12-11-2004

Predicting Open-Source Software Quality Using Statistical and Machine Learning Techniques

Amit Ashok Phadke

PREDICTING OPEN-SOURCE SOFTWARE QUALITY USING STATISTICAL AND MACHINE LEARNING TECHNIQUES

By

Amit Ashok Phadke

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

December 2004

PREDICTING OPEN-SOURCE SOFTWARE QUALITY USING STATISTICAL AND MACHINE LEARNING TECHNIQUES

By

Amit Ashok Phadke

Approved:

Edward B. Allen
Associate Professor of Computer Science and Engineering, and Graduate Coordinator, Department of Computer Science and Engineering
(Major Professor)

Julian E. Boggess
Associate Professor of Computer Science and Engineering
(Committee Member)

Susan M. Bridges
Professor of Computer Science and Engineering
(Committee Member)

W. G. Steele
Interim Dean of the James Worth Bagley College of Engineering

Name: Amit Ashok Phadke

Date of Degree: December 11, 2004

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Edward B. Allen

Title of Study: PREDICTING OPEN-SOURCE SOFTWARE QUALITY USING STATISTICAL AND MACHINE LEARNING TECHNIQUES

Pages in Study: 128

Candidate for Degree of Master of Science

Developing high quality software is the goal of every software development organization. Software quality models are commonly used to assess and improve the software quality. These models, based on the past releases of the system, can be used to identify the fault-prone modules for the next release. This information is useful to the open-source software community, including both developers and users. Developers can use this information to clean or rebuild the faulty modules thus enhancing the system. The users of the software system can make informed decisions about the quality of the product. This thesis builds quality models using logistic regression, neural networks, decision trees, and genetic algorithms and compares their performance. Our results show that an overall accuracy of 65 – 85% is achieved with a type II misclassification rate of approximately 20 – 35%. Performance of each of the methods is comparable to the others with minor variations.

DEDICATION

To my family and friends.

ACKNOWLEDGMENTS

This work is supported in part by grant CCR-0132673 from National Science Foundation. The findings and opinions in this thesis belong solely to the author, and are not necessarily those of the sponsor.

I thank Dr. Edward B. Allen for his expert suggestions, constant support, encouragement, and for guiding me throughout the thesis. I also thank Dr. Gene Boggess for his expert guidance on the neural network and genetic algorithms research. The genetic algorithms research was a result of a project done for Dr. Boggess's Genetic Algorithms class. I am also thankful to Dr. Susan Bridges for her expert guidance on machine learning, classifiers, decision trees, and C 4.5.

I am grateful to Barry Smith, Matthew Knepley, Satish Balay and the entire PETSc team at Argonne National Laboratory for helping me with the source code acquisition, installation and compilation issues. Datrix® is a registererd trademark of Bell Canada. I am grateful to Bell Canada for providing the license for the use of their product Datrix. I express my gratitude to BitMover Inc., for the use of the BitKeeper configuration management system. I am also thankful to Dr. Quinlan for allowing me the use of the C 4.5 tool. Neural network analysis could not have been possible without the Stuttgart Neural Network Simulator (SNNS) tool. I am grateful to folks at the Institute for Parallel and Distributed High Performance Systems (IPVR), University of Stuttgart for the SNNS tool.

Last but not the least, I thank all Empirical Software Engineering research group members for their valuable suggestions on the research.

TABLE OF CONTENTS

LIST OF TABLES

TABLE Page

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

Building high quality software is the goal of every software organization. Especially in the case of open-source software, quality is a big concern. Identifying the components which are fault-prone can help in achieving this goal. A new release of a software product generally contains bug fixes for the bugs found after the previous release, and enhancements to the functionality. Obviously, there is some relationship between a software product's source code and the post-release bugs found in that software product release. We try to model this relationship. The model can then be used to identify the fault-prone modules in the next release of the software system. Once these fault-prone modules are identified, developers can clean or rebuild these modules. Identifying fault-prone modules also allows the users to make informed decisions about the software system.

1.1 Hypothesis

The hypothesis of this work is as follows:

> By analyzing past releases of an open-source software product for high performance computing, one can identify high-risk faulty modules. This analysis can be performed by statistical and machine learning modeling techniques.

The statistical and machine learning techniques that will be employed in this research are logistic regression, neural networks, genetic algorithms (GAs), and decision trees.

1.2 Research Questions

The approach used in this research is an empirical case study of the past releases of Portable Extensible Toolkit for Scientific computation (PETSc). The research questions that are interesting for this case-study are as follows:

1. Does the case-study provide evidence for or against the hypothesis?
2. Which modeling approach, statistical or machine-learning is more accurate for the system being studied?
    (a) How does a neural network's accuracy compare with logistic regression?
    (b) How does a decision tree's accuracy compare with logistic regression?
    (c) How does a neural network's accuracy compare with decision tree?
    (d) How does the accuracy of GA trained logistic regression compare with traditional logistic regression?
    (e) How does the accuracy of a GA trained neural network compare with a traditional backpropagation neural network?

1.3 Relevance

This research will provide evidence for or against the proposed hypothesis. Supporting evidence will help the developers to clean or rebuild the faulty modules, thus producing higher quality software systems. It will also help the users of open-source software to analyze the benefits of a newer release of a product before they decide to use it. Evidence against the hypothesis will lead to the analysis of why the modeling process failed to give appropriate predictions of the fault-prone modules. The analysis can help understanding the problems in software modeling, which parameters and relationships are important for successful modeling, and possible solutions for overcoming them [1].

1.4 Overview

Chapter II presents an overview of related work in the field of software metrics and empirical software engineering. Chapter III presents the various tools that were used in the thesis. Chapter IV describes the methodology and the experimental design. Chapter V presents the case-study system on which the research was carried out. Chapter VI discusses the application of logistic regression modeling on the different data sets and presents its results. Chapter VII discusses the application of neural networks to the same problem and presents its results. Chapter VIII describes the application of genetic algorithms for learning logistic regression and neural network models to this problem. Chapter IX presents the application of decision trees to the same problem. Chapter X provides answers to the research questions and the hypothesis. Chapter XI evaluates the hypothesis, summarizes contributions and provides ideas for further research.

CHAPTER II

RELATED WORK

2.1 Empirical Studies in Software Engineering

This section presents the various issues in conducting empirical software engineering experiments and case-studies. In a recent study, Zelkowitz and Wallace [40] observed that the work regarding new technology or techniques in computer science is not carried out the ideal way. They propose a taxonomy of 12 different experimental approaches covering three broad areas — observational, historical, and controlled — in empirical software research. They discuss these approaches with their advantages and disadvantages. In their work, they surveyed 612 different articles related to software engineering and classified them according to their experimental approaches. Results show that the top two categories in these papers are assertions (biased experiments) and no-experiments. Researchers often fail to state their goals clearly, or show how they validate their hypothesis. The authors recommend that researchers characterize the experiments according to their data in order to enhance the researcher's ability to report on experiments. Following this approach, our experiment would fit in the case-study category.

There are various ethical issues in empirical software engineering experiments relating to informed consent, confidentiality, scientific value and beneficence [34]. Informed consent implies getting the consent of the subjects to perform the experiment without any coercion, with complete knowledge about the goals of the experiment and freedom to withdraw at any time without explanation. Confidentiality concerns maintaining anonymity of the subjects by avoiding including the identity details in the results. The scientific value of an experiment concerns the importance of the research topic to society and the validity of experimental results. If the use of a particular approach is debatable, then the validity of the experimental results is doubtful. Any results that could harm an individual subject or an organization should be not be published. This refers to beneficence. In essence, proper planning and research should be performed in advance, in order to abide by these ethical constraints to avoid difficult circumstances later. With our case-study approach, we do not identify individuals, thereby taking care of confidentiality. The research, by itself, has significant scientific value in the software industry.

Kitchenham et al. [18] propose a set of guidelines for conducting and reviewing empirical software-engineering results. These guidelines are based on the authors' experience of reviewing empirical software engineering research studies and the research guidelines in the medical field. These guidelines cover different aspects of the research including experimental context, experimental design, data collection, analysis, presentation of results, and interpretation of results. The aim of such guidelines is to ensure that other researchers can replicate the experiments discussed, that the conclusions are not an effect of applying incorrect statistical techniques, and that the conclusions do not follow as a result of manipulated data. Such guidelines can help improve the hitherto poor standard of empirical software-engineering research. These are not the definitive all-inclusive set of guidelines, but they provide a good place to begin. We will use these guidelines to allow other researchers to replicate our experiments and to maintain the validity of our results.

Jeffery and Votta's [13] special introduction to an IEEE Transaction issue on Software Engineering discusses the issues regarding the processes, design and structure of empirical studies. They focus on the goals of developing repeatable experiments, developing ways of providing cost-effective and timely results, and improving the empirical methods.

Generally, empirical software engineering research involves quantitative data. However, studies involving human behavior are best expressed using qualitative data. Seaman [31] discusses several qualitative methods for data collection and analysis and describes how they can be combined with quantitative data analysis. These methods can be used for analyzing the bug reports in our data.

2.2 Implementing Software Metrics

The software industry has recently begun to take advantage of software metrics. Pfleeger [27] describes the maturity of software measurements in an introduction to a special issue on software measurements in *IEEE Software*. These metrics programs have been found to be highly successful in reducing project costs and helping to produce quality software. The Capability Maturity Model developed by SEI also promotes the metrics program. CMM level 4 (Managed) and level 5 (Optimized) require use of process metrics to be measured and used for continuous process improvement [24]. This section describes some of the metrics programs implemented in the industry.

Grady [8], based on his experience at Hewlett-Packard, suggests that by carefully measuring and analyzing product and process metrics, it is possible to develop software products with fewer bugs and better quality and to avoid costly post-release updates. In most cases, only one set of metrics is collected and analyzed by different groups. However, the needs of higher management and the project manager differ in what they want to measure through the metrics program. To satisfy the needs of both groups, Grady suggests classifying the metrics program into four main areas: project estimation metrics, product metrics, process metrics, and rankings of the metrics among these. Examples of metrics in these four areas are tracking functionality, found-and-fixed defects for project estimation, cyclomatic complexity, fan-out squared for product metrics, tracking defect patterns for process improvement and comparing testing strategies for ranking best metrics.

Paulish and Carleton [23] discuss the results of applying the software process improvement methods in an industry setting at Siemens. Using a case-study approach, they measure the process, product, and environment at different sites working on different projects at periodic intervals. These metrics include defect detection distribution, defect rate, project productivity, schedule adherence, etc. By analyzing these metrics, improvements to the process are implemented. The gains of these improvement methods are calculated by measuring these same metrics after a year. They also present some of the lessons learned in initiating and implementing such a process improvement program and recommendations for implementing it.

Daskalantonakis [4], based on his experience at Motorola, stresses the importance of a prerequisite infrastructure that should be installed and working for a successful metrics program. Furthermore, metrics to be collected and tracked should be considered in various dimensions such as their utility, metric type, audience for metrics, metric user needs and the levels of metric application. He also gives a set of metrics implemented at Motorola, spanning a cross-section of the software development process ranging from requirements, design, schedule and effort estimation to defect density and post-release bugs. He explains how it helped to track and monitor progress. With respect to costs associated with the metrics program, Daskalantonakis argues that these are not too significant as compared to the benefits they offer. He stresses the fact that software metrics measurement is not the goal; the goal is continuous process improvement through measurement, analysis, and feedback.

Myrtveit and Stensurd [22] discuss a series of experiments carried out with industry practitioners to determine if they can perform better at the task of project-effort estimation with the aid of different tools, and if so which tool is the best. Their experiments extended earlier work and measured the performance of practitioners with a history aid, an analogy tool, and a multiple linear regression tool. They also measured the performance of these tools by themselves without the practitioners. Their results showed that while comparing only the tool's performance, the regression tool outperformed the analogy tool, which was not in accordance with the previous work. Their results also showed that when tested individually, neither of these tools was found to be better than the practitioners with a history aid. As the environment, nature of the projects, experimental design, and analysis methods change, empirical estimation results do not converge and hence these results can be difficult to generalize.

All of these examples were discussed with respect to an industrial setting which generally develops closed-source products. However, open-source products vary from their counterparts in many different ways. The following section gives an overview of open-source software.

2.3 Open-Source Software

Wu and Lin [38] explain that free software does not have any relation to the cost of the software. Free emphasizes the freedom to run the program, modify it to suit your needs, and redistribute the modified copies so that society can benefit from your modifications. One of the major differences of many open-source software projects from the closed-source software projects is that anyone in the world who is capable and wishes to join the development effort can join in. This is not the case with closed-source software. Supporters of open-source software claim that this is the basic reason that open-source software is more stable. Some of the development issues associated with open-source software are the use of version control systems and various licensing models.

Lawrie and Gacek [20] discuss the various issues surrounding the dependability of open-source software. Dependability is considered to be a broad term covering many features including reliability, security and availability. A system can be said to be dependable if it meets certain assurance criteria to demonstrate the above qualities. In general, open-source software products are not necessarily more dependable than the non-open-source software products. However, research suggests that, despite their ad-hoc nature and chaotic characteristics, open-source software development is highly organized in many cases. Since open-source software projects are initiated based on personal interest, they are more concentrated towards system software products, whereas non open-source software are available in a variety of domains. The open-source software development process differs from its counterpart in many factors such as tools/methods used, focus of these tools, and the constraints on these projects. So, a better way to analyze dependability would be on a project-by-project basis, rather than as open-source or non-open-source.

The introductory article by Feller et al. [6] in *ACM Software Engineering Notes* discusses papers on open-source software development with regards to meeting the challenges and surviving the success seen so far by the open-source software community. The papers in this special issue discuss (1) quality, maintainability, portability, replicability of open-source software products, (2) stability and sustainability of developer and user communities, and (3) viability and profitability of open-source business models.

Paulson, Succi and Eberlein [25] compare the evolutionary and static characteristics of open-source and closed-source software. This study is based on examination of six software systems, three from the open-source community and three from a closed-source community. Their results show that open-source software projects foster more creativity and, generally the software has fewer defects because the bugs are found and fixed more rapidly than in closed-source software. This does not necessarily imply that open-source software projects are more modular, foster faster system growth, or are simpler than the closed-source projects.

Given these differences, it is worth pointing out that even though our research methodology is the same for both open-source and closed-source projects, the results from this case study of an open-source system may not apply exactly to a closed-source software product.

2.4 Predicting Quality Metrics

Generally, an industry implemented metrics program collects the metrics for every release and makes improvements by analyzing the metrics in between the releases. A better approach can be to predict the value of software metrics for the release and make informed decisions about improving its quality before the software product is released. This has been an interesting topic for the software-engineering community for quite a while. This section describes some of the successful attempts in this area.

Khoshgoftaar et al. [16] discuss the use of software development databases to predict high risk modules. The hypothesis of their work is that by applying data mining techniques to the software development databases from current and past releases, a considerable percentage of high risk modules in the current release can be identified early in the development process. The idea is to build a model of the software using the past release data. The data mining approach used for building this model is classification and regression trees (CART). This model is then used on the current release data to decide whether a module is faulty or not. The technique correctly classified 63% of the fault prone modules and 81% of the not-fault prone modules. This can be considered a good estimate, given the huge costs of rework after the software release.

Von Mayrhauser et al. [36] discuss an approach for identifying fault prone components of a software system by studying the past releases and defect reports. Their hypothesis is that by analyzing the defect reports of the past releases of a software system, the fault architecture of the system can be derived. The idea behind this is that if two components are to be modified in order to fix a defect, then these two components share a fault relationship. The strength of this relationship is based on the number of times these two components appear together in defect reports. Such an architecture can reveal the most fault-prone system components over the past releases. If a component repeatedly appears in the fault architectures for several consecutive releases, then it can be identified as an architectural fault. The paper demonstrates this approach of deriving a fault architecture on a large system of C, C++ and microcode spread through about 130 components measuring approximately 800 KLOC. Performing a similar kind of analysis on our system would enable us to compare our results with this approach. However, given the scope of this comparison, we defer this work until later.

Khoshgoftaar and Allen [15] present a new type of model for software quality, named module-order models, and discuss their evaluation and use. This model uses an underlying statistical multiple regression model to predict the dependent quality variable for the various modules. These modules are then ranked according to the extent to which they are fault-prone. Analysis based on two case-studies shows that this model is fairly accurate in identifying the top fraction of most fault-prone modules. Moreover, as the results of the underlying multiple regression model are used to get the module order ordering, the cut-off for identifying fault-prone modules need not be fixed before the model is built. This allows the managers to concentrate the quality management effort on only as many modules as is permitted by the resources.

Neural networks have also been used in this classification task. Khoshgoftaar et al. [17] describe a neural network based software quality model for predicting the modules as high risk or low risk, early in the development cycle. The hypothesis of their work is that the neural network classification model performs better than the discriminant classification model for classifying the modules as high risk or low risk. These conclusions are based on the case-study involving data from a large military telecommunications system. This paper also discusses a few techniques for tweaking the performance of their neural network. The misclassification performance in terms of type I and type II errors was about 13% and 7% respectively which can be considered as excellent.

Rapur [30] presents an implementation of a statistical software quality modeling approach that uses the amount of code-churn as a predictor variable. Multiple linear regression and module-order module analysis was carried out on the data. The experiment was based on data collected for three past releases of the MPICH system. A multiple linear regression model was built on the training data, (data between the changes in release 1 and 2) and evaluated on the test data (data between the changes in release 2 and 3). The analysis used the number of lines of change between the different releases as the predictor variable. This change could be due to many different reasons other than bug fixes. Also, since the number of large-change modules is only a small fraction of total number of modules, the experimental accuracy was limited. The analysis, however, did provide a formal approach for conducting such an experiment.

Selby and Porter [32] demonstrate the application of decision trees to the problem of identifying components or modules which had high fault rates. The authors use the ID3 decision tree algorithm to build their trees from a set of 74 input attributes. Their results show that they had an average accuracy of about 80% with low type I and type II misclassification rates. The decision tree classification process makes it easy to understand the classification task and figure out the main factors that result in a high number of faults in a module. The authors' data for this experiment was based on sixteen moderate to large size software systems from NASA developed for ground support for unmanned spacecraft control. The decision tree was used to model the development process for these projects. Since we are concentrating on multiple releases of the same software product, we are trying to model the traits of a particular software product.

2.5 Modeling Techniques

This section discusses principal components analysis, and various modeling techniques that will be used in this research. These include logistic regression, neural networks, genetic algorithms, and decision trees.

Principal Components Analysis Principal components analysis (PCA) as defined by Dunteman [5] is a "statistical technique that linearly transforms an original set of variables into a substantially smaller set of uncorrelated variables that represents most of the information in the original set of variables."

A typical metric analyzer would measure a large set of metrics on a set of modules. Some of these metrics would be correlated with each other. However, modeling processes require a small subset of these metrics that are uncorrelated. Hence, principal components analysis is carried out. A step by step description of PCA, adapted from Khoshgoftaar and Allen [15] is shown below.

1. Let us assume that our metric analyzer collected m different metrics over n modules. The result is an $n \times m$ matrix, denoted by $\mathbf{X}$. We then standardize the matrix $\mathbf{X}$ as follows to obtain a standardized measurement matrix $\mathbf{Z}$, where an element is

$$z_{ij} = \frac{x_{ij} - \mu_j}{\sigma_j} \tag{2.1}$$

   where x_{ij} represents a measurement value of metric j for module i, μ_j represents the mean of metric j and σ_j stands for the standard deviation of metric j.

2. Principal components are linear combinations of standardized random variables such as $Z_1, Z_2, ..., Z_m$. These principal components represent the same data in a new coordinate system where variability is maximized in each direction and the principal components are uncorrelated.

3. Next, we calculate the covariance matrix $\mathbf{\Sigma}$ of $\mathbf{Z}$. The covariance of a pair of components, X_i and X_j is defined as follows [33].

$$\mathit{Cov}(X_i, X_j) = E[(X_i - \mu_i)(X_j - \mu_j)] \tag{2.2}$$

4. If the covariance matrix $\mathbf{\Sigma}$ is a real symmetric matrix with distinct roots then we can calculate the eigenvalues λ_j and eigenvectors $\mathbf{e}_j$ of $\mathbf{Z}$.

5. Each eigenvalue λ_j is the variance of the corresponding principal component. Since eigenvalues form a non-increasing series $\lambda_1 \geq \lambda_2 ... \geq \lambda_m$, we can reduce the dimensionality by considering only first p components according to some stopping rule, where $p \ll m$. For example, choosing p such that $\Sigma_{j=1}^{p} \lambda_j / m \geq 0.9$ captures 90% of total variance in the principal components.

6. Next, we calculate the standardized transformation matrix $\mathbf{T}$ of size $m \times p$. The columns $\mathbf{t}_j$ are calculated as follows.

$$\mathbf{t}_j = \frac{\mathbf{e}_j}{\sqrt{\lambda_j}} \tag{2.3}$$

7. Let D_j represent a principal component variable (in our case, *FACTOR*) and $\mathbf{D}$ represent the $n \times p$ matrix with D_j values for $j = 1, ..., p$. These are calculated as follows.

$$\begin{aligned} D_j &= \mathbf{Z}\mathbf{t}_j \\ \mathbf{D} &= \mathbf{Z}\mathbf{T} \end{aligned} \tag{2.4}$$

We calculate the transformation matrix $\mathbf{T}$ based on our training set. To transform the test set, we then use this transformation matrix $\mathbf{T}$ generated during PCA.

Logistic Regression: The logistic regression model is used for classifying modules into one of two classes, fault-prone or not-fault-prone. The general form of this model is given as

$$\log\left(\frac{p}{1-p}\right) = a_0 + a_1 x_1 + a_2 x_2 + a_3 x_3 + ... + a_n x_n \tag{2.5}$$

where p is the probability of a module belonging to the fault-prone class, $\mathbf{x}$ is the input vector for a module and $\mathbf{a}$ is the coefficient vector [19]. If this probability is greater than a certain threshold then the module is classified as being in the fault-prone class, else it is in the non fault-prone class. This follows from the logistic regression function discussed by Khoshgoftaar and Allen [14].

$$Class(\mathbf{x_i}) = \begin{cases} fault - prone & if\ \frac{\hat{p}}{1-\hat{p}} \geq \left(\frac{C_I}{C_{II}}\right)\left(\frac{\pi_{nfp}}{\pi_{fp}}\right) \\ not - fault - prone & otherwise \end{cases}$$

where $\mathbf{x_i}$ is a module, $\hat{p}$ is the estimated probability of a module being in a fault-prone class, π_{nfp}, π_{fp} are the prior probabilities of membership in the not-fault-prone and the fault-prone classes respectively in the training set, and C_I, C_{II} are the costs of misclassification.

The modeling process involves estimating the coefficient vector $\mathbf{a}$ from the training set, given the class of each module as fault-prone or not-fault-prone and the input vector $\mathbf{x}$. SAS provides a procedure for doing this type of modeling, which uses a maximum likelihood estimation method to calculate the coefficients [12]. Stepwise logistic regression involves adding one variable at a time to the model, starting with the intercept a_0. After a significant variable is added to the model based on a Chi-Square test, the least significant variable in the model is removed.

Neural Networks: Neural networks are one of the premier machine learning tools for classification tasks. There are different kinds of neural networks. The simplest form and the one that is relevant to our work is the backpropagation neural network. A typical three-layer backpropagation network is shown in Figure 2.1. The network consists of three layers: an input layer, a hidden layer, and an output layer. Because the nodes at the input layer do not perform any processing, this network is generally called a two-layer network. It is a feed-forward, fully connected network, which means that all nodes in one layer connect to all the nodes in the next layer. These nodes are connected by links. Each of these links has a weight associated with it. At every processing node in the hidden layer and the output layer there is a combination of a summation and a sigmoid unit. The summation unit sums the product of inputs and the weights of each link connected to that particular node. This value is then fed to the sigmoid unit where the input is transformed to a value between 0 and 1 using a sigmoid function. This is illustrated in Figure 2.2 and Figure 2.3. This illustrates the forward pass through the network. The following equations demonstrate the forward pass. S stands for the stimulation caused at the input of each node, W for weight of the link between nodes i and h and A stands for the activation output at each node.

$$S_h = \sum_{i=0}^{N_i} A_i W_{i,h} \tag{2.6}$$

$$A_h = \frac{1}{1 + e^{-S_h}} \tag{2.7}$$

Next we discuss the backward pass. In this pass, the error at the output layer is calculated as the difference between the target value and the output value. This error is then propagated back to the hidden layer. The weights of the network are then adjusted, so that

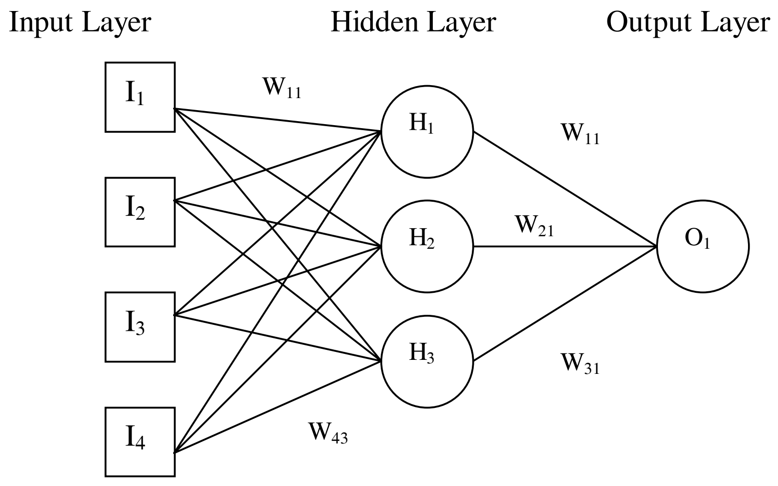

Figure 2.1 A Typical Two-Layer Backpropagation Neural Network

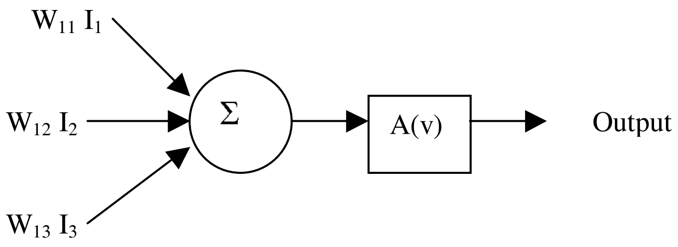

Figure 2.2 Processing at each Node

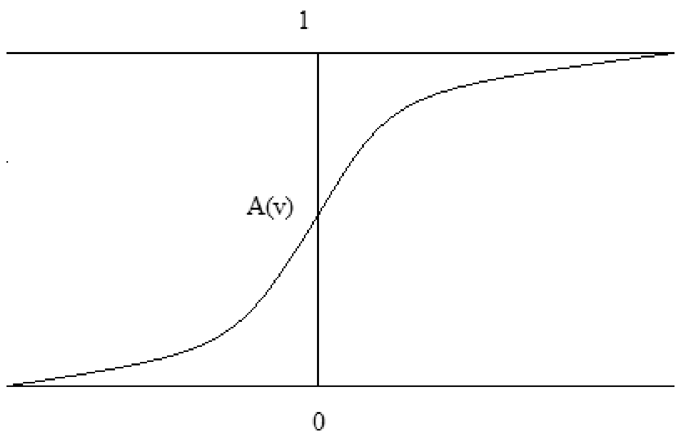

Figure 2.3 Sigmoid Curve

if that example is presented again the error of the network will be reduced. The equations for the updating the weights of the network are discussed by Haykin [9]. This series of a forward pass and a backward pass is repeated for all the examples in the training set. This is known as one epoch. Training a neural network can take up to several thousand epochs depending on the nature of the problem [9]. Once the network is trained, it is ready to use. The unclassified input samples are presented to the network and the classification score is obtained at the output layer. Since there are only two classifications involved in this example, the output layer contains a single node. If the classification score is greater than 0.5 then the input sample belongs to say class A, else it belongs to the class B. All the information in a neural network is stored in the form of the weights of the connections between the nodes. So, it is not trivial to abstract that information from a network.

Neural networks generally perform better when there are equal proportions of data in the training set for each class being identified. Since in our case-study, where only a small number of the modules are fault-prone, we will investigate techniques to counter problems arising from unequal distributions of data.

Genetic Algorithms: A genetic algorithm (GA) is a technique initially developed by Holland et al. for solving optimization or classification problems. Goldberg [7] applied genetic algorithms to the gas pipeline industry demonstrating their usefulness in solving real world problems. This method uses an analogy from biology and nature where chromosomes contain the information required to describe an individual. This information is encoded in the form of genes on the chromosome. The main idea behind a GA is the principle of natural selection, where the best fit members in a population tend to survive and produce offsprings, resulting in the evolution of better individuals.

A typical GA works as follows. A random population of individuals is created for the first generation. These individuals are then evaluated and ranked according to a fitness function. Higher fit individuals from one generation get a chance to produce offspring for the next generation. As the population evolves, better and better individuals are generated by crossing over or mutating the parent chromosomes. This process is repeated until a satisfactory solution is found or a fixed number of generations have passed. During the entire process, the best fit individual found so far, is saved. This best fit member is then used on the test set to get the results. The theory behind how a GA works is explained using the schemata theorem [7].

Our problem is to identify the fault-prone modules in the system. Typically, the ratio of fault-prone modules to not-fault-prone modules in a mature software system is approximately 1:9. Hence, the standard classification methods, which require equal class distributions do not perform as well on these data sets. A genetic algorithm allows us to express these conditions explicitly in the fitness function and guide the evolutionary process towards identifying better classifiers. Hence, genetic algorithms are more likely to be successful on this type of problem.

There are different methods to select the parents in one generation for producing the offspring in the next generation. The most commonly used is Roulette wheel selection.

The two most commonly used operators for producing the offspring are crossover and mutation operators. Crossover operators make cuts in the two parent chromosomes, and swap the material between the cuts to create two offspring chromosomes. The mutation operator involves changing the gene value on a chromosome. For real valued genes, as in our case, Gaussian mutation is preferred over random mutation [7].

Decision Trees: A decision tree is a machine-learning approach which is generally used for classification tasks. One of the advantages of the decision-tree approach is that it is very easy to abstract the information in the resulting model and understand how the classification decision was made. This can allow one to conclude facts like such and such parameters cause a module to be faulty. Figure 2.4 shows a simple decision tree. The decision tree is built by starting with a root node and a set of independent parameters. A parameter is selected at each level of the tree, and the input data is divided on that parameter. The parameter chosen is generally the one which would result in the smallest tree. The information gain heuristic is one of the methods employed to determine this parameter [29]. This process is continued until all the samples at a node are of one classification or they are too small in number. At this stage, the classification of the samples is assigned to that leaf node.

Once, the tree is built, classifying an unseen example involves a simple pass through the tree. The leaf which the input sample reaches is the classification for that example.

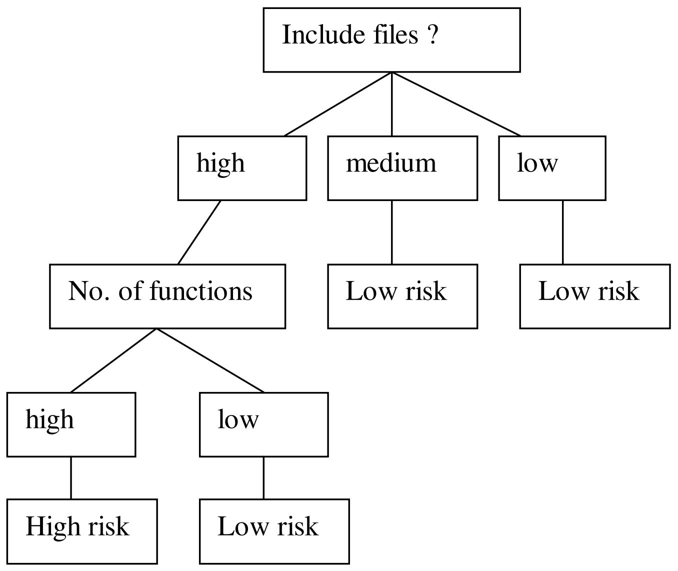

Figure 2.4 A Simple Decision Tree

C4.5 provides built-in support for pruning decision trees. We will investigate the effects of C4.5 pruning and study other techniques for pruning.

CHAPTER III

TOOLS

A variety of tools were used in this research. This chapter presents the pointers to the guides and descriptions about the tools used. Table 3.1 lists all the tools used in the thesis.

Table 3.1 Tools

Tools	Description	Developed by
Bitkeeper	Configuration Management System	BitMover Inc.
Datrix	Metrics Collection Tool for C, C++ code	Bell Canada
Perl Scripts	Processing, Collecting metrics	This Research
SAS	Statistical Analysis Tool	SAS Institute
Backprop	Backpropagation Neural Network	This Research
SNNS	Neural Network Simulator	IPVR
Custom GA	Genetic Algorithm Programs	This Research
C4.5	Decision Tree Tool	Quinlan

3.1 BitKeeper

BitKeeper [11] is a configuration management system used by the PETSc developers. It is similar to other configuration management systems. This configuration management system was used to obtain the update logs for each file in the system's data repository.

BitKeeper has a feature which allows one to get all the update logs from the time the system was being used to the current date. We used this feature to get our update log.

3.2 Datrix

Datrix$^{®}$ [2] is used for measuring the source code metrics for the system being studied. The following components of the Datrix tool set were used in this study.

1. `dxprepc`: This component preprocesses the macros and preprocessor directives like `#include`, `#ifdef`, `#ifndef` etc. For this study, it was found that preprocessed code with `dxprepc` resulted in errors in the next stage of measurement. Hence, the g++ preprocessor was used.
2. `dxmetc`: This component actually collects the metrics from the preprocessed source code. We measure file-level and routine-level metrics using `dxmetc`.

3.3 Perl Scripts

The metrics collected by Datrix are in a nested block-text format. For our analysis, we want the metrics to be in row-columns format. We use the scripts `sfmet.pl` and `srmet.pl` to switch to the row-columns format.

The file-level metrics are collected one row per source file and the routine level metrics are collected one row per routine. So, if a file had three routines, it will have one row of file-level metrics and three rows of routine level metrics. However, we want to have one row of metrics for each file. We merge the routine level metrics into one row, depending on the definition of each routine-level metric. For example, for a routine-level metric such as *RtnLnsNbr*, we merge the individual routine-level metrics by simply adding them. For a metric like *RtnScpNstLvlMax*, we merge the individual metrics by taking their maximum. Similarly, for metrics like *RtnCplCtlAvg*, we take the average by summing up the numerators' metric values and dividing by the sum of denominators' metric values. All of this aggregation is performed by the `aggrmet.pl` script. After the aggregation, the routine and file-level metrics are combined using the script `mergerfmet.pl`.

The output data is collected by the script `colloutmetrics.pl` from the Bitkeeper configuration management system. This script mines the entire update log of the system. It searches for keywords such as "bugs", "defects", "problem", etc. in the comments of each of the update logs. The outputs produced are bug counts for each file, release by release. The input and output metrics are then merged using the script `mergerfoutmet.pl`.

A set of utility scripts were also developed for finding the size of the system, and the percentage of bugs in `C` files, header files, etc. All the scripts are developed in Perl. Wall, Christiansen and Orwant [37] is a good resource for learning to build Perl scripts.

3.4 SAS

SAS [12] is statistical analysis package that was used for building various models in this study. Logistic regression models are built using the `proc logistic` procedure. A variety of other procedures in SAS like `proc means`, `proc univariate`, `proc score`, `proc freq`, etc. were used in the study.

3.5 Backprop

Standard backpropagation neural networks were implemented to carry out the neural network research. This implementation used the concepts discussed in Haykin [9] and also included features such as momentum, early stopping, committee machines, etc. A variation also included the use of a no-hidden-layer neural network and no-hidden-layer committee machines.

3.6 SNNS

SNNS is a very sophisticated neural network tool developed by the Institute for Parallel and Distributed High Performance Systems (IPVR) at the University of Stuttgart, Germany. Along with the basic neural network features, this tool also provides visualization of the error and the weights of the network. It also has a handy feature for conducting multiple tests at one time.

3.7 Custom GA

Genetic Algorithms were implemented as a part of the project for Dr. Boggess's Genetic Algorithms class [28]. Two genetic algorithms were developed. One GA program was used to learn the parameters for logistic regression and other to learn the weights of a neural network.

3.8 C4.5

C4.5 is a decision tree tool developed by Quinlan [29]. The main programs for this tool are `c4.5`, `c4.5rules`, `consult`, `consultr`, and the script `xval.sh`. The `c4.5` program builds a decision tree from the given data and `c4.5rules` builds the production rules from the `c4.5` generated tree. The programs `consult` and `consultr` can then be used to classify the unseen examples from the decision tree and production rules respectively. The script `xval.sh` (run using `csh`) is an automated script for performing cross validation. It builds the decision tree model, evaluates the test data and stores the result in the `filstem.tres` and `filstem.rres` files.

The input file names for the C4.5 system have a format of `filestem.ext`. `Filestem` denotes the current problem you are working on, say `petsc211` and `ext` denotes the type of the input file. Different values for `ext` are `names`, `data`, `tree`, `unpruned`, etc. The `names` file contains information about the data, classifications for the data, input parameters, their types — discrete, continuous, etc. The data file contains the data for building the tree with each line describing one input sample. The `test` file contains the testing data.

CHAPTER IV

METHODOLOGY

A case study methodology is followed for this research. This chapter explains the procedure for carrying out the experiments and the set-up for each experiment for our system. This methodology is also applicable for conducting a similar study with any other system.

4.1 Procedure

The procedure for our carrying out the experiments is listed in Table 4.1.

Table 4.1 Procedure

	Tasks
1	Collect the system's source code data using Datrix for each release.
2	Collect the system's bugs data for each release from configuration logs.
3	Perform principal components analysis on each data set.
4	Form training and validation data sets.
5	For every release
	5a Build logistic regression models.
	5b Build neural network models.
	5c Build GA trained logistic regression and neural network models.
	5d Build decision tree models.
	5e Analyze and compare the performance of the above models.

1. Collect the system's source code data using Datrix for each release: The data is collected from the most recent five releases of the system. Datrix is used for collecting this metric data.

2. Collect the system's bugs data for each release from configuration logs: The number of bugs found in each module for each release is collected from the configuration management system's logs.

3. Perform principal components analysis on each data set: Correlated features are undesirable for a classification task. Hence, principal components analysis is performed on each training data set to obtain uncorrelated features for the input data.

4. Form training and validation data sets: Since, the data collected is for five releases, we form more than a single set for training and validation. For example, we train a model on release 1 and test on release 2. The next model is trained on release 2, test on release 3 and so on.

5. For every release, we

   (a) Build logistic regression models: A statistical analysis package SAS [12] was used for building statistical classification models.

   (b) Build neural network models: Backpropogation neural network classification models were explored using a custom NN and a freely available SNNS [35] software.

   (c) Build GA trained logistic regression and neural network model: Logistic regression and neural network models trained using genetic algorithms were also built.

   (d) Build decision tree models: Quinlan's [29] C4.5 was used for building a decision tree based classification model.

   (e) Analyze and compare models: This involves the analysis of each model's performance and comparison of their performance – how similar or different are the results, what factors affect the model's performance, which model is better suited to this kind of a task, which factors make the modules fault-prone etc.

4.2 Experimental Design

Our aim is to compare the accuracy of each of the proposed methods, logistic regression, neural networks, decision trees, and GA trained logistic regression and neural network models from a software engineer's point of view. To compare these methods, there are a few terms that need to be explained.

Typically, a software system would have fault-prone to not-fault-prone modules ratio of 1:9. The two classes involved in the classification are not-fault-prone and fault-prone classes. Using the terminology from statistics, classifying a fault-prone module as not-fault-prone is called a type II misclassification error and classifying a not-fault-prone module as fault-prone is called type I misclassification error. The cost of misclassification for each class is different. C_{II}/C_I, known as the cost-ratio, is a subjective choice and measures how much a misclassification of type II costs compared to a type I misclassification. This implies how much classifying a fault-prone module as a not-fault-prone module (type II misclassification) would cost us later as compared to classifying a not-fault-prone module as a fault-prone module (type I misclassification). Clearly, the misclassification cost for type II is much higher than for type I. We adjust our cost ratio such that the type I and type II error rates are approximately equal. We do this cost ratio adjustment on our training set.

To compare two models, we then consider the type I and type II errors. We calculate these type I and type II errors using the confusion matrix obtained from the classifier.

$$\textit{Confusion Matrix} = \begin{pmatrix} C_1 & M_1 \\ M_2 & C_2 \end{pmatrix}$$

C_1 defines the number of modules that were correctly classified as being not-fault-prone. C_2 defines the number of modules that were correctly classified as being fault-prone. M_1 represents the number of misclassifications of type I, i.e. the number of modules that were not-fault-prone but were classified as fault-prone. M_2 represents the number of misclassifications of type II, i.e. the number of modules that were fault-prone but were classified as not-fault-prone. The Type I and Type II error rates are then calculated as shown below.

$$\textit{TypeI} = \frac{M_1}{M_1 + C_1} \tag{4.1}$$

$$\textit{TypeII} = \frac{M_2}{M_2 + C_2} \tag{4.2}$$

Knowing that type II errors are more important, a lower type II error rate is preferred. Also, because of the 1:9 distribution of fault-prone to not-fault-prone modules, higher type I error rate means that a larger number of modules are misclassified. Comparison of two models is a subjective choice combining these two issues.

Table 4.2 Comparing Different Models

	$TypeI$ (%)	$TypeII$ (%)
Model 1	20	45
Model 2	35	20
Model 3	30	25

For example, in Table 4.2, one can definitely say model 2 and model 3 are better than model 1. However, the comparison between models 2 and 3 is subjective. The decrease in one of the errors is associated with an increase in another.

CHAPTER V

CASE STUDY

5.1 Introduction

This research is based on the study of an open-source software system known as the Portable Extensible Toolkit for Scientific computation (PETSc) developed by the Argonne National Laboratory. This system was written in C and is comprised of approximately two million lines of code. We include five releases of the PETSc software in our study as shown in Table 5.1.

Table 5.1 PETSc Releases under Study

Release No.	Release Date
2.1.1	December 19, 2001
2.1.2	April 22, 2002
2.1.3	May 31, 2002
2.1.5	Jan 27, 2003
2.1.6	August 5, 2003

Release 2.1.4 was a private release and hence the source code was not available. We refer to these releases as 2.1.1, 2.1.2, 2.1.3, 2.1.5, 2.1.6 or releases 1, 2, 3, 5 and 6 interchangeably. Both actually refer to the same data sets.

5.2 Data Collection

In this research, we build software quality models for the PETSc software. As in any model, this involves dependent and independent parameters. The input data in our case were metrics measured from the source code of the above releases. The output parameters for this model were the number of bugs that were found in a particular module. A module was conceived of as a `*.c` file.

For measuring the input metrics, we used a metric analyzer tool, Datrix. The header files had to be preprocessed before measuring the metrics. The data collection process is shown in Figure 5.1.

Unfortunately, the PETSc developers did not directly keep track of any bugs data. However, they used the BitKeeper configuration management system. We used the configuration management entries to find the number of the bugs in a module during a particular release. Since the input data was collected only for a `*.c` file, only the bugs found in the configuration management reports pertaining to a `*.c` file were collected.

One of the issues was how to combine the input and the output parameters. We assumed that the number of bugs found after the release date of a particular release, say v1 until the next release date, v2, counts as the number of bugs for release v1. We matched our input and output data sets as shown in table Table 5.2.

One of the interesting things was to find out whether there were any modules that did not change since their first creation. By mining the configuration management system's logs, we found that there were around 150 modules that did not change from the first

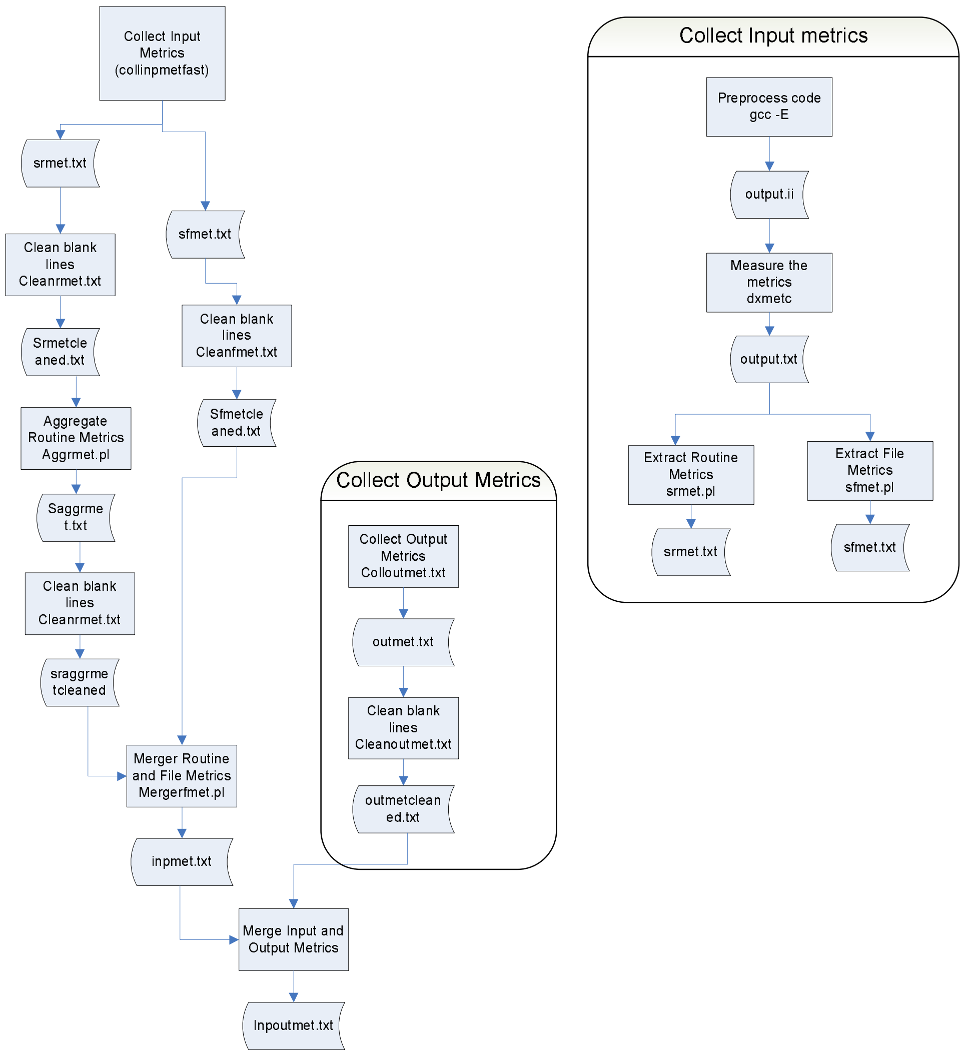
Collect Input Metrics (collinpmetfast)
srmet.txt
sfmet.txt
Clean blank lines Cleanrmet.txt
Srmetcleaned.txt
Clean blank lines Cleanfmet.txt
Aggregate Routine Metrics Aggrmet.pl
Sfmetcleaned.txt
Saggrmet.txt
Clean blank lines Cleanrmet.txt
sraggrmetcleaned
Merger Routine and File Metrics Mergerfmet.pl
inpmet.txt
Merge Input and Output Metrics
Inpoutmet.txt
Collect Output Metrics
Collect Output Metrics Colloutmet.txt
outmet.txt
Clean blank lines Cleanoutmet.txt
outmetcleaned.txt
Collect Input metrics
Preprocess code gcc -E
output.ii
Measure the metrics dxmetc
output.txt
Extract Routine Metrics srmet.pl
Extract File Metrics sfmet.pl
srmet.txt
sfmet.txt

Figure 5.1 Data Collection Process and Scripts Used

Table 5.2 Matching Input and Output Data

Release No.	`C` Bugs	`*.h` bugs	Input Release
Before Release 2.1.1	642	21%	
Between 2.1.1 and 2.1.2	131	29%	2.1.1
Between 2.1.2 and 2.1.3	107	22%	2.1.2
Between 2.1.3 and 2.1.5	42	8%	2.1.3
Between 2.1.5 and 2.1.6	68	15%	2.1.5
After 2.1.6	2285	24%	2.1.6

release of our study until the last release. Almost all of these modules were `*.c` files and very few were `*.h` files. We concluded that all of these modules were stable and bug-free over this period.

Our analysis only covers `*.c` files. However, it was interesting to discover what percentage of bugs were found in `*.h` files as compared to bugs found in `*.c` files. These data are shown in Table 5.2. The header file bugs accounted for roughly 20% of the bugs; 80% bugs were present in `*.c` files.

Once the data for a release was ready, summary statistics for the data set were calculated using SAS. Summary statistics provides number of entries, maximum, minimum, mean, and standard deviation for each variable. This helped identify any obvious errors in data collection and was a primary check for data validation. This procedure was repeated for each of the releases. The summary statistics revealed that the maximum value for the metric measuring the number of declarative statements in a module (*RtnStmDecNbr*) was found to be very high in releases 2.1.3 and 2.1.6. After performing a univariate analysis, it was found that the next highest values for this metric were considerably less giving the impression that this file performed some complex operation, that used a large number of variables. This may be the reason that this module had high values of *RtnStmDecNbr*.

For each of the releases, approximately 100 files had compilation errors which included macro and syntax errors. These files were about 10% of the data. On communicating with the PETSc team, we found that these files were not part of the core functionality and hence could be ignored in this study. We selected a Solaris 64 configuration of the PETSc software. For release 2.1.2 and 2.1.3, Datrix gave a few errors on `extern` definitions of some functions in a few header files. After slight modification, these errors were resolved. One of the examples of a modification is shown in Figure 5.2.

```
EXTERN int PetscViewerStringSPrintf(PetscViewer, char *, …) PETSC_PRINTF_FORMAT_CHECK(2,3);
EXTERN int PetscViewerStringSPrintf(PetscViewer, char *, …); // PETSC_PRINTF_FORMAT_CHECK(2,3);
```

Figure 5.2 Example Modification for Data Collection

5.3 PETSc Data

In this section, we discuss the different releases of the system. For each module, Datrix collected 39 software metrics, which form the inputs to the model. The output metric (number of bugs) was a discrete one. For our purposes of building a classification model, we needed to determine a threshold for the number of bugs, so that each module could be classified as being fault-prone or not-fault-prone.

We studied the bugs distribution for this purpose. Figure 5.3 shows the cumulative bugs added on the modules, sorted in decreasing order of number of bugs. Figure 5.4 shows the bugs distribution with respect the individual modules.

Since the proportion of modules having one or more bugs is so low (almost 10%), we kept the presence of any bugs as an indication of a fault-prone module. So a module is called as fault-prone if it had one or more bugs. A module is called as not-fault-prone if it

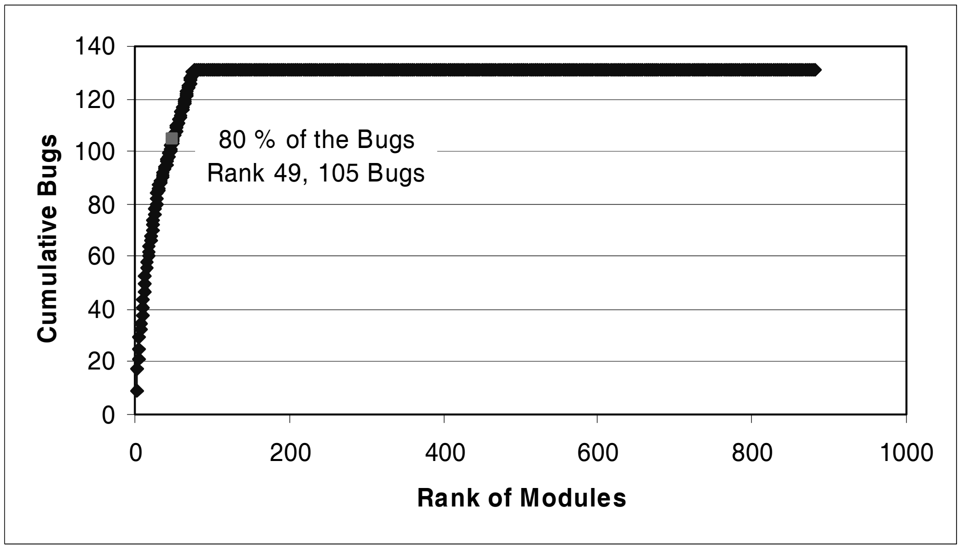

Figure 5.3 Cumulative Bugs for PETSc Source 2.1.1

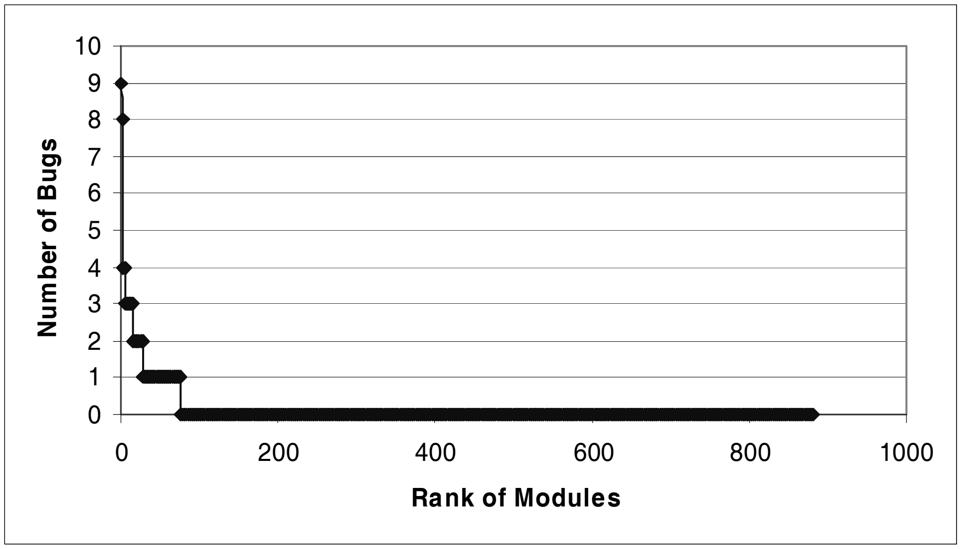

Figure 5.4 Bugs Distribution for PETSc Source 2.1.1

had no bugs. Using this classification, release 2.1.1 had 807 not-fault-prone modules and 75 fault-prone modules. Table 5.3 shows the fault-prone modules distribution over all the releases.

Table 5.3 Fault-Prone Distribution of all Releases

	fault-prone	not-fault-prone	total
Release 2.1.1	807	75	882
Release 2.1.2	809	95	904
Release 2.1.3	877	25	902
Release 2.1.5	860	52	912
Release 2.1.6	50	867	917

Releases 2.1.2, 2.1.3, and 2.1.5 had similar curves for cumulative bugs and bugs distribution. These are included in Appendix A.

However release 2.1.6 has a strikingly different distribution as compared to the other releases. Figure 5.5 and Figure 5.6 show the cumulative bugs and the bugs distribution for this release. This distribution is the reverse of that seen in other releases. Also, the number of bugs per module is also higher than the previous releases.

5.4 Building Models

Since we have five different releases, our strategy was to build a series of models using each release as the training set and testing with the rest of releases. Table 5.4 shows the formation of training sets and test sets for logistic regression and decision tree modeling.

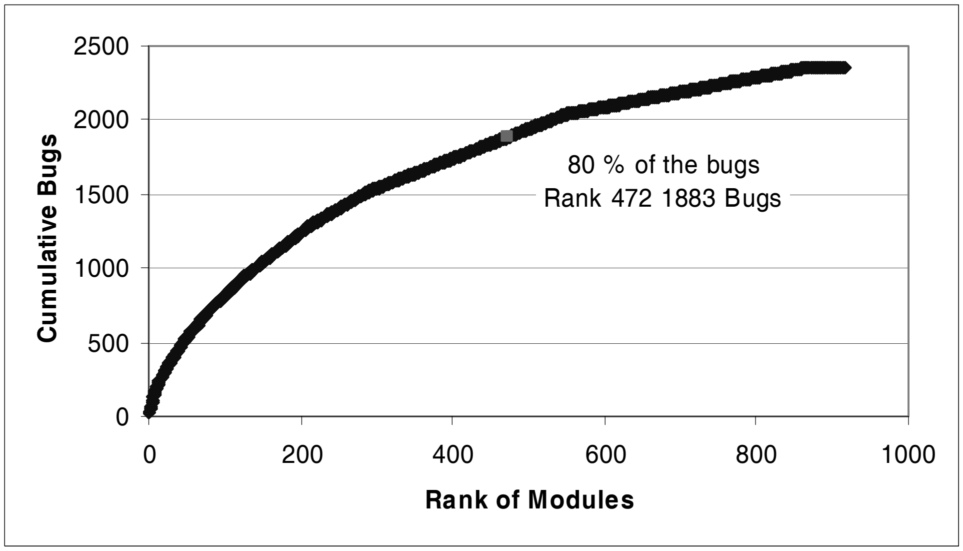

Figure 5.5 Cumulative Bugs for PETSc Source 2.1.6

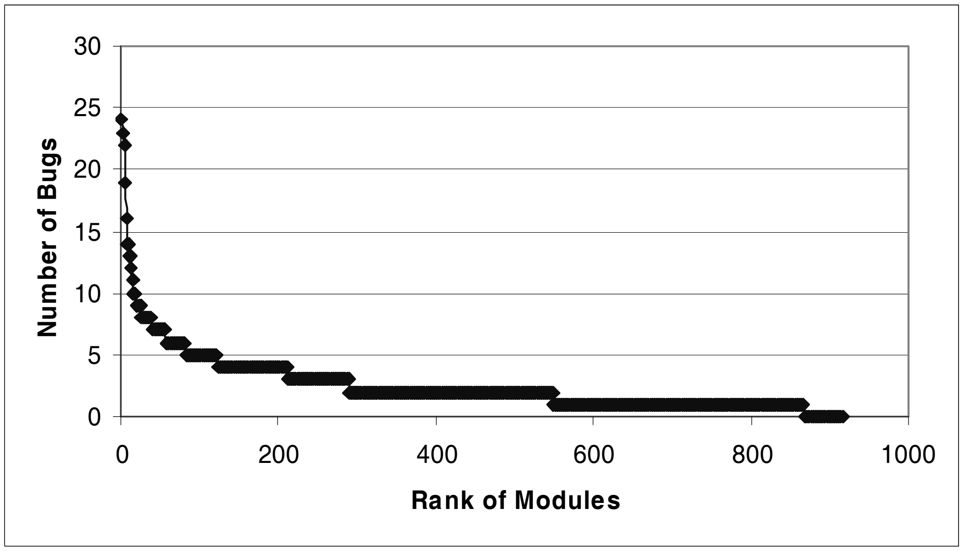

Figure 5.6 Bugs Distribution for PETSc Source 2.1.6

Table 5.4 Forming Data Sets for Logistic Regression and Decision Tree Modeling

Model	Training Set	Test Set(s)
R1	Release 2.1.1	Release 2.1.2, 2.1.3, 2.1.5, 2.1.6
R2	Release 2.1.2	Release 2.1.3, 2.1.5, 2.1.6
R3	Release 2.1.3	Release 2.1.5, 2.1.6
R5	Release 2.1.5	Release 2.1.6

Table 5.5 Training, Validation and Test Sets for Neural Network Modeling

Model	Training Set	Validation Set	Test Set(s)
N1	Release 2.1.1	Release 2.1.1	Release 2.1.2, 2.1.3, 2.1.5, 2.1.6

Table 5.5 shows the formation of training, validation and test sets for neural network modeling. Using each modeling technique, a model is built for each of these sets based on the training data. This model is then evaluated on each of the test sets. Since genetic algorithms were added on later, we build GA-trained models only on release 2.1.1 and test on 2.1.2.

5.5 Principal Components Analysis

For modeling purposes, we prefer uncorrelated data. Principal components analysis (PCA) was performed to reduce the data set containing a large number of correlated features to a small number of uncorrelated ones. The stopping criteria for principal components analysis was a critical decision. The aim was to have the maximum variance in the input data covered by the factors selected in the PCA. This ensured that we did not lose significant information in this step. PCA was carried out using SAS.

PCA was performed on only the training set for each of the models. The output of the PCA along with the above factors is a transformation matrix. This transformation matrix was used to obtain the factor values for each of the test sets.

PCA was initially run with a minimum eigenvalue of one as the stopping rule. This resulted in nine PCA factors on release 2.1.1. The tenth factor was close to the stopping criteria and PCA of most of the other releases resulted in ten factors. Hence we decided to add in the tenth factor for easy comparison. The eigenvalues for all the modules are shown in the Table 5.6.

The rotated factor pattern is shown in Table 5.7. The columns *F1* represents *FACTOR1*, *F2* represents *FACTOR2* and so on. Throughout our analysis, a Varimax [12] rotation was applied to the factor pattern to aid in interpretation. A row measures the correlation between a particular metric and each of the factors. The bold entries mark the metrics which have high correlation with the factors. This explains the aspect of the source code that each of the factors measure. Table 5.8 shows the meaning of the different factors in each of the releases. It can be seen that even though the factor variables change for the parameters over the different releases, they measure the same set of parameters for all the releases. Hence, all the releases seem similar to each other with respect to the PCA.

Eigenvalues and rotated factor pattern for other releases are similar and included in Appendix B.

Table 5.6 Eigenvalues for Release 2.1.1

Rank	Eigenvalues	Difference	Proportion	Cumulative
1	15.85	12.45	0.41	0.41
2	3.40	0.60	0.09	0.49
3	2.79	0.84	0.07	0.56
4	1.95	0.29	0.05	0.62
5	1.67	0.20	0.04	0.66
6	1.47	0.07	0.04	0.70
7	1.40	0.27	0.04	0.73
8	1.13	0.07	0.03	0.76
9	1.06	0.06	0.03	0.79
10	1.00	0.07	0.03	0.81
11	0.93	0.09	0.02	0.84
12	0.83	0.07	0.02	0.86
13	0.76	0.05	0.02	0.88
14	0.71	0.07	0.02	0.90
15	0.64	0.13	0.02	0.91
16	0.50	0.05	0.01	0.92
17	0.46	0.05	0.01	0.94
18	0.40	0.02	0.01	0.95
19	0.38	0.10	0.01	0.96
20	0.28	0.05	0.01	0.96
21	0.24	0.03	0.00	0.97
22	0.20	0.05	0.01	0.98
23	0.15	0.01	0.00	0.98
24	0.14	0.03	0.00	0.98
25	0.11	0.01	0.00	0.99
26	0.10	0.01	0.00	0.99
27	0.10	0.01	0.00	0.99
28	0.08	0.01	0.00	0.99
29	0.07	0.02	0.00	0.99
30	0.05	0.01	0.00	1.00
31	0.04	0.01	0.00	1.00
32	0.03	0.01	0.00	1.00
33	0.02	0.00	0.00	1.00
34	0.02	0.01	0.00	1.00
35	0.01	0.00	0.00	1.00
36	0.01	0.00	0.00	1.00
37	0.01	0.00	0.00	1.00
38	0.00	0.00	0.00	1.00
39	0.00	0.00	0.00	1.00

Table 5.7 Rotated Factor Pattern for Release 2.1.1

	FACTORS									
	F1	*F2*	*F3*	*F4*	*F5*	*F6*	*F7*	*F8*	*F9*	*F10*
RtnStmExeNbr	**0.89**	0.27	0.22	0.16	0.12	0.06	-0.03	0.03	0.02	0.03
RtnCplExeSum	**0.88**	0.22	0.19	0.14	0.21	0.01	-0.01	-0.01	0.05	0.04
RtnStmCtlLopNbr	**0.87**	0.17	0.26	0.14	0.01	0.06	-0.03	0.01	-0.04	-0.01
RtnStmXpdNbr	**0.85**	0.22	0.14	0.26	0.13	0.04	-0.11	-0.01	-0.02	0.00
RtnStmNstLvlSum	**0.82**	0.34	0.17	0.29	0.08	0.06	-0.08	0.01	0.02	0.02
RtnLnsNbr	**0.76**	0.45	0.24	0.07	0.09	0.05	0.10	0.19	0.00	0.02
RtnStmDecObjNbr	**0.75**	0.48	0.22	0.06	0.18	0.03	0.13	0.08	0.00	0.01
RtnCalXplNbr	**0.73**	0.47	0.10	0.08	0.08	-0.02	0.23	0.16	0.01	0.02
RtnArgXplSum	**0.69**	0.39	0.05	0.08	0.13	-0.03	0.30	0.17	-0.03	-0.02
RtnStmCtlCtnNbr	**0.48**	0.20	0.09	0.08	-0.01	0.01	0.08	0.15	-0.14	-0.13
RtnStmCtlIfNbr	0.19	**0.94**	0.06	0.10	0.05	0.01	0.01	-0.04	-0.05	-0.02
RtnScpNbr	0.33	**0.91**	0.11	0.13	0.05	0.02	0.03	-0.02	-0.04	-0.01
RtnCplCtlSum	0.33	**0.89**	0.08	0.12	0.10	0.01	-0.01	-0.05	-0.08	-0.03
RtnScpNstLvlSum	0.39	**0.86**	0.10	0.21	0.03	0.03	0.01	-0.03	-0.05	-0.02
RtnCastXplNbr	0.37	**0.80**	0.07	0.10	0.03	0.00	-0.01	-0.06	0.11	0.01
RtnStmCtlRetNbr	0.33	**0.78**	0.18	0.04	0.09	0.02	0.32	0.11	-0.01	0.08
FilDefRtnNbr	0.32	**0.77**	0.21	0.04	0.08	-0.01	0.33	0.16	0.03	0.04
RtnStmDecPrmNbr	0.32	**0.74**	0.18	0.06	0.12	0.01	0.29	0.09	0.01	0.01
FilDefObjGlbNbr	-0.09	**0.66**	-0.06	-0.12	-0.03	0.00	-0.30	0.16	0.07	0.01

Table 5.7 Rotated Factor Pattern for Release 2.1.1 (continued)

	FACTORS									
	F1	*F2*	*F3*	*F4*	*F5*	*F6*	*F7*	*F8*	*F9*	*F10*
FilLnsNbr	0.60	**0.66**	0.22	0.04	0.08	0.05	0.10	0.20	0.00	0.03
RtnStmCtlCaseNbr	0.23	0.09	**0.90**	0.05	-0.03	0.04	0.02	0.00	0.03	-0.01
RtnStmCtlSwiNbr	0.10	0.12	**0.89**	0.02	0.06	0.00	-0.01	0.03	0.00	0.00
RtnStmCtlBrkNbr	0.43	0.13	**0.76**	0.08	0.00	0.07	0.05	0.01	-0.01	-0.03
RtnStmCtlDfltNbr	0.28	0.20	**0.69**	-0.02	0.04	0.09	-0.05	0.11	-0.01	0.02
RtnScpNstLvlAvg	0.22	0.15	0.02	**0.89**	0.23	0.05	0.08	0.02	0.03	-0.03
RtnStmNstLvlAvg	0.31	0.04	0.05	**0.85**	0.19	0.04	-0.10	-0.01	0.14	0.02
RtnScpNstLvlMax	0.38	0.27	0.08	**0.73**	0.22	0.04	0.24	0.12	0.00	-0.01
RtnCplCtlAvg	0.01	0.15	-0.03	0.38	**0.79**	0.05	-0.07	0.12	-0.11	-0.03
RtnCplCtlMax	0.24	0.16	0.01	0.21	**0.75**	0.09	-0.19	0.14	-0.16	0.01
RtnCplExeMax	0.43	0.04	0.12	0.05	**0.61**	0.01	0.21	-0.05	0.11	-0.02
RtnCplExeAvg	0.07	-0.06	0.03	0.13	**0.56**	-0.24	0.23	-0.28	0.32	0.02
RtnLblNbr	0.03	-0.01	0.05	0.04	0.01	**0.96**	0.07	-0.02	-0.01	-0.01
RtnStmCtlGotoNbr	0.10	0.03	0.09	0.06	0.00	**0.95**	0.05	-0.04	-0.01	0.00
RtnStmDecRtnNbr	0.03	0.14	-0.04	0.05	-0.01	0.10	**0.72**	0.02	-0.01	-0.01
FilIncDirNbr	0.20	0.06	0.16	0.11	-0.03	-0.05	-0.03	**0.77**	-0.02	0.06
RtnStmDecTypeNbr	-0.02	0.02	0.05	0.14	-0.04	-0.04	-0.03	0.00	**0.82**	0.02
FilDecStruNbr	0.06	0.04	-0.11	-0.11	0.16	0.01	0.12	0.55	**0.57**	-0.07
FilIncNbr	0.21	0.13	0.01	0.00	0.29	-0.13	0.27	0.37	-0.43	0.03
FilDecObjExtNbr	-0.02	0.02	-0.01	-0.01	-0.02	0.00	-0.01	0.03	-0.01	**0.98**

Table 5.8 PCA Factors for each Release

Factors	Meaning for releases			
	2.1.1	2.1.2	2.1.3	2.1.5
FACTOR1	size	size	size	if statements
FACTOR2	if statements	if statements	if statements	size
FACTOR3	switch-case	switch-case	switch-case	nesting
FACTOR4	nesting	nesting	nesting	switch-case
FACTOR5	Datrix complexity	Datrix complexity	goto statements	goto statements
FACTOR6	goto statements	goto statements	data type declarations	data type declarations
FACTOR7	nested routine declarations	includes	includes	includes
FACTOR8	includes	data type declarations	nested routine declarations	nested routine declarations
FACTOR9	data type declarations	nested routine declrations	direct includes	direct includes
FACTOR10	extern objects	extern objects	extern objects	extern objects

CHAPTER VI

LOGISTIC REGRESSION MODELING

Logistic regression uses the following equation to build the model. The dependent parameter p in our case is the probability that a given module is fault-prone. We define a module as fault-prone if it had any bugs. The vector of dependent variables $\mathbf{x}$ are the factors obtained from the PCA analysis.

$$\log\left(\frac{p}{1-p}\right) = a_0 + a_1x_1 + a_2x_2 + a_3x_3 + ... + a_nx_n \tag{6.1}$$

Logistic regression analysis was performed using SAS. We review a few important parameters before going on to the modeling process. C_{II}/C_I, known as the cost-ratio, is a subjective choice and measures how a misclassification of type II costs compared to a type I misclassification. This implies how much classifying a fault-prone module as a not-fault-prone module (type II misclassification) would cost us later as compared to classifying a not-fault-prone module as a fault-prone module (type I miclassification). The logistic regression used the stepwise selection method. In the SAS logistic procedure, the selection criteria `slentry` and `slstay` define the stepwise criteria for the factors to enter in the model and stay in the model. SAS uses 15% as the default value. This has also been used successfully in the literature [14].

6.1 Release 1 Models

Using 2.1.1 release data as training or fit data set, we built a logistic regression model using SAS. Three different models were built by varying the parameters. The three models are described below.

Model 1: Model 1 used a 15% stepwise selection criteria for logistic regression. Figure 6.1 shows the classifier model performance by varying the cost ratio on the training set. We selected a cost ratio where the misclassification rates for type I and type II errors were approximately equal. In this case, C_{II}/C_I is 150. A cost ratio of 150 was justified by the practical penalty for letting a fault-prone module go undetected (C_{II}) compared to mistakingly treating a not-fault-prone module as if it were fault-prone (C_I). However, this choice is subjective and can be adjusted as needed. This applies only to the training set. The equation obtained from model 1 is the following.

$$\begin{aligned}\log\left(\frac{p}{1-p}\right) &= -2.71 + 0.40\,\textit{FACTOR1} + 0.41\,\textit{FACTOR2} - 0.33\,\textit{FACTOR3} \\ &\quad -0.25\,\textit{FACTOR5} + 0.22\,\textit{FACTOR7} + 0.40\,\textit{FACTOR8} - 0.43\,\textit{FACTOR9} \qquad (6.2)\end{aligned}$$

Model 2: For model 2, we selected a higher stepwise selection criteria of 25% for logistic regression. This would allow more factors to be present in the model. Figure 6.2 shows the classifier performance as the cost-ratio is varied on the training set. Again, we maintain the same ratio of 150, where the misclassification rates are equal.

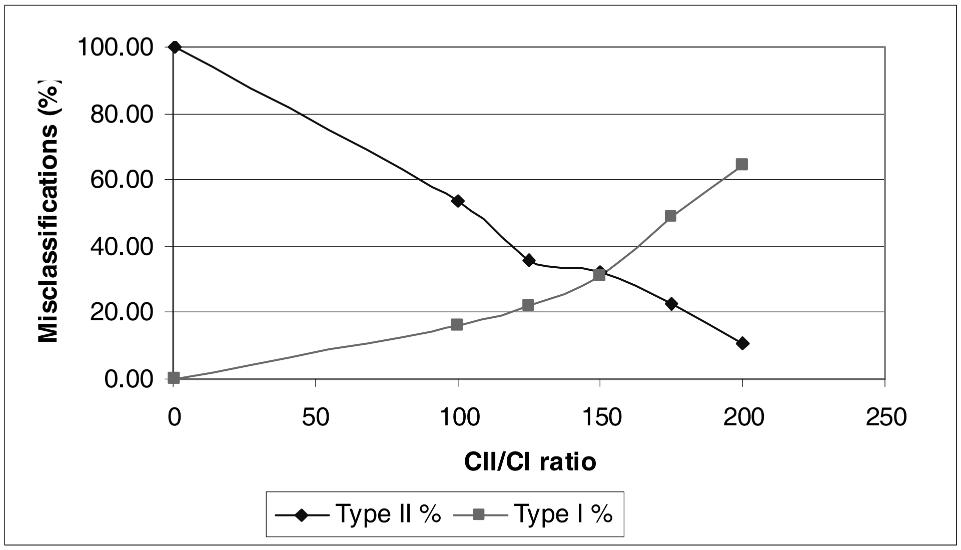

Figure 6.1 Logistic Regression Modeling R1: Cost Ratio Selection for Model 1

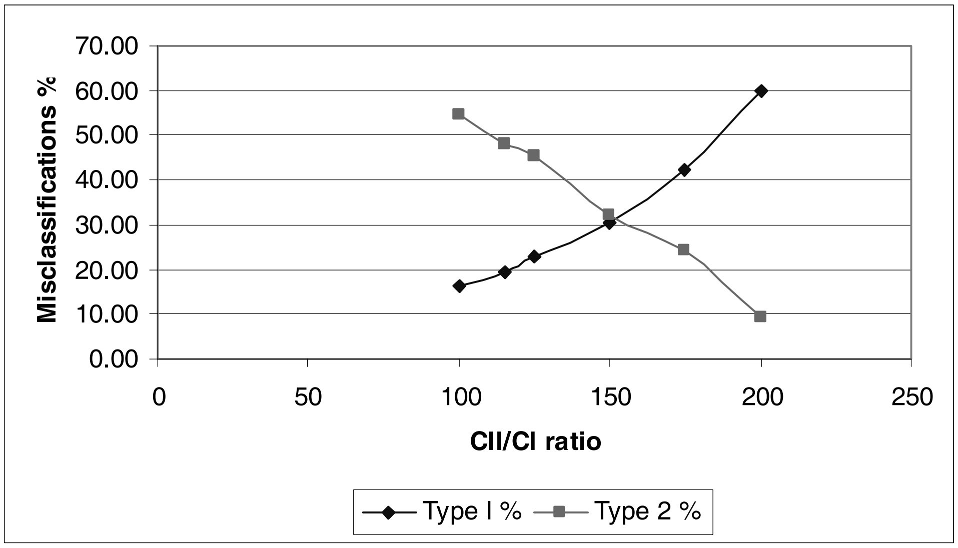

Figure 6.2 Logistic Regression Modeling R1: Cost Ratio Selection for Model 2

However, this model resulted in the same equation as model 1. So for this release, we used model 1 for further analysis.

Model 3: In both of the previous models we used the original data set where the proportion of fault-prone modules modules was very low (less than 0.1). In model 3, we use the duplicated data set. In this data set, we duplicated the fault-prone modules so that they are approximately equal to the number of not-fault-prone modules. The duplicated data set contained 807 not-fault-prone and 750 fault-prone modules. The stepwise selection criteria used was 15%. Figure 6.3 shows the misclassification rates for this model as the cost ratio is varied. We selected the cost ratio, where both type I and type II misclassification rates are equal, at 1.45. This new model resulted in one more parameter being added into the equation, *FACTOR6*.

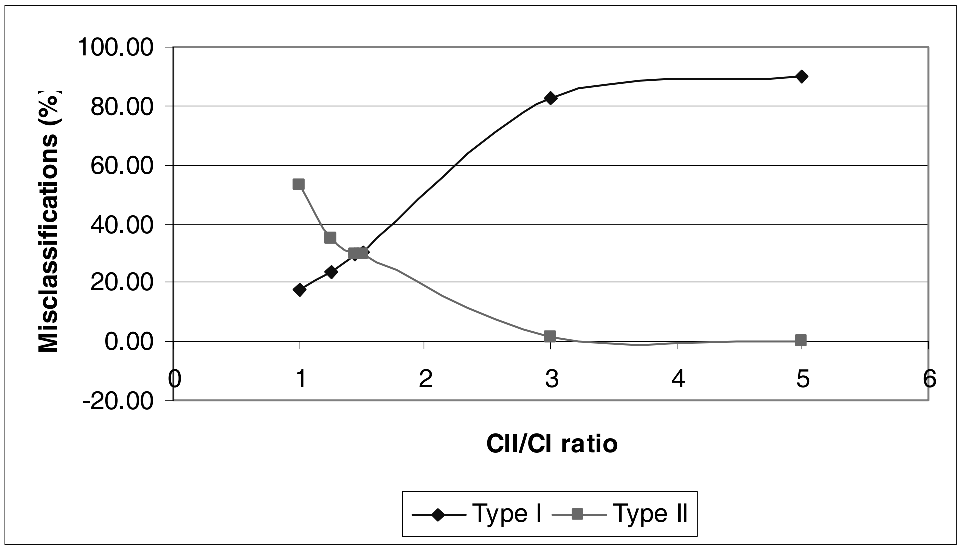

Figure 6.3 Logistic Regression Modeling R1: Cost Ratio Selection for Model 3

$$
\begin{aligned}
\log\left(\frac{p}{1-p}\right) \quad = \quad & -0.46 + 0.41\,\textit{FACTOR1} + 0.45\,\textit{FACTOR2} - 0.25\,\textit{FACTOR3} \\
& -0.17\,\textit{FACTOR5} - 0.48\,\textit{FACTOR6} + 0.24\,\textit{FACTOR7} \\
& +0.60\,\textit{FACTOR8} - 0.39\,\textit{FACTOR9} \qquad (6.3)
\end{aligned}
$$

Evaluation: The above models were tested with each of the four releases 2.1.2, 2.1.3, 2.1.5, and 2.1.6. The results are shown in Table 6.1 and Table 6.2. The type II misclassification rates are of more interest to us, as they tell us how many fault-prone modules the model actually missed. The results show that the models had approximately 69% accuracy on the training set with misclassification rates averaging to approximately 31%. On the 2.1.2 and 2.1.3, the test data accuracy decreased slightly to about 66%. The misclassification rates were slightly higher from 2.1.2 to 2.1.3. For the release 2.1.6 test set, the misclassification rates were lower and the overall accuracy was higher than the previous releases. This in part was because of the distribution of the fault-prone modules to no-fault-prone modules in this release. However the models had poor accuracy with 2.1.5 release test data.

We followed the same methodology to build models on other releases and test on the rest of the releases. We present this models in Appendix C.

Table 6.1 Logistic Regression Modeling Results: Trained on Release 1 (R1)

	Percent		Number of Modules	
	Model 1 (%)	Model 3 (%)	Model 1	Model 3
Training Accuracy 2.1.1				
Ok	68.93	70.52	608	1098
$TypeI$	30.98	29.62	250	239
$TypeII$	32.00	29.33	24	220
Test Accuracy 2.1.2				
Ok	65.27	68.69	590	621
$TypeI$	33.75	30.53	273	247
$TypeII$	43.16	37.89	41	36
Test Accuracy 2.1.3				
Ok	65.19	66.85	588	603
$TypeI$	35.23	33.30	309	292
$TypeII$	33.30	28.00	5	7
Test Accuracy 2.1.5				
Ok	26.97	27.19	246	248
$TypeI$	76.63	76.40	659	657
$TypeII$	13.46	13.46	7	7
Test Accuracy 2.1.6				
Ok	79.39	78.84	728	723
$TypeI$	30.00	34.00	15	17
$TypeII$	20.07	20.42	174	177

Table 6.2 Release 1: Training Set and Test Set Sizes

	Fault-prone	Not-fault-prone	Total
Release 2.1.1			
Model 1	75	807	882
Model 3	750	807	1557
Release 2.1.2	95	809	904
Release 2.1.3	25	877	902
Release 2.1.5	52	860	912
Release 2.1.6	867	50	917

6.2 Interpreting the models

From the logistic regression models, it can be seen that a module is fault-prone i.e. $(p > 1 - p)$ when the RHS of the model equation is positive. If we consider an average module, where all the factor values are zero, then the decision is made based on the intercept. A negative intercept would mean that the $p < 1 - p$ and the module is not-fault-prone. Similarly, a positive intercept will mean that the module was fault-prone. In all of the models discussed, the intercept is always negative. This implies that any model will classify an average module as not-fault-prone. This is justified from our training data distribution. Also, model 3 for each of the releases was based on duplicated data. This model had approximately equal numbers of fault-prone and not-fault-prone modules in each case. Hence the intercept for model 3 is less negative than the other models.

Typically, one would expect that as a factor's value increases, the probability that a module will be classified as fault-prone also increases. Hence, positive factor values increase the chances of a module being fault prone and negative factor values reduce them. In some of the models above, we found that some factors were negative in some models and positive in others. This occurs even when the factor measures the same aspect of the code, such as data type declarations. This suggests that a single factor is not decisive in determining whether a module is fault-prone or not.

CHAPTER VII

NEURAL NETWORK MODELING

This chapter presents the application of different variations of backpropagation neural networks to our classification problem. A backpropagation neural network typically requires an equal distribution of both the classes in its training set. However a typical software release has a fault-prone to not-fault-prone modules ratio of 1:9. Hence, we duplicate the fault-prone modules in our training set so that they are approximately equal to the number of not-fault-prone modules. Section 7.1 describes the various configurations of the backpropagation neural networks that were explored in this research. Section 7.2 presents the results of the best selected configurations on the future releases.

7.1 Experimenting with Backpropagation Neural Networks

We start with a standard backpropagation neural network.

Standard Backpropagation Neural Network: The standard neural network used was a two-layer backpropagation network. It employed a sigmoid function at the hidden and the output layers. All ten factors resulting from the principal components analysis (PCA) were input to the network. The network had ten input nodes, one for each of the ten PCA components and one output node. The number of nodes in the hidden layer were varied from one to six. The output of the network is a real number between zero and one. To get the classification, a simple rule was used. If the value of the output node was greater than 0.5 then the module was labeled as fault-prone, else not-fault-prone. The PCA components were real numbers in the range of approximately -20 to +20. Hence no encoding of inputs was required. Normalization of the PCA was carried out in one of the experiments. However, the standard neural network performed better without normalization. The performance of the neural network varies slightly with each run, since the initial weights are different for each run. We used a best of three technique, reporting the best results obtained on the test sets from three runs of the network for all the experiments. These results were found to be repeatable every few runs.

In this section, duplicated release 2.1.1 was divided into two sets, a training and a validation set. Release 2.1.2 was used for testing. The duplicated release 2.1.1 contained 807 fault-prone modules and 750 not-fault prone modules. Of these 1557 modules, 400 modules selected at random were used as the validation data set. The remaining 1157 modules formed the training set. Release 2.1.2 contained 809 fault-prone modules and 95 not-fault-prone modules.

Table 7.1 shows the results of the basic neural network with the above described data configuration. N_h defines the number of nodes in the hidden layer. *Momentum* defines the momentum value, η defines the learning rate, and *InitialWts* are the range of positive and negative values used to assign the initial weights of the network.

Table 7.1 Basic Neural Network Results

	NN #1	NN #2	NN #3	NN #4	NN #5	NN #6	NN #7
NN Parameters							
N_h	1	1	2	3	4	5	6
$Momentum$	0.00	0.20	0.20	0.00	0.00	0.00	0.00
η	0.30	0.20	0.35	0.18	0.15	0.20	0.15
$InitialWts$	0.50	0.25	0.50	0.10	0.10	0.25	0.50
Train Accuracy (%)							
Ok	74.50	75.28	74.16	81.68	85.31	89.71	91.53
$TypeI$	34.48	32.84	34.98	16.42	12.64	10.18	13.14
$TypeII$	15.51	15.69	15.69	20.44	16.97	10.40	3.28
Test 2.1.2 Accuracy (%)							
Ok	65.15	64.71	63.50	74.00	77.10	78.21	74.12
$TypeI$	36.22	35.97	37.95	21.63	19.16	17.06	22.37
$TypeII$	23.16	29.47	24.21	63.16	54.74	62.11	55.79

The neural network does well with 1 or 2 nodes in the hidden layer. As the number of hidden nodes is increased, the performance on the training set improves, but falls on the test set. This is because, the network memorizes the input patterns and hence does not perform well on unseen samples. To avoid overtraining, the neural network program had a feature, where if the error on the validation set increased continuously for a few epochs, say10, then the training would be stopped. However, it was observed that this option was never selected and the training continued for the maximum number of epochs scheduled, in our case, 500. We also experimented with a larger number of generations up to about 5000, but after approximately 500 generations, there was no significant improvement. To further improve the accuracy of the network, we built a committee machine of these individual networks.

Backpropagation Committee Machines: A committee machine consists of two or more neural networks, different from each other in one way or the other. In our case, a committee machine consisted of three neural networks. The final classification for a module was obtained by combining the classifications from individual networks, generally by voting. However, since the number of fault-prone modules was much less than the number of not-fault-prone modules, we had a different scheme for combining the outputs. If any of the individual networks classified the module as fault-prone, our final classification for the module would be fault-prone, otherwise not-fault-prone. We call this rule the OR rule. We selected the three best performing networks from our previous experiments to form our committee machine.

Table 7.2 Neural Network Committee Machine

	NN # 1	NN # 2	NN # 3	Committee
NN Parameters				
N_h	1	1	2	
$Momentum$	0.30	0.20	0.35	
η	0.00	0.20	0.00	
$InitialWts$	0.50	0.25	0.50	
Train Accuracy (%)				
Ok	74.76	75.28	74.85	
$TypeI$	37.44	32.84	8.87	
$TypeII$	11.68	15.69	43.25	
NN Test 2.1.2 Accuracy (%)				
Ok	61.39	64.82	80.75	64.71
$TypeI$	40.42	35.85	12.61	36.34
$TypeII$	23.16	29.47	75.79	26.32

Table 7.2 shows the results of this committee machine. The committee machine results are not significantly better than the individual committee machines. This can be attributed to the distribution of various samples in our test set. Since a large number of these modules are not-fault-prone, the majority voting rule tends to result in a higher type II error rate. If we use the OR rule, then the type II error rate decreases significantly, however it does so at the expense of type I misclassifications. Hence, it was found that individual networks are better than a committee machine in this case.

No Hidden Layer Neural Network: Since the standard backpropagation network performs best with just one node in the hidden layer, one school of thought is that the relationship being modeled may be linear. To test this hypothesis, we removed the hidden layer from our network. The network now contained an input layer and an output layer. The output layer employed sigmoid units as before.

Table 7.3 shows the results of no-hidden-layer neural networks. The no-hidden-layer neural networks are also called perceptrons. These results had higher type II error rates than a standard 2-layer backpropagation network. Just as with 2-layer networks, we also built committee machines with no hidden layer network.

No-Hidden-Layer Committee Machines: Table 7.4 shows the results of these committee machines. Again, these committee machines did not show significant improvement over the individual standard backpropagation networks.

Table 7.3 No Hidden Layer Neural Network Results

	NN #1	NN #2	NN #3	NN #4
NN Parameters				
$Momentum$	0.90	0.00	0.70	0.00
η	0.35	0.20	0.35	0.25
$InitialWts$	0.50	0.50	0.25	0.50
Train Accuracy (%)				
Ok	73.29	71.05	73.03	69.14
$TypeI$	26.27	19.54	27.59	31.36
$TypeII$	27.19	39.42	26.28	30.29
Test 2.1.2 Accuracy (%)				
Ok	69.80	74.67	68.14	67.26
$TypeI$	27.94	21.14	30.04	32.51
$TypeII$	49.47	61.05	47.37	34.74

Table 7.4 No Hidden Layer Neural Network Committee Machines

	NN # 1	NN # 2	NN # 3	Committee
NN Parameters				
$Momentum$	0.90	0.90	0.70	
η	0.35	0.40	0.35	
$InitialWts$	0.25	0.15	0.50	
Train Accuracy (%)				
Ok	73.29	72.43	72.17	
$TypeI$	21.51	30.17	34.48	
$TypeII$	32.48	24.09	20.44	
NN Test 2.1.2 Accuracy (%)				
Ok	72.68	66.70	60.95	59.07
$TypeI$	23.49	32.88	38.69	42.40
$TypeII$	60.00	36.84	42.11	28.42

Stuttgart Neural Network System: We decided to use a freely available neural network software package to compare and confirm our analysis. We chose the Stuttgart Neural Network Simulator (SNNS) [35]. SNNS is a very sophisticated neural network tool developed by the Institute for Parallel and Distributed High Performance Systems (IPVR) at the University of Stuttgart, Germany. We selected the standard feedforwad backpropagation neural network and used the same data sets to obtain a fair comparison.

Table 7.5 SNNS Results

	NN #1	NN #2	NN #3	NN #4	NN #5	NN #6	NN #7
N_h	1	2	3	4	5	6	7
η	0.2	0.3	0.2	0.2	0.2	0.2	0.2
$Momentum$	0	0.1	0	0	0	0	0
Train Accuracy (%)							
Ok	75.28	75.37	72.95	84.87	74.07	88.33	73.03
$TypeI$	35.30	25.29	40.89	18.39	42.04	14.12	35.80
$TypeII$	12.96	23.91	11.68	11.50	8.03	8.94	17.15
Test 2.1.2 Accuracy (%)							
Ok	63.16	68.25	60.18	71.35	59.18	74.78	63.27
$TypeI$	38.44	29.91	41.90	25.34	43.02	20.52	37.95
$TypeII$	23.16	47.37	22.11	56.84	22.11	65.26	26.32

Table 7.5 shows the results of SNNS on the same data sets. SNNS obtained results comparable to standard backpropagation networks. It also performed best with a single node in the hidden layer.

7.2 Performance On Future Releases

We selected the two best configurations from our previous sections to carry out the further analysis on future data sets. These two configurations were a standard backpropagation neural network with a single node in the hidden layer and an SNNS neural network with a single node in the hidden layer. The standard neural network used η = 0.30 and *InitialWts* in the range of -0.5 to +0.5.

Table 7.6 SNNS *vs.* Standard NN Test Data Sets

	SNNS Accuracy (%)	Standard NN Accuracy (%)
Test 2.1.3 Accuracy (%)		
Ok	57.10	57.32
$TypeI$	42.99	42.76
$TypeII$	40.00	40.00
Test 2.1.5 Accuracy (%)		
Ok	23.46	24.01
$TypeI$	80.70	80.12
$TypeII$	7.69	7.69
Test 2.1.6 Accuracy (%)		
Ok	83.97	83.64
$TypeI$	30.00	30.00
$TypeII$	15.22	15.57

Table 7.6 shows the results on the future data sets with the above two configurations. Both configurations had very similar results on the future releases. They performed on a moderate level on 2.1.3, but failed completely on 2.1.5 release. The performance of these networks on release 2.1.6 was worth noting, especially when this release had a strange

distribution of fault-prone and not-fault-prone modules (867 fault-prone module and 50 not-fault-prone).

This chapter also supported the hypothesis that release 2.1.5 is different from others in some sense, since both the modeling techniques failed to predict on this release.

CHAPTER VIII

GENETIC ALGORITHM TRAINED MODELING

We describe the application of genetic algorithms (GA) to our problem in this chapter. Two variants of GA's are discussed and analyzed. Section 8.1 deals with a logistic regression model learned by using a GA, and Section 8.2 deals with a GA trained neural network. Section 8.3 discusses the different evaluation functions used, and Section 8.4 presents the GA results.

8.1 Logistic Regression Approach

The logistic regression approach consists of fitting an equation as shown below.

$$\log\left(\frac{p}{1-p}\right) = a_0 + a_1x_1 + a_2x_2 + a_3x_3 + ... + a_nx_n \tag{8.1}$$

The class of a module is then determined by the following equation.

$$Class(x_i) = \begin{cases} fault - prone, & if\ \frac{\hat{p}}{1-\hat{p}} \geq \left(\frac{C_I}{C_{II}}\right)\left(\frac{\pi_{nfp}}{\pi_{fp}}\right) \\ not - fault - prone, & otherwise \end{cases} \tag{8.2}$$

where x_i is a module, $\hat{p}$ is the estimated probability of a module being in a fault-prone class, π_{nfp}, π_{fp} are the prior probabilities of membership in the non-fault-prone and the fault-prone classes, respectively, in the training set, and C_I, C_{II} are the costs of misclassification. SAS provides a procedure for doing the logistic regression, which uses a maximum likelihood estimate method to get the classification [12].

The GA approach to logistic regression is to evolve the coefficients a_0, a_1, ..., a_n rather than use a maximum likelihood estimate. The chromosome representation for the GA is a linear arrangement of genes, each gene representing a real valued coefficient $a_0, ..., a_n$. To calculate the fitness of the individual, the evolved coefficients in the individual are substituted into the above equations and the class for each module in the data set is calculated. Using this, a confusion matrix describing misclassifications and correct classifications is obtained. The confusion matrix for this problem has the following form.

$$\textit{Confusion Matrix} = \begin{pmatrix} C_1 & M_1 \\ M_2 & C_2 \end{pmatrix}$$

C_1 defines the number of modules that were correctly classified as being not-fault-prone. C_2 defines the number of modules that were correctly classified as being fault-prone. M_1 represents the number of misclassifications of type I, i.e. the number of modules that were not-fault-prone but were classified as fault-prone. M_2 represents the number of misclassifications of type II, i.e. the number of modules that were fault-prone but were classified as not-fault-prone. Type I misclassifications err on the safer side, whereas type II misclassifications cause some of the fault-prone modules to go undetected. Clearly a type II misclassification is more costly than a type I misclassification.

8.2 Neural Network Approach

In a typical neural network approach, our experiment would have ten input nodes for ten input variables, one output node for classification, and one to seven hidden nodes. The network would be trained using backpropagation algorithm. In our GA approach, we fix the network structure beforehand and then use the evolutionary method for developing the weights of the neural network. The chromosome representation is a linear arrangement of genes, each gene representing the weight between the input and hidden nodes and between hidden and output nodes. To calculate the fitness of the individual, the evolved weights are applied to the network. All the input samples are applied one by one to the network. If the output at the end of the network is greater than 0.5 it is classified as fault-prone, else it is classified as not-fault-prone. A confusion matrix is then built from these classifications.

8.3 Evaluation Function

Using the confusion matrix built above, we calculate the fitness of an individual classifier. The misclassification ratios of type I (*mt1r*) and type II (*mt2r*) are calculated from the confusion matrix. From these, ratios for correct classifications of Type I (*t1r*) and Type II (*t2r*) are calculated using the following equations.

$$mt1r = \frac{M_1}{M_1 + C_1} \tag{8.3}$$

$$mt2r = \frac{M_2}{M_2 + C_2} \tag{8.4}$$

$$\mathit{t1r} = 1 - \mathit{mt1r} \tag{8.5}$$

$$\mathit{t2r} = 1 - \mathit{mt2r} \tag{8.6}$$

Various fitness functions were applied for this task. In the first function, we calculate the fitness as shown in Equation (8.7).

$$\mathit{fitness} = (1 - \mathit{mt2r}) * (1 - \mathit{mt1r} * cost) \tag{8.7}$$

This function gives higher fitness values as the misclassification rates for type I and type II errors decrease. If a classifier classifies all the modules to be of one type, the fitness assigns the value of 0 to that classifier. The cost term enables us to scale a type I misclassification with respect to a type II misclassification.

The second fitness function is as shown in Equation (8.8). This fitness function uses both bonus and penalty.

$$\begin{aligned} \mathit{fitness} &= (100 + \mathit{t2r} * \mathit{costC2} - \mathit{mt1r}) * 10 \\ if\,(\mathit{fitness} &< 0)\; then\, \mathit{fitness} = 0 \end{aligned} \tag{8.8}$$

For every correct type II classification, a bonus is added to the fitness. *costC2* allows the type II misclassifications to be weighted as being more critical. Also, for every misclassification of type I, a penalty is subtracted from the fitness. Negative fitness is avoided by starting the fitness with a positive initial value and assigning a zero value if the fitness ends up being negative.

In the third function, we use the fitness evaluation method proposed by Hochman et al. [10]. We define minimal acceptable correct classification ratio for type I (*L1*) and type II (*L2*). Similarly, we define a higher desirable correct classification ratio limits for type I (*H1*) and type II (*H2*). We then use the following rules for fitness evaluation.

```
if ($t1r \leq L1$ or $t2r \leq L2$)
    $fitness = 0.01$
else if ($L1 < t1r \leq H1$ and $L2 < t2r \leq H2$)
    $fitness = 0.01 + \lambda*(t1r - L1) + (t2r - L2)$
else if ($t1r > H1$ and $L2 < t2r \leq H2$)
    $fitness = 0.01 + \lambda*(t1r - L1) + b1 + (t2r - L2)$
else if ($L2 < t1r \leq L1$ and $t2r > H2$)
    $fitness = 0.01 + \lambda*(t1r - L1) + (t2r - L2) + b2$
else if ($t1r > H1$ and $t2r > H2$)
    $fitness = 0.01 + \lambda*(t1r - L1) + b1 + (t2r - L2) + b2$
```

The parameter λ controls the effect of good class I identification. The parameters *b1* and *b2* are bonus points added for better type I and type II classifications. Usually they are kept low (approximately 0.05 to 0.25 in this case).

8.4 Results

We ran the experiments using all the three fitness functions. Crossover operators used were one-point crossover, and two-point crossover. Single-point and two-point crossover was used with equal probabilities. For mutation operators, we experimented with both random mutation and Gaussian mutation [3]. Roulette wheel selection is used for selecting the parents for producing offsprings.

Table 8.1 GA Parameters

GA Parameters	GA Logistic	GA NN
Population	200	100
Generations	200	100
Mutation Rate	0.03	0.03
Crossover Rate	0.9	0.9

Table 8.2 Data Sets Used in the Experiment

	not-fault-prone	fault-prone	Total
Train	807	75	882
Test	809	95	904

Table 8.1 show the parameters of the GAs used for the experiments. Table 8.2 describes the datasets used for this experiment. Release 2.1.1 is used as a training set and release 2.1.2 as the test set.

Table 8.3 shows the results of the GA-based logistic regression approach and Table 8.4 shows the results of the GA-based neural network approach. Results from various runs using different evaluation functions are presented. The parameters are presented depending on the evaluation function used.

All the evaluation functions were found to be comparable in terms of accuracy of results. Similarly the choice of standard or Gaussian mutation was not significant. The evolutionary power of the GA seems to cancel the effects, if any, of these parameters. It was also observed that the logistic regression approach had a lower type II error rate on the

Table 8.3 GA-Based Logistic Regression Results

	Model					
	# 1	# 2	# 3	# 4	# 5	# 6
Parameters						
cost	0.75	0.85	0.95	0.90	—	—
L1					0.4	0.4
L2					0.4	0.4
H1					0.65	0.75
H2					0.65	0.75
b1					0.15	0.15
b2					0.15	0.20
λ					1.0	0.9
Mutation Type	Standard	Standard	Standard	Gaussian	Standard	Gaussian
Evaluation Function	1	1	2	2	3	3
Train Accuracy (%)						
Ok	67.91	66.67	73.36	66.10	67.12	68.93
$TypeI$	33.46	34.70	26.77	35.69	34.32	32.34
$TypeII$	17.33	18.67	25.33	14.67	17.33	17.33
Test Accuracy (%)						
Ok	65.15	64.27	69.47	63.83	64.60	62.17
$TypeI$	35.35	36.71	28.68	37.45	35.23	37.58
$TypeII$	30.53	27.37	46.32	25.26	36.84	40.00

Table 8.4 GA-Based Neural Network Results

	Model					
	# 1	# 2	# 3	# 4	# 5	# 6
Parameters						
cost	0.75	0.8	—	—	0.95	0.95
L1			0.4	0.4		
L2			0.4	0.4		
H1			0.75	0.75		
H2			0.75	0.75		
b1			0.15	0.225		
b2			0.15	0.225		
λ			0.9	0.9		
Mutation Type	Standard	Standard	Standard	Gaussian	Standard	Gaussian
Evaluation Function	1	1	3	3	2	2
Train Accuracy (%)						
Ok	69.61	63.61	66.21	64.40	62.24	68.59
$TypeI$	29.37	36.68	34.45	34.57	38.04	29.62
$TypeII$	41.33	33.33	26.67	46.67	34.67	50.67
Test Accuracy (%)						
Ok	72.23	65.60	67.37	68.47	63.83	72.23
$TypeI$	27.94	35.85	33.87	32.88	38.32	27.94
$TypeII$	26.32	22.11	22.11	20.00	17.89	26.32

training data, but for the test data, the neural network approach had better accuracy than logistic regression.

CHAPTER IX

DECISION TREE MODELING

Decision trees work by building a tree-based model from the training set and using this tree to classify the samples from the test set. The tool used for decision tree modeling in this thesis is C 4.5 developed by Quinlan [29].

Section 9.1 describes how the C4.5 tool builds a decision tree and uses the tree to classify the data. Section 9.2 presents release 2.1.1 models and Section 9.3 discusses release 2.1.2 models.

9.1 C 4.5

The tree building algorithm as explained by Quinlan [29] is as follows.

Let T denote the set of training samples at a node, initially the root node. Let C_1, C_2, C_3 ... , C_k denote the different classes for these samples. There are four possible cases.

1. All the samples at the node T belong to one class C_i. The algorithm creates a leaf at the node and the classification for that leaf node is C_i.
2. There are no training samples at the node T, then a leaf node is created and the classification at that node is the majority classification at the parent node.
3. There are training samples belonging to multiple classes in T, but they cannot be divided further because of too few samples. In such a case, the class having the majority is the class assigned to the leaf.
4. There are samples belonging to multiple classes in T and are large enough to be split further. An attribute is chosen and the set T is partitioned into mutually exclusive

sets T_1, T_2, T_3, ..., T_n based on that attribute. This process is repeated iteratively on every partitioned set. The attribute chosen to partition the set T is chosen in such a way that it would result in a smallest decision tree. The heuristic used for this purpose is the gain ratio criterion from information theory.

After the tree is built using the above recursive procedure, it is generally found that it overfits the training data. C 4.5 uses error based pruning to produce a simpler tree which can generalize better. It starts from the bottom of the tree estimating the error with each subtree and the error when it is pruned to a leaf. If pruning lowers the error rate of the subtree, then the subtree is pruned to a leaf. This process is repeated until we reach the root of the tree.

The pruned tree is then used for classification. The parameters of the new sample to be classified are used to trace down a path from the root to the leaf of the tree. The classification of the new sample is the classification of the leaf.

The following are a couple of options in C 4.5 that were useful in building the tree.

1. Pruning Confidence Level (*CF*): This option controls the decision tree pruning rate. The default value for this parameter is 25%. Lower values cause heavy pruning.
2. Weight(*m*): C 4.5 requires that every test in the tree should result in at least two outcomes each with a minimum number of cases(*m*). The default value of this parameter is 2.

However, we did not find a way of specifying that misclassifications of type II are more costly than misclassification of type I. C 4.5 's way of classifying a sample is biased in terms of number of training samples at the leaf node. Since our data is highly imbalanced in favor of not-fault-prone modules, C 4.5 would classify most of the modules as not-fault-prone.

We changed the code, so that we can weight these misclassifications. There are different ways of doing this, the way we implemented is that we keep the cost of type II (fault-prone class) misclassification as 1, and varied the type I misclassification cost (between 0 and 1).

9.2 Release 2.1.1 Models

We built two different models, one based on the original data set and the other based on a duplicated data set. For our initial analysis of selecting the best model, we used release 2.1.1 as the training set and 2.1.2 as the test set. The 2.1.1 original data set had 807 not-fault-prone modules and 75 fault-prone modules, whereas in the duplicated data set, the number of fault-prone modules was increased to 750, number of not-fault-prone modules remaining the same. Release 2.1.2 had 809 not-fault-prone modules and 95 fault-prone modules.

Original Data Set Model: The cost ratio variation for the original data set is shown in Figure 9.1. The graph shows a stepwise function. The point where the value changes is approximate. In our analysis, we are only concerned about the values of the stepwise graph and not the actual point, where the value changes. We selected the cost ratio, where both the type I and type II misclassification rates were approximately equal. In this case, we selected our cost ratio to be 0.1. We resorted to very little or no pruning with this model.

This was to avoid pruning of the branches which might contain the very few fault-prone modules. The *CF* parameter was set to 100.

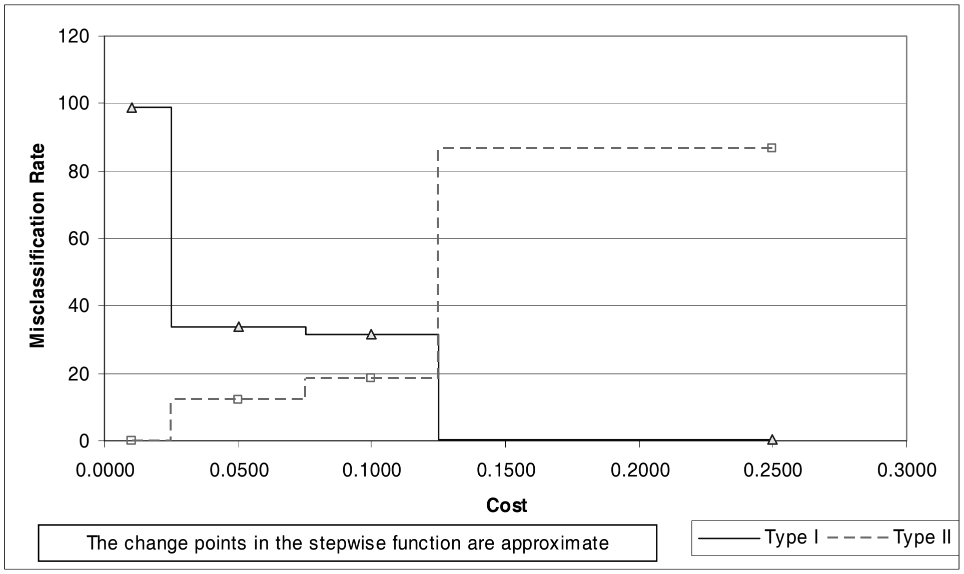

Figure 9.1 Cost Ratio Selection for C 4.5 on Original Data Set

Duplicated Data Set Model: For the duplicated data set, there was no variation with the cost ratio. This is explained by the fact, that the number of fault-prone modules to not-fault-prone modules is approximately equal. Maximum pruning was selected in this model. The *CF* parameter was set to 1. However, we did observe a variation with the *m* parameter. Figure 9.2 shows this variation. The point where both type I and type II errors (lower values of *m*) were least, did not result into good predictions. This was because the tree was too complex and overfitted the data. We used a validation set similar to the one in neural networks. However, the validation set being from the same release did not help much. The results on the test set were poor. This is because the decision tree had become specific to one release. We then used a different release validation set. Thus, the duplicated set model required 2 releases to build the model, duplicated release 2.1.1 as a training set, and release 2.1.2 as the validation set. This model was then tested on future releases. We decided to train the model at the point, where it performs best on the validation set.

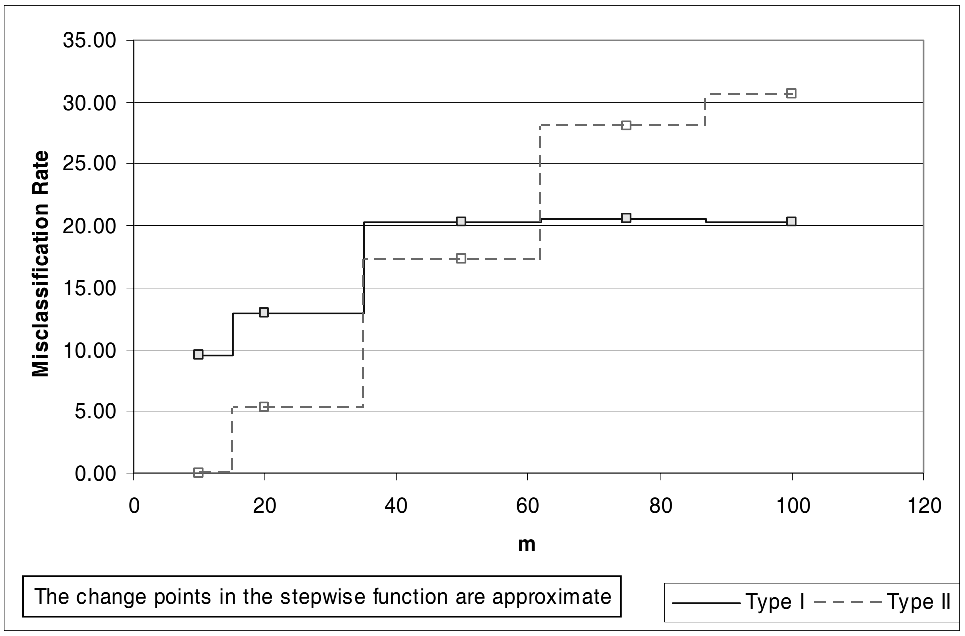

Figure 9.2 Cost Ratio Selection for C 4.5 on Duplicated Data Set

Table 9.1 and Table 9.2 shows the results obtained by using the decision tree built on these two models. The original data set model seemed to perform well on releases 2.1.2,

Table 9.1 C 4.5 Results on Original Data Set Model

	Original Data Set
Train 2.1.1 Accuracy (%)	
Ok	69.50
$TypeI$	31.60
$TypeII$	18.67
Test 2.1.2 Accuracy (%)	
Ok	65.27
$TypeI$	33.75
$TypeII$	43.16
Test 2.1.3 Accuracy (%)	
Ok	64.30
$TypeI$	35.80
$TypeII$	32.00
Test 2.1.5 Accuracy (%)	
Ok	66.45
$TypeI$	33.60
$TypeII$	32.69
Test 2.1.6 Accuracy (%)	
Ok	39.37
$TypeI$	20.00
$TypeII$	62.98

Table 9.2 C 4.5 Results on Duplicated Data Set Model

	Duplicated Data Set
Train 2.1.1 Accuracy (%)	
Ok	71.61
$TypeI$	37.42
$TypeII$	18.67
Validation 2.1.2 Accuracy (%)	
Ok	62.39
$TypeI$	38.81
$TypeII$	27.37
Test 2.1.3 Accuracy (%)	
Ok	58.09
$TypeI$	42.30
$TypeII$	28.00
Test 2.1.5 Accuracy (%)	
Ok	37.06
$TypeI$	64.65
$TypeII$	34.62
Test 2.1.6 Accuracy (%)	
Ok	66.41
$TypeI$	34.00
$TypeII$	33.56

2.1.3, and 2.1.5 but failed on release 2.1.6. The duplicated data set model worked on releases 2.1.2, 2.1.3, and 2.1.6 but failed on release 2.1.5.

It was observed that the original data set model had a lower type I error rate and required only one release to build the model. Hence we selected this model for further analysis.

9.3 Release 2.1.2 Models

The original data set model built earlier was trained on release 2.1.1 and tested on the rest of the releases. The next step was to build a series of models, trained on 2.1.2, 2.1.3, and on 2.1.5 and test each of them on their future releases.

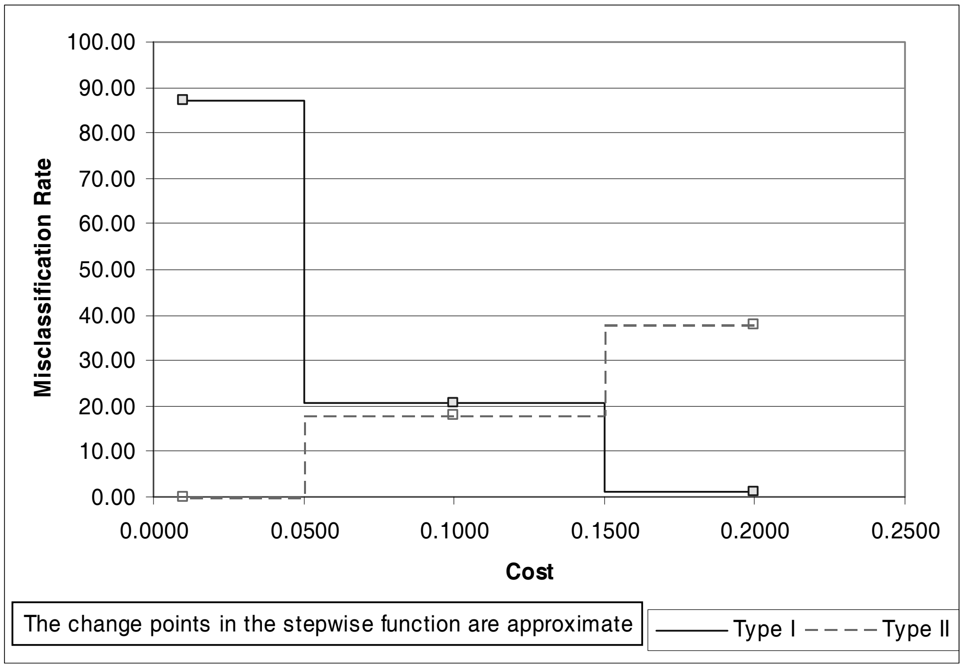

Figure 9.3 Cost Ratio Selection for C 4.5 Release 2.1.2 Model

Table 9.3 C 4.5 Release 2.1.2 Model Results

	Original Data Set Model
Train 2.1.2 Accuracy (%)	
Ok	79.65
$TypeI$	20.64
$TypeII$	17.89
Test 2.1.3 Accuracy (%)	
Ok	73.06
$TypeI$	26.34
$TypeII$	48.00
Test 2.1.5 Accuracy (%)	
Ok	60.64
$TypeI$	39.88
$TypeII$	30.77
Test 2.1.6 Accuracy (%)	
Ok	46.67
$TypeI$	22.00
$TypeII$	55.13

Figure 9.3 shows the cost ratio selection of 0.1 for this model. The results are shown in Table 9.3. The release 2.1.2 model exhibited moderate performance on releases 2.1.3 and 2.1.5. The type I and type II error rates are higher than with the release 2.1.1 models. The models completely failed to predict correctly on the release 2.1.6 data.

Similar models were also built on 2.1.3 and 2.1.5 releases. The 2.1.3 release had 25 out of 902 modules that were fault-prone and 2.1.5 had 52 of 912 modules that were fault-prone. However, because of the smaller number of fault-prone modules, the cost ratio variation had only two points, one where it classified almost all modules as not-fault-prone and other where it classified almost all modules as fault-prone. So for this kind of analysis, a substantial amount of fault-prone modules are required.

Decision tree models had a different trend than the other models. Where the other models failed on release 2.1.5, the decision tree seemed to work, but it failed on release 2.1.6 where the other models worked fine. Release 2.1.6 is the data set that has more fault-prone than not-fault-prone modules. The trend with release 2.1.5 is interesting and remains to be explained.

9.4 Interpretation of the Model

One of the main advantages of decision trees is that they provide a visual representation of the classification process. Figure 9.4 shows the tree obtained by training the decision tree algorithm on release 2.1.1. *FACTOR2* measures `if` statements, *FACTOR7* measures nested routine declarations, *FACTOR8* measures includes, *FACTOR9* measures data type declarations, and *FACTOR10* measures extern objects.

If we consider these factors independently, *FACTOR2*, *FACTOR7*, *FACTOR9*, and *FACTOR10* are easy to explain. High values of these factors mean the module is fault-prone, low values mean not-fault-prone. *FACTOR8* measuring the number of includes is difficult to explain. There are more divisions than just low or high for this factor. The alternating pattern of 1's and 0's at the leaf show that this is not a linear problem.

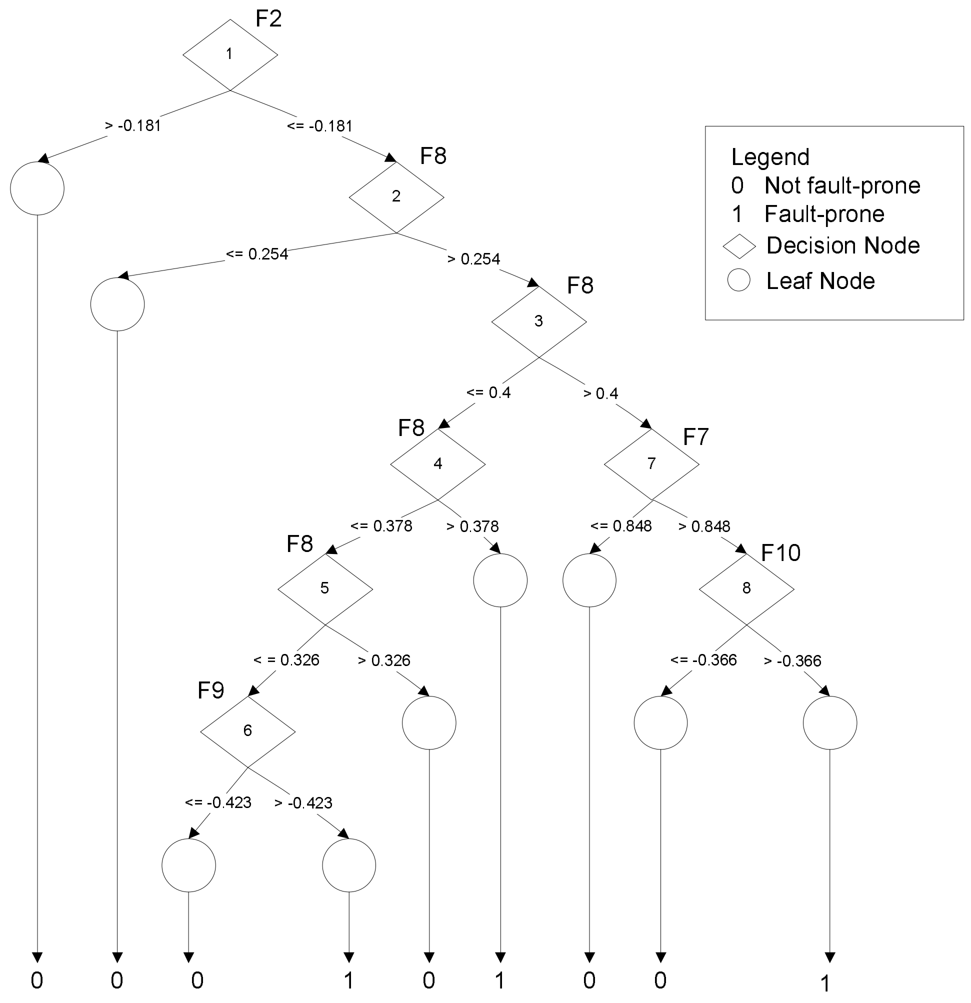

Figure 9.4 Model Trained on Release 2.1.1 Original Data Set

CHAPTER X

ANALYSIS

This chapter presents the discussion of our results, provides answers to our research questions and analyzes various factors affecting the results. Our analysis included only the files which were modified at least once during the period of five releases. There were about 150 unchanged bug-free files which were stable and hence not included in our analysis.

10.1 Research Questions

Table 10.1 shows the results of a statistical t-test [21, 26] obtained by comparing each of the modeling techniques with a random model. The hypothesis tested was that the mean of the predictions of a modeling technique is equal to the mean of the predictions of the random model. The probability column shows the probability that this hypothesis is true. This test was carried out on 2.1.2 release data with 904 observations. Since these probabilities are very small, it is clear that our modeling techniques are significantly better than the random model.

How does a neural network's accuracy compare with logistic regression?

Table 10.2 shows the neural network results compared with logistic regression results. For release 2.1.2, the neural network had a lower type II error rate but a higher type I error

Table 10.1 Statistical t-Test for each Technique *vs.* the Random Model

Model	t statistic	$\Pr$(Model = Random model)
Logistic regression	8.01	3.51E-15
Neural network	7.48	1.80E-13
GA trained logistic regression	6.30	4.69E-10
GA trained neural network	7.97	4.83E-15
Decision tree	6.54	1.01E-10

rate compared to logistic regression. Since the difference in type II error rates is more important, the neural network gets the edge over logistic regression for this release. For release 2.1.6, the neural network had lower type I and type II error rates compared to logistic regression and was thus better.

For release 2.1.3, logistic regression performed better than the neural network. However, it is interesting to note that release 2.1.3 had only 25 modules which were fault-prone. So, the percentages provide a magnified difference. As for release 2.1.5, both the methods failed on that release.

Overall, neural networks and logistic regression results are comparable with neural networks outperforming on a few occasions.

How does a decision tree's accuracy compare with logistic regression?

Table 10.3 shows the decision tree results compared with logistic regression results. For release 2.1.2 and 2.1.3, logistic regression had lower type I and type II error rates than decision trees. The logistic regression failed on release 2.1.5, whereas the decision

Table 10.2 Neural Network *vs.* Logistic Regression Results

	Logistic Regression Accuracy (%)	Neural Network Accuracy (%)
Train 2.1.1		
Ok	70.52	75.37
$TypeI$	29.62	35.14
$TypeII$	29.33	12.96
Test 2.1.2		
Ok	68.69	63.27
$TypeI$	30.53	38.32
$TypeII$	37.89	23.16
Test 2.1.3		
Ok	66.85	57.32
$TypeI$	33.30	42.76
$TypeII$	28.00	40.00
Test 2.1.5		
Ok	27.19	24.01
$TypeI$	76.40	80.12
$TypeII$	13.46	7.69
Test 2.1.6		
Ok	78.84	83.64
$TypeI$	34.00	30.00
$TypeII$	20.42	15.57

tree seemed to work well. For release 2.1.6, the decision tree model failed and logistic regression performed better.

Overall, logistic regression seemed to work better than the decision trees. An interesting fact is that where both logistic regression and neural networks failed on release 2.1.5, decision trees worked well.

How does a neural network's accuracy compare with decision trees?

Table 10.4 shows the neural network results compared with decision tree results over the five releases. The neural network performs better than decision trees on release 2.1.2 and 2.1.6 by the virtue of lower type II errors. Neural networks, as logistic regression, failed on release 2.1.5, where the decision tree performed better. The decision tree also performed better than the neural networks for release 2.1.3.

Overall, the performance of both the methods is comparable, with neural networks being slightly better by virtue of lower type II errors.

How does the accuracy of GA trained logistic regression compare with traditional logistic regression?

Table 10.5 presents the logistic regression model learned by GA and traditional logistic regression results. Traditional logistic regression has a higher type II error rate. The GA trained model had a significantly lower type II error rate and an increased type I error rate. The GA trained model can be judged as being better since the decrease in the type II error rate is greater than the increase in the type I error rate.

Table 10.3 Decision Tree *vs.* Logistic Regression Results

	Logistic Regression Accuracy (%)	C4.5 Decision Tree Accuracy (%)
Train 2.1.1		
Ok	70.52	69.50
$TypeI$	29.62	31.60
$TypeII$	29.33	18.67
Test 2.1.2		
Ok	68.69	65.27
$TypeI$	30.53	33.75
$TypeII$	37.89	43.16
Test 2.1.3		
Ok	66.85	64.30
$TypeI$	33.30	35.80
$TypeII$	28.00	32.00
Test 2.1.5		
Ok	27.19	66.45
$TypeI$	76.40	33.60
$TypeII$	13.46	32.69
Test 2.1.6		
Ok	78.84	39.37
$TypeI$	34.00	20.00
$TypeII$	20.42	62.98

Table 10.4 Neural Network *vs.* Decision Tree Results

	Neural Network Accuracy (%)	C4.5 Decision Tree Accuracy (%)
Train 2.1.1		
Ok	75.37	69.50
$TypeI$	35.14	31.60
$TypeII$	12.96	18.67
Test 2.1.2		
Ok	63.27	65.27
$TypeI$	38.32	33.75
$TypeII$	23.16	43.16
Test 2.1.3		
Ok	57.32	64.30
$TypeI$	42.76	35.80
$TypeII$	40.00	32.00
Test 2.1.5		
Ok	24.01	66.45
$TypeI$	80.12	33.60
$TypeII$	7.69	32.69
Test 2.1.6		
Ok	83.64	39.37
$TypeI$	30.00	20.00
$TypeII$	15.57	62.98

Table 10.5 Logistic Regression Based GA *vs.* Traditional Logistic Regression Results

	True Logistic regression Accuracy (%)	Logistic regression based GA Accuracy (%)
Train 2.1.1		
Ok	70.52	66.10
$TypeI$	29.62	35.69
$TypeII$	29.33	14.67
Test 2.1.2		
Ok	68.69	63.83
$TypeI$	30.53	37.45
$TypeII$	37.89	25.26

How does the accuracy of a GA trained neural network compare with a traditional backpropagation neural network?

Table 10.6 Neural Network Based GA *vs.* Traditional Backprop Neural Network Results

	True Neural networks Accuracy (%)	Neural network based GA Accuracy (%)
Train 2.1.1		
Ok	75.37	66.21
$TypeI$	35.14	34.45
$TypeII$	12.96	26.67
Test 2.1.2		
Ok	63.27	67.37
$TypeI$	38.32	33.87
$TypeII$	23.16	22.11

Table 10.6 presents the results for a neural network learned by GA compared with backpropagation neural network results. The backpropagation neural networks had better results on the training set, but the GA trained network performed better on the test set. The GA trained model had a slightly lower type II error rate than neural network and a significantly lower type I error rate.

10.2 Analysis of Risks

As we found out from the previous section, the predictions for release 5 from logistic regression, and neural nets and for release 6 from decision trees were poor. We attempted to discover the reasons behind this failure. Can we uncover this risk before the predictions? This would allow us to assign a quality factor to our predictions.

We analyzed the summary statistics of each of the releases. Table 10.7 shows the values for the metric *FilIncNbr*, which measures the number of files included in a given source file. It often approximates the number of interfaces a given file has with other files. This metric had a significant increase between releases 2.1.3 and 2.1.5. All the other metrics did not show any significant change between the releases.

Table 10.7 *FilIncNbr* Metric Values

Release	Mean Value	Std. Deviation	Minimum	Maxium
2.1.1	37.60	10.37	0	91
2.1.2	38.66	10.71	0	89
2.1.3	38.71	10.67	0	92
2.1.5	55.96	14.09	0	98
2.1.6	56.15	13.95	0	98

Such an increase in the number of include files could be due to breaking of a few large include files into many smaller include files or due to adding several new features. Such a transition in the metric can caution the quality analyst that these predictions are not very dependable. This information may explain the failure of the above models on releases 5 and 6.

It was noted from the overall observations that the decision tree models performed poorly on release 6, whereas logistic regression and neural networks performed well on release 6. Release 6 had a 9:1 fault-prone to not-fault prone ratio compared to the 1:9 ratio in other releases.

Finally, there is no one methodology that works best in all situations among the ones we discussed here. Neural networks trained with GA do come close to the best, but we do not have enough results to justify that. Neural networks and logistic regression have comparable results with the former outperforming the latter on a few occasions. These two are still better than the decision tree approach.

10.3 Threats to Internal Validity

This kind of modeling requires a large amount of data. Even though we had approximately 800 not-fault-prone modules and about 75 fault-prone modules, there appeared to be insufficient fault-prone modules. To overcome this, we duplicated the fault-prone modules so that the class distribution was approximately even. Although it is rare to find evenly distributed data sets in this field, a larger number of fault-prone modules helps the task.

Our modeling is based on source code metrics. There are several other factors that are useful for predicting if modules are fault-prone, but we were unable to gather data for these metrics. These include various process metrics such as number of inspections for a module, number of bugs found in the inspections, etc. The limitations of the data available make it difficult for our method to yield predictions with very high accuracy rates.

10.4 Threats to External Validity

Our models are based on the PETSc releases. Therefore, these models are expected to work for PETSc releases only. However, the methodology is general and can be extended to build similar models for other software.

CHAPTER XI

CONCLUSIONS

11.1 Evaluation of Hypothesis

The hypothesis we were evaluating is as follows:

> By analyzing past releases of an open-source software product for high performance computing, one can identify high-risk faulty modules. This analysis can be performed by statistical and machine learning modeling techniques.

From our analysis in this research, we can say that our results are good evidence that support our hypothesis. The accuracy of the results, though, is affected by various factors such as skewed distribution of the data, few data of the fault-prone class, lack of process metrics etc. Also, it is possible to identify beforehand whether the predictions are very uncertain. In essence, it is possible to identify the high-risk faulty modules by analyzing the open-source software product for high performance computing by statistical and machine learning techniques.

Of the techniques analyzed in this study, neural network results were comparable to logistic regression and better on a few occasions. Both these approaches worked better than the decision tree approach. The GA trained model for neural network gave the most accurate results for the only set it was tested on.

11.2 Contributions

The evidence presented in this case study confirms our hypothesis and other similar hypotheses found in the literature [14, 15, 16, 17].

The data collection technique of mining the configuration logs of a system to count bugs is a contribution to this field. Almost every software organization maintains a configuration management system making this type of data collection feasible.

This research also built upon the methodology developed by Rapur [30] to conduct an empirical study of open-source software systems. This methodology is applicable to the open-source software community as well as for the closed-source projects. The open-source software community can use this methodology to find and fix the fault-prone modules thereby producing better quality software. The closed-source software community can use this methodology to better allocate their resources for correcting fault-prone modules and uncovering any flaws in the software.

11.3 For Further Research

An interesting area is to build a fault architecture of the PETSc system over these five releases as suggested by Von Mayrhauser et al. [36]. Such an architecture can be used to identify the inter-related bugs between the modules while we treated each of these bugs independently.

An approach suggested by Ying et al. [39] for predicting source code changes by mining the change history is another good topic for further research. This can identify the files that need to be changed if a particular source file is changed. This is also based on the inter-relationship between the modules.

Another area to work on is to carry out additional case studies of different systems. This will make our evidence more convincing and help in verifying our conclusions.

In our case study, the point of stopping the training was determined using equal cost ratios for logistic regression and validation set for neural networks. Another approach is to use ROC curves. Determining if ROC curves result in better models is an interesting area for further work.

For decision trees and neural network, we fed the PCA outputs to build the model. One school of thought is to use the normalized raw data for building decision trees and neural network. It will be interesting to see if this technique results in significant improvements.

REFERENCES

[1] E. B. Allen, "CAREER: Assessment of Open Source Software for High-Performance Computing," *Proposal to National Science Foundation*, Mar. 2001.

[2] *Datrix Metric Reference Manual*, version 4.1 edition, Bell Canada, Montreal, Quebec, Canada, May 2001.

[3] E. Carter, *Generating Gaussian Random Numbers*, Taygeta Scientific Inc., 2004.

[4] M. K. Daskalantonakis, "A Practical View of Software Maintenance and Implementation Experiences," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, Nov. 1992, pp. 998–1010.

[5] G. H. Dunteman, *Principal Components Analysis*, chapter 1, SAGE Publications, Newbury Park, California, 1989, p. 7.

[6] J. Feller, B. Fitzgerald, F. Hecker, S. Hissam, K. Lakhani, and A. van der Hoek, "Meeting Challanges and Surviving Success: The Second Workshop on Open Source Software Engineering.," *Software Engineering Notes*, vol. 27, no. 5, Sept. 2002, pp. 69–71.

[7] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Boston, 1989.

[8] R. B. Grady, "Successfully Applying Software Metrics," *IEEE Computer*, vol. 27, no. 9, 1994, pp. 18–25.

[9] S. Haykin, *Neural Networks: A Comprehensive Foundation*, second edition, chapter 4, Prentice Hall, New Jersey, 1999, pp. 161–175.

[10] R. Hochman, T. M. Khoshgoftaar, E. Allen, and J. P. Hudepohl, "Using the Genetic Algorithm to Build Optimal Neural Networks for Fault-Prone Module Detections," *Proceedings: Seventh International Symposium on Software Reliability Engineering*, White Plains, NY, November 1996, IEEE, pp. 152–162.

[11] *BitMover, Inc*, http://www.bitkeeper.com, South San Fransico, California, February 2004.

[12] *Statistical Analysis System*, 4th edition, http://www.sas.com, Cary, North Carolina, Sept. 1991.

[13] D. R. Jeffrey and L. Votta, "Guest Editor's Special Section Introduction," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, July 1999, pp. 435–437.

[14] T. M. Khoshgoftaar and E. B. Allen, "Logistic Regression Modelling of Software Quality," *International Journal of Reliability, Quality and Safety Engineering*, vol. 6, no. 4, 1999, pp. 303–317.

[15] T. M. Khoshgoftaar and E. B. Allen, "Ordering Fault-Prone Software Modules," *Software Quality Journal*, vol. 11, no. 1, Mar. 2003, pp. 19–37.

[16] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl, "Data Mining of Software Development Databases," *Software Quality Journal*, vol. 9, no. 3, Nov. 2001, pp. 161–176.

[17] T. M. Khoshgoftaar, D. Lanning, and A. Pandya, "A Neural Network Modeling Methodology for the Detection of High-Risk Programs," *Proceedings: Fourth International Symposium on Software Reliability Engineering*, Denver, Colorado, 1993, IEEE Computer Society, pp. 302–309.

[18] B. Kitchenham, S. L. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary Guidelines for Empirical Research in Software Engineering," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, Aug. 2002, pp. 721–733.

[19] D. G. Kleinbaum, *Logistic Regression: A Self-Learning Text*, Springer-Verlag New York Inc, New York, 1994.

[20] T. Lawrie and C. Gacek, "Issues of Dependability in Open Source Software Development," *Software Engineering Notes*, vol. 27, no. 3, May 2002, pp. 34–37.

[21] D. C. Montgomery and G. C. Runger, *Applied Statistics and Probability for Engineers*, 2nd edition, John Wiley and Sons, Inc., New York, 1999.

[22] I. Myrtveit and E. Stensurd, "A Controlled Experiment to Assess the Benefits of Estimating with Analogy and Regression Models," *IEEE Transactions on Software Engineering*, vol. 25, no. 6, July 1999, pp. 510–524.

[23] D. J. Paulish and A. D. Carleton, "Case Studies of Software-Process-Improvement Measurement," *IEEE Computer*, vol. 27, no. 9, 1994, pp. 50–57.

[24] M. C. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber, "Capability Maturity Model Version 1.1," *IEEE Software*, vol. 10, no. 4, 1993, pp. 18–27.

[25] J. W. Paulson, G. Succi, and A. Eberlein, "An Empirical Study of Open-Source and Closed-Source Software Products," *IEEE Transactions on Software Engineering*, vol. 30, no. 4, Apr. 2004, pp. 246–256.

[26] J. Peters, “How to do a t-tests with MS Excel 2000,” 2003, http://www.cofc.edu/ petersj/SMFT639_MSExcel_T-test.html (current 5 Dec. 2003).

[27] S. L. Pfleeger, “Assessing Measurement,” *IEEE Software*, vol. 14, no. 2, Mar. 1997, pp. 25–26, Editor’s introduction to special issue.

[28] A. Phadke, *Identifying Fault-prone Modules using Genetic Algorithms*, term paper, Department of Computer Science and Engineering, Mississippi State University, Mississippi State, Mississippi, 2004.

[29] J. R. Quinlan, *C 4.5: Programs for Machine Learning*, 2nd edition, Morgan Kaufmann Publishers, San Mateo,CA, 1993.

[30] G. Rapur, *Assessment of Open Source Software for High Performance Computing*, master’s thesis, Department of Computer Science and Engineering, Mississippi State, Mississippi, December 2003.

[31] C. B. Seaman, “Qualitative Methods in Empirical Studies of Software Engineering,” *IEEE Transactions on Software Engineering*, vol. 25, no. 4, July 1999, pp. 571–572.

[32] R. W. Selby and A. A. Porter, “Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis,” *IEEE Transactions on Software Engineering*, vol. 14, no. 12, Dec. 1988, pp. 1743–1757.

[33] S. S. Shapiro and A. J. Gross, *Statistical Modeling Techniques*, 2nd edition, Marcel Dekker,Inc, New York, 1981.

[34] J. Singer and N. G. Vinson, “Ethical Issues in Empirical Studies of Software Engineering,” *IEEE Transactions on Software Engineering*, vol. 28, no. 12, Dec. 2002, pp. 1171–1179.

[35] *Stuttgart Neural Network Simulator User Manual, version 4.1*, University of Stuttgart, Tbingen, Germany, June 1995, http://www-ra.informatik.uni-tuebingen.de/SNNS/.

[36] A. von Mayrhauser, J. Wang, M. Ohlsson, and C. Wohlin, “Deriving a Fault Architecture from Defect History,” *Proceedings: 10th International Symposium on Software Reliability Engineering*, Boca Raton, Florida, 1999, IEEE Computer, pp. 295–303.

[37] L. Wall, T. Christiansen, and J. Orwant, *Programming Perl*, 3rd edition, O’Reilly, Sebastopol, California, 2000.

[38] M.-W. Wu and Y.-D. Lin, “Open Source Software Developement: An Overview,” *Computer*, vol. 34, no. 6, June 2001, pp. 33–38.

[39] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting Source Code Changes by Mining Change History," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, September 2004, pp. 574–586.

[40] M. V. Zelkowitz and D. R. Wallace, "Experimental Models for Validating Technology," *IEEE Computer*, vol. 31, no. 5, May 1998, pp. 23–31.

APPENDIX A

RELEASE DISTRIBUTIONS

This appendix presents the distribution of fault-prone modules for the releases 2.1.2 to 2.1.5. Cumulative bugs present the 80% cutoff of bugs and bugs distribution show the number of bugs per module.

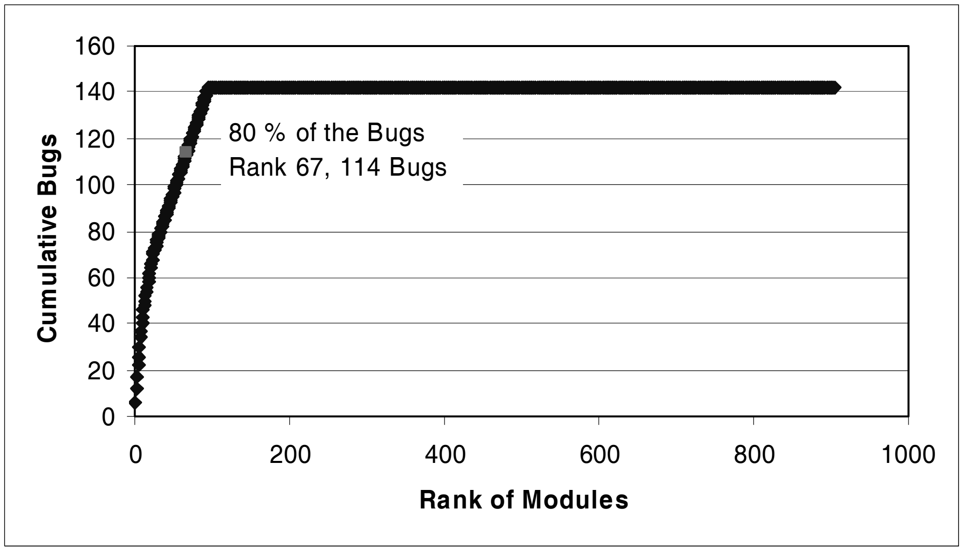

Figure A.1 Cumulative Bugs for PETSc Source 2.1.2

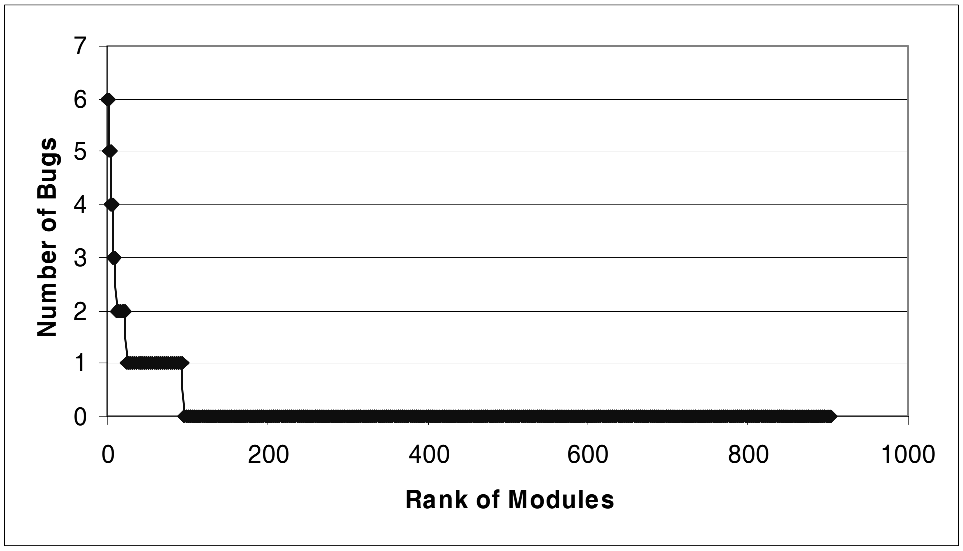

Figure A.2 Bugs Distribution for PETSc Source 2.1.2

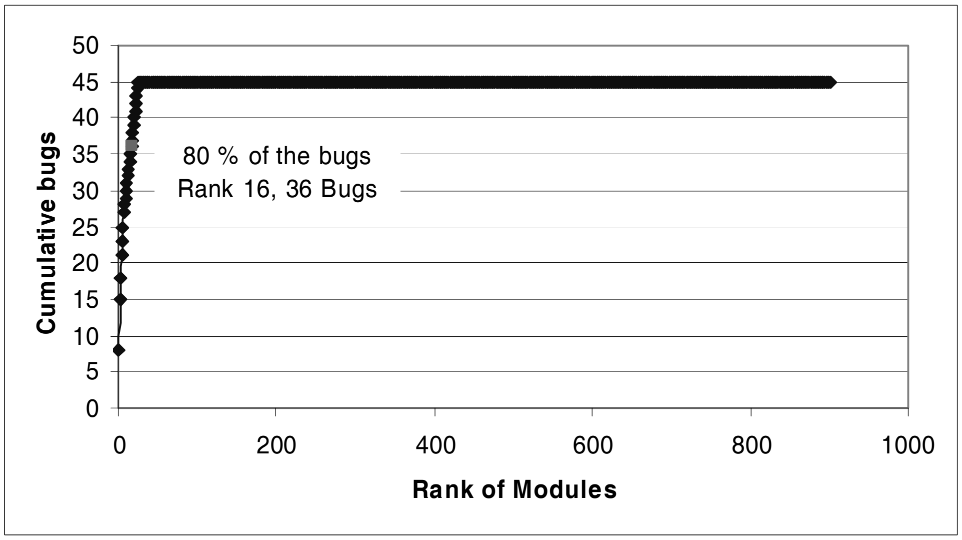

Figure A.3 Cumulative Bugs for PETSc Source 2.1.3

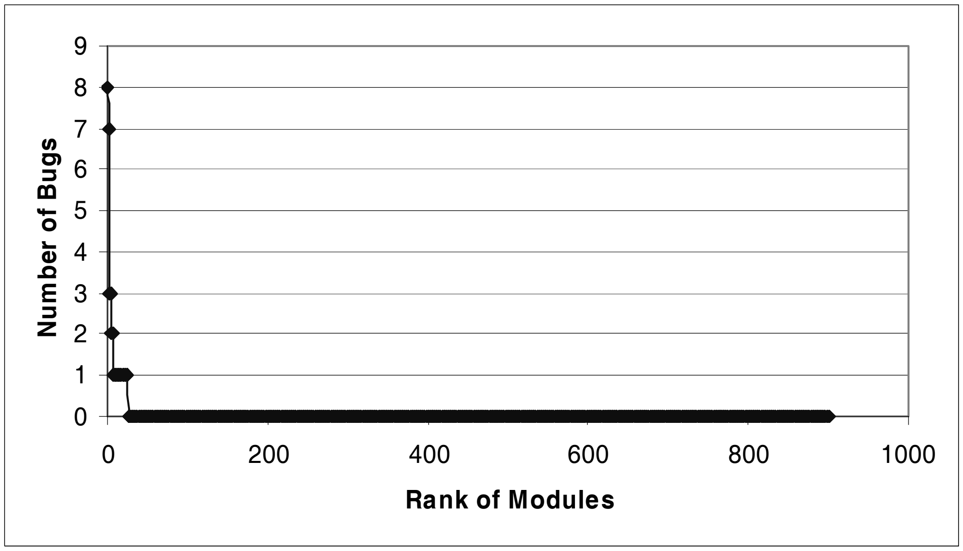

Figure A.4 Bugs Distribution for PETSc Source 2.1.3

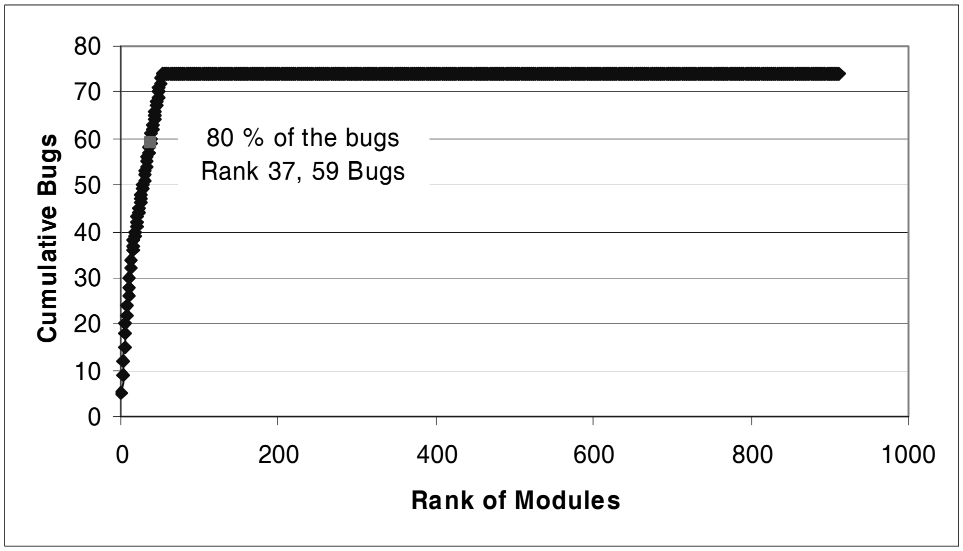

Figure A.5 Cumulative Bugs for PETSc Source 2.1.5

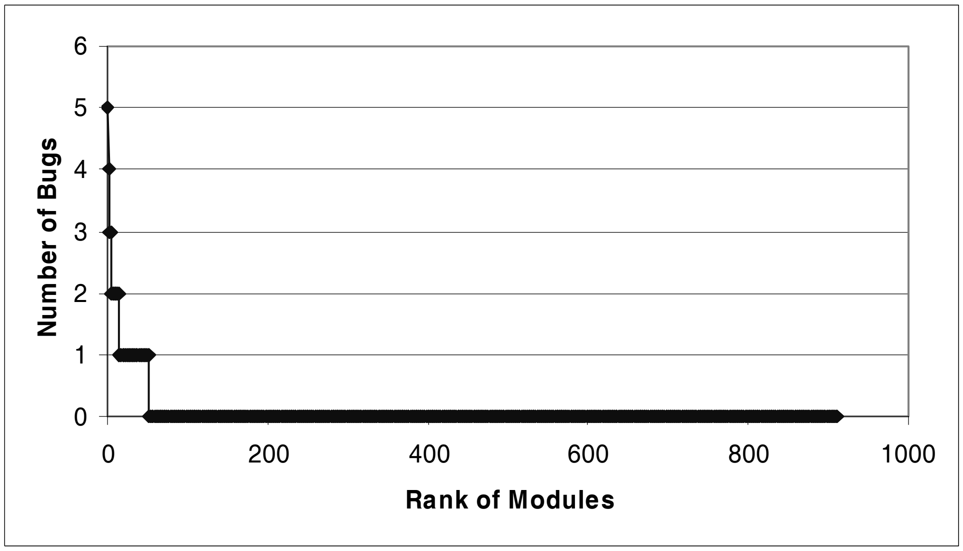

Figure A.6 Bugs Distribution for PETSc Source 2.1.5

APPENDIX B

PRINCIPAL COMPONENT ANALYSIS

This appendix presents the eigenvalues and rotated factor patterns for the releases 2.1.2 to 2.1.5. The PCA of all the releases resulted in ten factors. The stopping criteria used was a minimum eigenvalue of one.

Table B.1 Eigenvalues for Release 2.1.2

Rank	Eigenvalues	Difference	Proportion	Cumulative
1	15.76	12.52	0.40	0.40
2	3.25	0.43	0.08	0.49
3	2.82	0.87	0.07	0.56
4	1.95	0.20	0.05	0.61
5	1.75	0.27	0.04	0.65
6	1.48	0.08	0.04	0.69
7	1.40	0.26	0.04	0.73
8	1.14	0.07	0.03	0.76
9	1.06	0.06	0.03	0.78
10	1.00	0.08	0.03	0.81
11	0.92	0.09	0.02	0.83
12	0.84	0.06	0.02	0.86
13	0.78	0.04	0.02	0.88
14	0.74	0.09	0.02	0.89
15	0.65	0.09	0.02	0.91
16	0.56	0.05	0.01	0.93
17	0.47	0.06	0.01	0.94
18	0.41	0.02	0.01	0.95
19	0.39	0.15	0.01	0.96
20	0.24	0.02	0.01	0.96
21	0.23	0.03	0.01	0.97
22	0.20	0.05	0.01	0.98
23	0.15	0.02	0.00	0.98
24	0.13	0.02	0.00	0.98
25	0.11	0.01	0.00	0.99
26	0.10	0.01	0.00	0.99
27	0.10	0.01	0.00	0.99
28	0.08	0.01	0.00	0.99
29	0.07	0.02	0.00	0.99
30	0.05	0.01	0.00	1.00
31	0.04	0.01	0.00	1.00
32	0.03	0.01	0.00	1.00
33	0.02	0.00	0.00	1.00
34	0.02	0.00	0.00	1.00
35	0.02	0.01	0.00	1.00
36	0.01	0.00	0.00	1.00
37	0.01	0.00	0.00	1.00
38	0.00	0.00	0.00	1.00
39	0.00	0.00	0.00	1.00

Table B.2 Rotated Factor Pattern for Release 2.1.2

	FACTORS									
	F1	*F2*	*F3*	*F4*	*F5*	*F6*	*F7*	*F8*	*F9*	*F10*
RtnStmExeNbr	**0.90**	0.28	0.19	0.16	0.11	0.06	0.03	0.01	-0.03	0.03
RtnCplExeSum	**0.88**	0.23	0.16	0.14	0.20	0.01	0.00	0.04	-0.01	0.03
RtnStmCtlLopNbr	**0.88**	0.19	0.20	0.15	0.01	0.06	0.00	-0.03	-0.02	-0.01
RtnStmXpdNbr	**0.85**	0.23	0.10	0.25	0.14	0.04	-0.01	-0.02	0.09	-0.01
RtnStmNstLvlSum	**0.82**	0.35	0.13	0.29	0.07	0.06	0.01	0.01	-0.07	0.01
RtnLnsNbr	**0.76**	0.46	0.20	0.06	0.17	0.05	0.19	-0.01	0.09	0.02
RtnStmDecObjNbr	**0.75**	0.48	0.19	0.06	0.17	0.03	0.07	-0.01	0.12	0.07
RtnCalXplNbr	**0.71**	0.48	0.14	0.07	0.1	-0.02	0.18	0.00	0.22	0.02
RtnArgXplSum	**0.66**	0.40	0.12	0.06	0.16	-0.03	0.20	-0.04	0.29	0.01
RtnStmCtlCtnNbr	**0.49**	0.21	0.06	0.08	-0.01	0.13	0.15	-0.12	0.08	-0.15
RtnStmCtlIfNbr	0.19	**0.94**	0.04	0.10	0.04	0.01	-0.04	-0.05	-0.01	-0.05
RtnScpNbr	0.33	**0.91**	0.09	0.12	0.04	0.02	-0.02	-0.03	0.01	-0.02
RtnCplCtlSum	0.33	**0.90**	0.05	0.12	0.09	0.01	-0.04	-0.08	-0.03	-0.05
RtnScpNstLvlSum	0.40	**0.86**	0.08	0.21	0.02	0.03	-0.03	-0.04	-0.01	-0.03
RtnCastXplNbr	0.38	**0.80**	0.05	0.10	0.03	0.00	-0.04	0.11	-0.01	0.01
RtnStmCtlRetNbr	0.34	**0.78**	0.20	0.03	0.09	0.02	0.11	-0.01	0.29	0.13
FilDefRtnNbr	0.33	**0.78**	0.21	0.03	0.08	-0.01	0.15	0.02	0.30	0.13
RtnStmDecPrmNbr	0.34	**0.74**	0.17	0.05	0.12	0.01	0.08	0.00	0.26	0.14
FilLnsNbr	0.61	**0.66**	0.19	0.04	0.07	0.05	0.21	0.00	0.08	0.02
FilDefObjGlbNbr	-0.11	**0.54**	-0.02	-0.12	0.00	0.00	0.19	0.06	-0.36	-0.08
RtnStmCtlSwiNbr	0.09	0.11	**0.92**	0.00	0.06	0.00	0.02	-0.01	-0.02	0.00
RtnStmCtlCaseNbr	0.28	0.09	**0.84**	0.06	-0.06	0.06	-0.01	0.05	-0.01	-0.01

Table B.2 Rotated Factor Pattern for Release 2.1.2 (continued)

	FACTORS									
	F1	*F2*	*F3*	*F4*	*F5*	*F6*	*F7*	*F8*	*F9*	*F10*
RtnStmCtlDfltNbr	0.13	0.14	**0.79**	-0.04	0.06	0.03	0.08	-0.04	-0.01	0.01
RtnStmCtlBrkNbr	0.45	0.15	**0.73**	0.09	-0.013	0.08	0.00	0.01	0.04	-0.04
RtnScpNstLvlAvg	0.22	0.16	0.01	**0.88**	0.24	0.05	0.03	0.03	0.08	-0.02
RtnStmNstLvlAvg	0.31	0.04	0.03	**0.85**	0.20	0.03	0.00	0.14	-0.09	0.00
RtnScpNstLvlMax	0.39	0.29	0.06	**0.72**	0.23	0.05	0.12	0.00	0.24	0.05
RtnCplCtlAvg	0.01	0.16	-0.02	0.38	**0.77**	0.05	0.12	-0.12	-0.07	-0.03
RtnCplCtlMax	0.25	0.17	-0.01	0.22	**0.72**	0.09	0.13	-0.18	-0.20	0.00
RtnCplExeMax	0.45	0.05	0.09	0.06	**0.62**	0.03	-0.03	0.09	0.19	-0.01
RtnCplExeAvg	0.07	-0.04	0.03	0.13	**0.59**	0.96	-0.02	-0.01	0.06	0.00
RtnLblNbr	0.03	0.00	0.04	0.04	0.00	**0.96**	-0.02	0.00	0.23	0.05
RtnStmCtlGotoNbr	0.11	0.04	0.07	0.06	0.00	**0.95**	-0.04	0.00	0.04	0.00
FilIncDirNbr	0.23	0.06	0.10	0.14	-0.06	-0.04	**0.76**	-0.02	-0.08	0.07
FilDecStruNbr	0.03	0.04	-0.09	-0.11	0.18	0.02	**0.56**	0.54	0.14	-0.10
RtnStmDecTypeNbr	-0.02	0.02	0.04	0.13	-0.02	-0.03	0.02	**0.83**	-0.03	0.02
FilIncNbr	0.18	0.14	0.07	-0.02	0.29	-0.12	0.41	-0.45	0.21	0.02
RtnStmDecRtnNbr	0.02	0.17	-0.04	0.05	0.00	0.09	0.02	-0.02	**0.73**	-0.07
FilDecObjExtNbr	-0.02	0.05	-0.02	0.00	-0.01	0.00	0.04	-0.01	-0.05	**0.96**

Table B.3 Eigenvalues for Release 2.1.3

Rank	Eigenvalues	Difference	Proportion	Cumulative
1	15.77	12.50	0.40	0.40
2	3.26	0.41	0.08	0.49
3	2.85	0.90	0.07	0.56
4	1.95	0.20	0.05	0.61
5	1.75	0.27	0.04	0.66
6	1.49	0.08	0.04	0.69
7	1.40	0.27	0.04	0.73
8	1.13	0.08	0.03	0.76
9	1.05	0.05	0.03	0.79
10	1.00	0.08	0.03	0.81
11	0.92	0.08	0.02	0.84
12	0.83	0.05	0.02	0.86
13	0.79	0.04	0.02	0.88
14	0.74	0.09	0.02	0.90
15	0.66	0.09	0.02	0.91
16	0.57	0.10	0.01	0.93
17	0.46	0.06	0.01	0.94
18	0.41	0.02	0.01	0.95
19	0.39	0.15	0.01	0.96
20	0.24	0.02	0.01	0.97
21	0.22	0.03	0.01	0.97
22	0.19	0.05	0.01	0.98
23	0.15	0.02	0.00	0.98
24	0.13	0.02	0.00	0.98
25	0.11	0.01	0.00	0.99
26	0.11	0.01	0.00	0.99
27	0.09	0.01	0.00	0.99
28	0.08	0.01	0.00	0.99
29	0.07	0.02	0.00	1.00
30	0.05	0.01	0.00	1.00
31	0.04	0.01	0.00	1.00
32	0.03	0.01	0.00	1.00
33	0.02	0.00	0.00	1.00
34	0.02	0.01	0.00	1.00
35	0.01	0.00	0.00	1.00
36	0.01	0.00	0.00	1.00
37	0.01	0.00	0.00	1.00
38	0.00	0.00	0.00	1.00
39	0.00		0.00	1.00

Table B.4 Rotated Factor Pattern for Release 2.1.3

	FACTORS									
	F1	*F2*	*F3*	*F4*	*F5*	*F6*	*F7*	*F8*	*F9*	*F10*
RtnStmExeNbr	**0.89**	0.29	0.18	0.16	0.11	0.05	0.04	0.01	-0.02	0.04
RtnStmCtlLopNbr	**0.88**	0.19	0.18	0.15	0.01	0.07	0.01	-0.04	-0.02	-0.01
RtnCplExeSum	**0.88**	0.23	0.16	0.14	0.20	0.01	-0.01	0.04	0.00	0.03
RtnStmXpdNbr	**0.85**	0.23	0.10	0.25	0.14	0.04	0.00	-0.03	-0.09	0.01
RtnStmNstLvlSum	**0.82**	0.35	0.13	0.29	0.08	0.06	0.02	0.01	-0.07	0.02
RtnLnsNbr	**0.77**	0.47	0.21	0.07	0.09	0.05	0.19	0.01	0.10	0.02
RtnStmDecObjNbr	**0.75**	0.49	0.19	0.06	0.17	0.03	0.07	0.00	0.13	0.02
RtnCalXplNbr	**0.69**	0.49	0.17	0.07	0.10	-0.02	0.19	0.03	0.22	0.00
RtnArgXplSum	**0.64**	0.41	0.15	0.06	0.16	-0.04	0.21	-0.01	0.29	-0.03
RtnStmCtlCtnNbr	**0.49**	0.22	0.04	0.06	0.01	0.03	0.18	-0.09	0.06	-0.12
RtnStmCtlIfNbr	0.18	**0.95**	0.03	0.10	0.04	0.01	-0.02	-0.05	-0.02	-0.03
RtnScpNbr	0.33	**0.91**	0.09	0.12	0.05	0.02	-0.01	-0.03	0.01	-0.02
RtnCplCtlSum	0.32	**0.90**	0.05	0.11	0.10	0.02	-0.03	-0.08	-0.04	-0.04
RtnScpNstLvlSum	0.39	**0.87**	0.07	0.21	0.03	0.03	-0.01	-0.04	-0.01	-0.03
RtnCastXplNbr	0.37	**0.80**	0.04	0.10	0.03	-0.01	-0.05	0.11	-0.03	0.01
RtnStmCtlRetNbr	0.33	**0.79**	0.20	0.02	0.09	0.02	0.11	0.01	0.29	0.09
FilDefRtnNbr	0.32	**0.78**	0.21	0.03	0.08	-0.01	0.14	0.04	0.31	0.05
RtnStmDecPrmNbr	0.33	**0.75**	0.17	0.05	0.11	0.01	0.08	0.02	0.26	0.02

Table B.4 Rotated Factor Pattern for Release 2.1.3 (continued)

	FACTORS									
	F1	*F2*	*F3*	*F4*	*F5*	*F6*	*F7*	*F8*	*F9*	*F10*
FilLnsNbr	0.61	**0.67**	0.20	0.03	0.07	0.05	0.19	0.02	0.09	0.03
FilDefObjGlbNbr	-0.11	**0.54**	-0.02	-0.12	0.02	0.01	0.18	0.08	-0.38	0.01
RtnStmCtlSwiNbr	0.09	0.11	**0.93**	0.00	0.05	-0.01	0.03	-0.02	-0.02	0.00
RtnStmCtlCaseNbr	0.29	0.10	**0.83**	0.07	-0.06	0.07	-0.01	0.04	-0.02	-0.01
RtnStmCtlDfltNbr	0.12	0.14	**0.80**	-0.04	0.07	0.02	0.08	-0.03	0.00	0.01
RtnStmCtlBrkNbr	0.46	0.15	**0.72**	0.09	-0.03	0.09	0.00	0.00	0.03	-0.03
RtnScpNstLvlAvg	0.21	0.17	0.01	**0.88**	0.26	0.05	0.04	0.03	0.08	-0.03
RtnStmNstLvlAvg	0.30	0.04	0.03	**0.86**	0.20	0.03	-0.01	0.12	-0.09	0.01
RtnScpNstLvlMax	0.38	0.29	0.06	**0.72**	0.23	0.05	0.13	0.01	0.24	-0.01
RtnCplCtlAvg	0.02	0.15	-0.02	0.35	**0.80**	0.05	0.10	-0.08	-0.07	0.00
RtnCplCtlMax	0.26	0.16	-0.01	0.18	**0.75**	0.08	0.13	-0.13	-0.18	0.04
RtnCplExeMax	0.44	0.05	0.10	0.07	**0.60**	0.01	-0.06	0.11	0.22	-0.05
RtnCplExeAvg	0.07	-0.04	0.04	0.17	**0.54**	-0.24	-0.28	0.29	0.22	-0.03
RtnLblNbr	0.04	0.00	0.05	0.04	0.01	**0.96**	-0.02	-0.01	0.07	-0.01
RtnStmCtlGotoNbr	0.12	0.04	0.08	0.06	0.01	**0.95**	-0.04	-0.01	0.04	0.00
FilIncDirNbr	0.23	0.06	0.09	0.13	-0.06	-0.03	**0.77**	0.05	-0.05	0.05
FilIncNbr	0.17	0.15	0.09	-0.02	0.29	-0.13	**0.46**	-0.39	0.24	-0.02
RtnStmDecTypeNbr	-0.02	0.02	0.04	0.15	-0.06	-0.04	-0.08	**0.82**	-0.05	0.02
FilDecStruNbr	0.03	0.04	-0.09	-0.12	0.17	0.02	0.48	**0.62**	0.13	-0.06
RtnStmDecRtnNbr	0.02	0.17	-0.05	0.04	-0.01	0.10	0.03	0.00	**0.71**	0.00
FilDecObjExtNbr	-0.01	0.02	-0.01	-0.01	-0.01	-0.01	0.03	-0.01	0.00	**0.98**

Table B.5 Eigenvalues for Release 2.1.5

Rank	Eigenvalues	Difference	Proportion	Cumulative
1	15.69	12.45	0.40	0.40
2	3.25	0.32	0.08	0.49
3	2.92	0.94	0.07	0.56
4	1.98	0.23	0.05	0.61
5	1.75	0.23	0.04	0.66
6	1.52	0.11	0.04	0.70
7	1.41	0.31	0.04	0.73
8	1.10	0.05	0.03	0.76
9	1.05	0.06	0.03	0.79
10	0.99	0.05	0.03	0.81
11	0.93	0.11	0.02	0.84
12	0.82	0.04	0.02	0.86
13	0.78	0.04	0.02	0.88
14	0.74	0.06	0.02	0.90
15	0.68	0.08	0.02	0.91
16	0.60	0.13	0.02	0.93
17	0.47	0.08	0.01	0.94
18	0.39	0.03	0.01	0.95
19	0.36	0.13	0.01	0.96
20	0.23	0.02	0.01	0.97
21	0.21	0.03	0.01	0.97
22	0.19	0.04	0.00	0.98
23	0.15	0.02	0.00	0.98
24	0.13	0.02	0.00	0.98
25	0.11	0.01	0.00	0.99
26	0.10	0.01	0.00	0.99
27	0.09	0.01	0.00	0.99
28	0.08	0.01	0.00	0.99
29	0.07	0.02	0.00	0.99
30	0.05	0.01	0.00	1.00
31	0.04	0.01	0.00	1.00
32	0.03	0.01	0.00	1.00
33	0.02	0.00	0.00	1.00
34	0.02	0.00	0.00	1.00
35	0.02	0.01	0.00	1.00
36	0.01	0.00	0.00	1.00
37	0.01	0.00	0.00	1.00
38	0.00	0.00	0.00	1.00
39	0.00		0.00	1.00

Table B.6 Rotated Factor Pattern for Release 2.1.5

	FACTORS									
	F1	*F2*	*F3*	*F4*	*F5*	*F6*	*F7*	*F8*	*F9*	*F10*
RtnStmCtlIfNbr	**0.95**	0.18	0.10	0.02	0.02	-0.05	0.03	-0.03	-0.03	-0.03
RtnScpNbr	**0.92**	0.33	0.12	0.07	0.03	-0.03	0.02	0.00	-0.01	-0.02
RtnCplCtlSum	**0.90**	0.31	0.15	0.03	0.03	-0.04	0.07	-0.05	-0.04	-0.04
RtnScpNstLvlSum	**0.86**	0.40	0.18	0.05	0.04	-0.07	0.00	0.00	-0.01	-0.04
RtnCastXplNbr	**0.82**	0.34	0.10	0.03	0.00	0.02	-0.09	-0.01	-0.05	0.01
RtnStmCtlRetNbr	**0.79**	0.32	0.06	0.23	0.01	0.10	0.10	0.26	0.06	0.09
FilDefRtnNbr	**0.79**	0.31	0.06	0.24	-0.02	0.12	0.07	0.29	0.12	0.06
RtnStmDecPrmNbr	**0.76**	0.31	0.10	0.19	0.01	0.10	0.08	0.23	0.06	0.02
FilLnsNbr	**0.68**	0.59	0.05	0.21	0.04	0.13	0.07	0.08	0.17	0.03
FilDefObjGlbNbr	**0.54**	-0.13	-0.09	-0.01	0.00	0.08	-0.03	-0.38	0.23	0.01
RtnStmCtlLopNbr	0.20	**0.89**	0.12	0.16	0.08	-0.03	0.00	-0.01	0.06	-0.02
RtnStmExeNbr	0.30	**0.89**	0.17	0.19	0.05	0.08	0.04	-0.01	0.02	0.04
RtnCplExeSum	0.25	**0.87**	0.19	0.16	0.01	0.16	0.05	-0.02	-0.05	0.04
RtnStmXpdNbr	0.24	**0.86**	0.26	0.09	0.05	0.01	0.04	-0.07	0.00	0.00
RtnStmNstLvlSum	0.36	**0.83**	0.26	0.12	0.07	0.00	-0.02	-0.03	0.01	0.01
RtnLnsNbr	0.48	**0.76**	0.09	0.22	0.04	0.14	0.08	0.10	0.15	0.03
RtnStmDecObjNbr	0.50	**0.73**	0.12	0.20	0.03	0.16	0.11	0.09	0.04	0.02

Table B.6 Rotated Factor Pattern for Release 2.1.5 (continued)

	FACTORS									
	F1	*F2*	*F3*	*F4*	*F5*	*F6*	*F7*	*F8*	*F9*	*F10*
RtnCalXplNbr	0.50	**0.65**	0.07	0.25	-0.03	0.19	0.12	0.21	0.05	0.02
RtnArgXplSum	0.42	**0.61**	0.08	0.24	-0.05	0.24	0.19	0.25	0.02	0.00
RtnStmCtlCtnNbr	0.22	**0.52**	0.07	0.02	0.04	-0.06	0.08	0.05	0.21	-0.12
RtnScpNstLvlAvg	0.15	0.26	**0.86**	-0.01	0.05	-0.02	-0.11	0.16	-0.01	-0.05
RtnStmNstLvlAvg	0.03	0.34	**0.82**	0.02	0.04	-0.05	-0.25	0.01	-0.01	-0.01
RtnCplCtlAvg	0.15	-0.02	**0.74**	0.00	0.02	0.25	0.38	-0.15	0.05	0.03
RtnScpNstLvlMax	0.29	0.39	**0.71**	0.06	0.04	0.05	-0.02	0.29	0.09	-0.02
RtnCplCtlMax	0.17	0.20	**0.59**	0.00	0.06	0.21	0.41	-0.26	0.12	0.08
RtnStmCtlSwiNbr	0.11	0.10	0.01	**0.94**	-0.01	0.03	0.05	-0.02	-0.03	0.00
RtnStmCtlDfltNbr	0.14	0.11	-0.03	**0.84**	0.00	0.06	0.10	0.01	-0.02	0.02
RtnStmCtlCaseNbr	0.10	0.30	0.03	**0.82**	0.08	-0.07	-0.10	-0.02	0.07	-0.02
RtnStmCtlBrkNbr	0.16	0.46	0.06	**0.71**	0.10	-0.07	-0.06	0.02	0.08	-0.04
RtnLblNbr	0.01	0.05	0.04	0.04	**0.96**	0.00	0.00	0.06	0.00	-0.01
RtnStmCtlGotoNbr	0.05	0.12	0.05	0.07	**0.95**	-0.02	-0.01	0.04	-0.02	0.00
FilDecStruNbr	0.03	0.03	-0.02	-0.08	0.01	**0.78**	-0.16	0.08	0.26	-0.04
RtnCplExeMax	0.06	0.37	0.27	0.09	0.03	**0.63**	0.13	0.00	-0.18	-0.02
RtnCplExeAvg	-0.01	0.06	0.41	0.04	-0.21	**0.45**	-0.11	0.03	-0.40	0.00
FilIncNbr	0.13	0.17	0.06	0.09	-0.06	0.11	**0.67**	0.18	-0.05	-0.02
RtnStmDecTypeNbr	0.04	-0.03	0.11	0.05	-0.04	0.28	-0.74	0.01	-0.02	0.02
RtnStmDecRtnNbr	0.17	0.00	0.04	-0.05	0.09	0.07	0.09	**0.72**	0.05	0.01
FilIncDirNbr	0.06	0.24	0.11	0.06	-0.06	0.11	-0.03	0.03	**0.81**	0.03
FilDecObjExtNbr	0.02	-0.01	-0.02	-0.02	-0.01	-0.05	-0.02	0.00	0.02	**0.98**

APPENDIX C

LOGISTIC REGRESSION MODELS

This appendix presents logistic regression models built on release 2.1.2 and later. For every release, we build three models. The parameters for constructing these models are shown in Table C.1. Cost ratio is varied on the training set to obtain the type I and type II misclassification variations. The cost ratio where these two misclassification rates are equal is selected for building the model.

Table C.1 Parameters for Different Models

	Data Set	Selection Criteria
Model 1	Original	15%
Model 2	Original	25%
Model 3	Duplicated	15%

C.1 Release 2 Models

In this phase of modeling, we built a model based on release 2.1.2 and evaluated it on the subsequent releases. We built three different models as explained earlier. Figure C.1, Figure C.2 and Figure C.3 show the cost ratio selections for these models. The equations of these models are as follows.

$$\begin{aligned}\log\left(\frac{p}{1-p}\right) &= -2.33 + 0.27\,\textit{FACTOR1} + 0.27\,\textit{FACTOR3} + 0.22\,\textit{FACTOR5} \\ &\quad + 0.37\,\textit{FACTOR7} + 0.38\,\textit{FACTOR9} \qquad \text{(C.1)}\end{aligned}$$

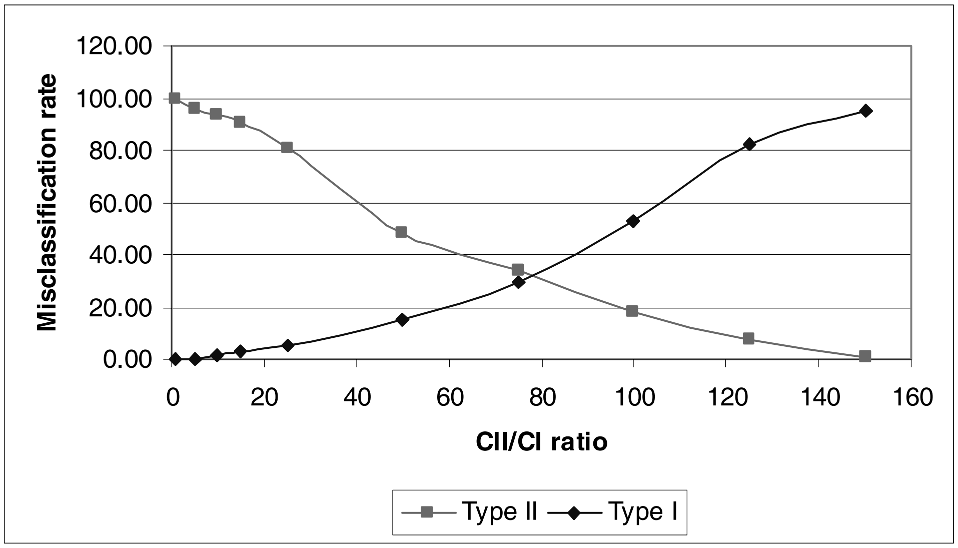

Figure C.1 Logistic Regression Modeling R2:Cost Ratio Selection for Model 1

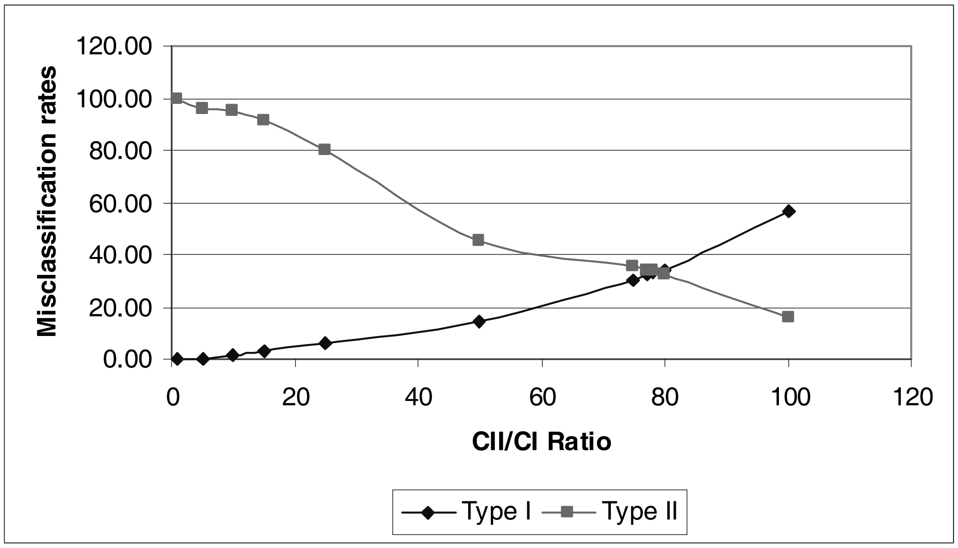

Figure C.2 Logistic Regression Modeling R2:Cost Ratio Selection for Model 2

$$\log\left(\frac{p}{1-p}\right) = -2.36 + 0.26\,\mathit{FACTOR1} + 0.28\,\mathit{FACTOR3} - 0.20\,\mathit{FACTOR4} + 0.21\,\mathit{FACTOR5} + 0.39\,\mathit{FACTOR7} - 0.16\,\mathit{FACTOR8} + 0.38\,\mathit{FACTOR9} \qquad \text{(C.2)}$$

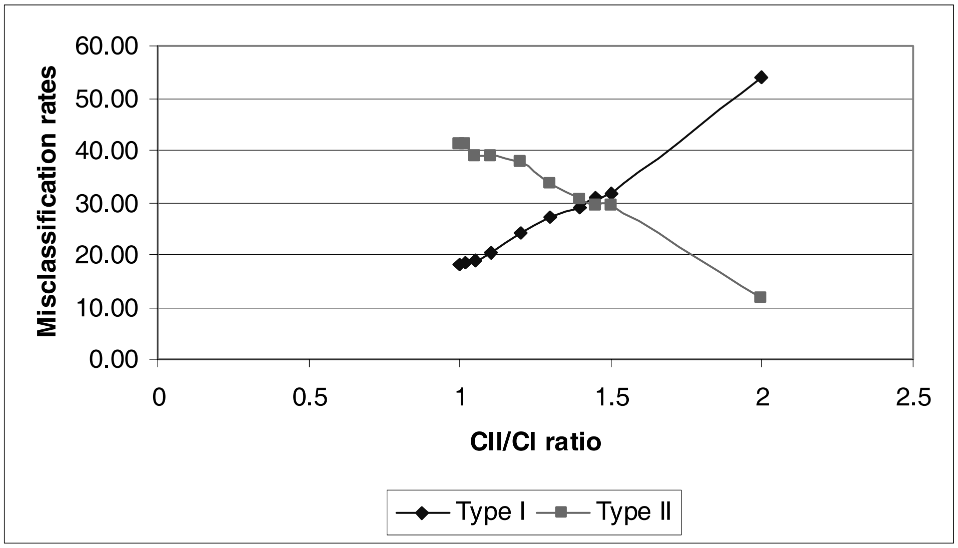

Figure C.3 Logistic Regression Modeling R2:Cost Ratio Selection for Model 3

$$\log\left(\frac{p}{1-p}\right) = -0.38 + 0.47\,\mathit{FACTOR1} + 0.21\,\mathit{FACTOR3} - 0.20\,\mathit{FACTOR4} + 0.17\,\mathit{FACTOR5} - 0.21\,\mathit{FACTOR6} + 0.53\,\mathit{FACTOR7} - 0.40\,\mathit{FACTOR8} + 0.61\,\mathit{FACTOR9} \qquad \text{(C.3)}$$

Table C.2 Logistic Regression Modeling Results: Trained on Release 2 (R2)

	Percent			Number of Modules		
	Model 1 (%)	Model 2	Model 3 (%)	Model 1	Model 2	Model 3
Training Accuracy 2.1.2						
Ok	68.36	66.92	65.52	618	605	1096
$TypeI$	31.64	33.00	34.32	256	267	249
$TypeII$	31.58	33.68	40.00	30	32	224
Test Accuracy 2.1.3						
Ok	65.08	64.08	65.52	587	578	591
$TypeI$	34.78	35.92	34.32	305	315	301
$TypeII$	40.00	36.00	40.00	10	9	10
Test Accuracy 2.1.5						
Ok	43.35	27.72	24.50	391	250	221
$TypeI$	59.42	76.16	80.00	511	655	688
$TypeII$	19.23	13.46	5.77	10	7	3
Test Accuracy 2.1.6						
Ok	64.99	78.63	82.88	596	721	760
$TypeI$	28.00	36.00	34.00	14	18	17
$TypeII$	35.41	20.53	16.15	7	178	140

Table C.3 Release 2: Training Set and Test Set Sizes

	Fault-prone	Not-fault-prone	Total
Release 2.1.2			
Model 1	95	809	904
Model 2	95	809	904
Model 3	760	809	1569
Release 2.1.3	25	877	902
Release 2.1.5	52	860	912
Release 2.1.6	867	50	917

Evaluation: The above models for revision 2 were tested with each of the subsequent three releases 3, 5, and 6. Table C.2 and Table C.3 show the results. The models had approximately 67% accuracy on the training set and misclassification rates of approximately 30%. The evaluation on 2.1.3 test set had slightly higher misclassification rates of approximately 40% and an accuracy of 65%. The model had better results with 2.1.6 test set, but poor results with the 2.1.5 test data. This suggests that the 2.1.5 release is really different in some way from the 2.1.1 and 2.1.2 releases.

C.2 Release 3 Models

In the third phase of modeling, we built three models based on release 2.1.2 as discussed earlier and evaluated it on the subsequent releases. Figure C.4, Figure C.5 and Figure C.6 show the cost ratio selections for these models. The equations of these models are as follows.

$$\begin{aligned}\log\left(\frac{p}{1-p}\right) &= -4.02 + 0.51\,\textit{FACTOR1} + 0.35\,\textit{FACTOR2} + 0.21\,\textit{FACTOR3}\\ &\quad + 0.21\,\textit{FACTOR6} + 0.26\,\textit{FACTOR7} + 0.18\,\textit{FACTOR10} \qquad \text{(C.4)}\end{aligned}$$

$$\begin{aligned}\log\left(\frac{p}{1-p}\right) &= -4.08 + 0.46\,\textit{FACTOR1} + 0.38\,\textit{FACTOR2} + 0.21\,\textit{FACTOR3}\\ &\quad -0.47\,\textit{FACTOR5} + 0.19\,\textit{FACTOR6} + 0.21\,\textit{FACTOR7}\\ &\quad +0.19\,\textit{FACTOR9} + 0.19\,\textit{FACTOR10} \qquad \text{(C.5)}\end{aligned}$$

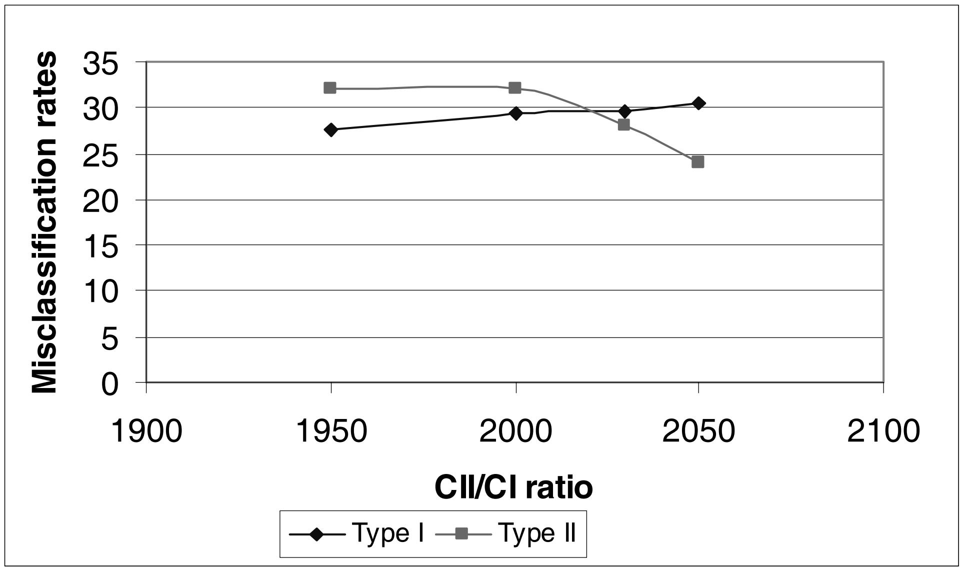

Figure C.4 Logistic Regression Modeling R3:Cost Ratio Selection for Model 1

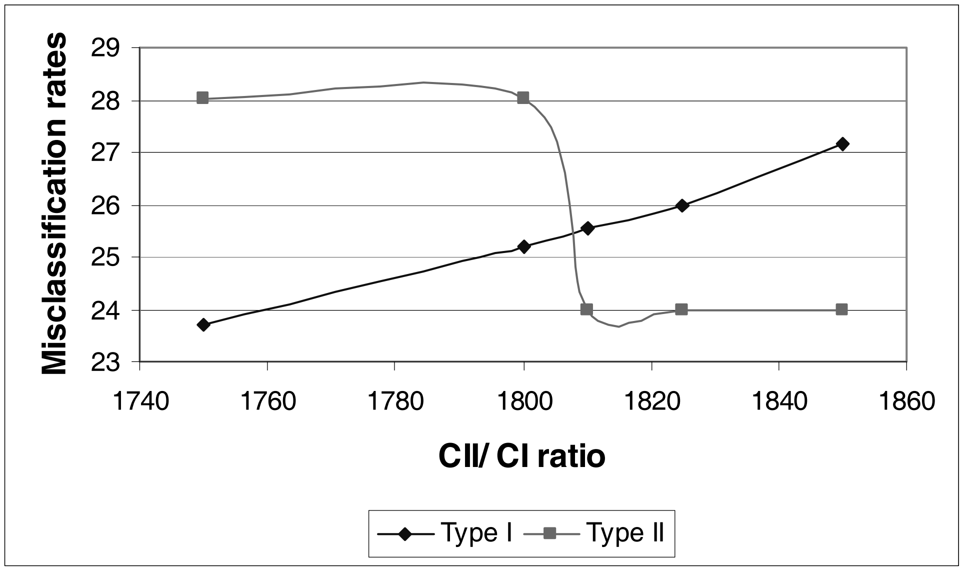

Figure C.5 Logistic Regression Modeling R3:Cost Ratio Selection for Model 2

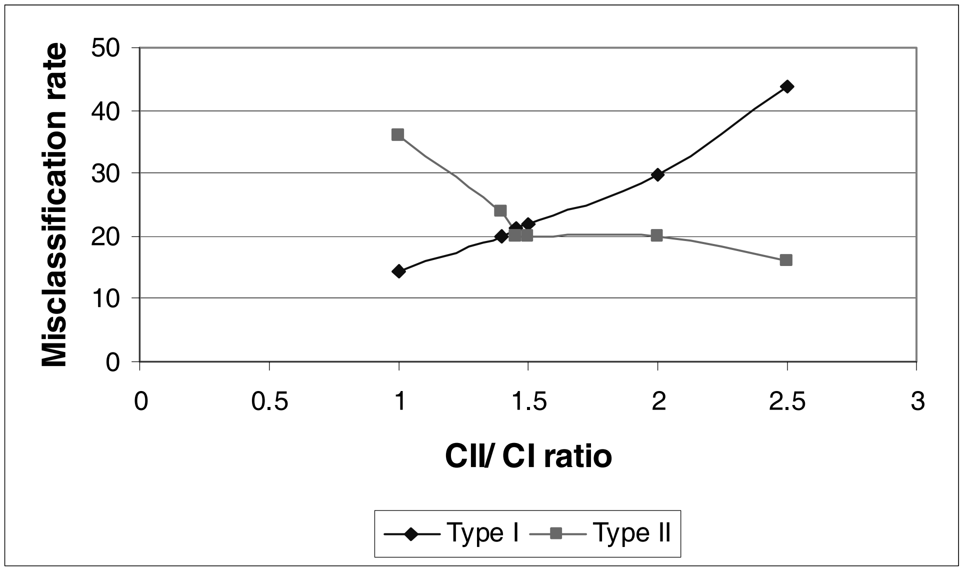

Figure C.6 Logistic Regression Modeling R3:Cost Ratio Selection for Model 3

$$\log\left(\frac{p}{1-p}\right) = -0.76 + 0.65\,\mathit{FACTOR1} + 0.38\,\mathit{FACTOR2} - 0.18\,\mathit{FACTOR5} + 0.25\,\mathit{FACTOR6} + 0.48\,\mathit{FACTOR7} - 0.63\,\mathit{FACTOR8} + 0.23\,\mathit{FACTOR9} + 0.55\,\mathit{FACTOR10} \tag{C.6}$$

Evaluation: The above models trained with 2.1.3 data were tested with each of the subsequent releases 2.1.5, and 2.1.6. Table C.4 and Table C.5 show the results for this evaluation. The training set had an accuracy of approximately 75% and misclassification rates of 25%. These are better than those found in the earlier training sets. The models performed relatively well on the 2.1.5 test data. It had an accuracy of approximately 65% and misclassification rates of approximately 35%. This suggests that closest match for 2.1.5

Table C.4 Logistic Regression Modeling Results: Trained on Release 3 (R3)

	Percent			Number of Modules		
	Model 1 (%)	Model 2 (%)	Model 3 (%)	Model 1	Model 2	Model 3
Training Accuracy 2.1.3						
Ok	70.51	74.50	79.45	636	672	1392
$TypeI$	29.53	25.54	24.00	259	224	185
$TypeII$	28.00	24.00	20.00	7	6	175
Test Accuracy 2.1.5						
Ok	65.46	74.56	62.83	597	680	573
$TypeI$	34.88	24.53	37.56	300	211	323
$TypeII$	28.85	40.38	30.77	15	21	16
Test Accuracy 2.1.6						
Ok	39.15	29.12	42.31	359	267	388
$TypeI$	24.00	18.00	16.00	12	9	8
$TypeII$	62.98	73.93	60.09	546	641	521

Table C.5 Release 3: Training Set and Test Set Sizes

	Fault-prone	Not-fault-prone	Total
Release 2.1.3			
Model 1	25	877	902
Model 2	25	877	902
Model 3	875	877	1752
Release 2.1.5	52	860	912
Release 2.1.6	867	50	917

is 2.1.3 in terms of the similarity of the cause-effect relationship for bugs. However, this model performed poorly with the 2.1.6 data.

C.3 Release 5 Models

In the last phase of modeling, we built three models based on release 2.1.5 and evaluated it on 2.1.6 release. Model 2 resulted in same equation as model 1. So we proceed with models 1 and 3 for further analysis. Figure C.7 and Figure C.8 show the cost ratio selections for these models. The equations of these models are as follows.

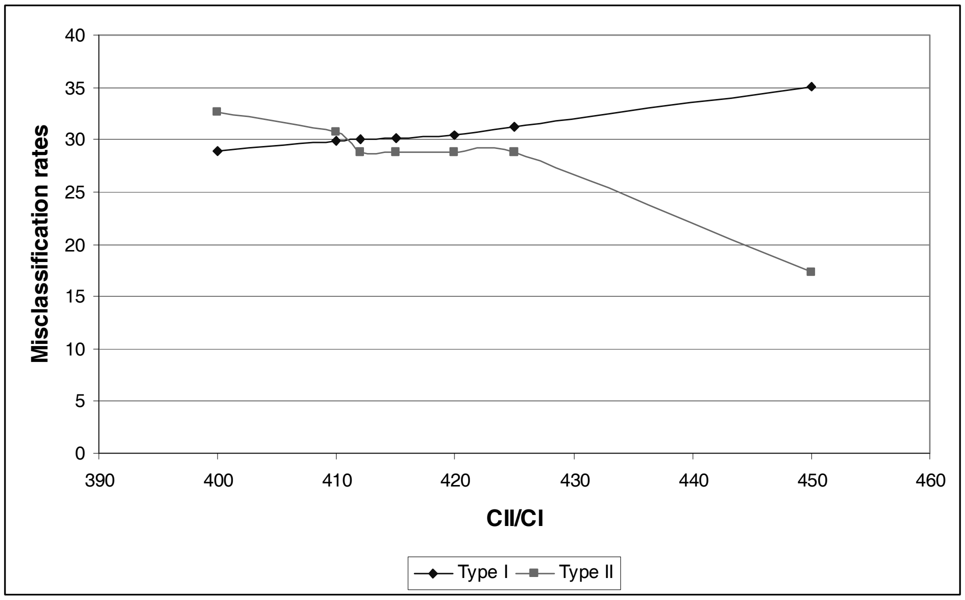

Figure C.7 Logistic Regression Modeling R5:Cost Ratio Selection for Model 1

$$\log\left(\frac{p}{1-p}\right) = -3.17 + 0.63\,\mathit{FACTOR1} + 0.34\,\mathit{FACTOR2} + 0.25\,\mathit{FACTOR3} + 0.23\,\mathit{FACTOR4} + 0.18\,\mathit{FACTOR8} + 0.12\,\mathit{FACTOR10} \qquad (C.7)$$

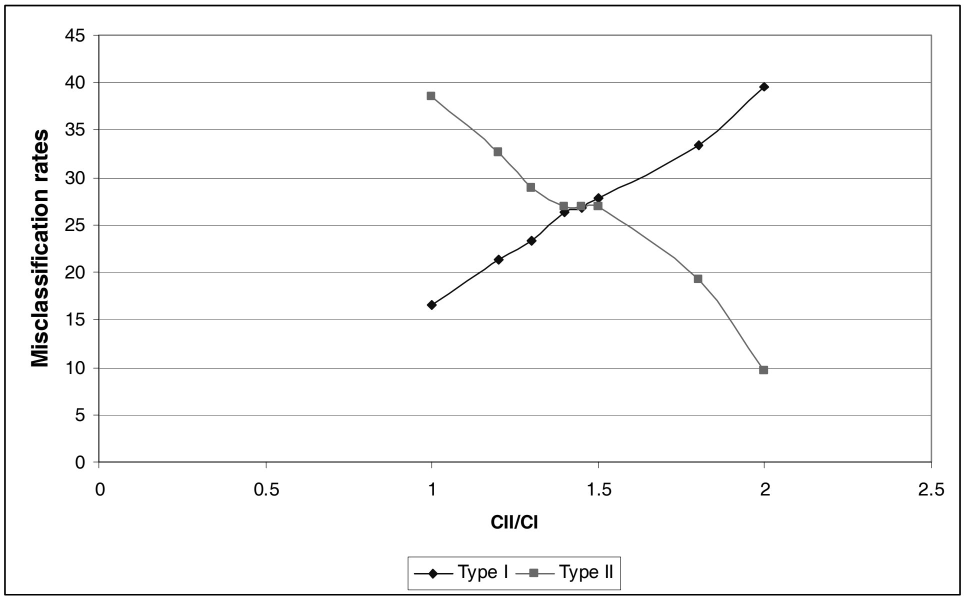

Figure C.8 Logistic Regression Modeling R5:Cost Ratio Selection for Model 3

$$\log\left(\frac{p}{1-p}\right) = -0.57 + 0.68\,\mathit{FACTOR1} + 0.46\,\mathit{FACTOR2} + 0.39\,\mathit{FACTOR3} + 0.19\,\mathit{FACTOR4} + 0.21\,\mathit{FACTOR6} + 0.39\,\mathit{FACTOR7} + 0.49\,\mathit{FACTOR8} + 0.22\,\mathit{FACTOR10} \qquad (C.8)$$

Table C.6 Logistic Regression Modeling Results: Trained on Release 5 (R5)

	Percent		Number of Modules	
	Model 1 (%)	Model 3 (%)	Model 1	Model 3
Training Accuracy 2.1.5				
Ok	70.07	73.17	639	1238
$TypeI$	29.88	26.74	257	230
$TypeII$	30.77	26.92	16	224
Test Accuracy 2.1.6				
Ok	13.30	6.65	122	61
$TypeI$	8.00	0.00	4	0
$TypeII$	91.23	98.73	791	856

Table C.7 Release 5: Training Set and Test Set Sizes

	Fault-prone	Not-fault-prone	Total
Release 2.1.5			
Model 1	52	860	912
Model 3	832	860	1692
Release 2.1.6	867	50	917

Evaluation: The above models trained on 2.1.5 data were then tested with release 2.1.6. Table C.6 and Table C.7 show the results for this evaluation. The training set had an accuracy of approximately 71% and misclassification rates of 29%. However, the model performed poorly on 2.1.6 test data. These results confirmed the hypothesis that release 2.1.5 is somehow different from the rest of the releases.