DEFECT PREDICTION IN EMBEDDED SOFTWARE SYSTEMS: CASCADING NAÏVE BAYES ALGORITHM WITH CROSS- VS WITHIN-COMPANY DATA

by

Ayşe Tosun

B.S., Computer Science and Engineering, Sabancı University, 2006

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

Graduate Program in Computer Engineering
Boğaziçi University

2008

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor Assistant Professor Ayşe Bener for her knowledge and encouragement during completion of this thesis. It would have been impossible to produce such comprehensive work without her guidance and patience.

I would also like to thank Professor Ethem Alpaydın and Associate Professor Necati Aras for being in my thesis committee. I am grateful to Professor Ethem Alpaydın who helped me clarifying ideas about machine learning algorithms during my course of study in the masters program.

I would like to express my special thanks to Burak Turhan, who is a member of Software Engineering Research Laboratory, for helping me whenever I needed help and sharing his valuable ideas with me.

I am grateful to many scholars from software engineering research community for giving me inspiration through their creative models, ideas and research results. Public data repositories such as Promise and NASA MDP Program have been very useful for providing data to the research community so that we were able to reproduce refute and improve the findings of other researchers.

My family deserves special thanks, especially my sister Merve, for her gentle support and love when my mood was up and down throughout my research.

I would like to thank Arçelik A.Ş. who provided new datasets to our research for nearly two years under Arçelik-Boğaziçi University Joint Graduate Research Program.

I would also like to thank Turkish Scientific Research Council (TÜBİTAK) who provided financial support for my graduate study. Finally I would like to thank Boğaziçi University Research Fund that support this research in part under the grant number BAP 06HA104 and TÜBİTAK under grant number EEEAG 108E014.

ABSTRACT

DEFECT PREDICTION IN EMBEDDED SOFTWARE SYSTEMS: CASCADING NAÏVE BAYES ALGORITHM WITH CROSS- VS WITHIN-COMPANY DATA

As mobile technologies advance and become part of our everyday life, we need nomadic access to information through intelligent and interoperable devices. Hence there is an increasing demand for intelligent embedded systems. The intelligence comes from the software that runs on them, therefore, the current problems in software engineering also hold true for embedded systems domain.

Embedded systems industry has its own unique challenges as well: tough competition and tight profit margins. The industry constantly seeks for creative solutions to improve existing processes, to increase the quality of the product, and to lower the costs. Since the embedded software increasingly dominates the end product, any improvement in software development lifecycle would bring tremendous benefit to the industry. The most costly and time consuming process area in software development is testing. Practitioners need oracles to help them decide how to allocate their limited time and effort effectively without affecting the quality of their embedded software. These oracles are basically learning-based predictive models that aim to provide effective and robust methodologies for testing phase by focusing on defect-prone parts of the software.

In this research, we propose a software defect prediction model for embedded software by analyzing specific characteristics of embedded systems. We employ a cascading learning mechanism to increase the prediction performance of the model by using the state of the art machine learning algorithm for software defect prediction.

We have examined the three pillars of defect prediction research and its practical challenges for embedded software domain: a) improving the prediction performance of the model, b) analysis of data collection effort and cost, and c) increasing the information content of data used in the model.

ÖZET

GÖMÜLÜ YAZILIMLARDA HATA TAHMİNİ: ŞİRKET İÇİ VE ŞİRKET DIŞI VERİLERLE AŞAMALI NAİVE BAYES

Mobil teknolojiler günlük yaşamımızın birer parçası haline geldikçe, bilgiye erişmek için ihtiyaç duyulan akıllı gömülü sistemlere olan talep hızla artmaktadır. Bu tip sistemlerdeki zeka faktörü, içlerindeki yazılımda geliştirilmektedir. Dolayısıyla, yazılım mühendisliğinin güncel problemleri, gömülü sistemler için de geçerli hale gelmiştir.

Gömülü sistem endüstrisinin, sert rekabet koşulları ve kısıtlı kar payı gibi kendine özgü problemleri bulunmaktadır. Endüstri, varolan süreçlerini iyileştirmek, ürünün kalitesini arttırmak ve maliyetini düşürmek için yaratıcı çözümler aramaktadır. Gömülü yazılım, son ürüne her geçen gün daha fazla hükmettiğinden, yazılım geliştirme döngüsündeki herhangi bir gelişme, endüstriye büyük faydalar sağlayacaktır.

Yazılım geliştirmede en masraflı ve fazla zaman alan aşama test sürecidir. Uzmanlar, yazılımın kalitesini bozmadan, sınırlı zaman ve eforlarını verimli bir şekilde kullanabilmek için, yeni çözümlere ihtiyaç duyarlar. Bu çözümler, öğrenmeye dayalı tahmin modelleridir. Amaç, yazılımdaki hatalı parçaları tespit ederek test süreci için etkili ve hızlı yöntemler sunmaktır.

Bu araştırmada, gömülü sistemlerin karakteristik özellikleri analiz edilerek, gömülü yazılımlara uygun bir hata tahmini modeli önerilmiştir. Hata tahmininde uygulanan en güncel makine öğrenme algoritması kullanılarak, aşamalı bir öğrenme mekanizması geliştirilmiş, böylece, modelin hata yakalama performansının arttırılması hedeflenmiştir.

Hata tahmini araştırması üç açıdan ele alınarak, gömülü yazılımlar alanındaki uygulanabilirliği araştırılmıştır. Temel aşamalar a) modelin performasını iyileştirmek, b) veri toplama maliyetini azaltmak ve c) verinin içeriğini arttırmak şeklinde özetlenebilir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

a	Static code attributes
bal	Balance
C	Class label of a module
d	Defect information
D	Defect matrix
FN	False negatives
FP	False positives
L	Software library
m	Number of modules
n	Number of attributes
pd	Probability of detection
pf	Probability of false alarm
S	Software system
TP	True positives
X	Attributes
θ	Confidence threshold
σ	Standard deviation
μ	Mean
ARM	Automated Requirements Management
CMMI	Capability Maturity Model Integration
CC	Cross-company
CNB	Cascading Naïve Bayes
MDP	Metrics Data Program
NB	Naïve Bayes
ROC	Receiver operator curve
VV&T	Verification, Validation and Testing
WC	Within-company

1. INTRODUCTION

A simple definition of quality can be "the degree to which a system, component, or process meets specified requirements" [1]. Software quality is interpreted differently in various studies. Fenton *et al.* examine the software quality in two different viewpoints, namely internal and external quality [2]. The first, internal quality view involves the control of software in terms of proper development processes and well-defined schedules [2]. The second, external quality view is often known as *quality-in-use* [2]. It measures software's characteristics from the user's perception of quality such that the product is defect-free and consistent with requirements [2]. External product quality is also defined as *quality of conformance* which measures conformance to requirements by investigating defects in the software [3].

Software quality is closely related to verification, validation and testing (VV & T) activities during software development lifecycle [4]. Many software quality attributes are defined such as reliability, usability, efficiency, transportability, maintainability and testability to ensure that the software has a good quality [4]. Quality attributes vary with respect to the type of system that software has been produced. Our work focuses on embedded software which has typical characteristics different than general notion of software. The distinguishing characteristic of embedded software is that they are small parts of a larger system [5]. Their principal role is the interaction with the physical world they are embedded [43]. They simultaneously react to multiple requests without any failures [43]. These requests, regularly or irregularly, occur from heterogeneous technologies such as various hardware and software designs [43]. Therefore, timeliness, liveness, concurrency, reactivity, heterogeneity and reliability are essential quality attributes of embedded software [43].

Rather than measuring good quality by observing quality attributes, poor quality of software can be easily observed with malfunctioning of the program or defects. To ensure the overall quality, each stage of the development process should be involved into VV activities [4]. However, it is very hard to manage this due to tight schedules and continuous modifications in project requirements. Traditional software development lifecycle

prioritizes the testing phase as the most critical to verify that the software is adequate and consistent with its requirements [4]. However, testing phase is very expensive and time consuming such that nearly 50 per cent of the project schedule is assigned to testing activities [6]. Different VV & T strategies, such as inspections, reviews, manual and automated methods, are proposed to use the time and resources effectively [4]. Among these, defect prediction models are helpful tools which operate static analysis on the software. Recent research proves that defect prediction models outperform other VV & T techniques by increasing the detection rate of actual defects in the software [7]. Rather than exhaustive testing, these automated oracles guide the user to specific parts of the software, which are more likely to fail. They use software metrics and defect information of past projects to calibrate their model and mark specific parts of a new project as defect-prone [5]. With the help of such tools, quality of software would be achieved; managers would allocate their resources more effectively and testing time would reduce.

1.1. Motivation

Embedded systems have become essential elements of many industries, including manufacturing, automotive, telecommunications and aerospace. Previously, software in these systems was only used for controlling the hardware [8]. However, the purpose of embedded systems has extended as a result of the increase in the demand and the competition in the market [8]. This increase in demand makes embedded software more important and popular than the hardware in these systems. Embedded software continuously interacts with the hardware and other resources to achieve real-time processing of the overall system [1, 8]. They are deployed pervasively once they are developed [8].

The increase in the demand also leads to increase in the size and complexity of embedded software [8]. Although size and complexity of the software has been steadily growing, demand for low cost, shortened time and high quality have become critical issues [8]. This demand for tightened schedules due to increasing competition enforces limited testing, which may leave unidentified defects in the embedded software. On the other hand, embedded systems must have highly reliable software, because defects in the field would be severe, extremely expensive and they would cause the failure of the overall system [5].

Therefore, developers need additional technologies to preserve the characteristics of embedded software such as reliability and ensure about the quality of their software without increasing time and resources.

Defect prediction models are helpful tools among other verification, validation and testing (VV&T) activities to catch as many defects as possible in the software [7]. These models reduce the amount of time required for thorough testing by guiding the developers through defect-prone parts in the software. With the help of defect predictors, managers would reduce testing resources and preserve the quality of their software at the same time. Defect prediction models are practical for embedded software systems to detect defective segments in software with a shorter time and fewer resources [9, 10]. They would help the companies survive against the tough competition and tight schedules in the embedded systems industry. Therefore, one of our motivations in this research is to model a successful defect predictor to solve current problems of embedded software. We have selected a learning-based methodology for our defect prediction model to detect defective modules in software as much as possible by learning from past projects' data.

Defect prediction research has often aimed to tackle three aspects of the practical software development problems. These three aspects are improving the performance of the defect prediction model, reducing the cost required to collect data for the model and the information content of the data used in these studies. We have studied all three aspects of defect prediction research to propose practical solutions for embedded software systems.

First aspect of defect prediction research is to construct a better defect predictor which catches all defects in the software with almost no false alarms. Software developers in the companies expect to use defect predictors which would guide them to actual defective parts in order to use their limited resources effectively. Therefore, our motivation has initially been defined as modeling a successful defect predictor for embedded software which has higher detection rates. Although previous research has proposed high detection rates for embedded software systems, it has a cost of high false alarms [10]. Therefore, we have re-designed our experiments to construct a defect prediction model for embedded software whose detection performance is higher while presenting lower false alarm rates.

We have investigated the second aspect of defect prediction research to apply it for embedded software systems. We have observed that time required to collect enough data on past projects within a company may be prohibitive. Developers are often too busy because of limited project schedules or they are less willing to collect defect data from completed projects. Second motivation of our research aims to reduce the data collection time and effort within a company by using data from other organizations. We aim to observe practical usage of cross-company data for defect prediction studies in embedded software systems.

Finally, we have observed *data* used in defect prediction studies to evaluate the information content of the size and complexity metrics. Recently, researchers proved that software metrics contain limited information such that performance of many algorithms could not go further with these metrics [12]. Our last motivation tries to assess the performance of defect predictors in embedded software using additional metrics. We believe that software systems that operate in the same domain should meet similar requirements. Embedded software systems, for instance, need to meet certain requirements in their application such as performance, time constraints and interaction between software and hardware. Defects related with weaknesses on those requirements can be shared with other embedded software. Therefore, we have investigated the performance of our defect prediction model by increasing the information content of cross-company embedded software data.

1.2. Outline

We give brief background information on software defect prediction with general concepts on the procedure, the proposed techniques and a discussion on the current trend of defect prediction research in the next section. We combine related defect prediction studies with embedded systems. Then, we conclude this section with related machine learning techniques that provide a new direction to our research.

Section 3 presents the problem statement of this research with criticizing possible approaches and justifying our research questions.

In the fourth section, our proposed model with the underlying mechanism is explained for solving the problems defined in the previous section. Details of the model with its advantages and limitations are presented.

In Section 5, we describe data used in this research and differences between data collection methodology of our experiments. Later, experiments with a discussion on the design of each research question are explained.

We present statistical results of these experiments by discussing on what we propose and what results indicate in Section 6. We evaluate threats to the validity of our model, experiments and results.

Final section summarizes the research and the conclusions derived with the experiments. Discussion on results, contributions of the research and a list of future directions are clearly expressed.

2. BACKGROUND

Researchers have been working on defect prediction and embedded software for many years. In this section, previous research on these topics will be examined and discussed. First two sections will demonstrate the fundamental aspects of defect prediction such as software metrics and prediction models as well as the algorithms used in the literature. The last section will focus on embedded software and specialized defect prediction studies for embedded software.

2.1. Software Metrics

A software metric can be defined as "a quantitative measure of the degree to which a system, component, or process possesses a given attribute" [1]. Software metrics can be used to gather information about the quality or cost of the software. Improper collection of metrics can cause additional cost without providing meaningful information [13].

Three different categories for software metrics are described in [14]. First category of the metrics, called project metrics, are used to collect information about the project such as number of software developers and development cost, to adapt the project schedule and avoid development schedule delays. Process metrics, on the other hand, focus on monitoring the processes that produce the software. These metrics are specifically used for software process improvement methodologies such as CMMI to measure defect removal rates, response times for defect fixes and change requests [15]. Finally, product metrics measure the size, complexity, design and performance of the software [14]. In this study, product metrics, as known as "static code attributes", are used to predict defects in the software.

The earliest research which relates defects with product metrics was done by Akiyama [16]. Akiyama stated that defects in a software system can be predicted through lines of code, number of decisions and number of subroutine calls [16]. Later, Halstead proposed certain size metrics which measure the complexity of a program using number of unique operators and operands [17]. Halstead argues that readability of a module can be

described in terms of unique operators and operands instead of lines of code. If a module is hardly to read, then it is more likely to contain too many operators and operands, which also leads to increase in defect density. More researchers experienced on different size metrics until McCabe'c cyclomatic complexity has been proposed as an additional predictor for defects [18]. McCabe's cyclomatic complexity measures the number of linearly independent paths by forming a connected graph from each module. Both McCabe and Halstead metrics are module-based metrics, where a module can be a file, function or method in a software program. These metrics with additional quantitative attributes (lines of code, source lines of code, blank lines of code, etc.) are widely used as "static code attributes" in defect prediction studies [6, 7, 11, 12, 19, 20, 21, 22]. Therefore, we will use this term for product metrics in the rest of this study.

Some researchers reject that static code attributes can be an indicator for defective modules in software [23, 24]. On the other hand, many defect prediction researches based their models on these attributes and the performance of such predictors produce better results than previous analysis techniques such as manual code reviews or full-Fagan inspections [4, 7, 17, 18, 25]. Therefore, we also assume that available code metrics (Appendix A lists the description of these metrics) provide as much information as possible so far to make accurate predictions in many software systems.

Apart from static code attributes, there are studies which collect Chidamber and Kemerer's object-oriented design metrics for their defect prediction models [26, 27, 28]. These set of metrics measure unique aspects of the object-oriented approach, for which function-oriented views of traditional approaches are not appropriate. CK metrics suite consists of six metrics: Weighted methods per class, depth of inheritance tree, number of children, coupling between objects, response for a class and lack of cohesion in methods. Since most of the embedded software we use in this research are not developed using an object-oriented approach, these metrics are unsuitable for our study.

So far, we examined current research on software product metrics used for defect prediction. However, we need to relate those studies with the area of embedded software and ensure that static code attributes can be easily used as successful indicators of defects in embedded software. Recently, a study done by Marchenko and Abrahamsson examined

the usefulness of static code attributes on deciding defect rates of embedded software [20]. They concluded that static code analysis tools provide a practical way for estimating the defects in the embedded product. Additionally, previous research completed by Oral strengthens the usefulness of static code attributes on the performance of defect prediction in embedded systems [10]. Therefore, aforementioned studies led us to use available static code attributes for defect prediction modeling to measure product characteristics of embedded software.

2.2. Defect Prediction Models

A complete verification of software to ensure the correctness and consistency of every element is frequently obtained by a thorough testing, i.e. exhaustive testing, in software development lifecycle [4]. Exhaustive testing is generally the most expensive and time consuming phase of this development process [6]. According to [29, 30], testing activities to ensure the quality of software require nearly 50 per cent of the project schedule. Various techniques have been suggested so far to provide effective and practical ways for the testing phase [4]. These are manual code reviews, walk-throughs, inspections and automated tools such as defect prediction models [4]. According to various researches on evaluation of these techniques, defect predictors perform noticeably better in terms of catching defects in the software [7, 31, 32]. They guide developers to specific parts of the software where they are more likely to fail during execution. Defect predictors would help the managers to decrease testing times and allocate resource more effectively [33].

A variety of software defect prediction techniques have been proposed over the years [6, 7, 19, 21, 34, 35, 36]. Most of them have combined well known methodologies and algorithms from various engineering domains to predict the actual defects in the software as much as possible. The most common methods used in defect prediction studies are statistical techniques and machine learning models. They require past data from completed projects in terms of software metrics and actual defect rates. Software metrics are quantitative measures of software in terms of size and complexity [1]. On the other hand, defect rates represent the location and amount of defects found in software [35]. Prediction models usually combine these metrics and defect information to learn which modules seem to be more defect-prone. Based on the knowledge gathered from completed projects, the

model can classify modules of a new project as defective or defect free. Some defect prediction models in the literature and their applicability will be discussed in the following subsections.

2.2.1. Defect Prediction Using Statistical Techniques

Regression as a statistical methodology has been widely used in many defect prediction studies [26, 28, 37, 38]. Researchers use linear, multinomial, Poisson, negative binomial, zero-inflated negative binomial and logistic variations of regression in their studies [26, 28, 37, 38].

In [28], object-oriented (OO) metrics are validated as appropriate indicators for defective modules using multivariate logistic regression. Logistic regression is a standard technique that is based on maximum likelihood estimation (Equation (2.1)) [28]. Basili, Briand and Melo choose this technique not to compare their prediction model with other multivariate approaches, but to validate the relationship between OO metrics and defective modules [28]. In logistic regression, a curve between each OO metric (X_i) and the probability (π) that a class is defective is formed with the formula below [28]. If the curve approximates a horizontal line or a step function, then it indicates that the metric is not significant [28].

$$\pi(X_1, X_2, ..., X_n) = \frac{e^{(C_0 + C_1 X_{i1} + .. + C_n X_{in})\gamma_i}}{1 + e^{(C_0 + C_1 X_{i1} + .. + C_n X_{in})}} \tag{2.1}$$

Basili *et al.* use the likelihood Equation (2.1) to form the logistic regression ratio as follows [28]:

$$R^2 = \frac{LL_S - LL}{LL_S} \tag{2.2}$$

Cartwright and Shepperd analyzed an object-oriented subsystem from a telecommunication product with a linear regression technique [26]. The main aim is once more to observe the relationship between defect densities and classes that represent object-

oriented characteristics such as inheritance. Their conclusion supports their claim that object-oriented metrics such as inheritance and polymorphism has a high correlation with defect densities.

Another study done in telecommunication software systems compares different regression techniques: Poisson, negative binomial and zero inflated negative binomial [38]. They use CK metrics [27] to investigate the relation between object-oriented design metrics and defects in software systems. Poisson distribution for a dependent variable y, and a vector of n independent variables $x=(x_1,\ldots,x_n)$ is given by

$$\Pr(y|x) = \frac{e^{-\mu}\mu^{y}}{y!} \tag{2.3}$$

where μ is the conditional mean value of dependent variable [38]. Generally, heterogeneity of data causes failures in Poisson distribution [38]. Moreover, dispersion of data affect overestimated statistical significance of the predictors [38].

Janes *et al.* propose negative binomial and zero inflated negative binomial regression models to handle the dispersion of data [38]. The resulting distribution for negative binomial regression is a combination of Poisson and probability distribution, as seen in Equation (2.4) [38].

$$P(y|x) = \int_{0}^{\infty} \left[\Pr(y|x,\delta)\Pr(\delta)\right] d\delta \tag{2.4}$$

Additionally, zero inflated models take into account the occurrences of zero counts [38]. Zero inflated negative binomial regression shows the best ability to describe the variability of the number of defects in classes of software [38].

Finally, Ostrand, Weyuker and Bell propose negative binomial regression to predict the expected number of faults in each file of the next release of a telecommunication system [37]. They collect file-based metrics such as number of lines, total number of faults and fault densities for each stage in development lifecycle, from each release of the system

[37]. Their results prove that files predicted as defective with their model actually contain more than 70 per cent of defects in the system [37]. On the other hand, it is hard to apply such a model due to lack of release history and file metrics.

All of the abovementioned studies examine the relation between software metrics and defects in different software systems. They do not propose a defect prediction model that aims to perform the best among others. Instead, they fit simple regression techniques to strengthen their claims about the relation between metrics and defects. Moreover, these prediction models have only local significance such that every developer should consider building their own prediction models [26]. Local prediction models, on the other hand, are practically not useful and they are costly since they are hardly to fit other environments.

2.2.2. Defect Prediction Using Bayesian Belief Networks

Machine learning techniques based on Bayesian probability have been proposed in various studies of defect prediction [34, 35]. Bayesian Belief Network (BBN) is one of these techniques, which is derived from Bayesian Theory [39]. It is a special type of graphical network that specifies the probabilistic relationships among variables [35]. It is simply a graph where nodes represent uncertain variables and arcs represent causal relationship between nodes. It is advantageous since it solves complex relationships with simple probabilistic statements and operates in the case of missing data [35].

Fenton *et al.* constructed their BBN that is specific to software systems where nodes represent different schools of thought that affect defect prediction problem such as “problem complexity”, “testing effort” and “defect density at testing” [34, 35]. Probabilistic relationships of these attributes help to predict defects in software systems. Although BBN is an effective method for software defect prediction, it can be used only when development processes of organizations are mature enough to collect such metrics [34]. Variables in the network are described with states such as “very low”, “low”, “medium”, “high” and “very high” [34]. Participants from the organizations rate these variables based on their opinion about development process of the organization. Since this kind of data collection may depend on the opinions of software development team, it raises the question of consistency of data as well as generalization of the results. Moreover, the

constructed network for one software system may not be applicable to another because of the difference in states of variables.

2.2.3. Defect Prediction Using Naïve Bayes Theory

Among the prediction models that use machine learning approaches, Naïve Bayes has been widely used and it performed as the best [6, 7, 11, 19, 22]. It is derived from Bayes' Theorem (Equation (2.5)), where the posterior probability of a class given the attributes is computed with prior probability of the class and likelihoods of attributes [39].

$$P(C\left|X_i\right.) = \frac{P(C)}{P(X)} \prod P(X_i\left|C\right.) \tag{2.5}$$

Attributes of defect prediction are software metrics, specifically static code attributes, whereas the classes are described as either "defective" or "defect-free". Therefore, Naïve Bayes computes the posterior probabilities of a module being in a defective or defect-free class, given static code attributes such as lines of code, McCabe complexity metrics and Halstead attributes [17, 18]. It then assigns the module to the class (defective or defect-free) that gives the highest posterior probability. The process is very straightforward such that it simply uses attribute probabilities (likelihoods) derived from historical data to make predictions [7, 39]. In order to compute likelihoods, normal (Gaussian) probability distribution function is used as follows;

$$f(x) = \frac{1}{\sqrt{2\pi}\delta} e^{\frac{-(x-\mu_i)^2}{2\delta^2}} \tag{2.6}$$

where the attributes (x_i) are assumed as independent, off-diagonals of shared covariance matrix (δ) are zero and there are distinct mean values (μ) for each attribute [39].

In many studies, Naïve Bayes predictor has been compared with machine learning algorithms such as decision trees, rule inductions, nearest neighbor, regression techniques and artificial neural networks [7, 21, 40, 41]. For example, Challagula *et al.* investigate

eleven different algorithms from statistical models, machine learning approaches and rule inductions for predicting number of defects in various public datasets [21]. Their study presents that Naïve Bayes and 1- Rule (1R) methods achieve the best two performances over all datasets.

A multivariate analysis based on Bayesian methods has been proposed by Turhan and Bener [19]. Their research contribution includes using multivariate approaches on static code attributes rather than univariate methods [19]. Univariate methods assume independence of attributes. In contrary, multivariate approaches take into account the correlations between attributes. They are more complex models compared to univariate models [19]. Authors compare quadratic discriminant, linear discriminant and Naïve Bayes with the best feature selection techniques. In majority of the results, Turhan and Bener once more validate the success of Naïve Bayes in defect prediction studies [19].

Menzies *et al.* focus on preprocessing of numeric attributes to increase the performance of Naïve Bayes data miner for defect prediction [7]. They prove the usefulness of static code attributes in defect prediction by emphasizing how attributes are used rather than which attributes are required [7]. With certain manipulations such as log filtering and a feature selection technique, i.e. InfoGain, they significantly increase the performance of Naïve Bayes, in terms of catching the actual defects of software [7].

According to Menzies *et al.*, Bayesian method with log filtering data works better than rule based methods because of two reasons [7]. First explanation could be the ability of this method to capture the relation between defects and attributes in a log-normal way [7]. Among all of the experiments in their work, Bayesian technique uses this association effectively. Secondly, Naïve Bayes collects signals from multiple attributes which have similar information content [7]. Therefore, it can combine several indicators from the attributes to increase the performance of prediction.

Although Menzies et al also analyze the effectiveness of a feature selection technique, they do not focus on whether each static code attribute has equal importance on measuring defects [7]. Therefore, Turhan and Bener propose various heuristics to assign weights to those attributes according to their importance [6]. They refer to three feature

selection and reduction techniques, GainRatio, InfoGain and PCA, to assign weights to likelihood probabilities of attributes with the Equation (2.7) [6].

$$P(C_i \mid X) = P(X \mid C_i)^{w_i} P(C_i) \tag{2.7}$$

Their results show that weighting attributes according to their importance outperformed simple Naïve Bayes. On the other hand, selection of the best heuristic is highly dependent on the software system [6].

Research in defect prediction consists of many studies that investigate better algorithms to increase the performance of catching defects in various software domains [11, 22]. Although exploring new algorithms can be one direction for future studies, it may not be productive and the attempts may fail to improve prediction research done so far [22].

One of the approaches that investigate new perspectives for defect prediction has been completed by Menzies *et al.* [22]. They inspect *data* used in defect prediction studies [22]. Collecting data is a very time-consuming and difficult task for organizations. Rapid changes in technologies also make it harder to use past projects, since they are not able to represent current practices [42]. Therefore, authors report an analysis that searches the usefulness of cross-company data in defect prediction. They claim that static code attributes are not ambiguous such that they are easily collected from different software with automated tools [11]. Therefore, predictions based on the attributes of cross-company data should not present fictitious results [11].

Model used to derive the output of this analysis is based on Naïve Bayes classifier on log-filtered data. Authors discussed the design of their two experiments, one of which uses static code attributes collected from different organizations for building the model. The other experiment is a replication of [7] to compare within-company prediction results with cross-company analysis. Results provide useful information to the academia. Authors observe that cross-company data dramatically increases the probability of detecting defective modules [11]. However, the trade-off for high detection rates is high false alarms

[11]. Therefore, authors suggest using cross-company data when organizations are able to pay off the cost of false alarms as a result of increased detection rates [11].

Abovementioned studies bring another perspective to defect prediction with an emphasis on the ceiling effects of data mining methods [22]. There are many methods that use static code attributes as input and output predictions for different software systems [6, 7, 11, 17, 18, 19, 21, 22]. These approaches unfortunately are not able to capture new information from data to do better predictions than Naïve Bayes [22]. The hypothesis in the work of Menzies *et al.* claims that these methods hit a "performance ceiling", i.e. an upper bound on the amount of information collected with static code attributes [22]. Menzies *et al.* seek an explanation for this ceiling effect by predicting that static code attributes may have limited information content [22]. They prove that using very few training instances in data mining methods would be sufficient to achieve the best performances, since information content is limited [22]. Additionally, authors speculate on improving the information content of training data with different methods such as case-based learning and additional requirements metrics [22]. They support that using early lifecycle data, i.e. textual metrics from requirements documents, significantly improves defect prediction performance on public datasets [12].

2.3. Defect Prediction in the Area of Embedded Systems

We have examined available software metrics and proposed algorithms so far for software defect prediction with a discussion on their application and performance. We have investigated the applicability of software metrics, namely static code attributes, to defect prediction in embedded software. In this section will briefly describe embedded systems and summarize previous defect prediction research in this area with a discussion on the selection of the algorithm for our prediction model in embedded software.

2.3.1. Embedded Systems

Before going further on specialized prediction studies, we need to define what embedded software is and discuss its significance in many industries. We can, first, define software as a list of mathematical procedures where they convert input data into the format

of required output [43]. Generally, procedures and their abstract properties in software are more important than the mechanism that carries out them [43]. Embedded software, on the other hand, is different than general software descriptions such that its principal role is the interaction of software with the physical world [43]. They perform some of the requirements of the overall system by communicating with the hardware and other components [1]. Due to close interaction between embedded software and the physical world, they must acquire specific properties of the physical world it is surrounded with [43]. Therefore, the physical world where embedded software operates in plays a critical role while measuring the correctness of the overall system.

Correctness of embedded software is often measured with several quality attributes that are specific to those systems [43]. First, timeliness is essential in embedded systems due to the fact that the physical system, embedded software interacts with, evolves over time [43]. In addition, software must simultaneously react with a variety of hardware and network components to be concurrent and reactive [43]. It must not terminate or run into a deadlock (liveness), and must communicate with a mixture of hardware and software designs (heterogeneity) [43].

Embedded systems must have highly reliable software, because the consequences of failures in the software severely impact other parts of overall system [5]. Failures may not be tolerated and repairing them may be very expensive [5]. On the other hand, increasing competition in the area of embedded systems forces managers to tighten schedules and demand for low cost and high quality software [5, 8]. This creates a dilemma such that to lower the cost, testing should be limited; on the other hand, to have high quality product, test coverage should increase. It also dramatically affects quality attributes that are specific to embedded systems. Automated warning mechanisms such as defect prediction models are helpful tools for developers in the embedded domain to overcome the effects of competition in the market and to improve quality in their software. We will examine some studies that focus on application of defect prediction in embedded systems in the following section.

2.3.2. Defect Prediction in Embedded Systems

A specialized prediction model for embedded systems is investigated by Khosghoftaar *et al.*, where they built a classification and regression tree for predicting high risk software modules in telecommunications system software [5]. A classification tree is an algorithm depicted as a tree graph, where each node represents a decision from software metrics and each edge represents possible result of the decision [5]. Input object is a software module which will be classified as fault-prone or not fault-prone at the end of the tree. Tree is formed with the Classification and Regression Tree (CART) algorithm, which recursively partitions the set into two nodes until a stopping criterion [5]. Authors use different releases of the software and add new process and execution metrics that keeps not only changes in the code, but requirements that affect the change and time constraints for transactions [5]. Although results are quite satisfactory for the company, new metrics included in the model and the specialized datasets make the research hard to reproduce.

Another study constructs a Bayesian Belief Network that merges both software and hardware development of embedded systems [8]. They mainly reconstruct Bayesian Network of Fenton *et al.* [34, 35] and modify it by using embedded system development lifecycle. The base data comes from completed projects. Software metrics used in the network are specifically chosen in five groups to represent the relationship between development process and final software quality [8]. They are size metrics, effort metrics, detected faults, test items and residual faults [8]. Amasaki *et al.* first construct abstract development process for the embedded model and the general model. Then, they define each stage in the development process (design, review, software and acceptance test) with a directed acyclic graph whose nodes are software metrics. Final form of the network can be seen in Figure 2.1.

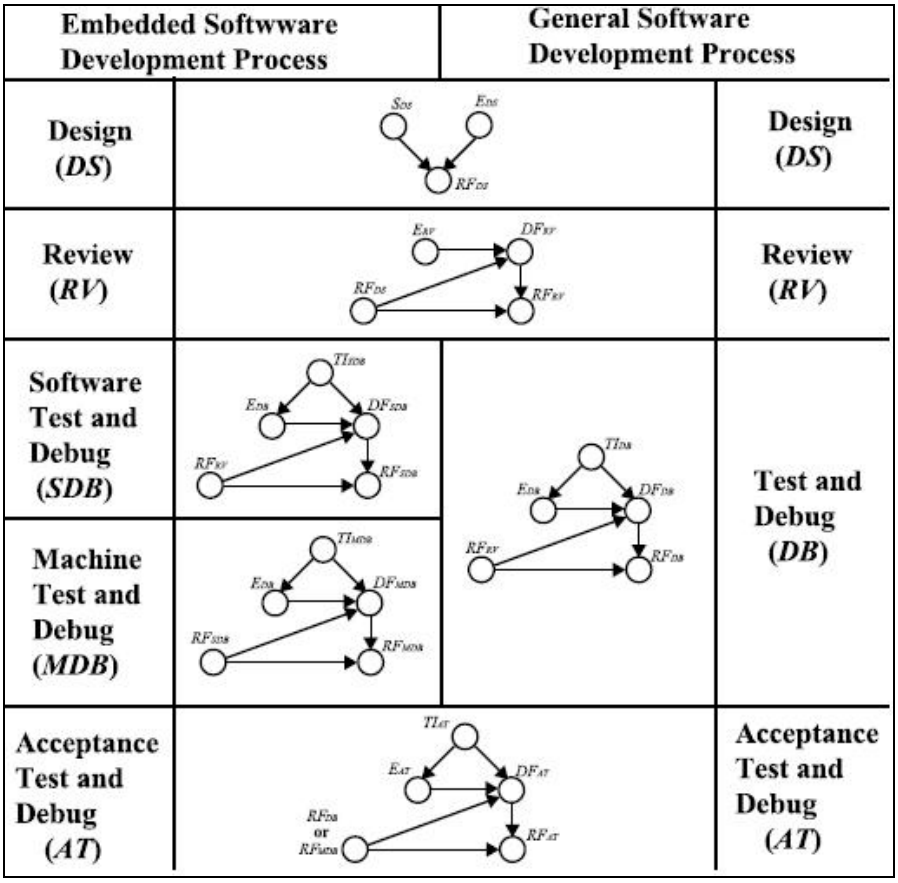


Figure 2.1 Bayesian network for the embedded and general model [8].

The results of embedded and general model once more indicate that it is necessary to use an embedded model for practical use. On the other hand, it is not practical for our purpose due to the fact that it is very hard to collect process metrics and analyze each development phase of an organization.

Khoshgoftaar and Gao proposed a multi-strategy classifier for embedded software, which cascades a rule-based approach (decision tree) with two case-based learning (1-nearest neighbor) [9]. They collected five software metrics which measure lines of code and commented codes delivered to system tests [9]. The model works as two-folds. First, the rule-based method classifies software modules. Misclassified defective and defect-free modules separately form two libraries (L_p, L_n). Instances in these libraries are further used in the testing phase, where each module is re-classified with case-based learners to check the similarity with the instances in the libraries. If a module is similar to any instance from the library, then it is assumed that rule-based method misclassified that instance [9]. It is observed that learning from previous misclassified instances increases the performance of

defect prediction. This research shows the advantage of using combined techniques and encourages researchers to try new approaches.

Finally, Oral and Bener proposed an ensemble of classifiers as a prediction model for embedded software [10]. They aim to capture the strengths of different methods such as Artificial Neural Network, Naïve Bayes and Voting Feature Interval to increase the probability of detection rates for embedded software. This research has become one of the guidelines for our study because of several reasons.

First, Oral and Bener constructed an automated metrics extraction tool for collecting special list of static code attributes as indicators of module characteristics of embedded software [10]. They managed to achieve significant detection rates with these attributes. Therefore, it encourages us to use the proposed list of static code attributes in our defect prediction model. Second, they used publicly available datasets that show embedded system characteristics to measure their model's performance. Public datasets increase repeatability of the previous work and comparison with new studies [46]. Thus, we can evaluate the performance of our defect prediction model by examining previous research. Third, although ensemble of classifiers increases detection performance of the prediction model in embedded software, it has a cost of high false alarm rates which should be investigated. We have extended the work of [10] by measuring how much ensemble would reduce the testing effort to catch defective modules [52]. Our observations show that ensemble reduces the verification effort in the testing phase with high detection rates [52]. However, false alarm rates cause additional effort during testing phase. Therefore, we have set our principal motivation on reducing false alarm rates of defect prediction in embedded software.

2.4. Related Machine Learning Studies

2.4.1. Combining Multiple Learners

Each learning algorithm applies a certain model which converges to a different solution and fails under different circumstances [39]. The performance of a learner may be limited to provide the highest accuracy on the given validation set, or any learners may not

be accurate on this dataset [39]. The idea of combining multiple learners is to learn from different algorithms and improve the accuracy. The easiest method is training different algorithms that make different assumptions on data [39]. Rather than depending on a single algorithm, we combine decisions of each algorithm given an instance, and a separate result has been computed by combining these decisions. This approach is called as *voting*, *ensembles* or *mixture of experts* [39]. On the other hand, we can use a single algorithm with different parameters [39]. Parameters can be k in k-nearest neighbor algorithm or covariance matrix in Gaussian approximation. Then we can combine results of the same algorithm, but with different parameters, to reduce the error. Another method can be using different learners based on different sources, i.e. different set of data, and combination of their predictions [39]. Different representation of data may help to observe different characteristics and increase the accuracy.

The final methodology for combining multiple learners is using different training sets to train different learning algorithms [39]. This approach is different than the previous one in terms of the sequence of trainers. In the previous method, we can train our algorithms, independent from each other, with different training data. On the other hand, the algorithms can be trained serially so that the later algorithm is trained with the instances which are predicted inaccurate by the previous algorithm [39]. This kind of learning is generally called *cascading* or *boosting* or *multi-stage combination* [39].

3. PROBLEM STATEMENT

Although high quality software with fewer or no defects has been in demand, comprehensive VV&T activities during the testing phase of software development lifecycle have become the most expensive and time-consuming activities [4]. Therefore, effective strategies are needed to solve the resource allocation problem in companies. Various review methods, inspections and static analysis tools are proposed to decrease time required during the testing phase [4]. According to recent studies, these methods improve process control and final quality of the software [44]. Review methods are capable of catching 60 per cent of defects in software, while inspections manage to catch, at maximum, 65 per cent of defects [31, 32]. These methods reduce the effort required for catching defects before the release of the software by 40-50 per cent [31]. Defect prediction techniques, on the other hand, has a better performance, on the average 71 per cent detection of defective modules, when compared to other VV&T activities [7]. In complicated and large software systems, defect predictors are able to catch 70 per cent of defects by inspecting only three per cent of the code [52]. They guide developers to specific parts of the software which seem defective [7]. Therefore, they are effective tools for solving the needs of developers in testing effort and managers in resource allocation.

Recently, embedded systems have encountered similar problems such as producing low cost, high quality software in a limited time. It is due to the fact that embedded software has become very popular, since there is a rapid competition and increasing demand for embedded software in the market [8]. Embedded software has different characteristics than general software systems such that it is deployed pervasively once developed [8]. Defects in embedded software not only affect the quality of the overall system, but also may be highly severe, expensive or not tolerable [5, 43]. Therefore, there is a strong need for effective testing strategies to decrease testing effort, defects and increase the quality in embedded software. We define the problem as the construction of a defect prediction model for embedded software that can successfully detect defective modules in the software.

There have been several studies in the area of defect prediction for embedded systems [5, 8, 9, 10, 45]. These studies include different machine learning algorithms and data from various organizations that operate on embedded systems. They collected software metrics including process metrics and static code attributes from embedded software systems. Unfortunately, they are hardly reproducible due to the impossibility of reaching their data sources. Diversity in collected metrics also makes it hard to compare the prediction performances of proposed models. Previous study that proposed a defect prediction model for embedded software used public embedded software data by collecting widely used static code attributes [10]. It is a valuable source for embedded software domain because of its reproducibility in terms of data and static code attributes. In addition, the results of this study point out new challenges such as finding ways to decrease false alarm rates of defect prediction in embedded software. Using public embedded software data with new projects, we aim to collect static code attributes as the inputs of our model. Static code attributes are selected in our study, since they are easily collected and widely used metrics in defect prediction studies [4, 6, 7, 10, 11, 12, 17, 18, 19, 21, 22].

In addition, we have observed that data collection in embedded software is a crucial and time-consuming problem for defect prediction studies. Rather than waiting to collect enough data for defect prediction, we need to be able to use other companies' data and predict defective parts of software within an embedded company. This problem would also be clarified with a cross-company analysis that trains the model with data from different organizations and predicts defective parts of software within a company [11]. We aim to analyze the performance of cross- vs. within company defect prediction on embedded software to see the applicability of cross-company data in embedded software domain.

Finally, our observations on defects in embedded software systems present that problematic areas in the software are not only caused by defects in design or code. It is also rooted from frequent changes in requirements during coding phase. We believe that these changes are causes of incomplete or vague descriptions of requirements. Problems in requirements documents should directly affect the problems (defects) in the software. Therefore, we have integrated requirements metrics to our defect prediction model in order

to increase the information content. The aim is to find a relation between requirements and defects in embedded software.

In summary, we have combined a list of research questions for defect prediction studies with specific problems of embedded software. These questions are based on three pillars of defect prediction research: improving defect predictors' performance, reducing data collection effort and increasing the information content of defect predictors. We have proposed our solutions for these questions listed below in the following sections.

i. Can we propose a better defect prediction model, in terms of high detection rates with low false alarm rates such that it would also be a solution for high false alarm rates of embedded software?
ii. Do we generate this model in such a manner that it would be practical for cross-company analysis where the main problem is high false alarm rates?
iii. Are cross-company embedded software data applicable to predict defective modules within our embedded software system?
iv. Can we increase the learning capability of defect predictors by adding new information from other stages of software lifecycle such as requirements phase?
v. Does this additional information contain similar characteristics for embedded software systems as in the case of static code attributes in cross-company analysis?

4. PROPOSED MODEL: CASCADING NAÏVE BAYES

.

We aim to propose a new learning mechanism for our defect prediction model that will solve the problem of high false alarm rates in embedded software. Moreover, our algorithm would be a solution to high false alarm rates of cross-company defect prediction in order to make its results practical for embedded data. Our first two research questions that are related with the first pillar will be solved with the proposed model. Third research question that aims to decrease data collection effort will be investigated by cross- vs. within-company experiments using the proposed model as our defect predictor. The last two research questions are related with data and the effects of increasing information content in defect prediction studies. Therefore, we aim to propose solutions for them in the later sections throughout the experiments.

To decrease the amount of misclassified defect-free modules (false alarms) in the software, we propose a cascading approach, where the preceding algorithm learns from the mistakes of the previous algorithms. We have selected Naïve Bayes as the base-learners of our cascading approach. Our model learns from inaccurate predictions of the first Naïve Bayes algorithm and train the next one with these misclassified instances. We have used past project data to adjust our model. Then, we classify the modules of new projects as defective or defect-free based on the knowledge gathered from previous projects.

Defect prediction models basically take a set of independent attributes and retrieve a class label from a set of outputs to define whether a module is defective or not. It consists of four steps, which will be explained in this section:

- “What are the inputs to the defect prediction model?”
- “What are the outputs of the defect prediction model?”
- “How does the model operate between the inputs and output?”
- “How can we assess the performance of the model?”

4.1. Inputs of the Model

As inputs of this study, we need completed embedded software systems. From these software systems, we collect static code attributes, defined by McCabe [18] and Halstead [17] which are module-based metrics. Module is defined as a functional unit in this study. Static code attributes can be easily collected from any software with automated metric extraction tools [11]. Moreover, they provide useful information about the characteristics of software in terms of complexity, modularity and lines of comments and codes. Previous research proves that static code attributes are better guiders to predict defects in a module than other industrial review methods [7]. Therefore, they are widely used as inputs to defect prediction studies. Combination of various static code attributes helps us define the characteristics of a defective module, because it is hard to understand whether a module is defective or not, based on the behavior of a single attribute such as lines of code or complexity.

Apart from static code attributes, we need defect information of these systems to find the relationship between attributes and defect content. Each module in a system should be labeled as defective or defect-free based on the results of testing phase. We require both defect information and software metrics to classify a module as defective or not, and evaluate the performance of our defect prediction model. So, our predictions would guide the developers through modules which seem to be defective.

We have a set of software as a library $L = \{S_1, S_2, \ldots, S_n\}$, where each software contains a list of modules whose attributes and defect information are in the form of $\{a_1, a_2, \ldots, a_n, d\}$. Defect information is set to either 0 (defect-free) or 1 (defective) depending on the defect content of each module. Therefore every software system is represented as an *m-by-n* matrix that includes *m* modules each with *n* attributes (Equation (4.1)). Additionally, we keep an *m-by-1* matrix for defect information of each module in the software as seen in Equation (4.2).

$$S = \begin{bmatrix} a_{11}, a_{12}, a_{13}, ..., a_{1n} \\ a_{21}, a_{22}, a_{23}, ..., a_{2n} \\ \\ a_{m1}, a_{m2}, a_{m3}, ..., a_{mn} \end{bmatrix} \tag{4.1}$$

$$D = \begin{bmatrix} d_1 \\ d_2 \\ . \\ . \\ d_m \end{bmatrix} \tag{4.2}$$

To test the prediction performance of our defect prediction model, we only use static code attributes for embedded software systems. Since we do not use actual defect information, each project has a matrix collected in the form of Equation (4.1). After training our model, these projects' defect data are predicted based on this information matrix. Then, real defect information in the form of Equation (4.2) is used to assess the performance of our predictions.

4.2. Outputs of the Model

Defect prediction can be described as a classification algorithm, which has independent attributes as the input and a class label, each module belongs to, as the output. We make predictions about the defectiveness of the modules in software with our defect predictor to label data as 0 or 1. Therefore, the expected outputs are from the set, C = {0,1}.

4.3. Algorithm of the Model

This section will describe the algorithm of our defect prediction model. The model takes static code attributes, a_i, from the modules of different embedded software systems, *S,* into the library. It then assigns a class, c_i, from the output set C, to each module of the software. The algorithm of the model is essential for defect prediction, since the performance of the selected algorithm directly affects the accuracy of the model.

In this research, we propose a cascading classifiers method for defect prediction in embedded software. Previous study presented that their defect prediction model for embedded software, *ensemble of classifiers,* has a high probability of detecting defective modules, with a cost of high false alarm rates [10]. In other words, the proposed defect prediction model catches the actual defective modules in embedded software successfully. However, it also wrongly classifies defect-free modules as defective. We state that false alarm rates of the previous study should be handled with a cascading approach.

Cascading classifiers consist of a sequence of learning algorithms such that later learners are used only if preceding learners are not confident on the prediction of the instances [39]. A confidence level for the outputs are defined and compared with the predictions of each learner [39]. If the output of a learner is smaller than the defined threshold, we use the next learner from the model. The algorithm stops at a learner, when previous learners are not confident and the output of selected learner is bigger than the confidence threshold. A schema of the cascading approach can be seen in Figure 4.1 [39].

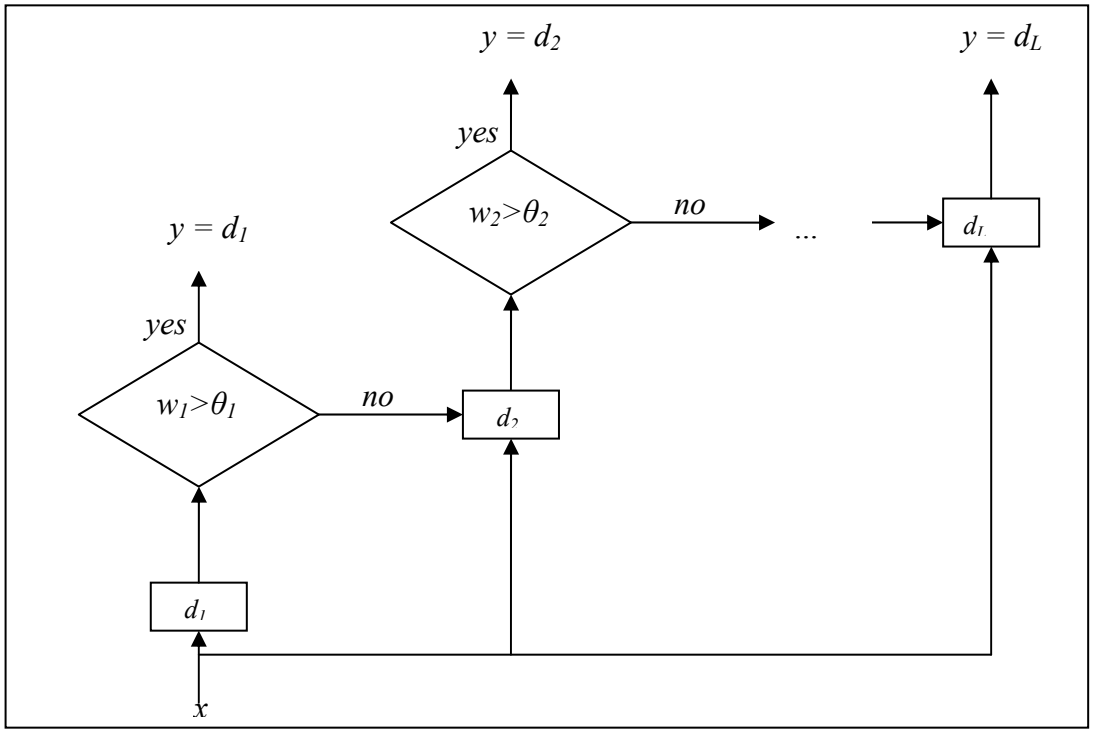


Figure 4.1. Schematic description for cascading classifiers [39].

Cascading classifiers work as follows: Starting with a training set, all instances from validation set, *x,* are classified with the first learner, d_1 [39]. Misclassified instances and the ones whose outputs, *w,* are smaller than the threshold, θ_1, are passed to the next learner, d_2 as training data [39]. The process is completed until there are no instances whose outputs are smaller than the threshold values. At each stage of the cascading classifiers, threshold values are selected such that $\theta_j \leq \theta_{j+1}$ [39].

For defect prediction of embedded software, we can construct cascading classifiers, where the next algorithm can learn from false positives and false negatives (misclassifications) of the previous algorithm. Non-confident instances, on the other hand, are left to the next algorithm to be re-classified. Thresholds for each step are defined by the posterior probability of a module that belongs to a class. Suppose we define the threshold of a learner as 0.60. If a module is classified as defect-free with 55 per cent probability, it is not labeled with the current learner. Instead, we use false positives and false negatives from the training set to re-classify that module with the next learner. This approach would improve the performance of defect predictors by increasing information content with misclassified instances. We can further decrease false alarms with the help of recorded instances of previous projects.

We examine several learning algorithms to decide which one will be used as base-learners of our cascading classifiers. From the previous studies, it is seen that Naïve Bayes produces the best results in terms of detecting defective modules on log-filtered data [7]. It calculates the posterior probability of a class (C), given evidence (X_i), by using the prior probability of the class, $P(C)$ and likelihoods of attributes, $P(X_i|C)$ [39].

$$P(C\left|X_i\right.) = \frac{P(C)}{P(X)} \prod P(X_i\left|C\right.) \tag{4.3}$$

We have decided that Naïve Bayes can be used as the base-learner at every stage of our cascading classifiers model. We additionally examine the performance of Voting Feature Interval (VFI) which is previously used in an embedded systems study [10, 52]. It is observed that VFI produces compatible or sometimes superior detection rates for embedded software. On the other hand, it produces high false alarm rates, which is not

preferred in our study. Therefore, we did not choose this algorithm as one of the base-learners of our approach.

Furthermore, we examine the posterior probabilities computed by Naïve Bayes for the modules of embedded software systems. Naïve Bayes compares the posterior probabilities of two classes (defective and defect-free) and selects the class which has the highest probability. It sometimes assigns a module to the defective class with 51 per cent probability. We want to increase the confidence level of these predictions with the help of cascading classifiers. Instead of using multiple base-learners in the cascading approach, we use multiple Naïve Bayes learners. At each stage, highest class probabilities of the modules are compared with the defined confidence thresholds. If that probability is less than the threshold, those instances are passed to the next Naïve Bayes without being classified as defective or defect-free. Non-confident instances are re-classified at the next stage using false positives and false negatives of the current stage. The model terminates when there are no instances whose highest class probabilities are not smaller than the threshold values.

Our defect prediction model has two phases, one of which is the training phase. During the training of our model, past projects from embedded domain are used to construct the model. First, Naïve Bayes algorithm classifies the modules of software as defective or not. Since we have actual defect information of past projects, predictions of the algorithm are compared with the actual defect values to construct false positive (FP) and false negative (FN) libraries. These libraries form the training set for the second stage of the model. Additionally, the modules whose posterior probabilities are smaller than the confidence threshold are left as unclassified and form the validation set of the second stage. Naïve Bayes classifier at the second stage uses the training instances to classify previous non-confident modules. If there are still instances left at the end of the second stage, which are misclassified or not confident enough to be classified, we form the training and validation set of the third stage with the same procedure. When training phase of the model terminates, we have a list of libraries which will be used at each stage of the cascading classifiers approach. Testing phase of the model uses these libraries to predict defective modules of new projects in a cascaded manner. A simple pseudo code for the construction of cascading classifiers can be seen in Figure 4.2.

```
S = past project
T =  Training set from S
V =  Validation set from S
FP = FalsePositive instances
FN = FalseNegative instances
NC = NotConfident instances
threshold = confidence threshold
using T
    (predictions, posteriors) = Apply Naive Bayes on V
    for each module m in V
        if prediction ~= actual label of m
            form FP,FN
        elseif posterior < threshold
            form NC
        end
    end
while FP,FN are not empty
    T' = FP + FN
    if NC is not empty
        V' = NC
        using T'
        (predictions',posteriors') = Apply Naive Bayes on V'
        for each module m in V'
        if prediction' ~= actual label of m
            form new FP,FN
        elseif posterior' < threshold
            form NC
        end
    end
end
```

Figure 4.2. Pseudocode for the construction of cascading classifiers.

4.4. Assessing the Performance of the Model

We need certain calculations to assess the prediction performance of our defect prediction model. Generally, *receiver-operator* (ROC) curve from signal detection theory has been employed into defect prediction studies [47]. In signal detection theory, ROC curves are offered to assess the performance of different predictors. A typical ROC curve, as represented in Figure 4.3, shows the hit rates of actual signals and false alarms of the predictor [47]. Similarly, we examine detection of actual defective modules in software (*pd*) and wrong signals (*pf*), i.e. detecting actually defect-free modules as defective. The line where (*pd,pf*) passes between (0,0) and (1,1) means that detection contains no information. Predictor never detects any defects in the software or all its predictions are false alarms. The ideal case should be (1,0) in terms of (*pd,pf*), where the predictor can detect all of the actual defective modules. Moreover, it should not call a defect-free module as defective. The ideal case is hard to achieve since higher probability of detection rates has a cost of false alarm rates. We need to form a line, near to *pd* axis, to achieve better performances in defect prediction models.

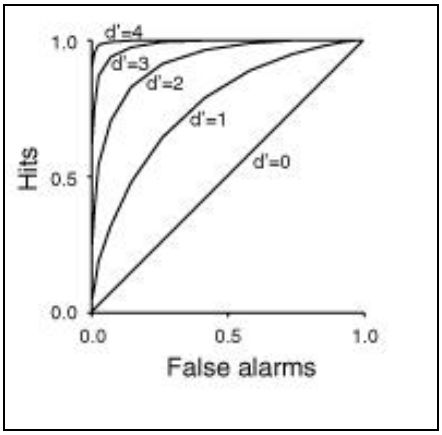


Figure 4.3. A typical ROC curve [47].

To measure hit rates and false alarm rates of defect predictors, a confusion matrix is used, as shown in Table 4.1 [7]. If a module is actually defective and the model is able to catch it, we increase the number of TP by 1. If our model wrongly predicts a defect-free module as defective, then we increase the number of FP by 1.

Table 4.1. Confusion matrix.

TP = true positives TN = true negatives FP = false positives FN = false negatives		Predicted modules containing defects?	
		Yes	No
Actual modules containing defects?	Yes	TP	FN
	No	FP	TN

After forming this confusion matrix, to calculate the probability of detecting a defective module (*pd*) and the probability of false alarm (*pf*), Equations (4.4), (4.5) and (4.6) are applied. Finally, we have used *balance* rate to see how close our estimates are to the ideal case [7]. *Balance* computes Euclidean distance of (*pd,pf*) to the point (1,0) and normalizes it with the point (1,1) [7].

$$pd = TP/(TP + FN) \tag{4.4}$$

$$pf = FP/(FP + TN) \tag{4.5}$$

$$bal = 1 - \sqrt{(0 - pf)^2 + (1 - pd)^2} \Big/ \sqrt{2} \tag{4.6}$$

5. EXPERIMENTS

Our first motivation is to construct a defect prediction model for embedded software, which produces higher *pd* (probability of detection) rates with lower *pf* (probability of false alarm) rates. We have proposed *cascading Naïve Bayes* to learn from misclassifications of the base-learner and decrease the false alarm rates. We explained the details of our proposed model and presented the construction of the model in Figure 4.2. In this section, we will describe data used in the model, general experimental setup and mention the differences in each analysis of this study to solve our research questions.

5.1. Data Used in the Model

We have used data from two different organizations. Most of them are publicly available in the Promise repository [46]. Data sources are NASA MDP Repository and SOFTLAB Repository, both of which contain software that have embedded systems characteristics. The reasons to use organizationally and operationally different software are as follows [45]:

i. To make a generalization by expanding the test-bed of embedded datasets by including projects from various sources.
ii. To prove the external validity of our results with different resources.
iii. To validate the performance of our cascading classifier on NASA datasets, which are reliable and used several times in other studies [7, 11, 21, 22].

We will give details about our datasets in the following four subsections. In the first two subsections, we will explain our datasets with the code metrics used in our defect prediction model. In the third subsection, we will explain the differences between cross- vs. within-company data in terms of the number of attributes and defect rates. Final section will explain datasets used in the analysis of cross- vs. within- company analysis using requirements metrics.

5.1.1. NASA MDP Repository

First data source used in this research is NASA MDP Repository, which contains 14 public datasets [48]. These datasets serve different characteristics and come from various software systems such as real-time applications, earth orbiting satellite system, space shuttle, flight software and dynamic simulator systems [10]. They are developed in different geographical locations across North America [7]. There are several reasons that explain our motivation of using NASA datasets. First of all, these datasets are accepted as reliable and used in many other defect prediction studies [7, 11, 21, 22]. Therefore, using these datasets would give us the opportunity to compare our results with the others from the field. Secondly, the representation of software in MDP Repository consists of static code attributes and actual defect information for each module, i.e. functional unit. Since we have selected module-based static code attributes as the inputs of our model, NASA datasets contain information that is large enough to evaluate our defect predictor's performance. Finally, this research contains an additional analysis with the requirements data to see the effects of increasing information content on the performance of defect prediction. The only available source that matches embedded software requirements with the modules is MDP Repository [48]. Thus, it is a great database for defect prediction studies with many aspects of data.

We have selected four projects, CM1, PC1, PC3 and PC4, from this repository that show embedded systems characteristics. Previous research completed by Oral has thoroughly analyzed requirements documents of NASA datasets and extracted specific attributes of embedded software such as timeliness, liveness, heterogeneity and reactivity [10]. Since four projects typically represent abovementioned embedded systems characteristics, we have decided to use them in our defect prediction study. Our experiments would require projects from the same domain to evaluate the similarities in their software patterns, i.e. defective module characteristics and requirements. A complete list of projects from NASA MDP Repository is given in Table 5.2 with other datasets from SOFTLAB Repository.

Each module in NASA datasets contains 19 static code attributes from Appendix A, and defect information, either *true* as defective or *false* as defect-free. We have converted

defect information into numeric values (0, 1) and use the full list of attributes to benefit from all of them. Some modules in some datasets contain zero lines of code (sum of lines of comment and executable lines of code). Although this situation is strange, we leave them as they are and try to collect information from their other attributes. Static code attributes used in NASA datasets are summarized in Table 5.1.

Table 5.1. Attribute list for NASA datasets [48].

McCabe metrics (3)	*v(g)*	Cyclomatic complexity
	ev(g)	Essential complexity
	iv(g)	Design complexity
Halstead metrics (11)	*V*	Halstead volume
	L	Halstead level
	D	Halstead difficulty
	I	Halstead content
	E	Halstead effort
	B	Halstead error estimate
	T	Halstead programming time
	uniq_op	# of unique operators
	uniq_opnd	# of unique operands
	total_op	# of total operators
	total_opnd	# of total operands
Lines of code (4)	*Loc*	Total lines of code
	lOCode	Executable loc
	lOComment	Commented loc
	locCodeAndComment	Code and commented loc
Miscellaneous (1)	*branchCount*	Number of branches

5.1.2. SOFTLAB Repository

Second data source of this research has come from a white-goods manufacturer in Turkey, which operates in embedded systems industry. We have been collaborating with the company for nearly two years to conduct our studies on defect prediction in embedded software. We have collected five projects' static code attributes as well as defect information. For the first three of them, Oral has obtained the source code and implemented a metrics extraction tool that calculates 29 static code attributes from the

software (marked in Appendix A) [10]. We have donated three projects to the Promise repository to make them publicly available [46]. For one year, we have been using this metrics extraction tool to obtain static code attributes of three more projects. Only two out of three projects have defect information available. Therefore, total number of local projects used in this research is five, namely AR1, AR3, AR4, AR5 and AR6. (AR2 is still under development and there is no defect information available for it.) Information about these datasets is available in Table 5.2.

Table 5.2. General information about the datasets.

Name	Attributes	# Modules	# Defectives	Defect rate
CM1	19	498	49	0.10
PC1	19	1109	77	0.07
PC3	19	1563	160	0.10
PC4	19	1458	178	0.12
AR1	29	121	9	0.07
AR3	29	63	8	0.13
AR4	29	107	20	0.19
AR5	29	36	8	0.22
AR6	29	101	15	0.15

All software systems from the company have been embedded into different white goods such as refrigerator, washing machine, drying machine, oven and dishwasher. These systems also represent typical embedded software characteristics, although they are very different from the safety-critical and mission-critical NASA projects. However, this also brings diversity into our study for generalization.

5.1.3. Data for Cross- vs. Within-Company Analysis

As we have mentioned before, one motivation of our research is to analyze whether embedded software collected from different organizations (cross-company) can be used to predict defective parts of embedded software within a company. This analysis was previously completed by [11], where they built their experimental setup with datasets coming from various software systems. Our aim is to narrow down the scope of this

experiment to embedded software domain. Therefore, we have collected static code attributes which are available in all embedded software systems. Attributes from both NASA and SOFTLAB datasets are inspected to form a list of common attributes for our defect prediction model.

We have observed that embedded software systems from NASA MDP Repository contain 19 static code attributes and one defect information (Table 5.1). On the other hand, local software systems from SOFTLAB Repository contain 29 attributes and defect information (Appendix A). A list of common attributes for cross-company analysis consists of 17 attributes, which is marked in Appendix A. Therefore, for cross-company experiments, we will change the number of static code attributes for all datasets listed in Table 5.2 to 17. Total number of modules and their defect rates in all datasets have been kept the same in cross- vs. within-company analysis.

5.1.4. Data for Additional Analysis Using Requirements Metrics

Third aspect of our defect prediction research includes increasing the information content of embedded software data used in our predictor. For this purpose, we have used requirements metrics of two projects from NASA MDP Repository. These metrics are extracted from textual requirements of two projects, namely CM1 and PC1. We were not able to reach information about requirements and matching modules for PC3 and PC4. Therefore, only two projects, CM1 and PC1 from MDP Repository, are traced module-by-module to relate them with their corresponding requirements. A module in a dataset can reflect one or more of the requirements. Furthermore, an individual requirement can be implemented by one or more modules in the software. There are also cases where we could not relate a requirement with a module from software. Similarly, some requirements could not be explained in terms of modules. Therefore, the number of modules, defect rates and number of attributes in *cross- vs. within-company analysis using requirements metrics* are very different than the original datasets.

To assess the effects of early lifecycle data on defect prediction, we have obtained AR1's requirements document to match each requirement with its corresponding modules. For this, we have observed the study done by NASA Goddard Space Flight Center to

automatically collect requirements metrics from requirements analysis documents [49]. They have implemented an Automated Requirements Measurement Tool (ARM) which traces the document extensively to catch simple words that indicate weaknesses, completeness, directives and continuances [49]. We have examined those 9 metrics (see Appendix B for the list of attributes.) and extracted them manually from AR1's requirements analysis document. It is due to the fact that there is not available measurement tool for documents written in Turkish. Number of requirements and related modules for all datasets used in within-company experiments can be examined in Table 5.3. Cross-company experiments will use 26 attributes (17 code metrics and 9 requirements metrics) and defect information from these three projects.

Table 5.3. Datasets in additional analysis using requirements metrics.

	#req. metrics	#code metrics	#modules	#modules that have req.s	#req.s associated with modules	#modules with req+code metrics	#defective modules with req+code metrics
CM1	9	19	498	109	114	266	83
PC1	9	19	1109	203	320	477	115
AR1	9	29	121	36	84	162	12

5.2. Experimental Design

5.2.1. Experimental Design for Cascading Naïve Bayes

We have used *M*N-way-cross validation* technique to overcome the ordering effects of data in our defect prediction study [50]. Certain orderings cause significant changes in data such that it degrades or improves the performance of the learner [7]. We avoid such extreme situations with randomizing the order of data M times. Different than the popular *N = 10-fold cross validation* technique [6, 7, 11, 22], we need to stratify data into *N = 3* bins, which are training set, validation set and test set. *Stratification* of data preserves prior probabilities for defective and defect-free classes in all subsets [39]. With training and validation sets, we constructed our cascading model and its misclassification libraries (false positives and false negatives). Then the remaining set is used as the test set whose modules will be predicted as *defective* or *defect-free* with our model. We randomize the

order $M = 10$ times to avoid ordering effects [50]. Finally, we repeat this procedure 10 times and the mean of 10 iterations is presented to assess the performance of our defect prediction model. We have also examined mean values of 10, 20 and 50 iterations and decided that there is not much difference in terms of *pd, pf* and *balance* rates. Therefore, we have agreed on 10 as the number of repetitions. For nine projects, we have iterated our model (10 repetitions) x (10 randomized orderings) x (9 test sets) = 900 times to present the performance of our cascading classifiers.

Since our base-learner in the cascading classifiers approach is Naïve Bayes algorithm, we took the logarithms of static code attributes (replace all numerics, *n,* with *ln(n)*) to increase the performance of the algorithm. It is proved that Naïve Bayes finds the relation between defects and static code attributes better in a log-normal way [7]. Besides, we did not apply a feature selection technique, since we do not only construct a model whose predictions will be superior, but also examine whether the information content of static code attributes is limited by adding new attributes to our data.

We have previously explained that cascading classifiers learn from misclassified instances to make predictions for non-confident instances. At each step, we have computed the posterior probabilities of module being in a defective or defect-free class. Then the class whose posterior probability is the highest is assigned to that module. We define a confidence threshold to select only the instances whose class probability is larger than that value. For non-confident instances, we keep on iterating our model as soon as none of the modules are left as non-confident. In order to define this confidence threshold, we have formed a list of possible threshold values, $\theta = \{0.55;0.60;0.65;0.70;0.75;0.80\}$. We have made an assumption that probabilities greater than 80 per cent are large enough to classify a module with Naïve Bayes classifier. We have selected the best threshold with an iteration that gives the highest improvement on the performance of cascading Naïve Bayes. All steps of our general experimental design can be seen in Figure 5.1.

```
DATA = {AR1 AR3 AR4 AR5 AR6 CM1 PC1 PC3 PC4}
FILTER = {none log}
THRESHOLD = {0.55 0.60 0.65 0.70 0.75 0.80}
for data in DATA
    for filter in FILTER
        data' = filter data
        for confidence from THRESHOLD
            for i from 1 to M
                data'' = randomize the order of data'
                (train,val,test) = stratify data'' into 3 bins
                CascadingNB = train the model using train, val, confidence
                predictions = CascadingNB(train,test,confidence)
                (pd,pf,bal) = assess the performance with (predictions,test)
            end
        end
    end
end
```

Figure 5.1. General experimental design for cascading Naïve Bayes.

5.2.2. Experimental Design for Cross- vs. Within-Company Analysis

We have made two experiments with embedded software systems using cross- vs. within-company data. Previous study completed by Menzies *et al.* observed the prediction performance of Naïve Bayes by selecting the training set from two different sources [11]. Their experimental design was formed with *N = 10-fold cross validation* technique such that 90 per cent of data was used as the training set. When within-company analysis is done, 90 per cent is selected from the projects within a company. Otherwise, it is selected from the projects of different organizations. Their results indicated that cross-company data has produced high detection rates with a cost of high false alarm rates [11]. Therefore, authors suggested using cross-company data only when the cost of high false alarm rates can be compensated with high detection rates [11].

Our purpose is to evaluate whether cross-company embedded software can be used for prediction of defective modules in new projects within a company. We have proposed *cascading Naïve Bayes* algorithm to decrease high false alarm rates of the previous study in order to increase the applicability of cross-company data for embedded software. We have constructed our experimental setup with small differences in the design of two experiments. First of all, all data has come from embedded software in our experiments which would give different prediction performances for cross-company analysis of [11]. Secondly, we have used *N=3-fold cross validation* technique to train and test our model.

For this purpose, we have used one third of data as the training set and another one third as the test set. This stratification has also increased varieties between two studies.

Figure 5.1 represents our general experimental design, where the *cascading Naïve Bayes* is trained and tested using the same project. We need to make several modifications on Figure 5.1, specifically at lines 11 and 12, to implement our cross- vs. within-company analysis for both sources of data. First of all, we need to define cross- and within-company data. We will interpret NASA datasets separately as they come from different organizations across North America. On the other hand, SOFTLAB datasets are implemented within the same company. For cross-company analysis, we will only use NASA datasets to construct the model, since SOFTLAB datasets are not large enough to make a difference in training set. Besides, within-company experiments are designed such that NASA projects use the general experimental design, whereas SOFTLAB datasets form one shared training set to calibrate the model.

Here is a simple example to describe two experiments in detail. Suppose we have chosen CM1 as our data (line 4) whose modules will be classified as defective or defect-free with our defect prediction model. We will implement within-company analysis using the same procedure defined in Figure 5.1. On the other hand, cross-company analysis will use other NASA projects to train the model (line 11) and one third of CM1 as test data to assess the performance (line 12). Within-company analysis procedure is slightly different when we select one of the SOFTLAB datasets, such as AR3, as our data (line 4). We need to use other SOFTLAB datasets, AR1,AR4,AR5 and AR6, to construct the model (line 11) and use test data from AR3 project to assess the performance (line 12).

In all of these experiments, training, validation and test sets are log-filtered and their orders are randomized at every iteration. As mentioned above, the experiments are repeated 10 times to get the mean of the *cascading Naïve Bayes* performance.

5.2.3. Experimental Design for Adding Requirements Metrics into the Analysis

We have added one more analysis to cross- vs. within-company defect prediction analysis in embedded software. We have increased the information content by using

requirements metrics in addition to static code attributes of the software. Our aim is to capture additional information from requirements documents to increase the prediction performance of defect prediction. This approach has been introduced in a recent study, where requirements documents are traced to extract specific information from them [12]. The results proved that early lifecycle data has a direct impact on defects in software. We have combined our cross- vs. within-company analysis with requirements metrics to evaluate whether requirements documents contain similar characteristics of embedded software between different organizations that would lead us to higher detection rates.

Experimental design of this analysis is the same with the cross- vs. within-company experiments except the data comes from only three projects, AR1, CM1 and PC1. The only difference is the amount of projects that are used to train the model, since we need to match requirements with the modules. For SOFTLAB dataset, AR1, within-company experiments are done in the same manner with the experiments of NASA datasets, since there is only one SOFTLAB dataset that comes from the local company. Cross-company experiments do not have any difference from our previous analysis. Finally, we have log-filtered both static code attributes and requirements metrics before we fed them into our model.

6. RESULTS

Above mentioned experimental setups have been implemented one-by-one and the results of each analysis are presented in this section. We have compared the performance of our *cascading Naïve Bayes (CNB)* predictor with simple Naïve Bayes *(NB)* algorithm, which presents the best results so far in the defect prediction literature [7]. The first subsection will present the performance of two algorithms for NASA and SOFTLAB datasets. Subsequently, we will assess the prediction performance using cross-company (CC) data in embedded software systems, and compare the results with performance of within-company (WC) data. Finally, we will add textual requirements metrics to our analysis and present the results of our predictor, *CNB,* to evaluate the impact of early lifecycle data on CC vs. WC analysis.

We have previously mentioned that we will use probability of detection (*pd*), probability of false alarms (*pf*) and *balance* rates to assess the performance of our algorithm, *cascading Naïve Bayes. Pd* measures how many of actual defective modules are detected by the predictor. *Pf* rate of a predictor, on the other hand, measures how many of actual defect-free modules are wrongly classified as defective. To see how close our estimates are to the ideal case, which all modules are correctly classified, *balance* is used. What is expected from a defect predictor is generally high *pd* rate while keeping the *pf* rate at a minimum. Therefore, our defect prediction algorithm should catch as much defects as possible, while reducing false alarms in datasets. Sometimes, the learner can reduce false alarms with a cost of decreased *pd* rates. In such situations, we need to judge the performance in terms of *balance* rate.

6.1. Results of Cascading Naïve Bayes

In this section, we will propose the results of our defect predictor, *cascading Naïve Bayes (CNB)* by comparing with simple Naïve Bayes algorithm. We will assess the performance of our defect predictor in all datasets separately.

We have selected confidence threshold of CNB by measuring the performance of the model in different threshold values. Then, we have decided on a specific threshold, in which CNB is superior in terms of *pf*, *balance* and *pd* rates respectively. In Figure 6.1, an example for AR3 project is presented where red lines represent the performance measures for NB, and blue lines represent the measures for CNB.

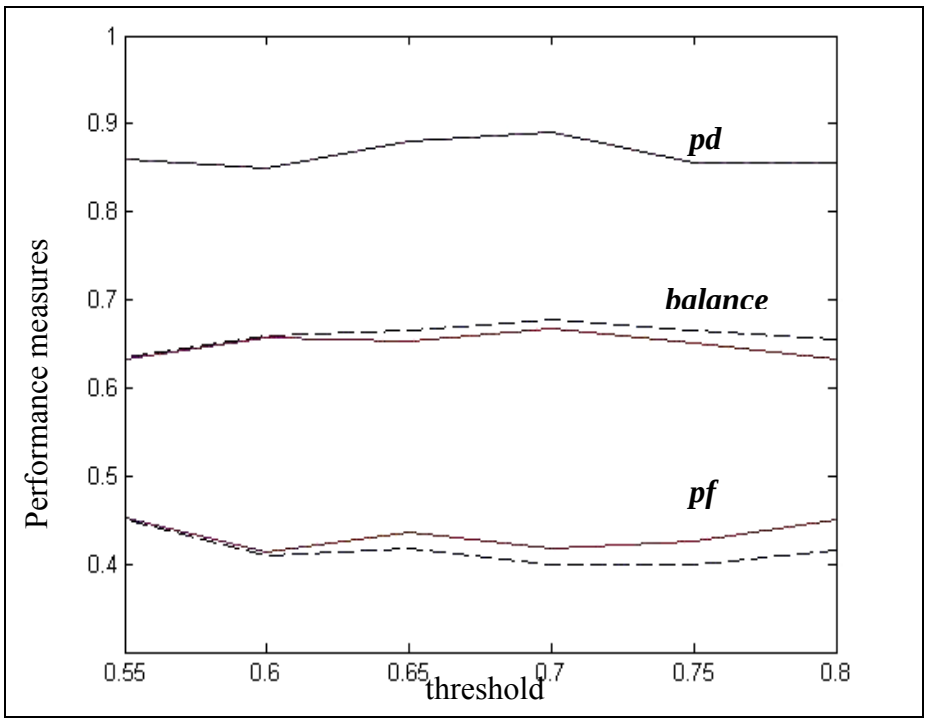


Figure 6.1. A sample representation for the selection of the best confidence threshold.

For thresholds bigger than 60 per cent, we can observe the increase in *balance* rate and decrease in *pf* rate of CNB. We select 70 per cent as the threshold for AR3, since higher *pd, balance* rates and lower *pf* rate can be observed for both NB and CNB algorithms.

The results of *cascading Naïve Bayes (CNB)* and *simple Naïve Bayes (NB)* for NASA and SOFTLAB datasets can be seen in Table 6.1 and 6.2 respectively. For each performance measure, mean values have been presented from (10 repetitions) x (10 randomized orderings) = 100 iterations. We compare our findings with Naïve Bayes algorithm, since our aim is to increment the learning capability of Naïve Bayes by using previous misclassifications. Results prove that false alarm rates have been decreased using our defect predictor by stabilizing detection rates (*pd*) and balance. In some datasets, our

defect predictor reduces *pd* rates by one to two per cent. However, we can still preserve the balance rate in the same range. On the average, CNB detects 74 per cent of defective modules in NASA datasets, while reducing false alarm rates from 30 per cent to 29 per cent. It also achieves to increase balance rate by one per cent. For SOFTLAB datasets, CNB produces one per cent decrease in both *pd* and *pf* rates. However, balance rates of CNB and NB show that we do not degrade the overall performance.

Table 6.1. Comparison of CNB with NB in NASA datasets.

	Pd (%)		**pf (%)**		**balance (%)**	
	CNB	**NB**	**CNB**	**NB**	**CNB**	**NB**
CM1	67	69	**32**	33	**67**	67
PC1	68	69	**26**	28	**70**	69
PC3	74	75	**29**	30	**72**	72
PC4	86	85	28	28	**78**	77
Avg.	**74**	**74**	**29**	30	**72**	71

Table 6.2. Comparison of CNB with NB in SOFTLAB datasets.

	Pd (%)		**pf (%)**		**balance (%)**	
	CNB	**NB**	**CNB**	**NB**	**CNB**	**NB**
AR1	59	59	36	36	**56**	56
AR3	80	82	**36**	37	**65**	65
AR4	76	77	**33**	34	**68**	68
AR5	90	90	**24**	25	**78**	78
AR6	81	82	**33**	34	**70**	70
Avg.	77	**78**	**32**	33	**67**	**67**

One interpretation of these results can be as follows: Cascading approaches are beneficial for defect predictors, since they learn from previous misclassifications, behave accordingly and do not perform worse in terms of *pd* and balance. On the other hand, improvements in three performance measures are so minimal. It is due to that fact that we can effectively combine the highest amount of information from static code attributes with a simple algorithm such as Naïve Bayes [22]. Therefore, new algorithms may not improve the performance of defect predictors.

Since we have achieved only minimal improvements in defect detection performance of several datasets, results of this section prove that we need to focus on data rather than the algorithm. Therefore, the following two sections will present two different perspectives that concentrate on embedded software data rather than the algorithm.

6.2. Results of Cross- vs. Within-Company Analysis

We have already observed that algorithmic changes in defect prediction studies show limited improvements. We should investigate data, specifically embedded software data, for additional analysis in this field. We have proposed a cross- vs. within-company analysis for embedded software to evaluate data among different organizations. Our aim is to measure whether cross-company embedded software systems can also be used to predict defects within a company from the same domain.

We have previously discussed a recent study indicating that using cross-company data increased detection rates as well as false alarm rates for NASA datasets [11]. Menzies *et al.* suggested using within-company data if high false alarm rates of cross-company analysis can not be compensated by high detection rates [11]. In this section, we will show the prediction performance of our algorithm on cross-company data, where companies are specifically chosen from embedded systems domain. We expect to reduce false alarm rates of cross-company analysis with our cascading approach to perform similar results to within-company analysis.

For each data repository, we form a table including the average performance of Naïve Bayes and cascading Naïve Bayes algorithm using within-company (WC) and cross-company (CC) analysis. Results for NASA datasets are presented in Table 6.3. It includes performance measures of our algorithm, compared to the Naïve Bayes learner, which is a replication of the previous study [11] using embedded software systems only. From the results of NASA datasets, we have derived several conclusions. First of all, we have once more realized that cascading Naïve Bayes algorithm does not produce the worst performance as well as the best performance in terms of *pd* and *bal* rates. On the contrary, it achieves to reduce *pf* rates for CC and WC experiments. Secondly, this analysis in

embedded domain shows the applicability of using domain-specific data to predict defects of new projects within a company. Previous study produced on the average 95 per cent detection rates with 72 per cent false alarms using cross-company data from various software domains [11]. On the contrary, our algorithm catches 72 per cent of defective modules and misclassifies 31 per cent of modules as defective, which is an acceptable rate compared to within-company analysis. However, when we observe within- vs. cross-company results, we realized that decrease in false alarm rates of cross-company analysis also brings decrease in other performance measures. Significance tests also prove that using within-company data is significant to cross-company data in terms of *pd* and balance rates. Minimal improvements in *pf* rates do not produce significant results for cross-company analysis. Therefore, companies should prefer cross-company data until they manage to collect their within-company data to train defect predictors.

Table 6.3. CC vs. WC analysis of NASA datasets.

		pd (%)		**pf (%)**		**bal (%)**	
		CNB	**NB**	**CNB**	**NB**	**CNB**	**NB**
CM1	**WC**	72	72	**32**	33	69	69
	CC	63	62	**25**	26	**68**	67
PC1	**WC**	68	**69**	**26**	28	**70**	69
	CC	72	74	**38**	41	**66**	65
PC3	**WC**	74	**75**	**29**	30	**72**	72
	CC	74	76	**30**	31	**72**	72
PC4	**WC**	86	85	**28**	28	78	77
	CC	79	81	**31**	32	**73**	73
Avg.	**WC**	75	75	**29**	30	**72**	71
	CC	72	73	**31**	32	**70**	69

Same experiments for SOFTLAB projects are represented in Table 6.4. We have included the results of cascading Naïve Bayes and Naïve Bayes for cross-company and within-company experiments. The results from SOFTLAB datasets have shown that cross-

company analysis on embedded software does not generally increase the pd rates of defect predictors compared to results using within-company data. On the other hand, our algorithm manages to lower false alarm rates of cross-company analysis by four per cent, which also improves balance by three per cent. When we observe each project from SOFTLAB repository, we have realized that cascading Naïve Bayes further reduces false alarm rates of projects whose defect rate is high. For instance, AR4 and AR5 have the highest two defect rates among other SOFTLAB projects, whose *pf* rates are decreased by nine and eight per cent respectively with our algorithm. Therefore, we can suggest using cascading approach for projects which may contain higher defects.

Table 6.4. CC vs. WC analysis of SOFTLAB datasets.

		pd (%)		**pf (%)**		**bal (%)**	
		CNB	**NB**	**CNB**	**NB**	**CNB**	**NB**
AR1	**WC**	47	47	**17**	18	**58**	58
	CC	43	45	**17**	18	57	**58**
AR3	**WC**	80	82	**36**	36	**65**	65
	CC	90	90	**37**	40	**69**	67
AR4	**WC**	95	95	**45**	45	**71**	71
	CC	78	78	**36**	37	**74**	73
AR5	**WC**	100	100	**42**	42	**71**	71
	CC	85	85	**34**	35	**72**	71
AR6	**WC**	40	41	**18**	19	**55**	56
	CC	49	44	**17**	20	**61**	57
Avg.	**WC**	72	73	**32**	32	**64**	**64**
	CC	**69**	68	**28**	30	**67**	65

In contrary to results of NASA projects, we have seen that cross-company analysis for SOFTLAB datasets is preferable to within-company analysis in terms of *pf* and *bal*. However, when significance tests are performed, we have found that none of the experiments are significant to each other. Although we have observed numeric differences

between CC and WC analysis, it is hard to conclude that companies should prefer cross-company data for their projects.

Finally, we have prepared box plots of each dataset using minimum, median and maximum values of performance measures for CC and WC analysis to better illustrate results of cascading Naïve Bayes vs. Naïve Bayes algorithms (Appendices C and D). Moreover, Appendix E includes a comparison between cross-company and within-company results in terms of balance rates.

6.3. Results of Additional Analysis Using Requirements Metrics

Final analysis in this research aims to increase the information content in defect prediction studies in order to improve the prediction performance. We have combined requirements and code metrics for three projects, CM1, PC1 and AR1, and completed cross- vs. within-company analysis using two set of metrics. Since number of modules and defect rates of the projects have been changed due to matching between requirements and modules (see Table 5.3), we have produced two experiments for each analysis. First, we have conducted cross- vs. within-company analysis using only code metrics of modified datasets. Then, we have measured the performance of this analysis by adding requirements metrics to our defect prediction model.

Table 6.5 summarizes cross- vs. within-company results with code metrics only and requirements and code metrics together. We present only the performance of cascading Naïve Bayes to make the results simple to interpret. Results of this analysis provide useful information about the requirements metrics and their similarity in similar software systems. First, we have seen that including requirements metrics improve both within-company and cross-company experiments. It motivates us to examine requirements metrics more carefully and extract those using automated tools such as ARM [49].

Second, requirements metrics increase the information content of cross-company experiments such that we can catch more defects in the software while reducing false alarms. We can state that requirements of different organizations whose software systems represent similar characteristics provide useful information for predicting defect-prone

modules of new projects within a company. One reason for this would be shared characteristics of embedded software such as timeliness, reactivity, and liveness [4].

Table 6.5. Results of CC vs. WC analysis using requirements metrics.

		Only Code metrics			**Req&Code metrics**		
		pd	**pf**	**bal**	**pd**	**pf**	**bal**
CM1	**WC**	61	47	56	**64**	49	**56**
	CC	43	34	49	**46**	**32**	**55**
PC1	**WC**	48	33	55	**56**	**26**	**63**
	CC	60	46	57	**62**	**36**	**62**
AR1	**WC**	90	31	70	**98**	**31**	**76**
	CC	20	3	42	11	3	37

On the contrary, we observed that cross-company data performs the worst for AR1 dataset. In Table 6.4, where we compare our findings using cross- vs. within-company embedded software, AR1 also produces low detection and false alarm rates in both experiments. That shows us AR1 may contain different characteristics from both within-company data and cross-company data. We observed requirements metrics and static code attributes of this project. Since within-company analysis results of AR1 in Table 6.5 are extremely satisfactory, we do not suspect from the correctness of both metrics.

We have carefully examined SOFTLAB projects' minimum, median and maximum values with respect to their attributes. From the analysis, we have realized that AR1 is very similar to AR6 and they represent different characteristics compared to AR3, AR4 and AR5. On the average, ratio of attributes in AR1 over attributes of AR3, AR4 and AR5 is 60 per cent, which indicates AR1 is a less complex project. We have replicated within company analysis of AR1 by using AR6 as the training data. The results show that we can increase within-company defect prediction results of AR1 (and similarly AR6) by 10 per cent in terms of *pd*. Therefore, we can assume that new projects (AR1 and AR6) present different characteristics which should be investigated further and those characteristics make them dissimilar to other projects used in this study.

Finally, we suspect from the experimental design of our cascading approach due to abnormal results of AR1. Cascading Naïve Bayes can stick into local minima since each base learner uses both code and requirements metrics. We change the design of our model such that the first base-learner classifies all instances using only code metrics. Then, the second learner classifies non-confident instances of the first learner using both code and requirements metrics. Since manual collection of the requirements needs too much time and effort, we can also decrease that effort by collecting requirements metrics only for the instances which can not be classified in the first learner. The results for our modified model are presented with results from the previous analysis in Table 6.6. We can observe that NASA datasets perform worse, in terms of *pd* and *bal* rates, with a modified model for cross-company and within-company experiments, although we achieve to increase detection rates for AR1. Therefore, we, once more, prove that using all attributes at every stage of the cascading approach is more useful for detecting defects in the software. Still, we can not conclude on a logical reason about the worse performance of AR1 which needs further investigation.

Table 6.6. Results of the additional analysis with the modified version of CNB.

		Code metrics using CNB			**Req&Code metrics using CNB**			**Req&Code metrics using modified CNB**		
		pd	**pf**	**bal**	**pd**	**pf**	**Bal**	**pd**	**pf**	**bal**
CM1	**WC**	61	**47**	56	**64**	49	**56**	43	**34**	52
	CC	43	34	49	**46**	**32**	**55**	23	**21**	50
PC1	**WC**	48	33	55	**56**	**26**	**63**	48	**24**	59
	CC	60	46	57	**62**	**36**	**62**	**65**	51	54
AR1	**WC**	90	31	70	**98**	**31**	**76**	96	35	73
	CC	20	**3**	42	11	**3**	37	**35**	12	**53**

6.4. Discussion on Results

We have set our experimental design on three aspects of defect prediction research using embedded software systems. All experiments have presented new solutions for the research questions derived from three pillars of defect prediction studies. First, we have investigated a new learning mechanism that would improve prediction performance of

current defect prediction models. Our motivation was to construct a defect prediction model for embedded software that has high detection rates as well as low false alarms. For this purpose, we have proposed a cascading approach that would store misclassifications of the previous learner to train the next learner. A cascading approach does not only learn from previous misclassifications, but also it increases the confidence levels of the predictions done by the learners. We have observed that combining multiple Naïve Bayes learners has positively affected the prediction performance of defect prediction in embedded software. Although the improvements in detection rates are quite minimal, we have achieved to decrease false alarm rates in all datasets. Therefore, it is worth using a learning mechanism that would help to learn from previous mistakes.

The first approach on defect prediction research proves that proposing new algorithms to improve the prediction performance would no longer be sufficient. We need to inspect data used in defect prediction studies to understand its information content. Therefore, second aspect of our research in embedded software has been based on cross- vs. within- company analysis. We have used data from different embedded software systems to predict defective modules of projects within a company. We have compared cascading Naïve Bayes algorithm with simple naïve Bayes to assess whether our algorithm would present a solution for high false alarm rates of cross-company predictions. We have obtained two results from cross- vs. within-company analysis in embedded software. First, it is seen that our algorithm manages to decrease false alarm rates of cross-company experiments while keeping balance and *pd* rates stable, compared to the Naïve Bayes results. Second, we have observed that cross-company experiments in embedded software are not far from within-company experiments in terms of *pd*, *pf* and balance rates. We can assume that embedded software systems contain similar information in their static code attributes. Therefore, they do not present significant changes in the predictions when embedded systems data from different organizations have been used to train the defect prediction model. Similarly, studies done in cost estimation area present that business-specific and company-specific approaches mostly deliver better predictions when compared to cross-company approaches [53]. While none of the first two approaches differ significantly, the results support the fact that cross-company analysis should be done using homogenous data [53]. Homogeneity is also an important issue for delivering better predictions in defect prediction, on the other hand, the best way for the companies is to

collect their own data. Cross-company data can only be an alternative for project managers to see the immediate results of their software and their defect prediction model.

Final analysis of our experimental results is related with data such that we have tried to increase information content to improve prediction performance of defect prediction in embedded software. We have used new metrics extracted from requirements documents of three embedded software. We have replicated cross- vs. within-company analysis with additional metrics. Results present that defects in software are closely related with early development lifecycle. Requirements which are not written well or subject to misinterpretation are one cause of defects in software. Therefore, they should be included into defect prediction studies and matched with modules that seem defect-prone. Developers can use this information to re-evaluate their requirements and modules at the same time. Moreover, it is seen that requirements also contain similar properties for the projects from embedded domain, since adding new metrics has increased the prediction performance of cross-company experiments. We can collect requirements metrics as well as code metrics from different organizations to catch defective modules of projects within a company. However, we can not conclude that defect prediction using cross-company data with both metrics has reached better predictions as using within-company data. We need more projects' requirements metrics to observe the performance of cross-company analysis with additional metrics.

6.5. Threats to Validity

First threat to validity is the sampling bias like in any empirical data mining experiments [7]. We try to surpass sampling bias by using variety of projects from different software systems in our comprehensive defect prediction study. The second threat is the ordering effect of data which is also avoided by randomizing the order at several iterations of the experiments. Another threat to validity is the learner used in the proposed defect prediction model. Data mining research contains various algorithms and methodologies such that trying all of them is impossible within the context of this study. However, we have used the most popular data mining method, Naïve Bayes, as the base-learner of our model. We know that Naïve Bayes proposes the best prediction performance so far among defect prediction studies [7]. To validate our results, we compare them with the results of

previous research that uses Naïve Bayes in its defect prediction model. Final threat to validity is difference between characteristics of the projects in NASA and SOFTLAB repositories. Projects from NASA are widely used in various defect prediction studies [7, 11, 21, 22]; their requirements documents are publicly available for analyzing which type of system they are operating in. Therefore, we ensure that they come from embedded systems and contain reliable data. We assume that SOFTLAB projects also represent similar characteristics such that they are parts of large embedded systems. We investigate several attributes to validate the type of software developed in the company.

7. CONCLUSION

Defect prediction research has become popular since companies started to invest their resources to process and quality improvement activities rather than time-consuming testing phases. We have proposed a defect prediction model that addresses problems of a specific domain, i.e. embedded systems. We have analyzed the embedded software and found unique characteristics that define the quality of embedded software. Since embedded software interacts with the overall system, the cost of fixing any defect or malfunctioning may be very expensive. Therefore, defect prediction models are paramount for companies that operate in embedded software industry.

We have built our research questions based on three pillars of defect prediction. First, we have focused on the performance of defect prediction algorithms. We have constructed a cascading learning mechanism where the model learns from the misclassifications of its preceding learners to train the next learners. Our proposed model takes static code attributes as inputs and classifies modules of software as defect-prone or defective. Results of our proposed model proved that we can decrease false alarm rates with a cascading approach for embedded software systems. However, the improvements are so minimal for both false alarm and detection rates. Therefore, we have expanded our defect prediction analysis by investigating data and the algorithm together for embedded software.

We have designed two experiments, in which we calibrate our model with projects either from past data within a company or from different embedded systems. We have replicated the study of Menzies *et al.* [11] for embedded software systems by introducing our defect prediction model that would reduce high false alarm rates of cross-company analysis. From the results of cross- vs. within- company analysis, we have observed that prediction performance of cross-company data is generally close to within-company experiments. In addition, cascading Naïve Bayes managed to decrease false alarm rates of cross-company analysis. This encourages us to use our proposed algorithm in domain-specific studies with homogenous data. On the other hand, the choice of cross- vs. within-company data in defect prediction still remains as an open question. We can propose cross-company analysis as an alternative plan to within-company defect prediction for

companies which can collect data from resources that are similar to themselves. On the other hand, results once more prove that the best predictions can be made using within-company data.

Finally, we have measured the information content of static code attributes by adding new metrics from early development lifecycle to our proposed defect prediction model. We have first defined that problems in requirements cause frequent changes in the code that leads to defects in embedded software. Then we proved our observation by adding requirements metrics as new inputs to our defect prediction model. Results show that for only three projects we can increase the information content with new metrics and improve our prediction performance. On the other hand, this analysis indicates that companies should use within-company data when they add requirements metrics to their defect prediction models.

Our research has both academic and practical contributions. We have proved to the academic community that algorithmic changes should only be the tip of the iceberg to construct a better defect predictor. Information content of data and the resources used to train the model directly influence the performance of a defect predictor. Researchers should carefully investigate data collected from projects by focusing on its homogeneity in terms of software domain to calibrate their model. Moreover, new metrics from other stages of software development lifecycle should be combined to predict defective modules in the software. The critical success factors in defect prediction research are to use a better designed data as well as a better algorithm and experimental design in software.

There are practical contributions of this study that guide software managers and developers through process improvement, accurate measurement and decision making. Learning based defect prediction models are proven to be useful as secondary decision tools for project managers to wisely allocate their resources and hence to save time and money during the test phase. In a highly competitive market with tight profit margins, those models guide developers to specific parts of the software so that they could effectively allocate their scarce resources.

Our proposed cascading Naive Bayes delivered successful predictions to practitioners from embedded software in terms of false alarm rates. Specifically, software managers should prefer using our model on embedded software which is estimated as highly defect-prone. With the help of a cascading approach, we can reduce the amount of files that are wrongly classified as defective. This improvement in false alarm rates also lowers the time allocated for inspecting the modules our defect prediction model catches as defective.

Another contribution to the embedded software industry is related with the data collection effort. We have proven that companies should prefer using their own data although it is time consuming until the company correctly establishes the process. In the absence of data and limited time for a prediction, software developers can use cross-company data. Moreover, cross-company data should be carefully selected from the projects that operate in embedded domain. Finally, requirements should be prioritized as well as testing phases in the companies in order to lower defect rates in the software product.

Our observations led future research directions for defect prediction studies. First of all, new algorithms could be added to the model to see the effects of cascading approach better. Nevertheless, proposing new algorithms for better defect prediction performance may not be useful, since the information content of static code attributes as the inputs of defect predictors is limited. We need to manipulate data to capture new information. We learned that requirements metrics have a positive impact on predicting defective modules in software. We should further examine design metrics to combine problems in design, requirements and coding phases. Additionally, we need to observe noise in data collected from the projects. For two new SOFTLAB projects, we could not effectively predict the defect-prone modules either with within-company or cross-company experiments. To overcome this problem, a preprocessing step that eliminates noise in data can be added to defect prediction studies. Finally as a future work, past projects with similar characteristics in terms of complexity, size and other measures can be allocated to train the defect prediction model for new projects, instead of using within- vs. cross-company data from the same domain.

APPENDIX A: STATIC CODE ATTRIBUTES

In Table A.1, the set of available static code attributes that are used in defect prediction models are presented with their descriptions [7, 10]. 29 of these attributes are used in SOFTLAB datasets (marked as S), while only 17 of them are common for both NASA and SOFTLAB datasets (marked as C).

Table A.1. Set of static code attributes collected from software [7].

Datasetused	**Name**	**Description**
S, C	Loc_total	*Total number of lines in a module.*
S	Loc_blank	*Lines with only white space.*
S, N	Loc_code_and_comment	*Lines that contain both code and comment.*
S, N	Loc_comments	*Source lines of code that are purely comments*
S, N	Loc_executable	*Source lines of code that contain only code and white space.*
S, N	Num_unique_operators (μ1)	*Number of unique operators found in a module.*
S, N	Num_unique_operands (μ2)	*Number of unique operands found in a module.*
S, N	Num_operators (N1)	*Total number of operators found in a module.*
S, N	Num_operands (N2)	*Total number of operands found in a module.*
S	Halstead vocabulary	*M1 + μ2*
S	Halstead length	*N =operands + operators*
S, N	Halstead volume	*V = N*log(n)*
S, N	Halstead level	*L = V*/V*
S, N	Halstead difficulty	*D = 1/L*
	Halstead content	*I = Ĺ/V where Ĺ = (2/ μ1)*(μ2/N2)*
S, N	Halstead effort	*E = V / L*
S, N	Halstead error_estimate	*B = V / S**
S, N	Halstead prog_time	*T = E / 18*
S, N	Branch_count	*Number of branches in a given module*
S	Call_pairs	*Number of calls to other functions in a module.*
S	Condition_count	*Number of conditionals in a given module.*

S	Decision_count	*Number of decision points in a module.*
S	Decision_density	*condition count / decision count*
S	Design_density	*Id(g) = Iv(g) / V(g)*
	Edge_count	*Number of edges found in a given module.*
	Modified_condition_count	*Every condition shown to independently affect a decision outcome.*
S	Multiple_condition_count	*Number of multiple conditions that exist within a module.*
	Node_count	*Number of nodes found in a given module.*
S, C	Cyclomatic complexity	*V(g) = edge count - node count + 2*num. unconnected parts in g*
S	Cyclomatic density	*V(g)/executable lines of code*
S, C	Design complexity	*Iv(g) = call pairs*
	Essential complexity	*Ev(g) = V(g) – num. D-structured subflowgraphs*
S	Normalized_cyclomatic_complexity	*Norm V(g) = V(g) / lines of code*
	Parameter_count	*Number of parameters to a given module.*
S	Formal parameters	*Number of formal parameters in a module.*
	Per cent_comments	*100*(Loc_comment/loc_total)*

APPENDIX B: TEXTUAL REQUIREMENTS METRICS

Requirements metrics extracted from requirements documents are given below [12]. For a detailed description with sample words for each attribute, see [49].

Table B.1. Textual requirements metrics collected from documents [12].

Name	**Description**
Action	Number of actions the requirement needs to be capable of performing.
Conditional	Represents whether the requirement will be addressing more than one condition.
Continuance	Phrases that follow an imperative and precede the definition of lower level requirement specification.
Imperative	Those words and phrases that command that something must be provided.
Incomplete	Phrases such as TBD or TBR.
Option	Those words that give the developer latitude in the implementation of the specification that contains them.
Risk_level	A calculated metrics based on weighted averages from metrics collected for each requirement.
Source	Represents number of sources requirement will interface with or receive data from.
Weak_phrase	Clauses that are apt to cause uncertainty and leave room for multiple interpretations.

APPENDIX C: GRAPHICAL REPRESENTATION OF NASA RESULTS

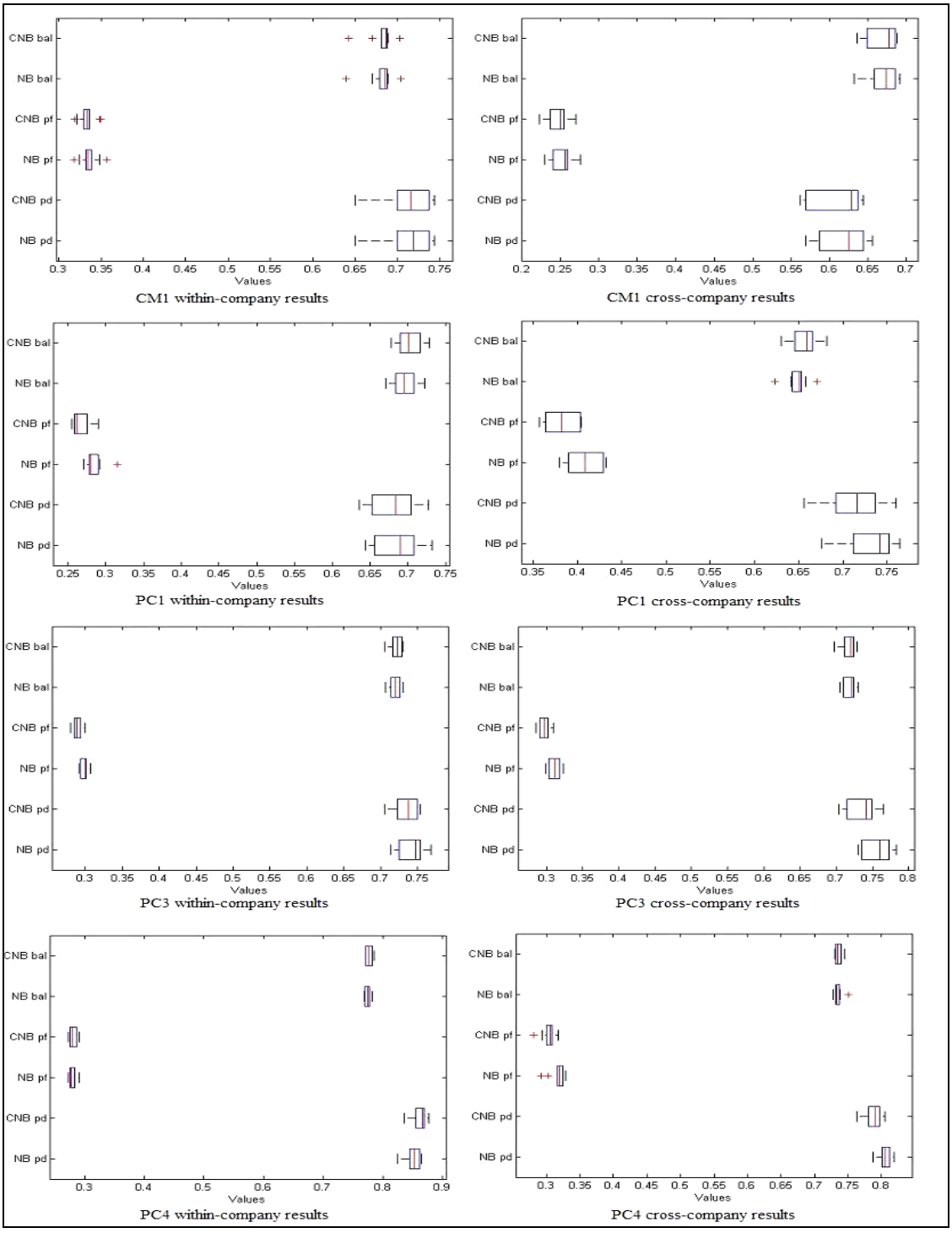
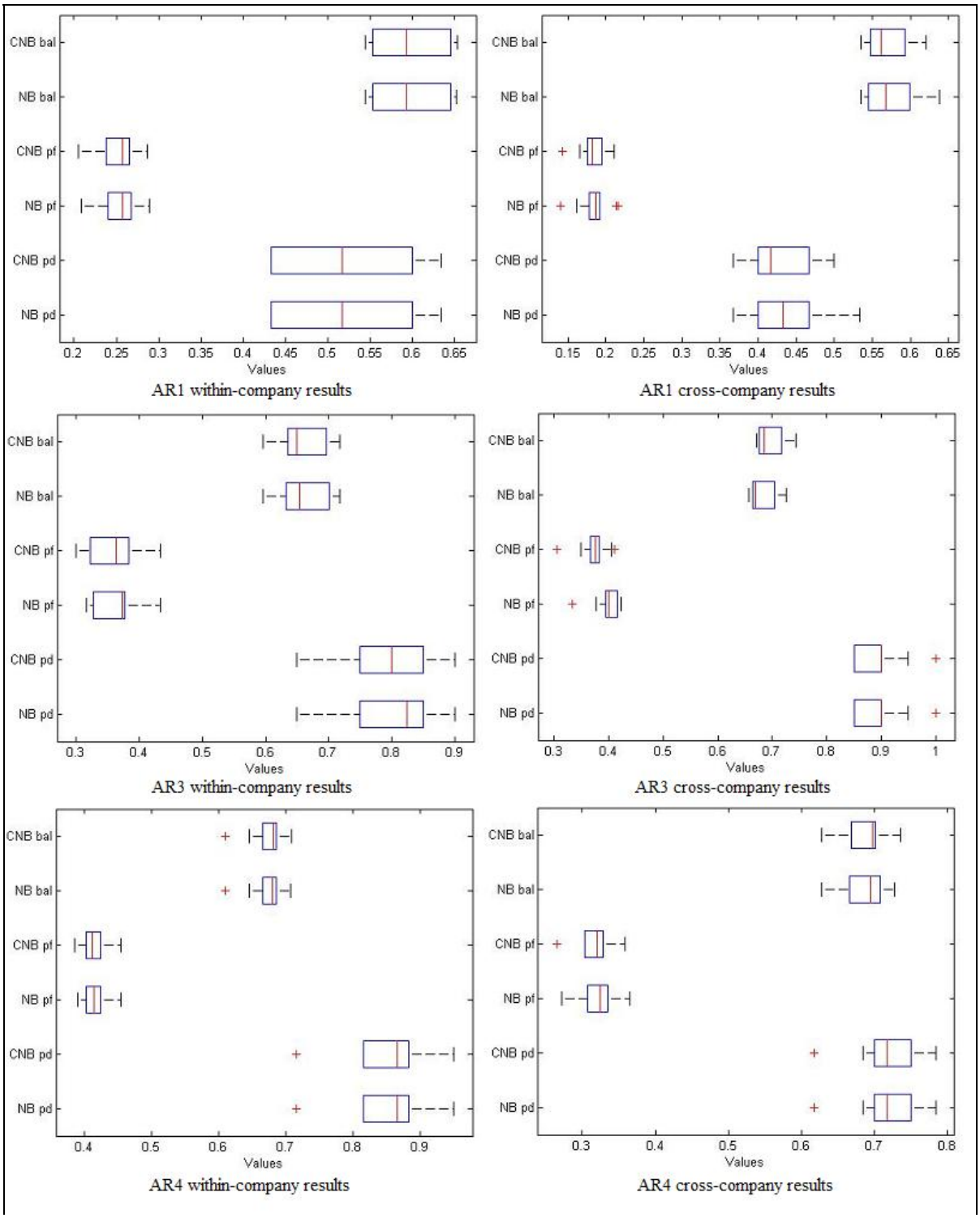


Figure C.1. Box plots of cross- vs. within-company analysis of NASA datasets.

APPENDIX D: GRAPHICAL REPRESENTATION OF SOFTLAB RESULTS



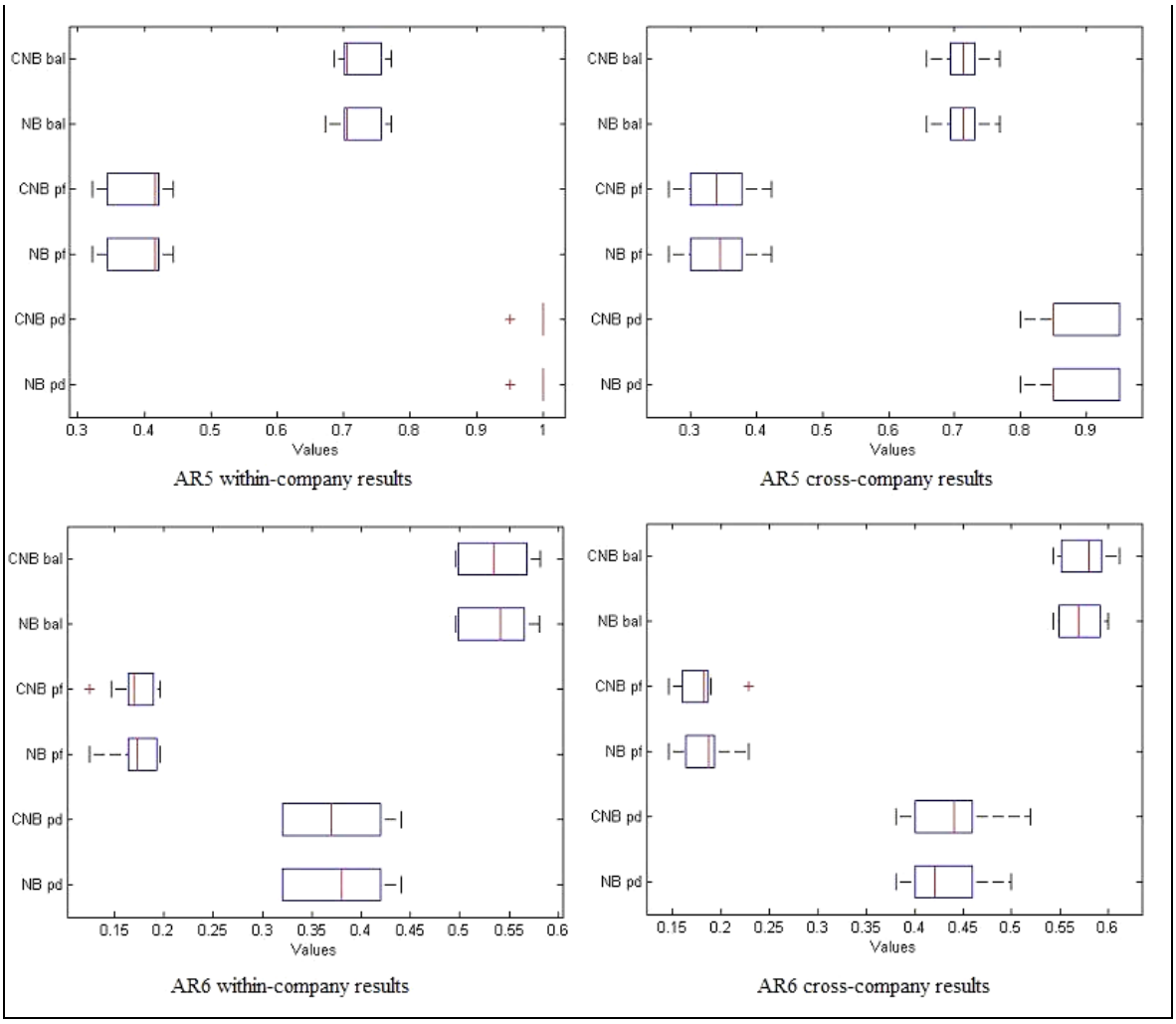


Figure D.1. Box plots of cross- vs. within-company analysis for SOFTLAB datasets.

APPENDIX E: GRAPHICAL REPRESENTATION OF CC VS. WC ANALYSIS

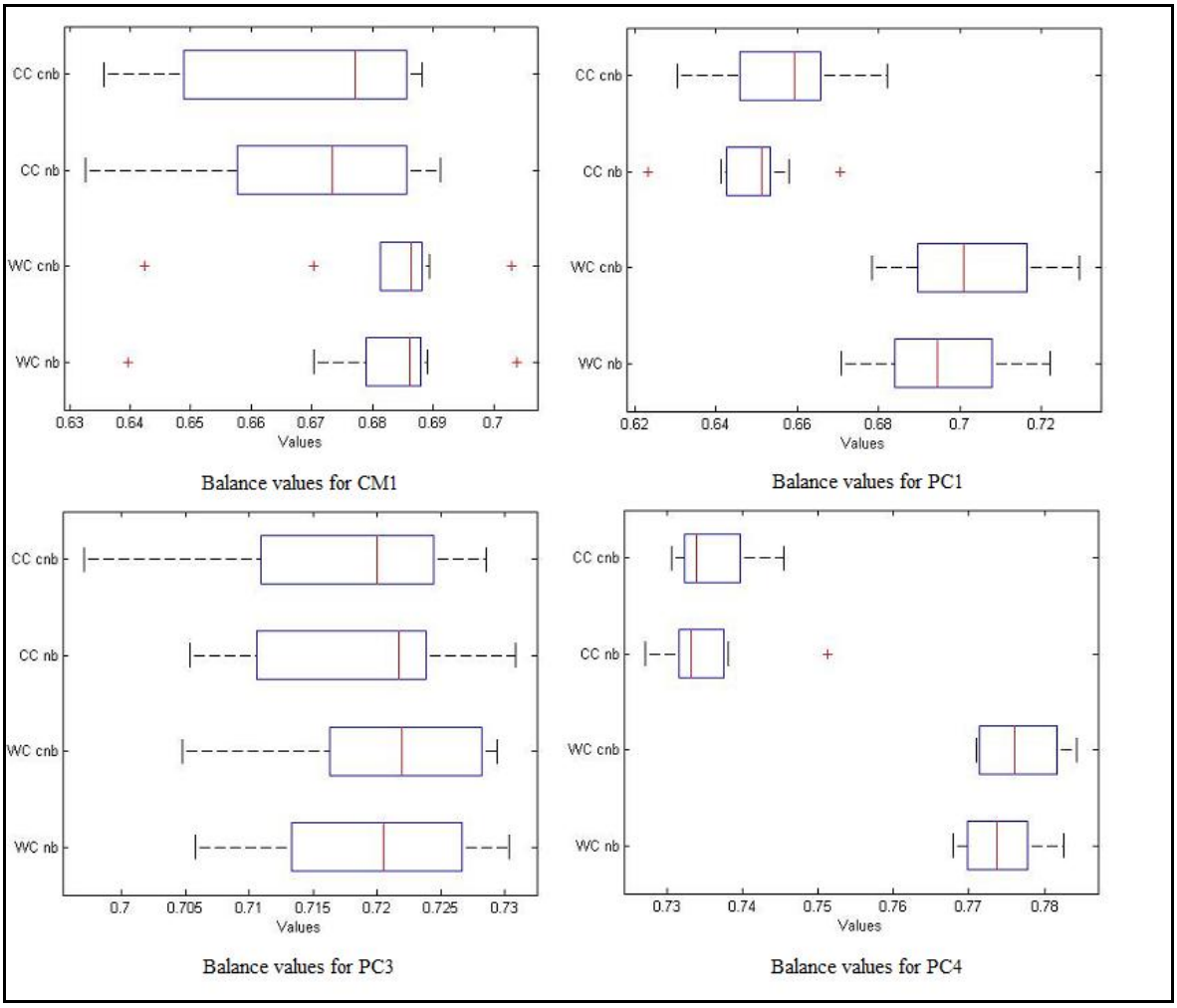


Figure E.1. CC vs. WC analysis using NASA datasets

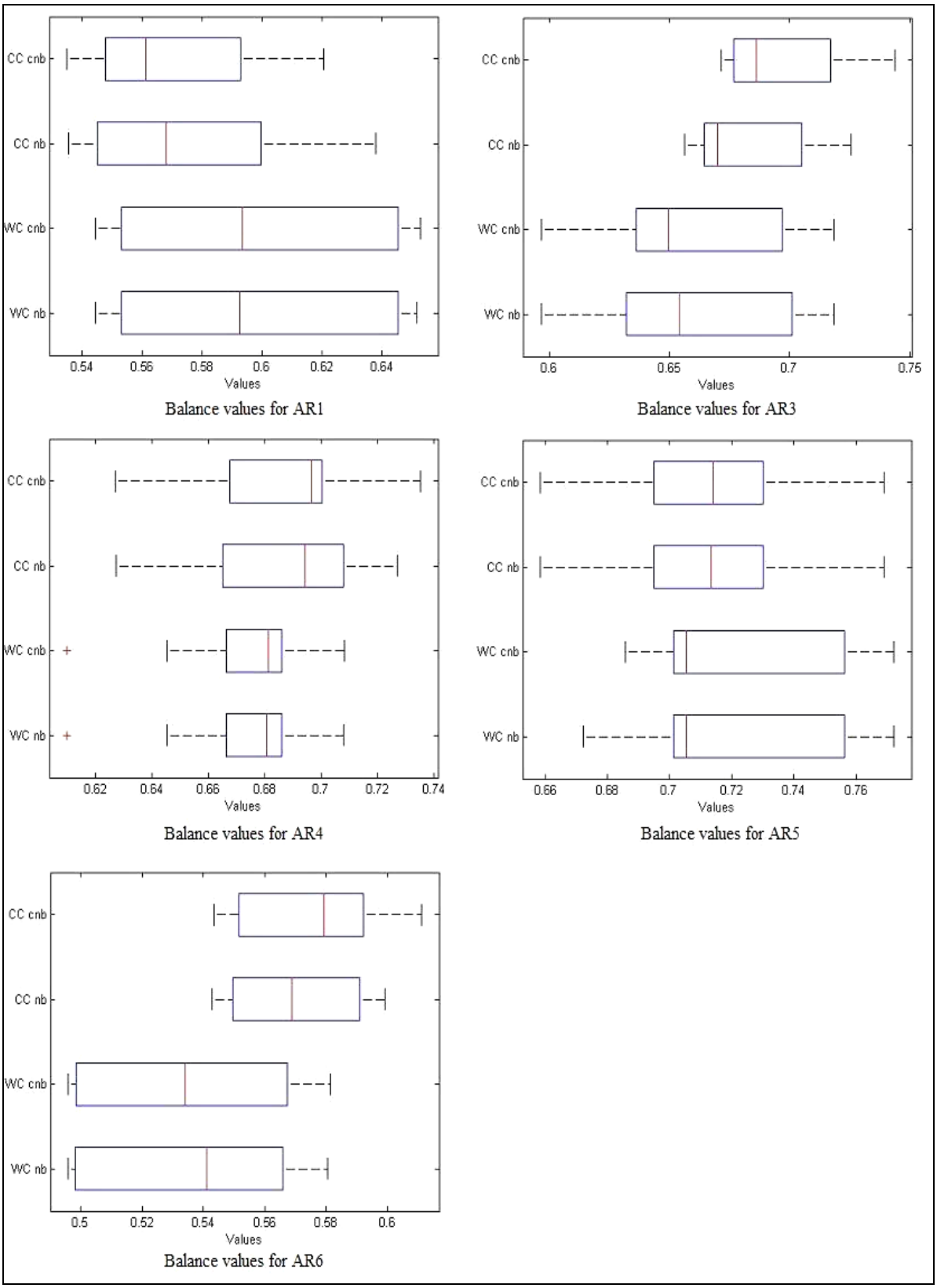


Figure E.2. CC vs. WC analysis using SOFTLAB datasets

APPENDIX F: SIGNIFICANCE T-TESTS OF THE MODEL

Below, t-tests are presented for two different experiments. First results represent the significance of *Cascading Naïve Bayes* on *simple Naïve Bayes* algorithm for three performance measures, *pd, pf* and *bal*. For each dataset, results of both cross-company and within-company experiments are presented in Table F.1. In Table F.1, if *CNB* significantly dominates the other for the selected performance measure of the datasets, then it is indicated as 1, otherwise, it is indicated as 0. If none of the algorithms is significant to each other, then it is marked as X. Cross-company experiments are marked with CC, whereas within-company is marked with WC.

Table F.1. T-tests for CNB vs. NB.

		pd	pf	bal
CM1	CC	X	X	X
	WC	X	X	X
PC1	CC	X	1	X
	WC	X	1	X
PC3	CC	1	1	X
	WC	X	1	X
PC4	CC	1	1	X
	WC	X	X	X
AR1	CC	0	X	X
	WC	X	X	X
AR3	CC	X	1	X
	WC	X	X	X
AR4	CC	X	X	X
	WC	X	X	X
AR5	CC	X	X	X
	WC	X	X	X
AR6	CC	1	1	1
	WC	X	X	X
TOTAL	wins	3	7	1
	ties	14	11	17
	losses	1	0	0

Second, we have computed t-tests, presented in Tables F.2 and F.3, for cross- vs. within- company experiments. We have analyzed whether cross-company or within-company data should be preferred for defect prediction in embedded software. If using WC data is significantly better than CC data, then it is indicated as 1, otherwise, it is marked as 0. If none of them is significant to each other, it is marked as X.

Table F.2. T-tests for CC vs. WC analysis using NASA datasets.

	pd	pf	bal
CM1	1	0	X
PC1	0	1	1
PC3	X	1	X
PC4	1	1	1
Avg	1	X	1

Table F.3. T-tests for CC vs. WC analysis using SOFTLAB datasets.

	pd	pf	bal
AR1	1	X	X
AR3	1	X	0
AR4	1	0	0
AR5	1	0	X
AR6	0	X	0
Avg	X	X	X

REFERENCES

1. IEEE Standards Association, *IEEE Standard Glossary of Software Engineering Terminology,* http://standards.ieee.org/reading/ieee/std/se/610.12-1990.pdf, 2007.

2. Fenton, N., P. Krause and M. Neil, "Software Measurement: Uncertainty and Causal Modeling"*, IEEE Software*, Vol. 19, No.4, pp. 116-122, July/August 2002.

3. Pressman, S., *Software Engineering: A Practitioner's Approach,* McGraw-Hill Education, 2005.

4. Adrion, R.W., A.M. Branstad and C.J. Cherniavsky, "Validation, Verification and Testing of Computer Software"*, ACM Computing Surveys*, Vol. 14, No. 2, pp. 159-192, June 1982.

5. Khoshgoftaar, M.T., "Predicting Fault-prone Software Modules in Embedded Systems with Classification Trees"*, Proceedings of the Fourth IEEE International Symposium on High-Assurance Systems Engineering*, Washington, USA, pp. 105-112, 17-19 November 1999.

6. Turhan, B. and A. Bener, "Software Defect Prediction: Heuristics for Weighted Naive Bayes", *Proceedings of the Second International Conference on Software and Data Technologies,* Barcelona, Spain, 22-25 July 2007.

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

8. Amasaki, S., Y. Takagi, O. Mizuno and T. Kikuno, "Constructing a Bayesian Belief Network to Predict Final Quality in Embedded System Development", *IEICE Transactions on Information and Systems,* Vol. E88-D, No.6, pp. 1134-1141, June 2005.

9. Khoshgoftaar, M.T. and K. Gao, "Assessment of a Multi-Strategy Classifier for an Embedded Software System", *Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence,* Washington, pp. 651-658, 13-15 November 2006.

10. Oral, D.A., "Defect Prediction for Embedded Software", *MS Thesis,* Computer Engineering, Bogazici University, 2007.

11. Menzies, T., B. Turhan, A. Bener and J. Distefano, "Cross- vs. Within-Company Defect Prediction", *Technical Report,* Bogazici University, Turkey, 2008.

12. Jiang, Y., B. Cukic and T. Menzies, "Fault Prediction using Early Lifecycle Data", *Proceedings of the 18th IEEE International Symposium on Software Reliability,* Sweden, pp. 237-246, 5-9 November 2007.

13. Dawson, R. and J.A. Nolan, "Towards a Successful Software Metrics Programme", *Proceedings of the 11th Annual International Workshop on Software Technology and Engineering Practice,* Amsterdam, pp. 48-51, 19-21 September 2003.

14. Kan, S.H, *Metrics and Models in Software Quality,* Addison-Wesley, New York, 2005.

15. Chrissis, B.M., M. Konrad and S. Shium, *CMMI Guidelines for Process Integration and Product Improvement,* Addison-Wesley, USA, 2005.

16. Akiyama, F. "An Example of Software System Debugging", *Information Processing,* Vol. 71, pp. 353-379, 1971.

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

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

19. Turhan, B. and A. Bener, "A Multivariate Analysis of Static Code Attributes for Defect Prediction", *Proceedings of the Seventh International Conference on Quality Software,* Portland, USA, 11-12 October 2007.

20. Marchenko, A. and P. Abrahamsson, "Predicting Software Defect Density: A Case Study on Automated Static Code Analysis", *Proceedings of the 8th International Conference on Agile Processes in Software Engineering and Extreme Programming,* Como, Italy, pp. 137-140, 18-22 June 2007.

21. Challagula, U.B.V., B.F. Bastani, L. Yen and A.R. Paul, "Empirical Assessment of Machine Learning based Software Defect Prediction Techniques", *Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems,* Sedona, USA, pp. 263-270, 2-4 February 2005.

22. Menzies, T., B. Turhan, A. Bener, G. Gay, B. Cukic and Y. Jiang, "Implications of Ceiling Effects in Defect Predictors", *PROMISE 2008 Workshop (part of the 30th International Conference on Software Engineering),* Germany, pp. 47-54, 12-13 May 2008.

23. Shepperd, M. and D. Ince, "A Critique of Three Metrics", *J. Systems and Software,* Vol. 26, No. 3, pp. 197-210, 1994.

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

25. Nagappan, N. and T. Ball, "Static Analysis Tools as Early Indicators of Pre-Release Defect Density", *Proceedings of the 27th International Conference on Software Engineering,* USA, pp. 580-586, 15-21 May 2005.

26. Cartwright, M. and M. Shepperd, "An Empirical Investigation of an Object-Oriented Software System", *IEEE Transactions on Software Engineering,* Vol. 26, No. 8, pp. 786-796, August 2000.

27. Chidamber, R.S. and F.C. Kemerer, "A Metrics Suite for Object Oriented Design", *IEEE Transactions on Software Engineering,* Vol. 20, No.6, pp. 476-493, June 1994.

28. Basili, R.V., C.L. Briand and L.W. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators", *IEEE Transactions on Software Engineering,* Vol. 22, No.10, pp. 751-761, October 1996.

29. Harrold, M.J., "Testing a madmap", *Proceedings of the Conference on the Future of Software Engineering,* New York, pp. 61-72, 2000.

30. Tahat, B.V., B. Korel and A. Bader, "Requirement-Based Automated Black Box Test Generation", *Proceedings of the 25th Annual International Computer Software and Applications Conference,* Chicago, pp. 489-495, 8-12 October 2001.

31. Shull, F., B.V. Boehm, A. Brown, P. Costa, M. Lindwall, D. Port, I. Rus, R. Tesoriero and M. Zelkowitz, "What We Have Learned About Fighting Defects", *Proceedings of the 8th International Software Metrics Symposium,* Canada, pp. 249-258, 4-7 June 2002.

32. Shull, F., I. Rus and V. Basili, "How Perspective-Based Reading Can Improve Requirements Inspections", *IEEE Computer,* Vol. 33, No. 7, pp. 73-79, July 2000.

33. Song, Q., M. Shepperd, M. Cartwright and C. Mair, "Software Defect Association Mining and Defect Correction Effort Prediction", *IEEE Transactions on Software Engineering,* Vol. 32, No. 2, pp. 69-82, February 2006.

34. Fenton, N., M. Neil, W. Marsh, P. Hearty, D. Marquez, P. Krause and R. Mishra, "Predicting software defects in varying development lifecycles using Bayesian nets", *Information and Software Technology,* Vol. 49, pp. 32-43, 2007.

35. Fenton, E.N. and M. Neil, "A Critique of Software Defect Prediction Models", *IEEE Transactions on Software Engineering,* Vol. 25, No. 5, pp. 675-689, September/October 1999.

36. Arisholm, E. and C.L. Briand, "Predicting Fault-prone Components in a Java Legacy System", *Proceedings of the 5th International Symposium on Empirical Software Engineering,* Brazil, 21-22 September 2006.

37. Ostrand, J.T., J.E. Weyuker and M.R. Bell, "Predicting the Location and Number of Faults in Large Software Systems", *IEEE Transactions on Software Engineering,* Vol. 31, No. 4, pp. 340- 355, April 2005.

38. Janes, A., M. Scotto, W. Pedrycz, B. Russo, M. Stefanovic and G. Succi, "Identification of defect-prone classes in telecommunication software systems using design metrics", *Information Sciences,* Vol. 176, pp. 3711-3734, 2006.

39. Alpaydin, E., *Introduction to Machine Learning,* MIT Press, Massachusetts, 2004.

40. Munson, J. and T.M. Khoshgoftaar, "Regression modeling of software quality: empirical investigation", *Journal of Electronic Material,* Vol. 19, No. 6, pp. 106-114, 1990.

41. Padberg, F., T. Ragg and R. Schoknecht, "Using machine learning for estimating the defect content after an inspection", *IEEE Transactions on Software Engineering,* Vol. 30, No. 41, pp. 17-28, 2004.

42. Kitchenham, A.B., E. Mendes and H.G. Travassos, "Cross- vs. within-company cost estimation studies: A systematic review", *IEEE Transactions on Software Engineering,* Vol.33, No.5, pp. 316-329, May 2007.

43. Lee, A.E., "Embedded Software", *Advances in Computers,* vol. 56, Academic Press, London, 2002.

44. Fagan, M., "Design and Code Inspections to Reduce Errors in Program Development", *IBM Systems Journal,* Vol. 15, No. 3, pp.182-211, 1976.

45. Turhan, B., A. Tosun and A. Bener, "An Industrial Application of Classifier Ensembles for Locating Software Defects", submitted to *Data and Knowledge Engineering Journal*, 2008.

46. Boetticher, G., T. Menzies and T. Ostrand, PROMISE Repository of empirical software engineering data http://promisedata.org/ repository, West Virginia University, Department of Computer Science, 2007.

47. Heeger, D., *Signal Detection Theory,* http://www.cns.nyu.edu/~david/handouts/sdt/sdt.html, 1998.

48. NASA/WVU IV & V Facility, *Metrics Data Program,* http://mdp.ivv.nasa.gov, 2004.

49. ARM: Automated Requirements Measurement Tool, *NASA Goddard Space Flight Center Tools,* http://satc.gsfc.nasa.gov/tools/download, 2004.

50. Hall, M. and G. Holmes, "Benchmarking Attribute Selection Techniques for Discrete Class Data Mining", *IEEE Transactions on Knowledge and Data Engineering,* Vol. 15, No. 6, pp.1437-1447, 2003.

51. Kocak, G., B. Turhan and A. Bener, "Predicting Defects in a Large Telecommunication System", *Proceedings of the 3rd International Conference on Software and Data Technologies,* Portugal, 5-8 July 2008.

52. Tosun, A., B. Turhan and A. Bener, "Ensemble of Software Defect Predictors: A Case Study", *Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement,* Germany, October 2008.

53. Premraj, R. and T. Zimmermann, "Building Software Cost Estimation Using Homogenous Data", *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement,* Spain, pp. 393-400, September 2007.