DEFECT PREDICTION FOR EMBEDDED SOFTWARE

by

Ataç Deniz Oral

B.S., Computer Engineering, Bilkent University, 2005

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

Graduate Program in Computer Engineering

Boğaziçi University

2007

ACKNOWLEDGEMENTS

I would like to thank my supervisor Assistant Professor Ayşe Bener for her continuous support and guidance throughout my research. This thesis has only been possible with her unlimited patience for helping me in finding the best in what I thought and what I did.

I would also like to thank Prof. Fikret Gürgen and Prof. Emin Anarım for kindly accepting to be in my thesis jury.

I am grateful to the members of Software Engineering Research Laboratory, especially Burak Turhan, for their insights to my research and above all for their patience in listening what I had to say.

My parents deserve special thanks for letting me be a kid with no worries for another two years in my life. My sister Ayça, was always at the other end of the phone to listen to my sarcastic jokes amidst my studies and contributed to this research by being a good listener.

Çitil family made me feel home and they supported me in all stages of my research. Having them at my side was like having my personal black hole for troubles.

I would also like to express my gratitude to all of the anonymous contributors of the open source software community. Their efforts made tedious jobs easier for me.

Finally I would like to thank Arçelik A.Ş. who supported my research under Arçelik-Boğaziçi University Joint Graduate Research Program.

This research is supported in part by Boğaziçi University Research Fund under grant number BAP-06HA104.

ABSTRACT

DEFECT PREDICTION FOR EMBEDDED SOFTWARE

As ubiquitous computing becomes the reality of our lives, the demand for high quality embedded software in shortened intervals increases. In order to cope with this pressure, software developers need new approaches to manage the development cycle: to finish on time, within budget and with no defects. Software defect prediction is one area that has to be focused to lower the cost of testing as well as to improve the quality of the end product.

This research proposes a defect prediction model specifically for embedded software systems. We utilize machine learning techniques in order to identify the complex relationship between software metrics and defect-proneness. Our proposed model involves three different machine learning techniques which have useful characteristics for defect prediction in embedded software. We combine the strengths of these machine learning techniques in order to obtain a general model for defect prediction in embedded software.

The resulting model may be used to assist the embedded software developers in planning the future iterations of their development life-cycle and allocating their limited testing resources more effectively. This will help embedded software developers in increasing the quality of their products by increasing the efficiency of defect removal strategies.

ÖZET

GÖMÜLÜ SİSTEMLERDE HATA KESTİRİMİ

Gömülü sistemlerin hayatımızın her alanına yayılması kısa sürede kaliteli gömülü yazılım üretilmesi için talebi arttırdı. Bu durumla başedebilmek ve geliştirme sürecini iyi yönetebilmek için yeni yaklaşımlara gerek duyulmaktadır. Yazılımda hata kestirimi, test aşamasının maliyetini düşürmek ve son ürünün kalitesini arttırmak için odaklanılması gereken alanlardan birisidir.

Bu araştırmada özellikle gömülü yazılımlar için geliştirilmiş bir hata kestirimi modeli önerdik. Yazılım ölçütleri ve hataya meyillilik arasındaki karmaşık ilişkinin modellenebilmesi için makine öğrenimi teknikleri kullandık. Öne sürdüğümüz model gömülü yazılımlarda hata kestirimi için uygun özellikleri olan üç makine öğrenimi tekniğininin birleşiminden oluşmaktadır.

Sonunda ortaya çıkan model, gömülü yazılım geliştirenlere yazılım geliştirme döngüsünün gelecekteki yinelenmelerinin planlanmasında ve test kaynaklarının doğru bir şekilde kullanılmasında yardımcı olacaktır. Böylece gömülü yazılım geliştiricileri hata ayırım aşamalarının etkinliğini arttırarak ürünlerinin kalitesini arttırabileceklerdir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

a	static code attribute
C	class
c	instance of a class
cov	covariance
E	error
e	defect information for a module
fn	false negative
fp	false positive
M	module
O	Big Oh
P	a set of embedded software systems
p, P	probability
pd	probability of detection
pf	probability of false alarm
S	embedded software system
tn	true negative
tp	true positive
λ	eigenvalue
σ	standard deviation
ANN	Artificial Neural Network
LOC	Lines of Code
NB	Naive Bayes
PCA	Principal Components Analysis
VFI	Voting Feature Intervals

1. INTRODUCTION

1.1. Motivation

Software quality is perceived differently by different people in the industry. However some definitions have been proposed. In [1] quality is defined as "conformance to requirements". This definition treats nonconformance to requirements as defects. In [2], authors define quality as "fitness for use". This definition takes into account the customers' expectations and therefore ascribes a more central role to the customer. In this definition quality is the value of the product from the customers' perspective. This value can be determined by taking into account variables such as price, performance, reliability, and satisfaction [3]. These definitions are helpful in understanding the term quality and its meaning. However practitioners face certain problems when trying to measure quality in their products. This is because "conformance to requirements" and "fitness for use" are hard to, arguably impossible to quantify [3]. Therefore when measuring quality in products one must use a narrower sense.

The question that needs to be answered is then how to quantify quality. The importance of this question is due to the fact that the way we quantify quality is highly related to our understanding of quality. In [3] the most basic meaning of "conformance to requirements" is given as lack of defects in the product. This is justified by the fact that it is not possible to provide the functionality specified in the requirements with too many functional defects. Quantification of the quality from this perspective might be possible by measuring the defect rate and reliability of the product [3]. The other definition of quality emphasizes customer satisfaction. In order to quantify this view certain quality attributes have been defined, such as capability, usability, performance, reliability. Customers' view of these attributes for a specific product can be measured using questionnaires designed for this purpose [3].

Software quality has many attributes, such as capability, usability, reliability [3]. However, not all attributes of quality are equally important in different types of software systems [3]. The work in this thesis aims to assist embedded software development. It is

appropriate to discuss the areas in which embedded systems are used before embarking on a discussion on embedded software. There are variety of embedded software and they are pervasive [4]. These systems are used in many industries, including automotive industry, house-hold appliances, telecommunications, and aerospace [5]. As opposed to other software systems, the principal role of embedded software is interaction with the other parts of the system that it is embedded in or the physical world [5]. In certain types of embedded software real-time constraints can also be imposed as requirements. Whether they contain real-time requirements or not, timeliness, dependability, and reliability are considered of utmost importance in these systems [4]. This is also due to the fact that there are numerous applications of embedded systems which contain hard or soft real-time constraints, with mission-critical or safety-critical operations [26].

The presence of defects in a software system severely impacts the quality of a software system [3]. Defects in a software system indicate that some of the required functionalities of the system are missing or incorrect, and on top of that, important attributes of software quality such as reliability and dependability are reduced [3]. The presence of defects, and the consequent poor reliability and dependability have severe consequences in embedded systems [26], [27]. Therefore defect prevention, and elimination methods pose important research questions in embedded software systems [17].

Many tools and techniques have been proposed in various academic research for eliminating the defects in a software product [29], [30], [31]. If the function of a program is viewed as a transformation of its input to some specified output, testing can be considered as a technique that is used for ensuring that this transformation is correct [32]. There are many methods for this purpose, such as inspections and formal verification, they are collectively named as verification, validation, and testing (VV&T) [7]. Proper use of techniques are strongly related to software quality [7]. Prior research have revealed that efforts towards reducing the costs or improving the efficiency of verification, validation, and testing techniques result in substantial improvements in software quality [33]. One of the main problems with testing is that its efficiency increases with the amount of resources allocated to it, therefore management of VV&T phases of software development is considered to be an important research area [33].

Collecting and analyzing data pertaining to the development process is proposed as a means for improving the software products [33]. The aim in the collection and analysis of the data is to measure certain properties of the software development process and improve the process according to the results of these measurements [33]. A variety of software metrics have been proposed, including in-process metrics, code metrics, and availability metrics [3]. All of these metrics capture certain information about the software system that is being developed. These include understandability, testability, maintainability, and complexity [3]. If the information contained in these metrics are used for predicting the defects in parts of a software system, the efficiency of testing activities can be increased [22]. Defect prediction can be used to identify the defect prone parts of the system before they are tested [34]. Such identification enables developers to focus their limited effort to the testing of the defect prone parts of the system.

The demand for embedded systems has increased dramatically in the last decade. This increase in the demand has increased the pressure on the embedded software developers during the development process. This is mainly because the increase in the demand also resulted in an increase in the functionality of the embedded systems, and the size of the software running on them [9]. The increasing demand and the resulting increase in the size of the embedded software left shorter time intervals for development. In such a situation, developers need every methodology that they can use to ensure the reliability of the system being developed, because generally there is not enough time or resources to inspect or test the entire system [16].

There are many motivating factors for using defect prediction techniques in embedded software development. Embedded systems are deployed pervasively once they are developed and embedded software systems are generally part of larger systems [6], [9]. Therefore if defects are found in embedded software the impact might not only be on the operation of the embedded software itself. Due to the close coupling of multiple systems in an embedded system, once the overall system is deployed it is harder to remedy the faults [9]. Current reliability engineering practices base their assumptions on the number of defects identified during testing [3]. However the results obtained after the testing phase do not leave enough slack time to make significant changes to the embedded

software [10]. Therefore highly-mature organizations that operate in embedded systems market find defect prediction useful [10].

The diversity of the embedded systems is another motivation for the use of defect predictors in this area. Embedded computing plays a crucial role both in daily life and many industrial sectors [5], [17]. This puts a pressure on the developers both in terms of development time and pricing of the products [9], [17]. Studies have shown that developing highly-dependable and highly-reliable software costs fifty per cent more for each source instruction in the code [18]. On top of this it is not possible to ascertain field reliability (or true reliability) until the product is delivered to the customers [19]. This problem makes maintainability and testability (which directly impact reliability) important research challenges to be addresses for embedded software [17]. Prior research have demonstrated to a certain degree that early warning mechanisms of product defects can help software organizations in directing the corrective actions towards the quality of the software [19]. At the same time it is possible to use such warning mechanisms to concentrate effort to the defect-prone parts of the software in order to increase the true reliability of the software [20].

One other motivation for using defect prediction techniques in embedded software development is the frequent use of dynamic code synthesis techniques. In dynamic code synthesis techniques developers synthesize (generate) the code for parts of the new software system from formal abstractions [35]. However it is necessary to dynamically assess the dependability of the synthesized code [21]. If the dynamic assessment of the code contains signals for low dependability, this information can be used to give feedback to the system operators on altering mission-objectives. On top of that preventive measures can be placed in the code for masking the predicted defects and dependability can be managed pro-actively [21].

The motivation for this research is the belief that defect prediction for embedded software will address the aforementioned research challenges successfully. These research challenges include management of the testing phases, reliability assessment, and control and measurement of the overall system quality.

Defect prediction techniques are used to identify the relationships between software metrics and the quality attributes of software products [14], [20]. Researchers have used many techniques for identifying this relationship [9], [12], [13]. The relationship between the metrics data and quality attributes is complex and nonlinear [20]. This problem and the recent advances in artificial intelligence research encouraged researchers for using data mining techniques [14], [20]. There are many factors affecting the success of a defect predictor [14]. Some of these are the accuracy of the metrics data, the amount of information contained in the data, and the techniques used for building predictors [14], [21]. This research aims to utilize the insights of the previous work in developing a useful defect predictor for embedded software. The characteristics of embedded software and the characteristics of metrics data gathered from these systems will be used for determining the requirements of an appropriate model. On top of that there are certain problems with the proposed methods for defect prediction [14], [21], this research aims to tackle those problems in the context of embedded software systems.

1.2. Outline

A brief background information on the software systems under consideration is presented in the next chapter. After the discussion of the systems of concern, ways of achieving quality in those systems is summarized, and finally the chapter is concluded with a discussion of the state of the art defect prediction techniques, and artificial intelligence methods used for this purpose.

Third chapter is devoted to the presentation of the problem statement of this research. The choice for this statement is justified by scrutinizing the hypothesis, and other possible approaches.

Chapter 4 contains a presentation of the proposed model for solving the problem under consideration. The details of our model, its limitations, and advantages are presented in this part.

The data used in this research and data collection methodology are discussed in Chapter 5. The information contained in this data and its validity are discussed.

The experiments performed in order to utilize the data available to this research is given in detail in Chapter 6. The statistical analysis of the results of these experiments are also given in this chapter.

The last chapter is reserved for a discussion of the results of the study carried out for this research. A summary of the research is presented and conclusions drawn from this research is stated. Finally contributions of the research is highlighted and the discussion is concluded with a listing of future research directions.

2. BACKGROUND

2.1. Overview

Both defect prediction and embedded software are broad areas of research. Therefore we must examine the prior research in these areas when making a research at their intersection. We examine embedded software and their characteristics in the following section. The later sections are devoted to a discussion of defect prediction techniques and models. Finally some machine learning techniques that are related to our research is presented.

2.2. Embedded Software

The research in this thesis targets embedded software. Before defining which systems are embedded systems, we should stress that a single statement which comprehensively defines all embedded systems does not exist [5]. Therefore it might be appropriate to view the situation from a variety of perspectives. The first perspective is the requirements perspective. This perspective states that since these systems are embedded in other systems, we can say that they are systems which does not function, or cannot function on their own [5]. Their purpose is to perform a certain amount of the requirements of the overall system [6]. Therefore embedded software is the software part of these systems.

Another perspective takes into account the hardware and software parts of these systems, and chooses to look at the picture from a systems design perspective. This perspective claims that when the system of concern has a tight hardware software coupling, which are both designed for a particular purpose, we can consider it as an embedded system [5].

The final and perhaps most comprehensive perspective does not try to give a definitive statement for embedded systems. In this approach the characteristics of embedded systems are captured. The starting point for this view is that any statement like

software on small computers will fall short, and can even be considered naive [4]. The operation of software is often viewed as a transformation of its inputs to some output [32]. However interactions with other parts of the system or the environment play a more central role in the operation of embedded software. These interactions require that the embedded software should conform to the time constraints coming from the environment (timeliness). Besides, these systems should be able to react continuously to their environment (reactivity) and they must not block waiting for events that will not occur (liveness). Therefore whenever these characteristics are required in the operation of a system it is considered embedded software [4].

The aim of this research is not demonstrating the differences of embedded systems, or trying to define them. Therefore in light of the aforementioned definitions we do not choose to adopt one, instead we embrace all of them. In this thesis the term embedded software is used for software that is characterized by any of the perspectives mentioned in this section.

2.3. Software Metrics

Software metrics are the name given to a wide variety of measurable attributes that can be observed in the development of software systems. They can be used for gaining a better understanding of the process, product, or the project being measured [22].

In [3] a comprehensive study of many software metrics has been given. In this research software metrics have been categorized as follows: product metrics, process metrics, and project metrics. Product metrics quantify certain attributes of the software product that is being developed, such as size, complexity, and performance. Process metrics quantify the attributes of the process which produces the software. These attributes range from defect removal effectiveness to the response time of the defect fixing activities. The final class, project metrics, measure the attributes of the software projects in an attempt to understand their characteristics and execution. Examples of these metrics are number of software developers, development cost, and development schedule constraints.

The software metrics that are used in this research belong to the product metrics category. These metrics will be called as static code attributes henceforth. Appendix A presents a complete list of the metrics under consideration, and how they are calculated. Static code metrics can be divided into three categories, McCabe metrics, Halstead metrics, and various quantitative metrics. McCabe metrics are proposed in [23]. McCabe argues that a quantification of complexity can be achieved via counting the paths among the constructs in a module. An example of this is the McCabe's cyclomatic complexity [23]. When calculating cyclomatic complexity, we transform a module into a graph, where vertices are the statements in the module, and edges are the flow of control among these statements [23]. Cyclomatic complexity of a module is the number of linearly independent paths in this graph. Halstead's software science is based on the assumption that a code that is harder to read, is more likely to be fault-prone [24]. In order to quantify readability, these metrics count the total number of operator occurrences, total number of operand occurrences, number of unique operands, and number of unique operators. The final class of metrics counts various attributes of the code, such as lines of code, blank lines of code, source lines of code, and number of calls to other functions in a module.

2.4. Defect Prediction

2.4.1. Motivation

There are various software systems around us today, each has different requirements. However nearly all of them require dependability and reliability, or at least it is expected of them to behave reliably to certain degrees [3]. Although this is the case with requirements, increasing complexity of software systems and tight schedules for development keeps developers from producing defect-free software without emphasis on quality [18]. Studies in the past have shown that isolating and solving problems in software after shipment is prohibitively expensive [18]. Contrary to this some studies show that defects generally tend to be concentrated in certain parts of the software system [18]. Since the actual reliability of a system cannot be determined until the software product is used by actual users of the system [19], it is unavoidable to seek new techniques to remedy this situation. Defect prediction is one of the techniques for assessing parts of a software system before the product is actually deployed. Researchers have demonstrated that defect prediction can

be used to assess the dependability of a system and to make sure that the system will not fail unexpectedly [21]. Therefore we may state that in the competitive and demanding environment of software development, defect prediction can be used to increase the agility of developers' corrective actions for removing defects from software.

There are other pressures for using defect prediction techniques in the industry. Capability Maturity Model is a widely accepted certification process for software development companies [36]. This model requires effective measurement of processes in its higher levels. The results of the measurements are then used for software process improvement, validating previous actions, and decision making. [22]. This enables optimization of software development processes [36]. Defect prediction can be used to assist software managers in this process by interpreting the results of the measurements [14]. The aim in defect prediction is to find a relationship with the results of the measurements and the defect content of the software system [20].

2.4.2. Defect Prediction Models

Many defect prediction models have been proposed in academic research, and in the industry [9], [12], [14], [37]. One characteristic of these models is that they generally use software metrics data to predict defect content of parts of software projects. It is necessary to understand the general approach in these models before discussing them. Software is often viewed as a collection of modules [9]. Depending on the paradigm of development, these modules can be classes [37], or functions [14]. Software metrics are quantifications of various attributes of these modules, such as size and complexity [3]. Generally the aim in defect prediction is to find a relationship between the measured attributes of the modules, and their defect content [9], [14], [37].

Defect prediction models generally use statistical or machine learning techniques [9], [12], [37]. These techniques have been used for pattern recognition, empirical modeling of relationships, and signal analysis [25], [40]. They enable us to make use of example data or past experience [40]. Defect prediction is one such problem, where we use past experience for taking actions based on predictions. Advances in statistical analysis and artificial intelligence provided researchers many tools for solving this complex problem

[25], [40]. It is possible to propose various defect prediction models using these techniques. However there are few studies that specifically target embedded software systems [9], [10], [20]. This might be partly due to the assumption that the proposed models will generalize and may also be used for embedded software. However one of the major problems with defect prediction is that none of the models have been consistently accurate [21]. Some of these models are given in the remainder of this section along with a discussion of their use in embedded software development.

2.4.2.1. Regression Based Defect Prediction Models.

Regression has been widely used for defect prediction [37], [38], [39]. In regression the aim is to find a model for the unknown variable [40]. In linear regression the model for an observed target value y_i is as follows [41],

$$E(y_i) = \beta_0 + \beta_1 x_{1i} + ... + \beta_p x_{pi} \tag{2.1}$$

In logistic regression a linear model for the log-odds is hypothesized [41],

$$\ln\left(\frac{p_i}{1-p_i}\right) = \beta_0 + \beta_1 x_{1i} + ... + \beta_p x_{pi} \tag{2.2}$$

Logistic regression can be used for classification by assigning cut-off values to *p* [41]. In [37] authors use multivariate logistic regression for defect prediction. In this case the model is as follows [37],

$$p(x_1, x_2, ..., x_n) = \frac{e^{(\beta_0 + \beta_1 x_{1i} + ... + \beta_n x_{ni}) y_i}}{1 + e^{(\beta_0 + \beta_1 x_{1i} + ... + \beta_n x_{ni})}} \tag{2.3}$$

In [37] Basili, Briand, and Melo propose a defect prediction model for object-oriented systems. The goal of the study is to analyze the relationship between object-oriented metrics and the occurrence of defects in parts of a software system [37]. The hypothesis in the study is as follows:

"Assuming testing was performed properly and thoroughly, the probability of fault detection in a class during acceptance testing should be a good indicator of its probability of containing a fault, and, therefore, a relevant measure of fault-proneness." [37].

In order to validate this hypothesis Basili, Briand, and Melo use data coming from eight medium-sized information systems [37]. The data contain measurements 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 (Appendix B presents descriptions of these metrics). The data also contain error counts of each class [37]. Basili, Briand, and Melo then use logistic regression to validate their hypothesis. The x_i values in the model are the metrics that were measured during the development process, and p denotes the probability that a defect was found in a module [37].

In [38] and [39] authors propose using similar models utilizing logistic regression. However the proposed models are not appropriate for our purposes. First of all use of object-oriented metrics limits the applicability of these models to embedded systems, because not all embedded systems are developed using the object-oriented paradigm. Therefore we need a different set of metrics for our purposes. On top of that, some other studies have shown that logistic regression is not always enough for describing the relationship between the metric values and fault-proneness [20].

2.4.2.2. Defect Prediction Models Based on the Bayesian Methods.

Machine learning techniques based on the Bayesian Theorem have been used to built successful defect prediction models [9], [21]. The Bayesian Theorem states that [42],

$$P(C \mid A) = \frac{P(A \mid C)P(C)}{P(A)} \tag{2.4}$$

This theorem enables us to compute the conditional probability of C given A [42]. In [9] and [57] authors use Bayesian Belief Networks for defect prediction, and in [21] authors use the naïve Bayes classifier for defect prediction.

A Bayesian Belief Network is a directed acyclic graph [42]. Random variables are used as nodes in this graph, and nodes are connected with directed links. The graph is annotated with conditional probability distributions that lists the effects of incoming links to a node [42]. Amasaki et al. use Bayesian Belief Networks for predicting the number of residual faults in an embedded software system after the testing phase [9]. In order to do this they capture the abstract development process in a company working in the embedded systems market. This process is given in Figure 2.1 [9].

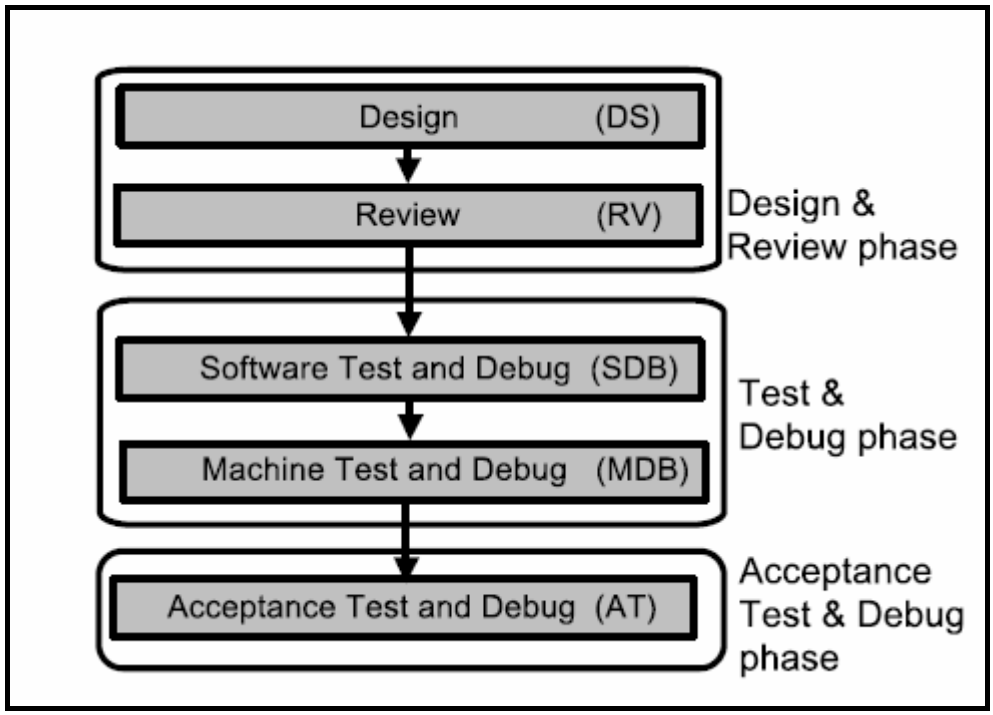


Figure 2.1. An abstract development process for embedded systems [9]

Amasaki et al. propose measuring the following metrics at each phase of the development process: product size (S), effort (E), detected faults (DF), test items (TI), residual faults (RF) [9]. Authors construct a Bayesian Belief Network using the relationships among these metrics. This network is given in Figure 2.2.

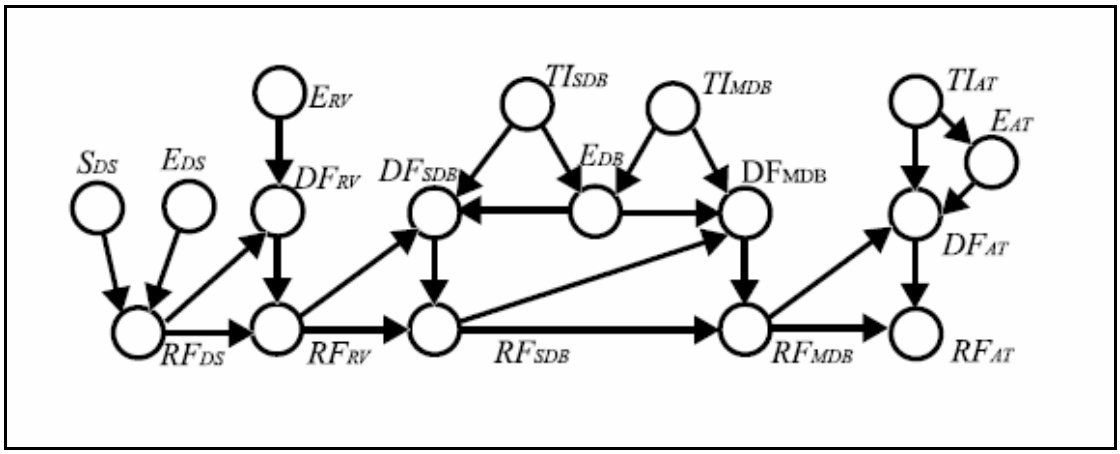


Figure 2.2. A Bayesian Belief Network for defect prediction in embedded systems [9]

Amasaki et al. hypothesize that the number of residual faults is a factor of software quality, therefore it is possible to evaluate the quality of a system by predicting the number of residual faults [9].

Challagulla et al. propose using naive Bayes classifier for defect prediction [21]. Naive Bayes is based on the following model:

$$P(C \mid A) = \frac{P(C)}{P(A)} \prod_k P(A_k \mid C) \tag{2.5}$$

This model lets us calculate a posteriori probability P(C|A), given evidence A_i and a priori probability for a class P(C) [42]. It is called naive because it ignores the possible dependencies between the A_i [25]. Challagulla et al. use software metrics' values as the evidence (A_i) for defects, and being defective or not as the class (C). After that they obtain a probability of being defective for modules in a software system [21]. The data for this study are taken from the NASA MDP repository, which contains software projects with varying requirements [15].

Similarly Fenton et al. use a Bayesian Network to predict the number of residual defects in an embedded software after the development [57]. In order to do this they first determine the factors that affect the number of residual defects in software. They categorize these factors as specification and documentation process, new functionality, design and development process, testing and rework, and project management. Fenton et

al. use a questionnaire for acquiring data on these factors and the participants rate these factors in their development processes. Besides these factors, they also obtain quantitative data from the projects in his research. These data include size, effort, programming language, and number of defects. Finally, Fenton et al. model the relationship between these data and the number of residual defects as a Bayesian Network.

One disadvantage of the model proposed by Amasaki et al. is that it has to be tailored for different software development processes in different companies. Authors claim that one of the advantages of the Bayesian Belief Networks is the ability to handle missing data, and therefore it can be used before the testing phase [9]. However they fail to provide any evidence for this, instead they only examine the results after the testing phases [9]. The model proposed by Challagulla et al. have also been used by others to build useful defect predictors [14]. However these models are not tailored for embedded software systems. Fenton et al. aim to find the amount of residual defects in embedded software. However he uses certain amount of qualitative data, such as the answer to the question "In your opinion, how effective was your review procedure?" [57]. The use of such qualitative data rises questions about the consistency of the answers since they measure opinions not the actual procedures. On top of that their aim is to predict the number of residual defects and the overall quality, they do not propose a method for assisting developers in testing phases during the development process [57].

2.4.2.3. Defect Prediction Models Based on Artificial Neural Networks.

When using neural networks as classifiers our aim is to construct a network of neurons, that takes as input some attributes, and gives an output which we can interpret as being in some class. The neurons in this network are organized into layers, the layer connected to the inputs is called the input layer, the layer from which the output is obtained is called the output layer, and in between these two layers are the hidden layers [25]. Figure 2.3 contains a simplified representation of this structure. Although it is omitted in Figure 2.3 for simplicity, every neuron in a layer is connected to all of the neurons in the previous layer.

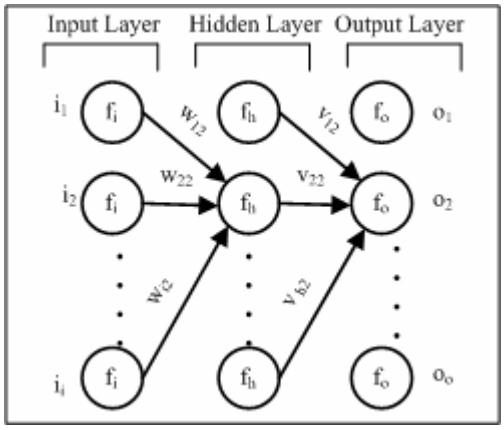


Figure 2.3. A neural network [25]

The input of a given neuron is the weighted sum, using the weights on the connections, of its inputs, and the output is a function, called the transfer function, of its input. The knowledge is represented using the weights in the network. Naturally the training process is an adjustment of these weights by trying to minimize a predefined error function. Artificial neural networks can be trained for learning complex and nonlinear patterns in data [42], that is why they have been used for building models for defect prediction [12].

Khoshgoftaar and Szabo use neural networks for predicting defects in systems designed for providing interfaces in operating kernels and their hardware [12]. The data contain metrics values for ten different software metrics and errors found in software modules. Khoshgoftaar and Szabo use principle components analysis (PCA) before feeding this data to the neural network. In principal components analysis the aim is to reduce the dimensionality of the data [25]. In order to do this the data are centered, and then the axes are rotated to line up with the directions with highest variance [25]. There are several reasons for using PCA before feeding the data to an artificial neural network [25]:

- If the size of the data is smaller (which is the case after PCA) then the memory and computation cost during training is reduced,
- Simpler models give more robust results on small datasets, which means their results vary less depending on noise and outliers,
- If the data are explained with fewer features knowledge extraction is easier.

Khoshgoftaar and Szabo train a neural network using metric values as input, and obtain classification as defective and defect-free as the output [12].

There are other studies that demonstrate the usefulness of this model [20], [43]. One disadvantage of this model is that all of these studies use different data, and only one of them comes from an embedded software system [20]. Because of this it must be studied further before proposing it as a model for embedded software. Another disadvantage of this model is the fact that the training algorithms for neural networks has a higher time complexity compared to other techniques, such as naive Bayes [25].

2.4.2.4. State of the Art.

We can consider the study presented in [14] as the state of the art in defect prediction. There are several reasons for this. First of all authors use a publicly available dataset for testing the model proposed in the study. The dataset is taken from the NASA MDP Repository, which contains 13 projects at the time this thesis was written [15]. Using a publicly available dataset provides other researchers with the opportunity to reproduce and compare the results of their studies. Another important contribution of this study is that it answers an important research question in defect prediction. One of the most important questions in defect prediction research was how to select an appropriate subset of available metrics [14], [37], [46]. This study has shown that the choice of the machine learning method is more important than choosing a subset of the available data [14]. The results of this study are far better than other similar models proposed in the literature [20], [44]. However it is important to note that not all of the studies in the literature give us results in the same form, therefore this study is state of the art among the studies with comparable results.

Menzies et al. propose building a defect predictor using the naive Bayes classifier. They use eight projects from the NASA MDP Repository [14]. As a first step the data are log filtered, that is all of the numeric values less than 10^{-6} are replaced with their natural logarithms. Menzies et al. claim that log-filtering spreads the data more evenly across the y-range, and it makes it easier to reason about them. After that Menzies et al. use exhaustive and iterative InfoGain subsetting for determining an appropriate subset of

metrics for each project. InfoGain subsetting is based on information theory, and it ranks attributes in a set according to the amount information gain they provide [14]. In order to rank the attributes the following model is used [14]:

$$InfoGain(A_i) = H(C) - H(C \mid A_i) \tag{2.7}$$

Here *H(C)* is the minimum number of bits required to encode a class distribution, and it is calculated using the following formula [14]:

$$H(C) = -\sum_{c \in C} p(c) \log_2 p(c) \tag{2.8}$$

p(c) is the probability of observing a class in the above formula. *H(C|A)* is computed using the following formula [14]:

$$H(C \mid A) = -\sum_{a \in A} p(a) \sum_{c \in C} p(c \mid a) \log_2 (p(c \mid a)) \tag{2.6}$$

In the model Menzies et al. use, the classes ($c \in C$) are defect-free and defective, and attributes ($a_i \in A$) are the metrics available in the data. Every $a_i \in A$ are ranked according to their *InfoGain* values. In InfoGain subsetting all subsets of the top *j* attributes are selected or the subset size is increased incrementally [14]. After this step they build naive Bayes classifiers using all of these subsets. The results of the best predictors in this research show that their usefulness can be compared to that of manual code reviews [14].

The model proposed by Menzies et al. is especially important for our purposes. This is because four of the projects used in this study come from embedded software systems. Therefore we can use their results in order to benchmark our results. However there are certain drawbacks of this model, first of all InfoGain subsetting is a fully supervised method [14]. This is because it requires a count of defective and defect-free modules in a project. On top of that this model uses a different subset of the available metrics for each of the four embedded software projects that were used [14]. Therefore it is not possible to propose a defect prediction model for embedded software based only on this study.

2.5. Related Machine Learning Techniques

We discuss two machine learning techniques that have not been used in defect prediction research before. These techniques play crucial roles in our research, and the reasons for this are given below.

2.5.1. Voting Feature Intervals (VFI)

The VFI classifier was proposed in [47] as an alternative to naive Bayes classifier. The primary advantage of this classifier is that the time complexity of its training algorithm is lower than the training algorithm of the naive Bayes classifier [47]. The input to this classifier is as follows:

$$I = \begin{bmatrix} f_{11} & f_{21} & . & . & . & f_{1m} & c_1 \\ f_{21} & f_{22} & . & . & . & f_{2m} & c_2 \\ . & . & . & & & . & . \\ . & . & & . & & . & . \\ . & . & & & . & . & . \\ f_{n1} & f_{n2} & . & . & . & f_{nm} & c_n \end{bmatrix} \tag{2.7}$$

where columns from *1* to *m* are the features, and column *m+1* is the actual classes of the samples.

The training algorithm for VFI uses a dimension for each feature, and places the minimum and maximum values for that feature in each sample on this dimension. The difference between the values make up the intervals on each dimension. After that it counts the number of samples of each class between each value. The number of samples of each class in an interval makes up the votes of that interval. In order to make classification it takes as input a vector *V*,

$$V = \begin{bmatrix} f_1 & f_2 & . & . & . & f_m \end{bmatrix} \tag{2.8}$$

The classification process places the elements of *V* on the corresponding intervals on each dimension. After that the votes of the intervals are summed and the class for *V* is determined by selecting the class with the highest vote [47]. A demonstration of the classification process is given in Figure 2.4.

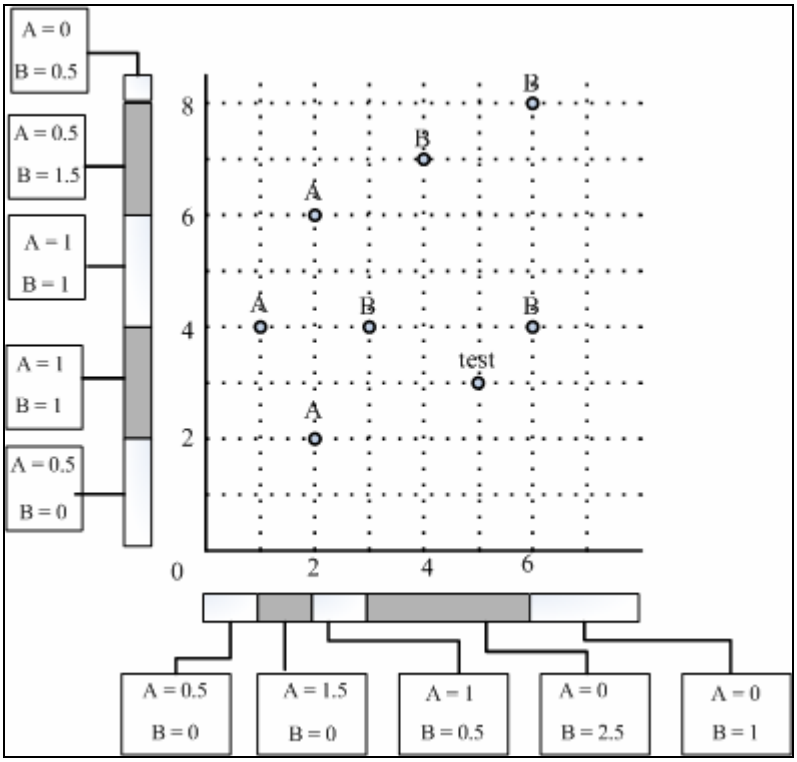


Figure 2.4. A demonstration of the VFI classifier. The test sample is classified as belonging to class B

This classifier has never been used in defect prediction to the best of our knowledge. However, it has a trait that makes it useful for defect prediction in embedded software systems. VFI works with minimum and maximum values for each feature [47], because of this it may be used without any preprocessing of the data. This means that none of the information contained in the data is lost. This is important because companies developing embedded software generally produce multiple software systems with different characteristics [46]. Therefore we should expect that data coming from different projects may vary significantly, and information that is not important in one project might prove to be useful in another. However existing preprocessing approaches such as feature reduction eliminate the outliers from the data [25]. Therefore outliers are not included in the models. The problem here is that outliers in one project that is used for building the model may

contain valuable information for new projects, for which we need to make defect prediction. A classifier based on VFI does not ignore these outliers, because it can be used efficiently without preprocessing the input. Therefore VFI may be used with minimal data loss across projects.

2.5.2. Ensemble of Learners

In an ensemble of learners the aim is to combine the strengths of multiple machine learning techniques [25]. In order to do this a linear combination of the results of multiple learners are used to determine the final result [25]. In many cases there is not any learning algorithm that always induces the most accurate learner in any domain, therefore combining multiple learners can yield higher accuracy [25]. Our aim in this research is to build models with a *probability of detection* as high as possible. Therefore an ensemble can be tested as a strategy for building a defect prediction model for embedded software systems.

3. PROBLEM STATEMENT

We have mentioned earlier that our motivation in this research is to assist software developers in allocating their resources during the defect removal phases of their development process. There are many defect removal strategies requiring a certain amount of the available resources. Manual code reviews, testing, and formal verification are some examples of these strategies [7]. One common characteristic of these techniques is that they all require large amounts of resources and time. In developing software, project managers have to choose where to spend their valuable resources wisely. Defect prediction helps managers focus their defect removal efforts to the defect prone parts of the system. On top of that in processes where developers work in small iterations planning of the future iterations becomes important. Defect predictors are useful in such situations because they give clues about the defect contents of the parts of the software.

The problem can be stated as a resource utilization or optimization problem in view of the above statements. Defect predictors help managers to solve this problem by guiding them in allocating their scarce resources during the defect removal stages of the development process.

We see the problem of building a defect predictor as obtaining a binary classifier which classifies modules either as defective or defect-free. This is because a model capable of doing that can be used for assessing individual parts of software in terms of reliability and dependability [21]. On top of that it can be used for decision making, and resource allocation in different phases of the development process [10], [14].

The data that can be used to build useful defect predictors varies depending on the development phase that the predictor is going to be used. In order to obtain useful predictors it is vital to use data that carry as much information as possible pertaining to the end product. We plan to achieve this by using static code attributes, since they are the last deliverable of the software development process, before the actual product [18].

In identifying parts of a source code as defective or defect-free, the first step is to define the parts under consideration. We take these parts to be the modules (individual functional units) of the source code. The technique used for the classification process should take as input the source code attributes and it should be able to classify a module as defective or defect-free. There are many approaches for doing this. The idea common to many of them is setting the available software metrics as our independent variables, and the class to which a particular module belongs to as the dependent variable [12], [13], [14], [20]. After that it is a matter of finding a pattern in the independent variables so that this pattern can be recognized as a dependent variable. This problem is exemplified in Figure 3.1 with two independent variables.

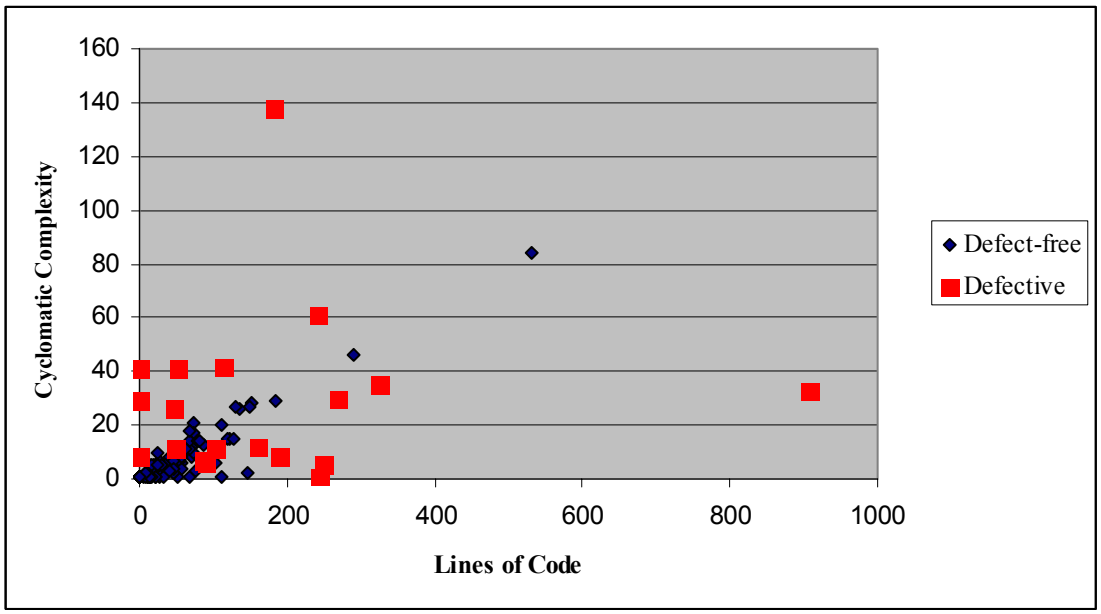


Figure 3.1. Spread of defective and defect-free modules with two source code attributes on the axes, data taken from [15]

It has been shown in the literature that conventional statistical models are successful to a certain degree [20]. Machine learning techniques have been used to build the most useful predictors [14], [20]. However there is still room for improvement, and improvement is necessary to use defect prediction on embedded software.

Achieving high reliability and dependability is very important in embedded software development. This is because embedded software is expected to have timeliness, liveness,

and reactivity as its characteristics [4]. These characteristics require checking parts of the software thoroughly in order to achieve the desired dependability and reliability. Defect predictors can be utilized in embedded software development to check whether the software under consideration meets the quality standards that are previously set inside the organization. However, we believe that it is necessary to perform a performance evaluation before proposing a defect predictor for embedded systems. In [14] an appropriate measure for evaluating defect predictor performance is given. This performance measure utilizes a table called the confusion matrix [14]. A sample confusion matrix is given in Table 3.1.

Table 3.1. A confusion matrix

		actual	
		defective	defect-free
predicted	defective	tp	fp
	defect-free	fn	tn
tp: True positive, defective module correctly classified.			
fp: False positive, defect-free module classified as defective.			
fn: False negative, defective module classified as defect-free.			
tn: True negative, defect-free module classified correctly.			
pd = tp/(tp+fn)			
pf = fp/(fp+tn)			

Basili claims that a useful defect predictor should maximize the probability of detection (*pd*), and minimize probability of false alarm (*pf*) [14]. This is because an ideal predictor should have *pd = 1* and *pf = 0*. Therefore if we maximize *pd* and minimize *pf* we would be getting closer to the ideal predictor [14]. It is possible to measure this as follows [14]:

$$balance = 1 - \frac{\sqrt{(0 - pf)^2 + (1 - pd)^2}}{\sqrt{2}} \tag{3.1}$$

The *balance* value of a predictor measures how close it is to the ideal predictor [14]. It is important to note one limitation of this approach, the *balance* for different predictors could be the same. Table 3.2 gives three different sample predictors with the same *balance* value. We believe that a predictor should have a *pd* value as high as possible in order to be used confidently in embedded software systems. This due to the fact that embedded

software systems are expected be highly-reliable and highly-dependable [4], [5]. Therefore a defect predictor designed for embedded software should have a high *balance* value and a high *pd* value at the same time. This will make sure that developers are warned about the existence of maximal number of defective modules.

Table 3.2. Example pd and pf values which yield the same balance value

	pd	*pf*	*balance*
Predictor 1	0.48	0.17	0.61
Predictor 2	0.55	0.33	0.61
Predictor 3	0.73	0.48	0.61

There are also other factors that should be kept in mind when building defect predictors for embedded systems. Various technologies, programming languages, and development methodologies are used in embedded software development [46]. Because of this a defect prediction model for embedded software should not make any assumptions on the programming paradigm, programming language, or the software process that is being used. The data used in prediction must be easily collected in different development processes. On top of this many embedded software development companies produce various embedded systems [46]. This means that different approaches or technologies may be used within a single company [46].

Therefore the problem is to build a defect prediction model that can be used with different technologies and development processes. Besides that the number of defective modules that are predicted accurately must be maximized, while keeping the number of false alarms at a minimum.

4. PROPOSED MODEL

Our aim is to make defect prediction for embedded software systems. In order to do this we utilize past experience. This past experience comprises metrics data gathered from old projects along with their defect content. The defect content should specify which modules of the system contained defects. The past experience will be used to build a model for predicting the defect content of new systems before the testing phase. The next section gives the details of the inputs to the model. Then the outputs of the model is discussed, and finally the details of the model are given in the last section.

4.1. Inputs to the Model

We aim to make defect prediction prior to the testing phase. In order to do this we use static code attributes as our input. A full list of these attributes are presented in Appendix A. We see a new software system S_n as a collection of modules M_i, where M_i is a vector of static code attributes,

$$M_i = \begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix} \tag{4.1}$$

Then an embedded software system S_n , where M_i are its modules, is a matrix:

$$S_n = \begin{bmatrix} a_{11} & a_{12} & . & . & . & a_{1n} \\ a_{21} & a_{22} & . & . & . & a_{2n} \\ . & . & . & & & . \\ . & . & & . & & . \\ . & . & & & . & . \\ a_{1n} & a_{2n} & . & . & . & a_{mn} \end{bmatrix} \tag{4.2}$$

where m is the number of modules, and n is the number of attributes.

The proposed model will use past data in order to learn the relationship between the static code attributes and defect-proneness. Therefore the other input to the model is

embedded software systems that were developed in a company before the new system. This data should also contain defect information for the modules. This defect information should indicate whether defects were found in those modules after the testing phases or production. This data may contain multiple systems, therefore it is a set of old embedded software projects, *P,*

$$P = \{S_1, S_2, \ldots, S_{n-1}\} \tag{4.3}$$

where S_i is as follows for $i=1,n-1$,

$$S_i = \begin{bmatrix} a_{11} & a_{12} & . & . & . & a_{1n} & e_1 \\ a_{21} & a_{22} & . & . & . & a_{2n} & e_2 \\ . & . & . & & & . & . \\ . & . & & . & & . & . \\ . & . & & & . & . & . \\ a_{1n} & a_{2n} & . & . & . & a_{mn} & e_m \end{bmatrix} \tag{4.4}$$

where e_i is 1 for defective modules, and 0 for defect-free modules.

Therefore the inputs to the model are an *mxn* matrix which is an abstraction of a new embedded software system *S*, and *P* a set of past data containing multiple *mx(n+1)* matrices.

4.2. Outputs of the Model

Our aim is to make binary classification. Therefore the outputs of the model are two classes defective and defect-free. Formally:

$$C = \{c_1,c_2\}, \text{ where } c_1 = \textit{defective} \text{ and } c_2 = \textit{defect-free} \tag{4.5}$$

4.3. The Details of the Model

The model should be able take as input an embedded software system S_i, and give as output a $c_i \in C$. If S is the set of embedded software systems, our proposed model is a function whose domain is $S_i \in S$, and whose range is $c_i \in C$.

We propose using an ensemble of classifiers for defect prediction in embedded software systems. The ensemble will be formed by an artificial neural network classifier, a naive Bayes classifier, and a Voting Feature Intervals classifier. In order to obtain better accuracy these classifiers will be supported by a feature reduction phase. Our hypothesis is that we can achieve the required performance for predicting defects in embedded software by using this model. This is because an ensemble can combine the strengths of multiple predictors. By combining machine learning methods that have been proven to be useful in defect prediction and other machine learning techniques that can help the model generalize, we claim that it is possible to build defect predictors for embedded software systems.

The necessary steps for building the model is given in pseudocode in Figure 4.1. An overview of the proposed model is given in Figure 4.2.

```
P is the past data with error information.
S is the data from a new project without error information .
   Build ANN classifier.
      Perform PCA on P.
      Train the ANN clasifier.
   Build naive Bayes classifier.
      log filter P.
      Use InfoGain subsetting to obtain the best subset.
      Train the NB classifier.
   Build the VFI classifier.
      Train the VFI with P.
   On S.
      Run the ANN classifier.
      Run the NB classifier.
      Run the VFI classifier.
      Combine outputs via voting.
   Return the output of the ensemble as the prediction result.
```

Figure 4.1. Pseudocode for building the proposed model

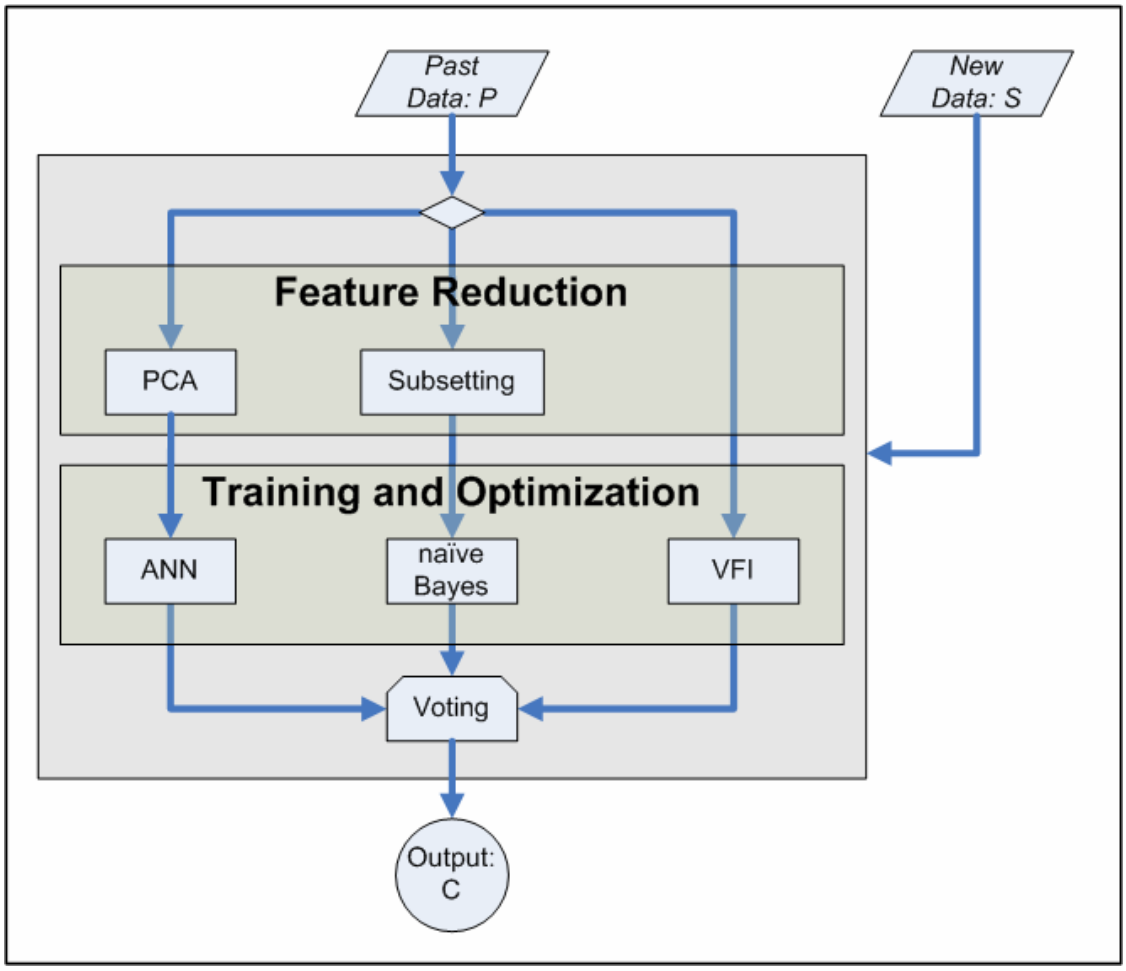


Figure 4.2. An overview of the proposed model

Defect prediction in embedded software systems requires learning complex relationships between static code attributes and the defect content of modules. Artificial neural networks are able to capture the patterns in those relationships. Besides that a defect prediction model for embedded software systems should be able to distinguish static code attributes that are more important for each project and for embedded software. Naive Bayes with subsetting is able to capture the amount of information contained in static code attributes, because it is able to rank the static code attributes according to their information content [14]. Besides those companies that are developing embedded software generally develop various software systems with different characteristics. Therefore valuable information across projects should not be lost. Voting Feature Intervals based classifier is able to do that by keeping the information related to outliers in the samples [47]. These are the reasons why we propose an ensemble for combining the strengths of multiple classifiers for defect prediction model in embedded software systems.

5. DATA COLLECTION

We have proposed a defect prediction model for embedded software systems. Data collected from embedded software projects is necessary to ascertain the performance of the proposed model. It is also necessary to determine the confidence levels for the utilization of the model in practice.

The first section is devoted to the specification of the requirements for the data. The format of the data and its characteristics will be discussed in detail in that section. Second section is reserved for the data extraction methodology. Finally the last section presents an overview of the embedded software projects used in this research, along with the characteristics of the data extracted from them.

5.1. Requirements for the Data

The proposed model tries to classify modules in an embedded software system as defect-prone or defect-free. In order to do so it utilizes past experience. The past experience refers to the defect information from old software projects. Therefore the projects that are going to be used in this research should include defect information for the modules of those projects.

Our aim is to find a relationship between the characteristics of embedded software modules and their defect-proneness. We have discussed that certain characteristics of software modules can be captured by using static code attributes. We propose to identify defect-prone modules of an embedded software system using static code attributes of these modules. This means that the data should also contain static code attributes for each module.

It is also necessary to obtain clues about the applicability of the model to different types of embedded software systems. In order to do this the performance of the model should be assessed on data coming from different embedded software projects [45]. Therefore the data that are going to be used to support this research should also include

multiple embedded software projects. Besides that static code attributes do not measure characteristics of the software project or the development process. These attributes quantify properties of the source code such as readability and complexity [3]. It is not possible to claim that the proposed model can be used in different development processes and across projects with varying requirements unless the model is tested on different datasets. Therefore the data should include different types of embedded software projects.

We employ machine learning techniques for defect prediction in embedded software systems. Recent studies show that the amount of information about the software modules should be as much as possible in order to achieve acceptable performance [14]. Therefore the data should also comprise as many static code attributes as possible in order to obtain a successful model.

The last but perhaps the most important attribute of the data is reliability. Any potential inconsistencies in the data would rise questions about the validity of the research. Therefore the data should be obtained from reliable sources, and it should not contain any discrepancies.

5.2. Data Extraction

The data used in this research comes from two different sources. One of them is the NASA MDP Data Repository [15]. The other source is a local company operating in the embedded systems market. The details of the procedure for extracting data from these sources are discussed in the remainder of this section.

5.2.1. Data extraction from the NASA MDP Data Repository

Data from 13 projects were available in the MDP Repository at the time this thesis was written. The code names and short descriptions for these projects are given in Table 5.1. The decision for using the data from this source was based on several factors. First of all the amount of information available on the projects (i.e. number of static code attributes) is enough for our research. Besides that this dataset have been used in other studies extensively and it is considered to be a reliable dataset [14], [21], [50]. Finally, the

fact that they have been used in other studies for defect prediction gives us the opportunity to compare our results with those studies.

Table 5.1. An overview of projects in the NASA MDP Data Repository [15]

Code Name	Description
CM1	A science instrument.
JM1	A real-time C project.
KC1	A computer software configuration item within a large ground project.
KC3	A system for collection, processing and delivery of satellite metadata.
KC4	A ground-based subscription server.
MC1	A combustion experiment that is designed to fly on the space shuttle.
MC2	A software designed to run a video guidance system.
MW1	A from a zero gravity experiment related to combustion.
PC1	A flight software from an earth orbiting satellite.
PC2	A dynamic simulator for attitude control systems.
PC3	A flight software from an earth orbiting satellite.
PC4	A flight software from an earth orbiting satellite.
PC5	A cockpit upgrade that has a series of safety enhancements.

The first step is to determine which of the available projects are appropriate for our purposes, such that we can decide on which ones can be considered as embedded software systems. In order to do this we examined the requirements for these projects.

The requirements traceability data for CM1 was available in [49]. An excerpt from these set of requirements is presented in Appendix C. A close examination of the requirements of this project led us to decide that this system can be considered as an embedded software system. This decision is based on several factors. The characteristics of embedded systems such as timeliness, liveness, heterogeneity, and reactivity are required in the operation of this software system. On top of that the requirements indicate a high degree of hardware-software coupling. Since these are the typical characteristics of embedded software systems we have taken CM1 project to conduct our experiments.

The rest of the projects selected from this data repository are the flight software systems with code names PC1, PC3, and PC4. Flight software systems are embedded software distributed over multiple different flight computers [26]. These are highly interactive systems and are required to monitor various hardware and the environment [26]. Because of this we decided that these projects are appropriate for our research as

well. The remainder of the projects do not fit for our purposes since there were not much information on their requirements descriptions in MDP Data Repository [15].

As we mentioned earlier our metric data source is MDP Data Repository. This dataset comprises static code attributes of the modules on each project and the number of defects found in these modules during development. We have seen that some of the static code attributes have zero variance. This means that they carry no valuable information for our purposes [25]. Therefore these static code attributes were removed from the dataset before we used the data.

5.2.2. Data Extraction from Local Embedded Software Projects

We have discussed that it is necessary to evaluate the performance of the model on different projects. In order to do this we have cooperated with a local company operating in the embedded software systems market. We have obtained the source code for three embedded software projects and their defect data. These software projects were developed in white goods industry. This means that their functional requirements are very different from the projects taken from the NASA MDP Data Repository [15]. The diversity they provide to our dataset is beneficial because it will help us in evaluating how well our model can generalize across different projects.

One approach for extracting the static code attributes from the source codes of these projects would be using off-the-shelf tools. However this approach has disadvantages due several reasons. First of all in order to increase the success of our research we need the same set of metrics for all the projects in our dataset. Therefore the tool used for this purpose should be able extract same set of attributes found in the NASA MDP Data Repository. On top of that there should be no discrepancies between the definitions of the attributes extracted from the data. For the validity of our results we had to make sure that the static code attributes were computed in the same way they were computed in NASA MDP Data Repository. Therefore we have developed our own tool to extract those attributes from the code.

A complete list of the static code attributes that should be extracted are given in Appendix A. Some of these attributes are derived from other ones. The rest of the attributes are directly extracted from the source code. It is necessary to process the source code in order to extract these attributes. This can be done by using techniques from compiler design [51].

An overview of a compiler is given in Figure 5.1 [51]. Extracting the base metrics given in Appendix A requires that we should analyze the lexical and the syntactic structure of the code. First of all since we need metrics data for each module, we need to be able to identify the function definitions in the source code. Furthermore in order to calculate metrics such as number of unique operators, number of unique operands, and cyclomatic complexity, it is necessary to identify the keywords, identifiers, and calls to other modules from a module.

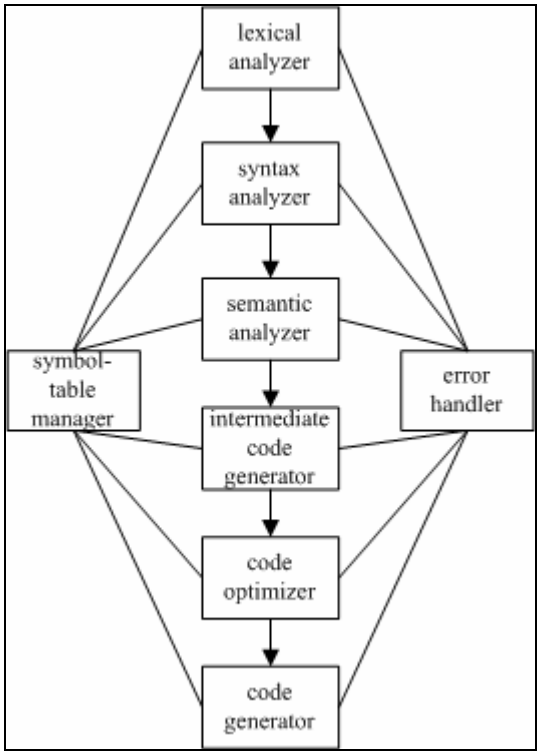


Figure 5.1. An overview of a compiler [51]

The first step in a compiler is the lexical analysis step [51]. The minimum meaningful units, which are called tokens, are identified in this step. After that syntax

analysis is performed. In syntax analysis the syntactic structure of the source code is obtained as a parse tree. This stage enables us to group the tokens into meaningful structures [51]. Appendix A lists what is required for calculating each metric. This information shows us that the first two stages of a compiler, lexical analysis and syntactic analysis are enough for our purposes [51].

Although it is only necessary to implement the first two stages of a compiler, doing this still requires a lot of effort. However we are able to make simple assumptions. First of all we do not need to obtain the parse tree of the code, we only the need the static code attributes which can be computed while scanning the input. Besides that an important amount of effort in designing compilers goes to developing error handlers [51]. In order to avoid this we assume that the source code has been compiled without errors prior to extraction. It is safe to make this assumption because the source codes under consideration have been compiled before, and actually is or was in production for sale. An overview of the tool we have developed is given in Figure 5.2.

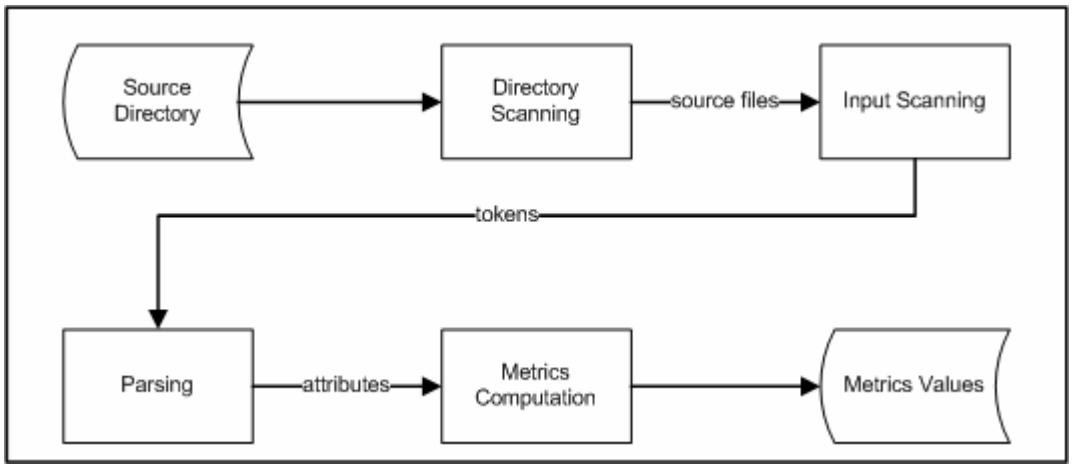


Figure 5.2. Overview of the software for metrics extraction

In the directory scanning phase the tool scans the directory containing the source files. The projects that we used are implemented using the C Programming Language. Therefore the source files that we are concerned with are the header files and the definition files [52]. The header files have a *.h* extension and the definition files have a *.c* extension. The directory is recursively scanned for files with *.h* or *.c* extensions and these files are given to the input scanning phase as input.

The input scanning phase contains an implementation of a lexical analyzer for the C programming language. This is basically an implementation of a transition diagram for the C programming language. Transition diagrams are used to depict the actions taken by a lexical analyzer [51]. The transition diagram implemented in this phase is presented in Appendix D. This phase processes the contents of each source file character by character. The outputs of this phase are the tokens. These tokens are handed to the parsing phase one by one upon request. After that the parsing phase constructs the syntactic structure of the source code using the tokens and counts the attributes in the source code.

Finally the metrics computation phase receives the source code attributes for each module. The metrics that are calculated by composing the base attributes are computed in this step and the results are stored in a file. This file contains the attributes of each module in a comma separated file format.

5.3. Overview of the Projects

We used seven projects in this research. Three of the projects were obtained from a local company and four of them were taken from the NASA MDP Data Repository [15]. The total lines of code and the number of modules in these projects are listed in Table 5.2. We will refer to the projects taken from the local company as LP1, LP2, and LP3 (an abbreviation for Local Project) for the sake of brevity.

Table 5.2. Projects used in this study

Project	Total LOC	# of Modules	Average LOC per Module
CM1	20000	505	40
PC1	40000	1107	36
PC3	40000	1563	26
PC4	36000	1458	25
LP1	3800	38	100
LP2	8300	65	128
LP3	15400	135	114

The sizes, in terms of lines of code, of the projects vary from 3800 to 40,000. Furthermore the average lines of code in each module range from 25 to 114, and tend to be much smaller in projects taken from NASA. These types of variances are desirable for our

purposes, because we would like to test the performance of our model on projects with varying characteristics.

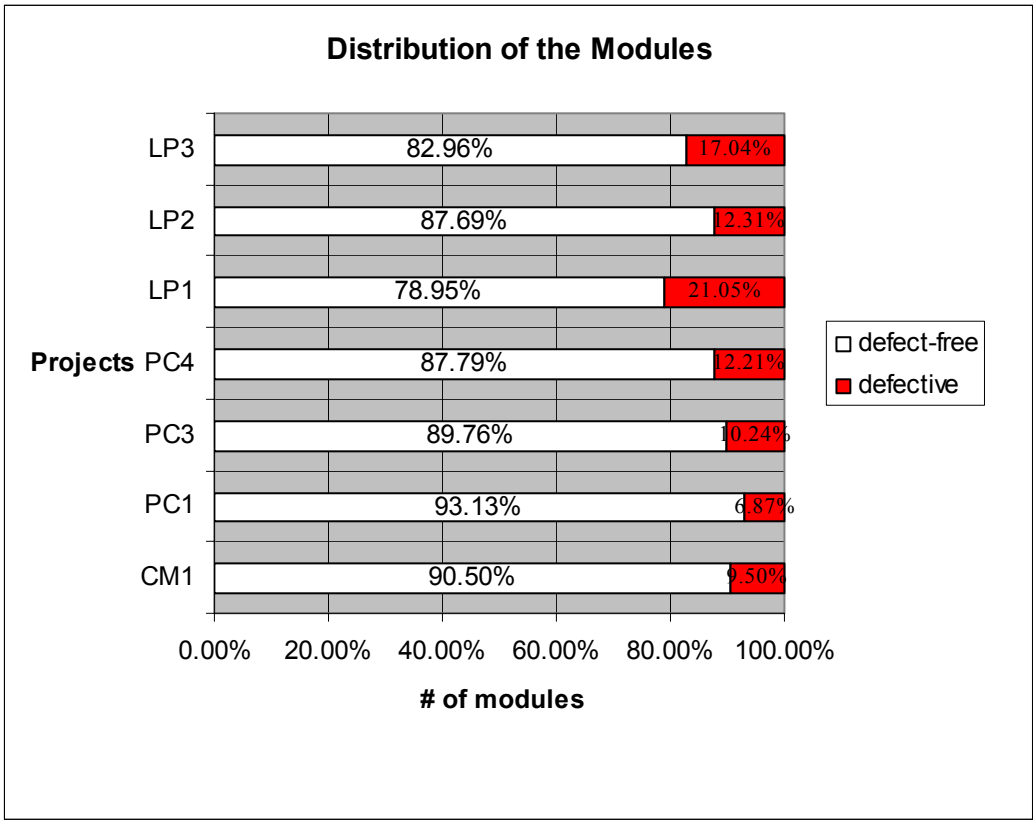


Figure 5.3. Percentage of the defective modules in each project

Figure 5.3 gives the percentage of defective modules in each project. Defects are found in 12.74 per cent of the modules on the average. In view of other studies stating that nearly half of the modules are defective in other software projects, this distribution can be considered unexpected [18]. Our purpose is not to investigate the cause of this discrepancy, however it is important to note that the target class is in minority. This fact should be taken into account while evaluating the performance of the proposed model. Prior studies have shown that it is important to use appropriate measures for assessing the performance of classifiers on datasets like ours [14].

6. EXPERIMENTS AND RESULTS

We present the design and details of the empirical study for validating our hypothesis in this part of the thesis. In the next section we discuss our experimental design. After that we give the details of the experiments we made. Finally this part is concluded with a discussion of the results of our experiments.

6.1. Experimental Design

A careful experimental design is required for measuring the generalization ability of a model [25]. A plausible method proposed in the machine learning literature for this measurement is the *N-way cross-evaluation* technique [53]. In *N-way cross-evaluation* the data are divided to ten equal parts, and then *N-1* of these parts are used for training. This data are called the training set [53]. The remaining part is used for testing and is called the test set [53]. This ensures that the generalization ability of the model is measured as accurately as possible [25]. In *M*N-way cross-evaluation* the *N-way cross-evaluation* is repeated *M* times by randomizing the order of the samples in the training set each time [53]. This is necessary to avoid the effects of ordering on our experiments [54]. Order effect might dramatically increase or decrease the performance of certain algorithms [14], [54]. Randomizing the order of the samples and repeating the experiments multiple times is also suggested for assessing the performance of defect predictors in other research [14]. It is also necessary to preserve the characteristics found in the data in the training and the test sets [25]. This is achieved by preserving the prior class probabilities in the training and test sets. This process is called stratification.

Common value for *M* and *N* in machine learning literature is *10* [14], [25], [53]. Therefore we performed our experiments by using *10*10-way stratified cross-evaluation*. The pseudocode for this scheme is given in Figure 6.1.

```
M <- 10
N <- 10
DATA <- {CM1, PC1, PC3, PC4, LP1, LP2, LP3}

for each element in DATA
   defective <- samples of class defective
   defectfree <- samples of class defectfree
   defectiveParts <- generate N equal parts from defective
   defectfreeParts <- generate N equal parts from defectfree
   for j <- 1 to M
      for i <- 1 to N
         testSet <- defectiveParts[i] + defectfreeParts[i]
         trainingSet <- (defectiveParts + defectfreeParts) - testSet
         randomize the order in the training set
         train the model on the training set
         test the model on the test set
```

Figure 6.1. Pseudocode for the experiment methodology

6.2. Experiments

We employ three different machine learning techniques and a method for combining these techniques. We present our experimental studies with each of these techniques in the rest of this section.

6.2.1. Experiments with ANN

We demonstrate the decisions we made for building a defect predictor using an artificial neural network in this section. The steps of this experiment can be summarized as, the preprocessing of the data to yield optimum performance from the network, and then determining the parameters of the network. All of the experiments with neural networks are performed using MATLAB [56].

We have discussed the reasons for using PCA before feeding the data to the network. Our samples are in an *m x 32* matrix where *m* is the number of modules in a project and *32* is the number of static code attributes. PCA enables us to reduce this matrix to an *m x p* matrix where $p < 32$ [25]. PCA achieves this in a way that this new matrix still carries most of the variation in the original dataset. This ensures that the amount of information loss is minimal. This analysis is performed as follows [25]:

i. Find the covariance matrix , *cov(m)* of our set of *m* variables,

ii. Find the eigenvectors of *cov(m)*,

iii. The total variance is the sum of the eigenvalues, λ, of these eigenvectors,

$$total\ variance = \sum_{j=1}^{m} \lambda_j \tag{6.1}$$

iv. Find how much of the *total variance* is explained by each component using,

$$\frac{\lambda_j}{\sum_{j=1}^{m} \lambda_j} \tag{6.2}$$

There are two common ways for selecting the components to use [55]. One of them is to select components with $\lambda_j > 1$, the other way is to select the components that explain a certain amount of the *total variance* [55]. We adopted the second strategy and selected the components that explain *95* per cent of the *total variance*. This was because we want to propose a model that can be used with a wide variety of embedded software systems. Therefore we did not want to make any assumptions on the size of the eigenvalues.

The following step is the design of the neural network. There are many approaches to do this. One common approach is trying many different architectures, and then choosing the architecture that generalizes the best. It has been shown that when the complexity of the network is too high generalization may not be well.

The space and time complexity of a neural network is as follows [25]:

$$\mathrm{O}(H \cdot (K + d)) \tag{6.3}$$

where H is the number of hidden units, d is the number of inputs, and K is the number of outputs.

In binary classification problems a single output unit suffices [25], therefore $K=1$, in our case. The number of inputs, d is equal to the number of components selected in the

PCA step. The unknown variable is *H*. Since we favor simplicity over complexity we adopted a constructive approach in determining *H*. Two constructive approaches have been given by Alpaydin [25]. One of them is dynamic node creation and the other is cascade correlation. In both methods the initial network has a single hidden layer and one neuron in this layer. Dynamic node creation method adds a neuron to the hidden layer one by one. Cascade correlation adds another hidden layer with a single neuron at each step and every new hidden layer is connected to all of the previous layers. Both approaches continue adding new neurons until an acceptable error is reached. Alpaydin states that an ideal constructive method should be able to decide when to add a new hidden layer or a neuron to an existing layer. This is an open research problem in neural networks [25]. Because of this it is hard to decide which method to use. However a network constructed with dynamic node creation has fewer connections compared to a network of the same size constructed with cascade correlation. The number of connections increases the time complexity of the training algorithms [25]. We prefer simplicity over complexity in constructing our network, therefore we decided to use the dynamic node creation.

When the number of neurons in the hidden layer are less than the number of input units the ANN may perform dimensionality reduction [25]. We perform dimensionality reduction before giving the data to the ANN by using PCA. Therefore we need to start with a hidden layer size that is equal to the number of input units. After that we increase the number of hidden units in the hidden layer one by one using dynamic node creation.

We have discussed the ways of evaluating the performance of defect predictors. We used *balance* proposed in [14] to evaluate the performance of the networks. We have tested different network structures constructed by dynamic node creation on our data. The *balance* values for these networks are given in Figure 6.2.

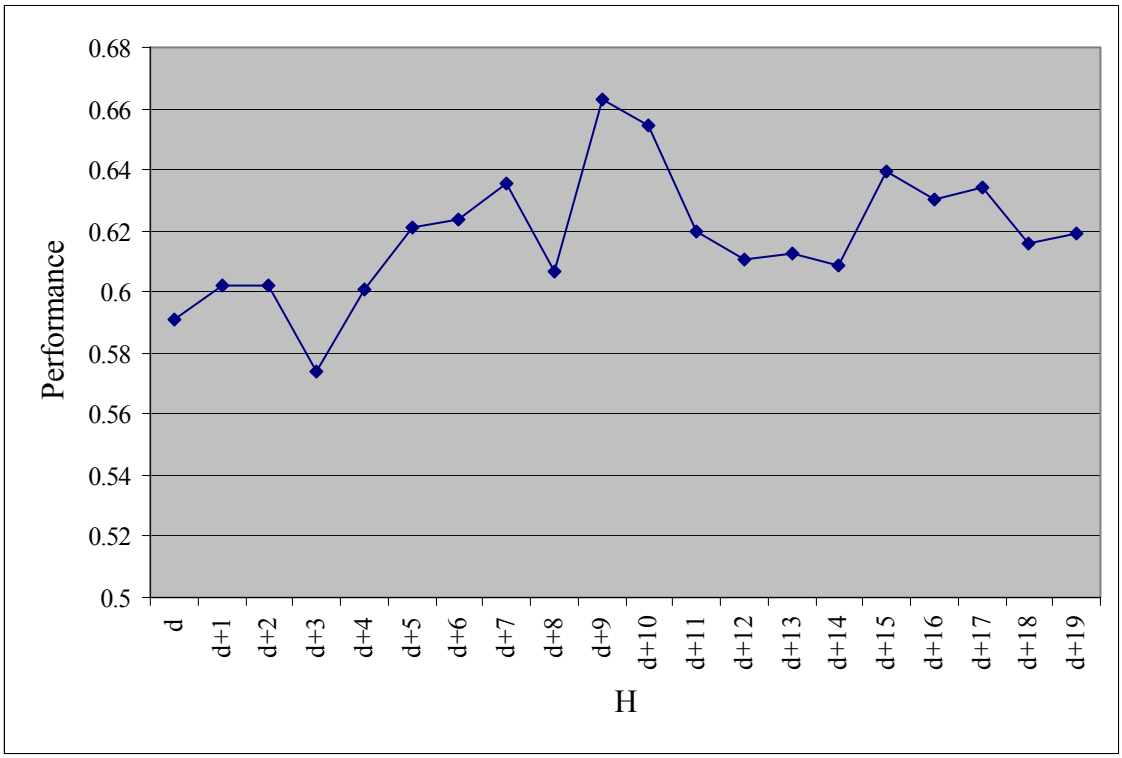


Figure 6.2. Performance of networks with varying hidden layer size

Figure 6.2. shows that adding hidden units more than *9* does not increase the performance of the network. Because of this the hidden layer size of the best network is:

$$\textit{hidden layer size} = \textit{number of input units} + 9 \qquad (6.4).$$

The final structure of the network is given in Figure 6.3. The number of input units is equal to the number of principal components selected in PCA. The hidden layer contains nine units more than the input layer. The output layer contains a single unit.

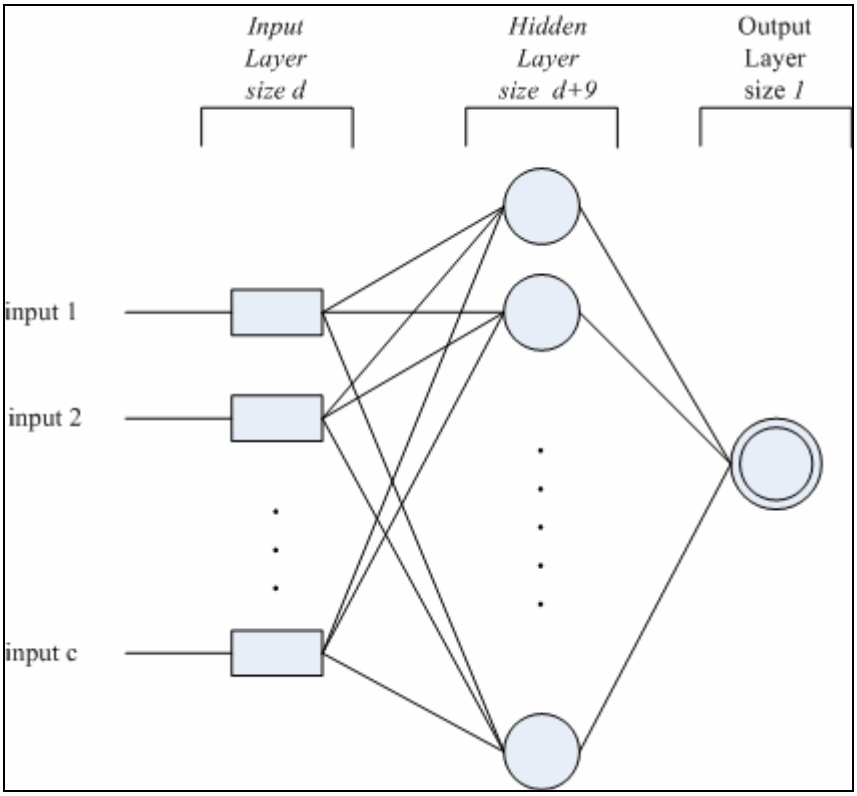


Figure 6.3. The structure of the ANN

6.2.2. Experiments with Naive Bayes

Menzies et al. have shown that a Naïve Bayes classifier using subsetting of features can be useful [14]. Menzies et al. rank the static code attributes according to their information content. After that he selects a subset of static code attributes for each project. Our aim in using this scheme is finding the best subset of attributes for defect prediction in embedded software. We performed our experiments with the naive Bayes classifier with the WEKA Data Mining Toolkit [58].

We first log-filter the dataset in order to spread the data more evenly. This makes it easier to reason about the data [14]. This process is performed by replacing every numeric value with their natural logarithms. We then use the InfoGain algorithm [14] in order to rank the static code attributes. The best five attributes for each project are given in Table 6.1.

Table 6.1. Top five code attributes in projects

	1	2	3	4	5
LP1	Conditions	Decisions	Cyc. Complexity	Hal. Level	Total LOC
LP2	Decisions	Cyc. Complexity	Conditions	Operators	Total LOC
LP3	Total LOC	Hal. Length	Hal. Volume	Operators	Cyc. complexity
CM1	LOC comments	Unique operators	Unique operands	Hal. content	Number of lines
PC1	Number of lines	Blank LOC	Unique operands	Hal. content	Total LOC
PC3	Blank LOC	Number of lines	Hal. content	Unique operands	Hal. volume
PC4	Percent comments	Code-Comment LOC	Blank LOC	Conditions	Mult. conditions

Our aim is to find the best subset of metrics for all of the projects we used in our research. Therefore we calculated the standard score of the ranks computed by InfoGain,

$$standard\ score = \frac{rank - mean}{\sigma} \tag{6.5}$$

This value denotes the distance of a given *rank* to the *mean* in terms of the *standard deviation*. The *standard score* of these ranks is a dimensionless value, and it enables us to compare the different distributions we observed in different projects. The attributes with top 15 *standard score* values are given in Table 6.2.

Table 6.2. Top 15 attributes listed in decreasing order according to their standard scores

Rank	Attribute	Standard Score
1	Percent comments	2.33
2	LOC comments	2.28
3	Code-Comment LOC	2.08
4	Unique operands	1.67
5	Total LOC	1.50
6	Unique operators	1.46
7	Executable LOC	1.24
8	Hal. volume	1.22
9	Conditions	1.16
10	Decisions	1.16
11	Operators	1.14
12	Hal. length	1.10
13	Cyc. complexity	1.06
14	Hal. effort	0.52
15	Hal. programming time	0.52

We build predictors using iterative InfoGain subsetting [14]. This means that we start with the best attribute and build a predictor using that attribute. After that we

iteratively increase the number of attributes. We stop when the performance of the predictor ceases to improve. The final subset of attributes is selected as the best subset of attributes. This procedure enables us to identify the static code attributes that contain the highest amount of information for embedded software. The performances of the classifiers using iterative InfoGain subsetting is given in Figure 6.4.

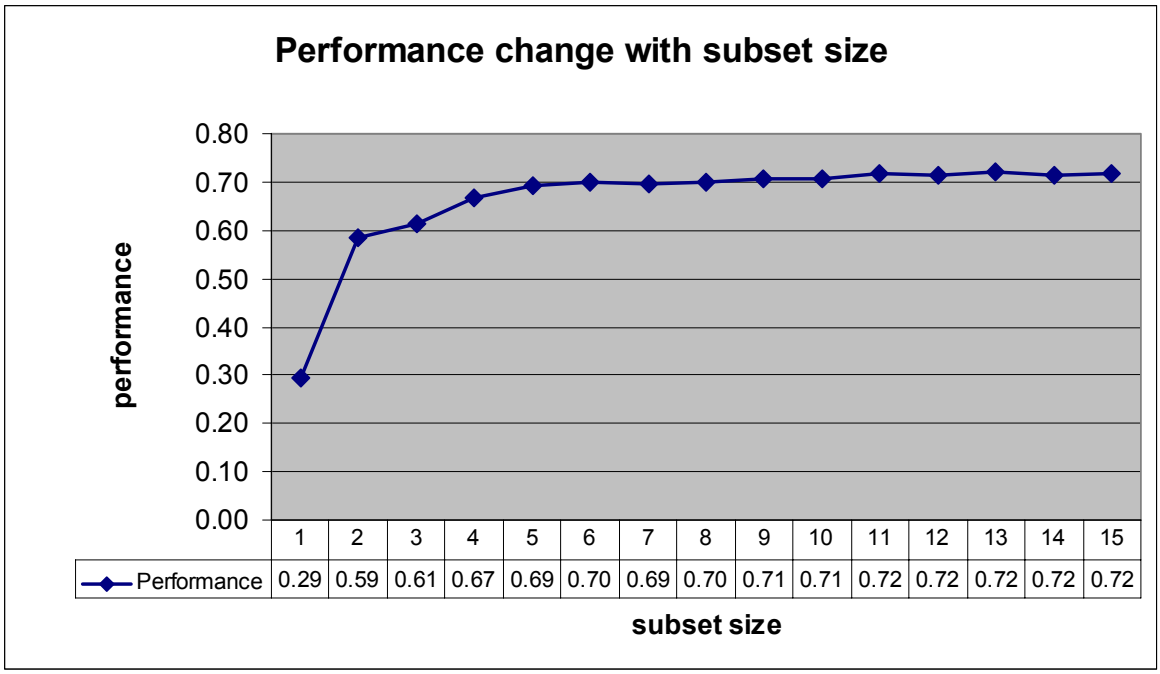


Figure 6.4. Performance of classifiers with different subset of attributes

Figure 6.4 shows that the performance of the classifier does not increase significantly after the addition of the sixth attribute to the subset. However performance reaches its maximum after the addition of the 11th attribute. Therefore we select the top 11 attributes listed in Table 6.2 for using with naive Bayes classifier.

6.2.3. Experiments with Voting Feature Intervals

Our aim in using the classifier based on VFI is obtaining a classifier which does not incur any information loss. Therefore we did not preprocess the data used by the VFI classifier. The data coming from the seven projects were directly fed to the VFI based classifier. These experiments were performed using the WEKA Data Mining Toolkit [58].

6.2.4. Experiments with Ensemble of Classifiers

Our aim in forming an ensemble of classifiers is combining the strengths of multiple classifiers. There is no single model that is capable of generalizing a wide variety of projects. We propose using an ensemble of classifiers to overcome this problem. In order to combine the classifiers we adopt a voting scheme. We classify a module as defective if the majority of classifiers classify it as defect-prone, which means that at least two of the classifiers has to signal a module as defect-prone.

6.3. Threats to Validity

The first threat to validity is the sampling bias. We have used multiple data sources with varying functional requirements and development environments to get around this problem. However sampling bias threatens any data mining research, as Menzies et al. delicately express "what matters there may not be true here" [14]. Nevertheless, we still can say that our research uses a more comprehensive dataset compared to prior research on defect prediction on embedded software, which draws conclusions about the usefulness of defect prediction [9], [58].

Another threat comes from the observation that not all defects can be explained with the information in static code attributes. There are other human related defect triggers such as lack of domain knowledge or working under time pressure [59]. Another bias comes from the fact that we used three machine learning techniques for building an ensemble. However machine learning literature is vast and many other techniques are available [25]. It is not feasible to test all of these techniques in an experimental research.

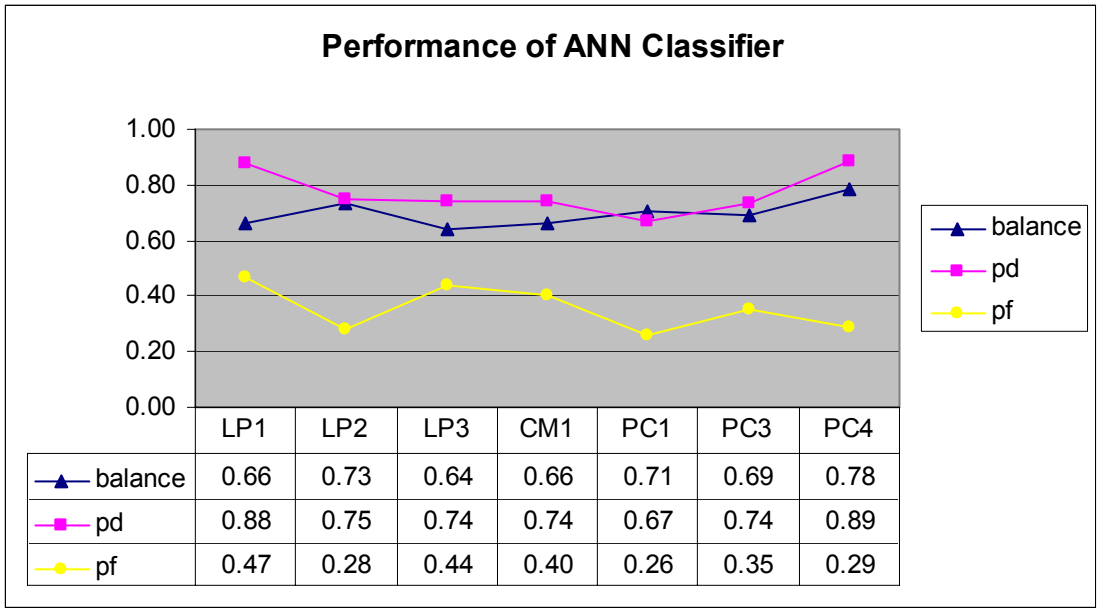
6.4. Results

We discuss the results of our experiments in this section. We have stated that we use *balance*, *probability of detection* (*pd*), and *probability of false alarm* (*pf*) values to evaluate the predictors. The *pd* of a predictor measures how good is a predictor in finding the defective modules. The *pf* of a predictor measures the probability of classifying a defect-free module as defective. The *balance* of a predictor measures how close a predictor is to

the ideal predictor when used as a percentage. Therefore if we want to compare two predictors we can say that the one with a higher *balance* is preferable [14]. Our model comprises four different machine learning techniques. First the results of our experiments with each of these techniques is presented. Second, we discuss these results from a data mining perspective and their utility in embedded software development.

6.4.1. Performance of the Classifiers

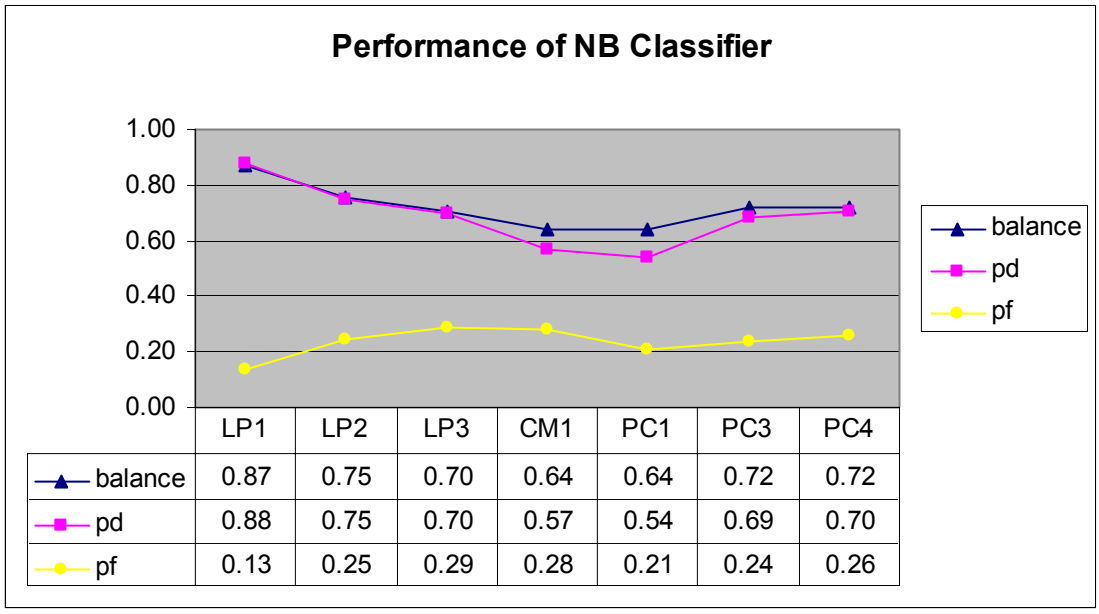
The performance of the ANN classifier is presented in Figure 6.5. We see that the ANN classifier detects at least 64 per cent of the defective modules in all projects. However it classifies 47 per cent of the defect-free modules as defective in the worst case. The average *balance* of this predictor is 70 per cent.



	LP1	LP2	LP3	CM1	PC1	PC3	PC4
balance	0.66	0.73	0.64	0.66	0.71	0.69	0.78
pd	0.88	0.75	0.74	0.74	0.67	0.74	0.89
pf	0.47	0.28	0.44	0.40	0.26	0.35	0.29

Figure 6.5. The performance of ANN classifier on our dataset

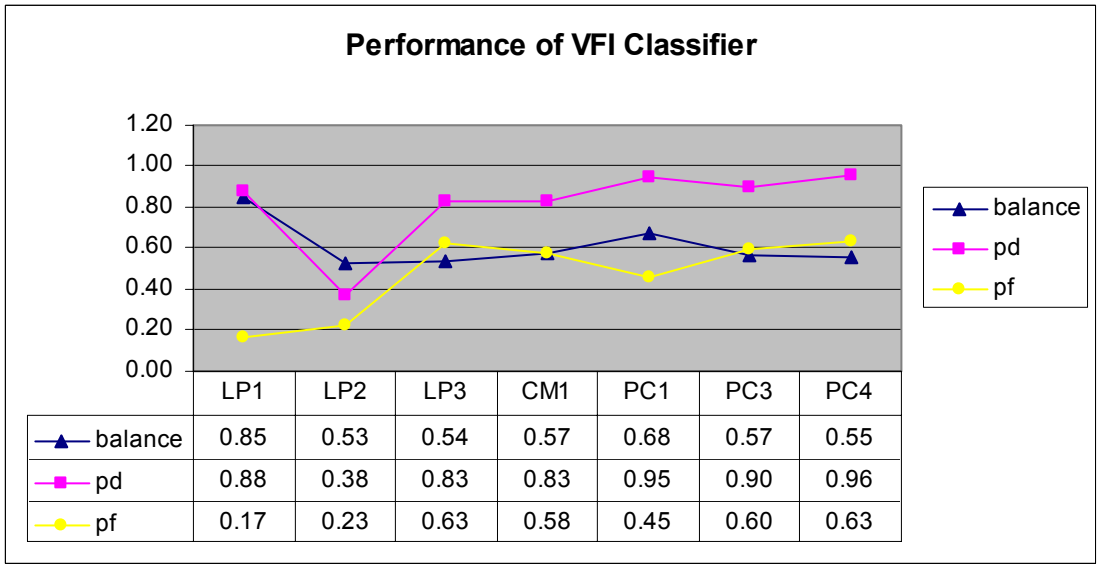
The performance of the NB classifier is given in Figure 6.6. We see that the this classifier detects 64 per cent of defective modules in the worst case. However it also classifies as much as 29 per cent of the defect-free modules as defective. The average *balance* of this classifier is 72 per cent.



	LP1	LP2	LP3	CM1	PC1	PC3	PC4
balance	0.87	0.75	0.70	0.64	0.64	0.72	0.72
pd	0.88	0.75	0.70	0.57	0.54	0.69	0.70
pf	0.13	0.25	0.29	0.28	0.21	0.24	0.26

Figure 6.6. The performance of NB classifier on our dataset

The performance of the VFI classifier is shown in Figure 6.7. The results show that this classifier detects at least 38 per cent defective modules. Its *probability of false* alarm is 63 per cent in the worst case. Average *balance* of this classifier is 61 per cent.

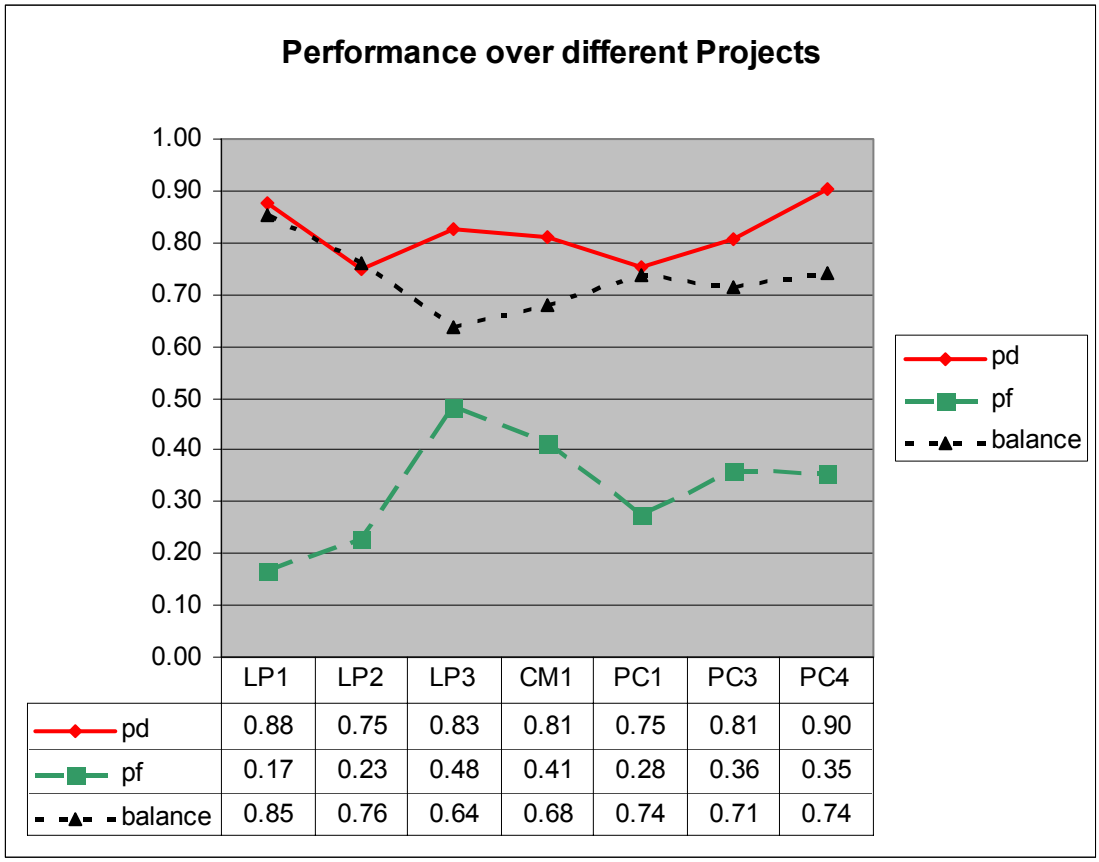


	LP1	LP2	LP3	CM1	PC1	PC3	PC4
balance	0.85	0.53	0.54	0.57	0.68	0.57	0.55
pd	0.88	0.38	0.83	0.83	0.95	0.90	0.96
pf	0.17	0.23	0.63	0.58	0.45	0.60	0.63

Figure 6.7. The performance of VFI classifier on our dataset

Our aim in this research is to combine multiple classifiers for predicting the defects in embedded software systems. We let the majority decide and classify a module as

defective if at least two predictors signal it as defect-prone, this scheme is called an ensemble of classifiers [24]. The performance of the ensemble is presented in Figure 6.8.



	LP1	LP2	LP3	CM1	PC1	PC3	PC4
pd	0.88	0.75	0.83	0.81	0.75	0.81	0.90
pf	0.17	0.23	0.48	0.41	0.28	0.36	0.35
balance	0.85	0.76	0.64	0.68	0.74	0.71	0.74

Figure 6.8. Performance of our model

Our model has a *pd* of 64 per cent in the worst case. It classifies at most 48% of the defect-free modules as defective. It has an average *balance* of 73 per cent.

6.4.2. Analysis of the Results

We have demonstrated the performance of our model on our dataset. However it is necessary to discuss these results from a data mining perspective. We also need to analyze their utility in embedded software development. The performance of our model on all of our datasets is presented in Table 6.3.

Table 6.3. The performance of our model

Technique	pd	pf	balance
ANN	0.77	0.35	70%
NB	0.69	0.24	72%
VFI	0.82	0.47	61%
Ensemble	0.82	0.33	73%

Our aim in using an ANN classifier was to capture the complex relationship of static code attributes and the presence of defects in the code. Our results show that ANN based classifier is actually useful for this purpose. It is able to identify 77 per cent of the defective modules and on top of that it incurs only 35 per cent *probability of false alarm* on the average.

We used a naive Bayes classifier with InfoGain subsetting in order to identify the best subset of static code attributes for defect prediction in embedded software and build a predictor using this information. Our results show that the *balance* of the NB classifier is higher than two other predictors used in this study.

We also needed a machine learning technique that does not lose any valuable information in any of our datasets. We selected the VFI classifier for this purpose. The *probability of detection* of the VFI classifier supports our claim for defective modules. The average *pd* of this classifier is higher than the other two machine learning techniques in our experiments.

In order to obtain a useful defect predictor for embedded software systems we need a classifier which has a high *balance*, and on top of that it should have high *probability of detection*. Our model combines the strengths of three different techniques to achieve this. The ANN based predictor has a good *balance* along with a high *pd*. The NB classifier has a very low *probability of false alarm*. The strength of the VFI classifier is that it achieves a very high *pd*. Our model outperforms all three techniques by combining them. Our results show that the ensemble of classifiers is the best predictor by having the highest *balance* value. On the other hand it misses only 18 per cent of the defective modules on our datasets.

One criticism to our results could be that the ensemble outperforms the naive Bayes predictor by only 1 per cent in *balance*. Therefore it is not necessary to use other techniques. This criticism might be valid from a data mining perspective. However we believe that it is not valid in the context of embedded software development. The ensemble has a *pd* that is 5 per cent higher than its best contributor naive Bayes on our dataset. This means that our model may be more useful in embedded software development perspective where reliability is of utmost importance.

We also performed t-tests for the sake of entirety. The results of these tests are given in Appendix F.

6.4.3. Comparison of our Results to Other Experiments

Four of the projects we used in our research are also used by Menzies et al. for defect prediction [14]. It is important to note that Menzies et al. do not propose a general defect prediction model for all of these projects. Instead they propose using a different subset of static code attributes for each project and they also use a different subset selection algorithm for one of the projects. They aim to discuss the results of best predictors on a public dataset. However our aim is to propose a defect prediction model that is applicable to a wide variety of embedded software. Their results and our results for the 4 projects that appear in both research are given in Table 6.4.

Table 6.4. Our results versus results in [14]

Project	pd		pf		balance	
	Our Model	*Model of Menzies et al.*	Our Model	*Model of Menzies et al.*	Our Model	*Model of Menzies et al.*
CM1	0.81	*0.71*	0.41	*0.27*	68%	*72%*
PC1	0.75	*0.48*	0.28	*0.17*	74%	*61%*
PC3	0.81	*0.80*	0.36	*0.35*	71%	*71%*
PC4	0.90	*0.98*	0.35	*0.29*	74%	*79%*
average	0.82	*0.74*	0.35	*0.27*	72%	*74%*

We can see in Table 6.4 that the *balance* values for both models are very close. Menzies et al. have better *pf* values compared to our model. However our results indicate a

higher *pd* in these projects. We believe that because of higher *pd* values our results are more promising for embedded software.

7. CONCLUSION

We have proposed a defect prediction model for embedded software. This model uses static code attributes for identifying the defect-prone modules of a software system. It utilizes an ensemble of three classifiers for combining their strengths. Our results show that this model can actually be useful in embedded software development.

In the remainder of this chapter we discuss our contribution to the defect prediction research. We start with a discussion of the importance of our results. Finally we conclude our discussion with possible future directions for improving our research.

7.1. Contribution

Many defect prediction models have been proposed in prior research however there are few that specifically target embedded software systems. Besides that there is not a defect prediction model that specifically aims embedded software systems and uses static code attributes. Our research fills this gap. It proposes a useful defect prediction model for embedded software that uses static code attributes.

One problem with prior research on defect prediction is that they generally use data coming from similar systems. Therefore it is not possible to argue that the results will generalize beyond the data used in those research. We used data coming from different embedded software systems. Our data include projects that have been widely used in defect prediction research and data coming from a local white goods manufacturer. The functional requirements of the projects used in our research vary significantly. We contributed to defect prediction research by testing the performance of different machine learning techniques on various datasets.

Defect prediction literature is vast and many statistical and machine learning techniques have been utilized in prior research. We tested the performance of two techniques that have not been used in defect prediction before. One of them is the

classification by Voting Feature Intervals and the other is an ensemble of classifiers. Our results show that both techniques can be useful for defect prediction in embedded software.

Our aim in proposing a model for defect prediction in embedded software was to aid software developers in the planning of future iterations and testing phases. Our model finds 82 per cent of the defective modules in our datasets. We believe that this model may be used by the software engineers in the industry to aid them in decision making. The knowledge of a large percentage of the defective modules can be used to focus on those modules. Moreover by using our proposed model software development managers may better design testing phases and hence they can capture more defects in the source code.

7.2. Future Work

The model we propose have promising results. However improvements are still possible. We use a set of static code attributes for predicting the defect-prone parts of an embedded software system. The set of attributes we use in our research is partially determined by the available data for our purposes. One direction for improvement could be to extend the set of metrics used in the model, especially with inter-module metrics. This will enable us to capture more information pertaining to the software under consideration.

We deployed several techniques from machine learning research. However there are many other techniques that may prove to be useful for defect prediction in embedded software. Our model can be extended by using other data mining techniques or perhaps improved by replacing the ones used in it. An ensemble is just one way of combining classifiers. We plan to investigate classifier cascading for better defect prediction in embedded software in our future work.

8. APPENDIX A: SOFTWARE METRICS USED IN THIS THESIS

This appendix presents the complete list of static code attributes used in our research [23]. [24]. These attributes can be categorized into two categories, base metrics and composite metrics. The base metrics are directly extracted from the source code. These metrics are given in Table A.1.

Table A.1. Base Metrics

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

The composite metrics are calculated using the base metrics. These metrics present an initial interpretation of the base metrics. The aim is to quantify various attributes of the source code. These metrics are given in Table A.2.

Table A.2. Composite Metrics

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

9. APPENDIX B: OBJECT-ORIENTED DESIGN METRICS

These metrics were proposed in [48]. Table B.1 contains the list of these metrics and their descriptions.

Table B.1. Object-Oriented Design Metrics

Metric	Description
Weighted Methods per Class	Number of methods in a class
Depth of Inheritance Tree	The maximum length from the node denoting this class in the inheritance tree, to the root of the tree
Number of Children	Number of immediate subclasses of a class
Coupling between Objects	The count of the number of other classes to which a class is coupled
Response for a Class	Set of all methods called by a methods in a class
Lack of Cohesion in Methods	This is a count of method pairs whose similarity is 0

10. APPENDIX C: REQUIREMENTS FOR THE PROJECT CM1

We present an excerpt from the requirements taken from the project CM1 [15]. Some selected requirements are as follows:

- The DPU-A shall include the capability to dynamically load object modules from the EEPROM.
- The DPU-A shall provide a function to allow an application program to write to the Real-Time Clock registers on the RAD6000SC CPU Module.
- The DPU-B shall create a fixed analog conversion scan list containing the A DPU analog values (active values plus one ground reference value).
- The DPU-B shall provide an application program with the capability to read the most recent A/D results in the scan list.
- The DPU-C shall provide the capability for an application program to enable/disable the VME Slave Interrupt on the D for the DPU-C Interface.
- The DPU-C shall read from the C registers in 16-bit words only.
- The DPU-E shall provide the capability for an application program to set the Address Limit at which the hardware will swap ping-pong buffers.
- The DPU-E shall provide the capability for an application program to set a timeout in milliseconds that specifies a period during which no new events have been received by the hardware, after which the hardware will swap ping-pong buffers and interrupt the software.
- The DPU-E shall provide an application program with the capability to command the E hardware to reorder the bit sequence of each event's original Detector Electronics Output Format as it is received.
- The DPU-E shall provide an application program with the ability to toggle the hardware interface between Normal mode and Test mode.

11. APPENDIX D: TRANSITION DIAGRAM FOR THE METRICS EXTRACTION TOOL

This appendix gives the details of the transition diagram that has been implemented in the metrics extraction tool. This diagram enables us to identify the lexemes in the source code as tokens. It is partially based on the syntax diagrams given in [52]. Some modifications have been made on those diagrams. The reason for this is that we do not need to whole semantic information in the source code. The diagram is very large therefore it is shown in Figures D.1, D.2, D.3, and D.4. The start state is repeated in each of the parts for the sake of clarity.

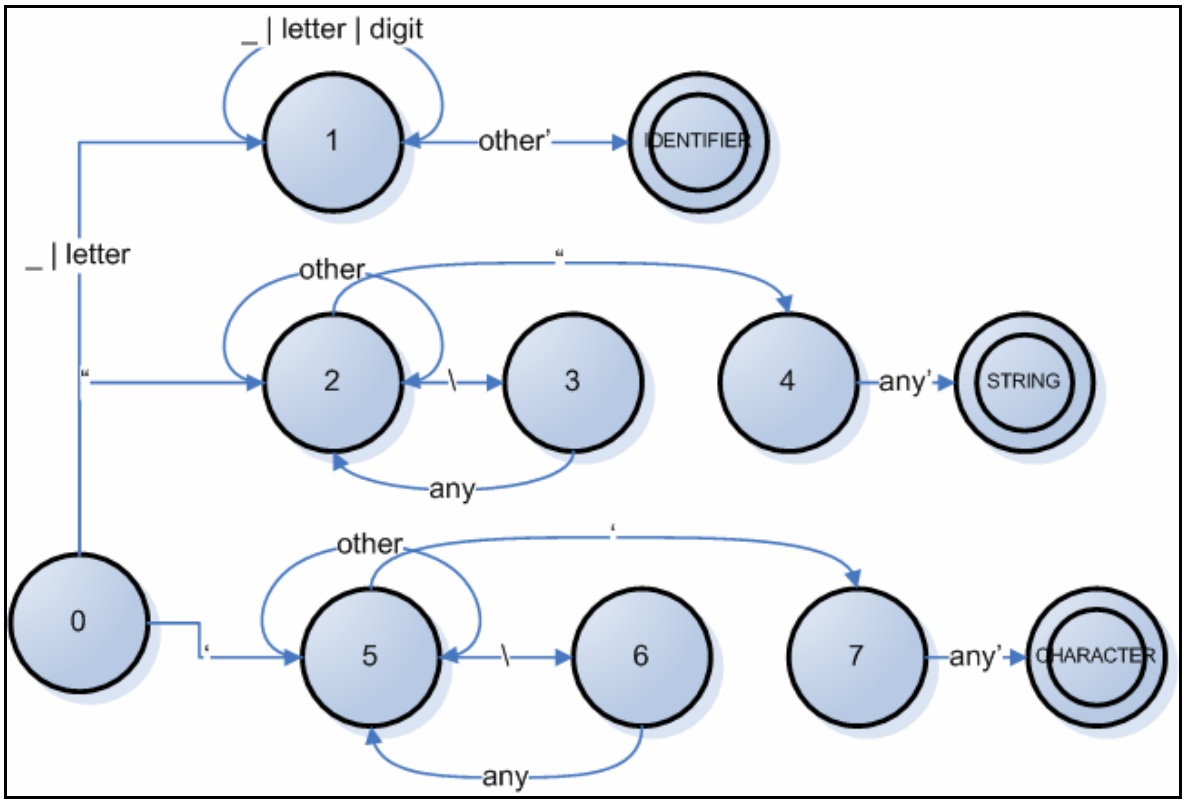


Figure D.1. Transition diagram, part 1

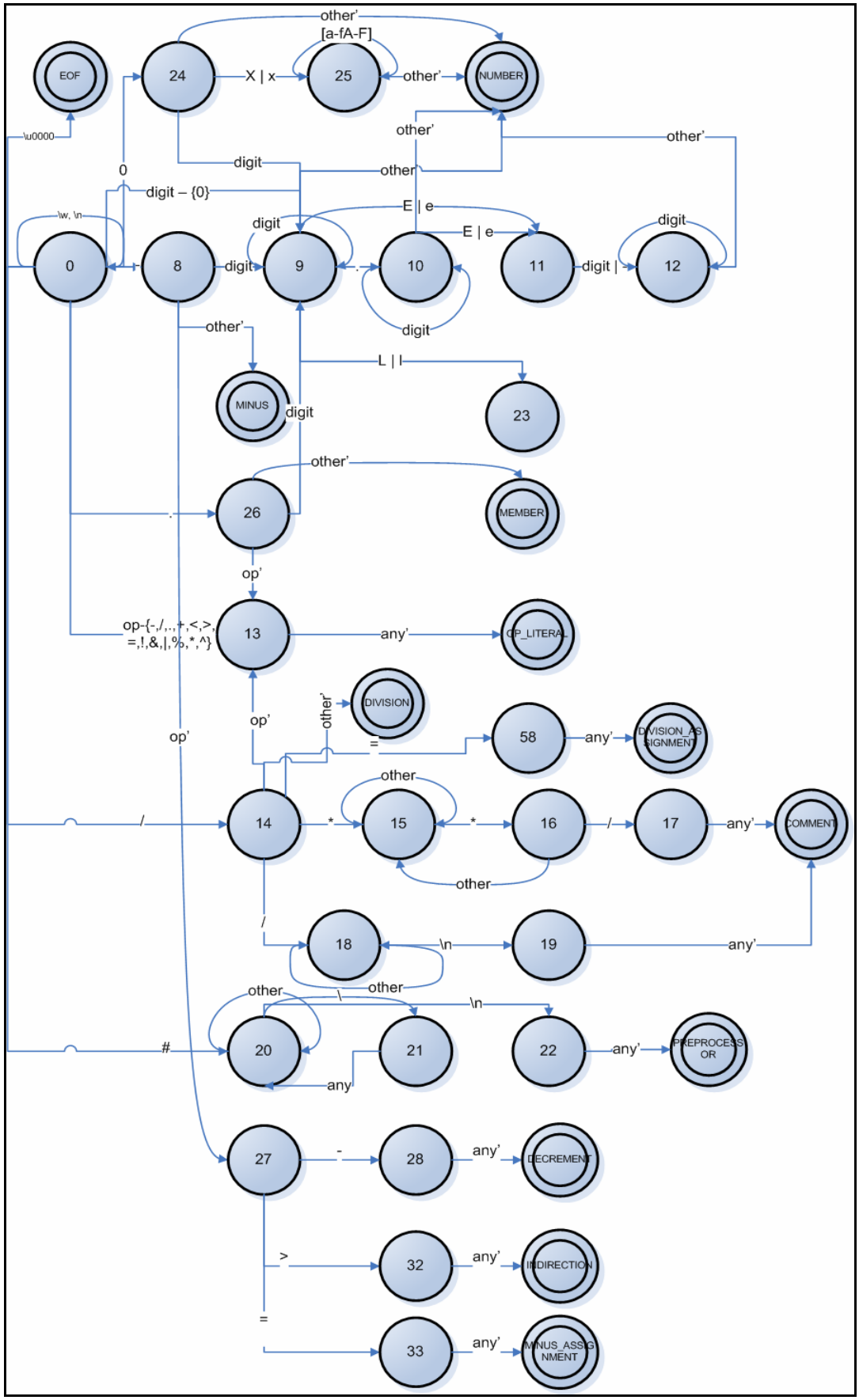


Figure D.2. Transition Diagram, part 2

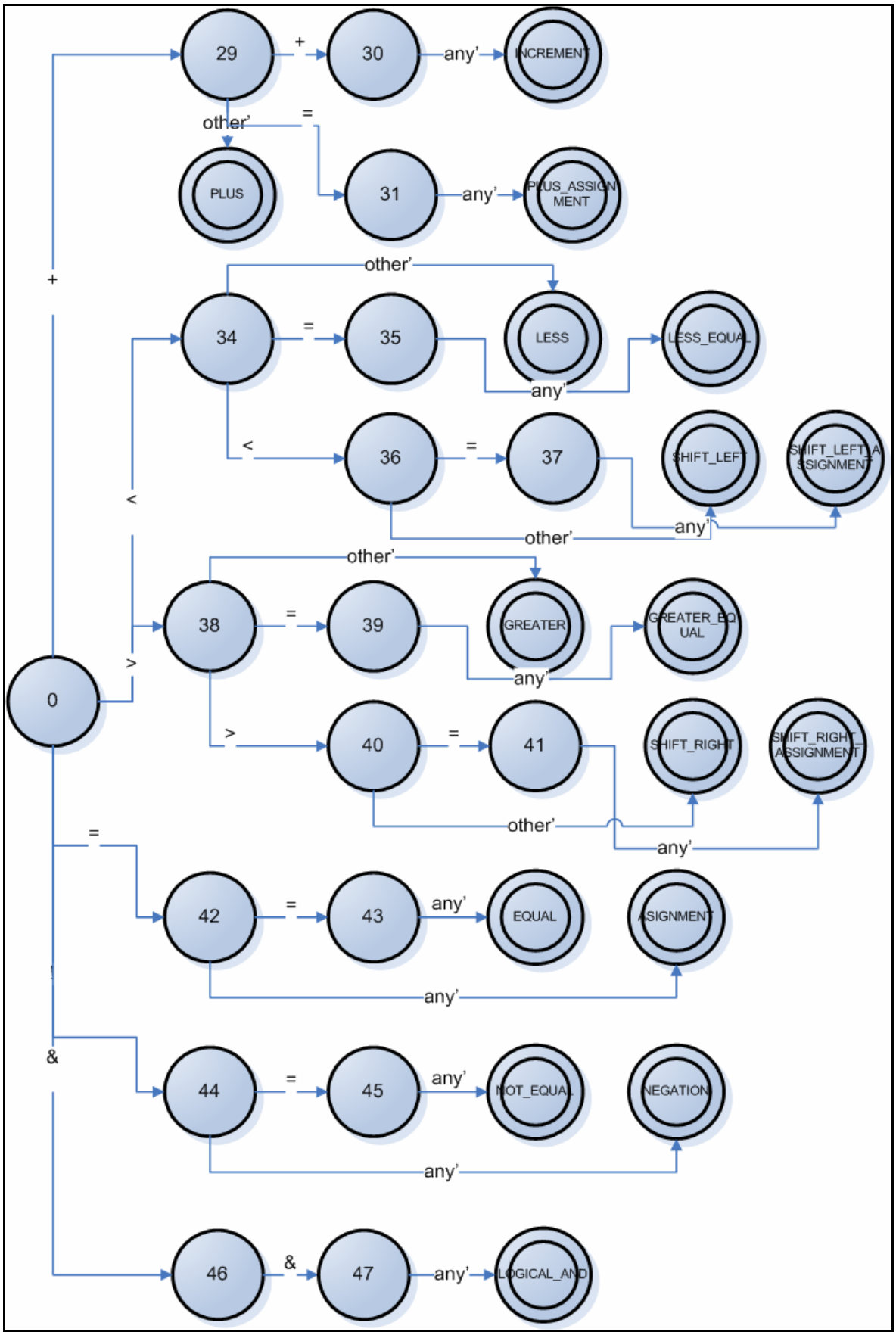


Figure D.3. Transition Diagram, part 3

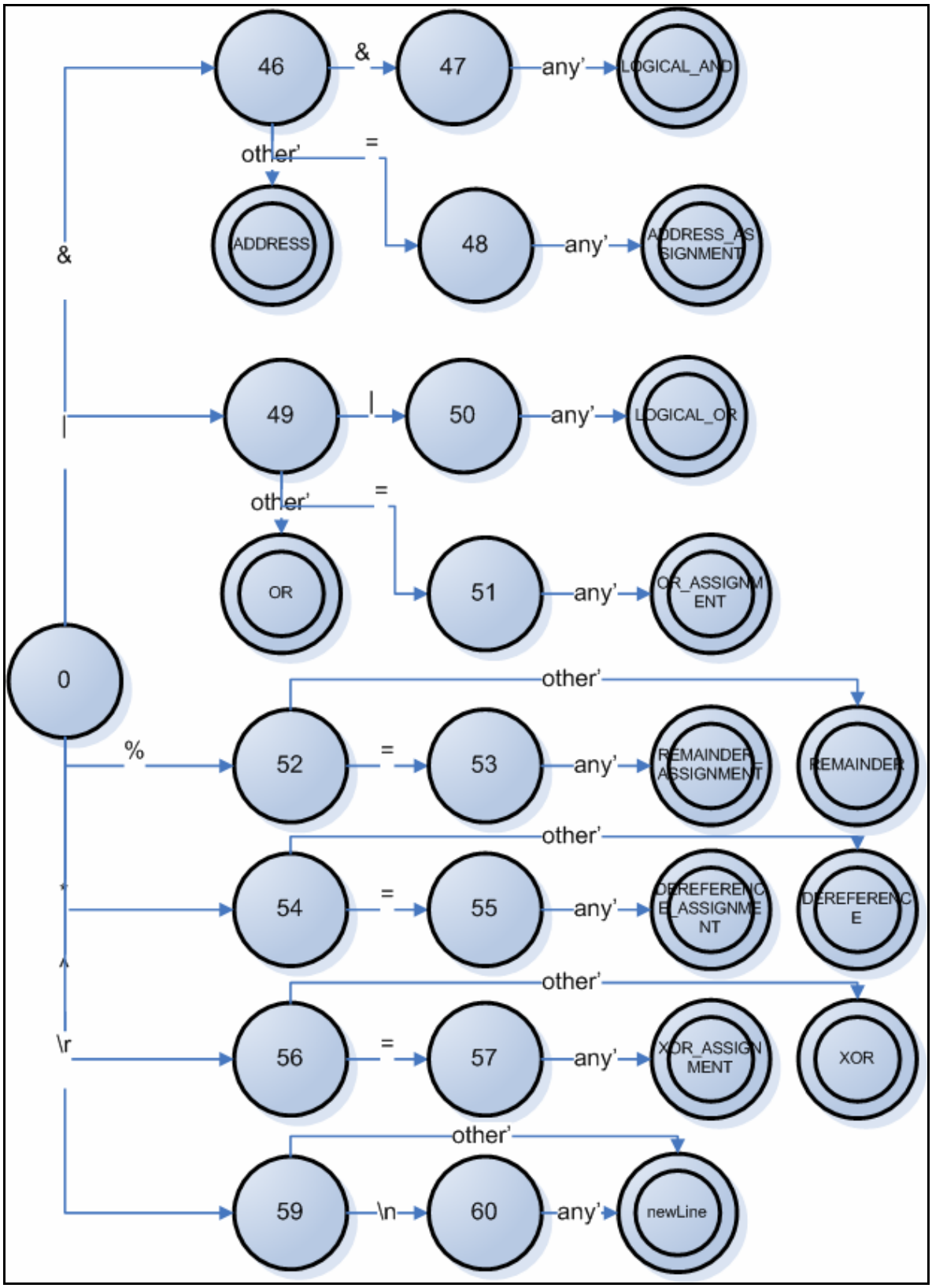


Figure D.4. Transition Diagram, part 4

12. APPENDIX F: RESULTS OF T-TESTS AMONG THE CLASSIFIERS

The results of the t-tests among the classifiers are given in Table F.1. If the null hypothesis was rejected it is denoted with a 1, and 0 otherwise. The table contains an X entry for the unnecessary entries.

Table F.1. The results of the t-tests

		ANN	NB	VFI	Ensemble
LP1	ANN	X	1	1	1
	NB	X	X	1	0
	VFI	X	X	X	0
	Ensemble	X	X	X	X
LP2	ANN	X	0	0	0
	NB	X	X	0	0
	VFI	X	X	X	0
	Ensemble	X	X	X	X
LP3	ANN	X	0	0	0
	NB	X	X	1	1
	VFI	X	X	X	0
	Ensemble	X	X	X	X
CM1	ANN	X	0	0	0
	NB	X	X	1	0
	VFI	X	X	X	0
	Ensemble	X	X	X	X
PC1	ANN	X	0	0	0
	NB	X	X	1	0
	VFI	X	X	X	0
	Ensemble	X	X	X	X
PC3	ANN	X	0	1	0
	NB	X	X	1	0
	VFI	X	X	X	1
	Ensemble	X	X	X	X
PC4	ANN	X	0	1	0
	NB	X	X	1	0
	VFI	X	X	X	1
	Ensemble	X	X	X	X

REFERENCES

1. Crosby, P.B., *Quality is Free: The Art of Making Quality Certain*, McGraw-Hill, NewYork, 1979.

2. Juran J.M. and F.M. Gryna, Jr., *Quality Planning and Analysis: From Product Development Through Use*, McGraw-Hill, New York, 1970.

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

4. Lee, E.A., "Embedded Software", in M. Zelkowitz (ed.), *Advances in Computers,* Vol. 56, Academic Press, London, 2002.

5. Li, Q., *Real-Time Concepts for Embedded Systems*, CMP Books, San Francisco, 2003.

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

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

8. Satzinger, J.W., R.B. Jackson and S.D. Burd, *Systems Analysis and Design in Changing World*, Course Technology, Massachusetts, 2002.

9. 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 & Systems*, Vol. E88-D, No. 6, pp. 1134-1141, June 2005.

10. Perez-Minana, E. and J.J. Gras, "Improving Fault Prediction Using Bayesian Networks for the Development of Embedded Software Applications", *Software Testing Verification and Reliability*, Wiley Interscience, Vol. 16, No. 3, pp. 157-174, 2006.

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

12. Khoshgoftaar, T.M. and R.M. Szabo, "Using Neural Networks to Predict Software Faults During Testing", *IEEE Transactions on Reliability*, Vol. 45, No. 3, pp. 456-462, September 1996.

13. Khoshgoftaar, T.M. 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*, Virginia, 13-15 November 2006, pp. 651-658, IEEE Computer Society, Washington DC, 2006.

14. 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.

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

16. Futrell, R.T., D.F. Shafer and L.I. Shafer, *Quality Software Management*, Prentice Hall PTR, Indiana, 2002.

17. Stankovic, J.A., "Strategic Directions in Real-Time and Embedded Systems", *ACM Computing Surveys*, Vol. 28, No. 4, pp. 751-763, December 1996.

18. Boehm, B. and V.R. Basili, "Software Defect Reduction Top 10 List", *Computer*, Vol. 34, No. 1, pp. 135-137, IEEE Computer Society Press, California, January 2001.

19. Nagappan, N. and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density", *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, 15-21 May 2005, pp. 284-292, ACM Press, New York, 2005.

20. Khoshgoftaar, T.M., E.B. Allen, J.P. Hudepohl and S.J. Aud, "Application of Neural Networks to Software Quality Modeling of a Very Large Telecommunications System", *IEEE Transactions on Neural Networks*, Vol. 8, No. 4, pp. 902-909, July 1997.

21. Challagulla, V.U.B., F.B. Bastani, I. Yen and R.A. 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, 02-04 February 2005, Vol. 00, pp. 263-270, IEEE Computer Society, Washington DC, 2005.

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

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

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

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

26. Lutz, R.R., "Analyzing Software Requirements Errors in Safety-Critical Embedded Systems", *IEEE International Symposium on Requirements Engineering*, San Diego, 1993, pp. 126-133, IEEE Computer Society Press, Washington DC, 1993.

27. Ward Jr., W.A., B. Venkataraman, "Some Observations on Software Quality", *Proceedings of the 37th Annual Southeast Regional Conference*, Mobile, AL, 15-18 April 1999, ACM Press, New York, 1999.

28. Garvin, D., "What Does "Product Quality" Really Mean?", *MIT Sloan Management Review*, Vol. 26, No. 1, pp. 25–43, 1984.

29. Clarke, E.M., J.M. Wing, "Formal Methods: State of the Art and Future Directions", *ACM Computing Surveys*, Vol. 28, No. 4, pp. 626-643, December 1996.

30. Schieferdecker, I., E. Bringmann, J. Grosmann, "Continuous TTCN-3: Testing of Embedded Control Systems", *Proceedings of the 2006 International Workshop on Software Engineering for Automotive Systems*, Shanghai, 20-28 May 2006, pp.29-36, ACM Press, New York, 2006.

31. Sturmer, I., M. Conrad, I. Fey and H. Dorr, "Experiences with Model and Autocode Reviews in Model-based Software Development", *Proceedings of the 2006 International Workshop on Software Engineering for Automotive Systems*, Shanghai, 20-28 May 2006, pp.45-52, ACM Press, New York, 2006.

32. Gelperin, D. and B. Hetzel, "The Growth of Software Testing", *Communications of the ACM*, Vol. 31, No. 6, pp. 687-695, June 1988.

33. Dalal, S.R., J.R. Horgan and J.R. Kettenring, "Reliable Software and Communication: Software Quality, Reliability, and Safety", *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, 1993, pp. 425-435, IEEE Computer Society Press, Los Alamitos, 1993.

34. Harrison, W., "Using Software Metrics to Allocate Testing Resources", *J. Management Information Systems*, Vol. 4, No. 4, pp. 93-105, April 1988.

35. Hsiung, P., "Formal Synthesis and Code Generation of Embedded Real-time Software", *Proceedings of the Ninth International Symposium on Hardware/Software Codesign*, Copenhagen, 25-27 April 2001, pp. 208-213, ACM Press, New York, 2001.

36. "Capability Maturity Model for Software", *Technical Report CMW/SEI-91-TR-24*, Carnegie Mellon University, 1991.

37. Basili, B.R., L.C. Briand and W.L. 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.

38. Gyimothy, T., R. Ferenc and I. Siket, "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction", *IEEE Transactions on Software Engineering*, Vol. 31, No. 10, October 2005.

39. Denaro, G., S. Morasca and M. Pezze, "Deriving Models of Software Fault-Proneness", *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, Ischia, 15-18 July 2002, pp. 361-368, ACM Press, New York, 2002.

40. Shiavi, R., *Introduction to Applied Statistical Signal Analysis*, Academic Press, San Diego, 1999.

41. Perlich, C., F. Provost and J.S. Simonoff, "Tree Induction vs. Logistic Regression: A Learning-curve Analysis", *The Journal of Machine Learning Research*, Vol. 4, pp. 211-255, December 2003.

42. Russell, S. and P. Norvig, *Artificial Intelligence A Modern Approach*, Prentice Hall, New Jersey, 2003.

43. Ceylan, E., O. Kutlubay, A.B. Bener, "Software Defect Identification Using Machine Learning Techniques", *Proceedings of the 32nd EUROMICRO Conference on Software*

Engineering and Advanced Applications, Dubrovnik, 29 August-1 September 2006, pp. 240-247, IEEE Computer Society Press, Los Alamitos, 2006.

44. Menzies, T., J. DiStefano, A. Orrego, and R. Chapman, "Assessing Predictors of Software Defects", *Proceedings of the Workshop on Predictive Software Modules*, Chicago, 17 September 2004.

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

46. van Ommering, R., "Software Reuse in Product Populations", *IEEE Transactions on Software Engineering*, Vol. 31, No. 7, July 2005.

47. Demiroz, G. and H.A. Guvenir, "Classification by Voting Feature Intervals", *Proceedings of the Ninth European Conference on Machine Learning*, Prague, 23-25 April 1997, pp. 85-92, Springer-Verlag, London, 1997.

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

49. Sayyad-Shirabad, J. and T.J. Menzies, *The PROMISE Repository of Software Engineering Databases*, http://promise.site.uottawa.ca/SERepository, 2005.

50. Kaminsky, K. and G.D. Boetticher, "Better Software Defect Prediction Using Equalized Learning with Machine Learners", *Proceeding of the Fourth Conference on Knowledge Sharing and Collaborative Engineering*, St. Thomas, 22-24 November 2004, Acta Press, Calgary, 2004.

51. Aho, A.V., R. Sethi, J.D. Ullman, *Compilers Principles, Techniques, and Tools*, Prentice Hall, New Jersey, 2003.

52. Johnsonbaugh, R. and M. Kalin, *Applications Programming in C*, Macmillan Publishing Company, New York, 1989.

53. 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, June 1003.

54. Fisher, D., L. Xu and N. Zard, "Ordering Effects in Clustering", *Proceedings of the Ninth International Workshop on Machine learning*, Aberdeen, 1-3 July 1992, pp.163-168, Morgan Kaufmann Publishers Inc., San Francisco, 1992.

55. Khoshgoftaar, T.M., R.M. Szabo and P.J. Guasti, "Exploring the Behavior of Neural Network Software Quality Models", *IEE Software Engineering Journal*, Vol. 10, No. 3, pp. 89-96, May 1995.

56. The MathWorks Inc., The MAthWorks – MATLAB and Sinulink for Technical Computing, http://www.mathworks.com/, 2007.

57. Fenton, N., M. Neil, W. Marsh, P. Hearty, L. Radlinski and P. Krause, "Project Data Incorporating Qualitative Factors for Improved Software Defect Prediction", *Proceedings of the International Conference on Software Engineering Promise 2007 Workshop*, Minneapolis, 20-26 May 2007, 2007.

58. Witten, I.H. and E. Frank, *Data Mining: Practical machine learning tools and techniques*, Morgan Kaufmann, San Francisco, 2005.

59. Leszak, M., D.E. Perry and D. Stoll, "A Case Study in Root Cause Defect Analysis", *Proceedings of the 22nd International Conference on Software Engineering"*, Limerick, 4-11 June 2000, pp. 428-437, ACM Press, New York, 2000.