134349`.

121. Nachiappan Nagappan, V. B., Brendan Murphy, “The Influence of Organizational Structure on Software Quality: An Empirical Case Study”, *ICSE’08*, 2008.

122. Guyon, I., A. Elisseefi, and L. Kaelbling, “An Introduction to Variable and Feature Selection”, *Journal of Machine Learning Research*, Vol. 3, No. 7-8, pp. 1157–1182, 2003.

123. chung Liu, A. Y., *The Effect of Oversampling and Undersampling on Classifying Imbalanced Text Datasets*, Master’s thesis, Graduate School, University of Texas at Austin, 2004, available from `http://www.lans.ece.utexas.edu/~aliu/papers/aliu_masters_thesis.pdf`.

124. Witten, I. H. and E. Frank, *Data mining. 2nd edition*, Morgan Kaufmann, Los Altos, US, 2005.

125. Kamei, Y., A. Monden, S. Matsumoto, T. Kakimoto, and K.-i. Matsumoto, “The Effects of Over and Under Sampling on Fault-prone Module Detection”, *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pp. 196–204, 20-21 Sept. 2007.

126. Seliya, N., T. Khoshgoftaar, and S. Zhong, “Analyzing software quality with limited fault-proneness defect data”, *High-Assurance Systems Engineering, 2005. HASE 2005. Ninth IEEE International Symposium on*, pp. 89 – 98, Sep 2005, `http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=1581286&isnumber=33403&punumber=10554&k2dockey=1581286@ieeecnfs`.

127. E, M., G. Dinakaran, and N. Mosley, “How Valuable is it for a Web company to Use a Cross-company Cost Model, Compared to Using Its Own Single-company

Model?", *16th International World Wide Web Conference, Banff, Canada, May 8-12*, 2007, available from `http://www2007.org/paper326.php`.

128. Abrahamsson, P., R. Moser, W. Pedrycz, A. Sillitti, and G. Succi, "Effort Prediction in Iterative Software Development Processes – Incremental Versus Global Prediction Models", *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pp. 344–353, 2007.

129. MacDonell, S. and M. Shepperd, "Comparing Local and Global Software Effort Estimation Models – Reflections on a Systematic Review", *Empirical Software Engineering and Measurement, ESEM 2007*, pp. 401–409, 2007.

130. Kitchenham, B. A., E. Mendes, and G. H. Travassos, "Cross- vs. Within-Company Cost Estimation Studies: A Systematic Review", *IEEE Transactions on Software Engineering*, pp. 316–329, May 2007.

131. Premraj, R. and T. Zimmermann, "Building Software Cost Estimation Models using Homogenous Data", *Empirical Software Engineering and Measurement*, Jan 2007, `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4343767`.

132. Boehm, B., "Safe and Simple Software Cost Analysis", *IEEE Software*, pp. 14–17, September/October 2000, available from `http://www.computer.org/certification/beta/Boehm_Safe.pdf`.

133. L.Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web", Technical report, Stanford Digital Library Technologies Project, 1998.

134. Alpaydin, E., *Introduction to Machine Learning*, The MIT Press, October 2004.

135. Zhang, H. and S. Sheng, "Learning weighted naive Bayes with accurate ranking", *Proc. 4th IEEE Int. Conference on Data Mining*, pp. 567–570, 2004.

136. Hall, M., "A decision tree-based attribute weighting filter for naive Bayes",

Knowledge-Based Systems, Vol. 20, pp. 120–126, 2007.

137. Mladenic, D. and M. Grobelnik, “Feature Selection for Unbalanced Class Distribution and Naive Bayes”, *Proc. Sixteenth International Conference on Machine Learning*, pp. 258–267, Morgan Kaufmann Publishers, 1999.

138. Boetticher, G., T. Menzies, and T. Ostrand, “The PROMISE Repository of Empirical Software Engineering Data”, 2007, `http://promisedata.org/repository`.

139. Menzies, T., B. Turhan, A. Bener, and J. Distefano, “Cross- vs within-company defect prediction studies”, Technical report, Computer Science, West Virginia University, 2007.

140. Zhang, H. and X. Zhangu, “Comments on ’Data Mining Static Code Attributes to Learn Defect Predictors” ’, *IEEE Transactions on Software Engineering*, September 2007.

141. Mann, H. B. and D. R. Whitney, “On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other”, *Ann. Math. Statist.*, Vol. 18, No. 1, pp. 50–60, 1947, available on-line at `http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&handle=euclid.aoms/1177730491`.

142. Demsar, J., “Statistical Comparisons of Clasifiers over Multiple Data Sets”, *Journal of Machine Learning Research*, Vol. 7, pp. 1–30, 2006, avaliable from `http://jmlr.csail.mit.edu/papers/v7/demsar06a.html`.

143. Lewis, D., “Naive (Bayes) at Forty: The Independence Assumption in Information Retrieval”, *Proc. 10th European Conf. on Machine Learning*, pp. 4–15, Springer Verlag, London, 1998.

144. E. Frank, M. H. and B. Pfahringer, “Locally weighted naive Bayes”, *Proc. of the Uncertainty in Artificial Intelligence Conference*, pp. 249–256, Morgan Kauf-

mann, 2003.

145. Auer, M., A. Trendowicz, B. Graser, E. Haunschmid, and S. Biffl, “Optimal Project Feature Weights in Analogy Based Cost Estimation: Improvement and Limitations”, *IEEE Trans. on Software Eng.*, Vol. 32, pp. 83–92, 2006.

146. Domingos, P. and M. J. Pazzani, “On the Optimality of the Simple Bayesian Classifier under Zero-One Loss”, *Machine Learning*, Vol. 29, No. 2-3, pp. 103–130, 1997, `citeseer.ist.psu.edu/domingos97optimality.html`.

147. Kutlubay, O., B. Turhan, and A. B. Bener, “A Two-Step Model for Defect Density Estimation”, *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on*, pp. 322 – 332, Jul 2007, `http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=4301095&isnumber=4301046&punumber=4301045&k2dockey=4301095@ieeecnfs`.

148. Chen, Z., T. Menzies, D. Port, and B. Boehm, “Feature subset selection can improve software cost estimation accuracy”, *PROMISE ’05: Proceedings of the 2005 workshop on Predictor models in software engineering*, pp. 1–6, ACM, New York, NY, USA, 2005.

149. Dekhtyar, A., J. H. Hayes, and T. Menzies, “Text is Software Too”, *International Workshop on Mining Software Repositories (submitted)*, 2004, available from `http://menzies.us/pdf/04msrtext.pdf`.

150. Menzies, T., “Practical Machine Learning for Software Engineering and Knowledge Engineering”, *Handbook of Software Engineering and Knowledge Engineering*, World-Scientific, December 2001, available from `http://menzies.us/pdf/00ml.pdf`.

151. Hayes, J. H., A. Dekhtyar, and S. K. Sundaram, “Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods”, *IEEE Trans. Software Eng*, Vol. 32, No. 1, pp. 4–19, 2006, `http://doi.ieeecomputersociety.org/10.`

1109/TSE.2006.3.

152. John, G. and P. Langley, “Estimating continuous distributions in Bayesian classifiers”, *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence Montreal, Quebec: Morgan Kaufmann*, pp. 338–345, 1995, available from http://citeseer.ist.psu.edu/john95estimating.html.

153. Basili, V., F. Shull, and F. Lanubile, “Building Knowledge through Families of Experiments”, *IEEE Transactions on Software Engineering*, Vol. 25, No. 4, pp. 456–473, 1999.

154. Kolodner, J., “Improving Human Decision Making Through Case-Based Decision Aiding”, *AI Magazine*, p. 68, Summer 1991.

155. Kolodner, J., *Case-Based Reasoning*, Morgan Kaufmann, 1993.

156. Compton, P., L. Peters, G. Edwards, and T. Lavers, “Experience with Ripple-Down Rules”, *Knowledge-Based Systems*, Vol. 19, No. 5, pp. 356–362, September 2006, available from http://www.cse.unsw.edu.au/~compton/#Starter_Papers.