IMPROVING PREDICTIVE MODELS OF SOFTWARE QUALITY USING SEARCH-BASED METRIC SELECTION AND DECISION TREES

by

Rodrigo Vivanco

A Thesis submitted to the Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements of the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada
August 2010

Abstract

Software engineering is a human centric endeavour where the majority of the effort is spent understanding and modifying source code. The ability to automatically identify potentially problematic components would assist developers and project managers to make the best use of limited resources when taking mitigating actions such as detailed code inspections, more exhaustive testing, refactoring or reassignment to more experienced developers. Predictive models can be used to discover poor quality components via structural information from the design and/or source code.

There exist many traditional source code metrics to capture the size, algorithmic complexity, cohesion and coupling of modules. Object-oriented systems have introduced additional structural concepts such as encapsulation and inheritance, providing even more ways to capture and measure different aspects of coupling, cohesion, complexity and size. An important question to answer is: *Which metrics should be used with a model for a particular predictive objective?*

In machine learning, large dimensional feature spaces may contain inputs that are irrelevant or redundant. Feature selection is the process of identifying a subset of features that improve a classifier's discriminatory performance. In analysis of software system, the features used are source code metrics. In this work, an analysis tool has been developed that implements a parallel genetic algorithm (GA) as a search-based metric selection strategy. A comparative study has been carried out between GA, the Chidamber and Kemerer metrics suite (for an objected-oriented dataset), and principal component analysis (PCA) as metric selection strategies with different datasets.

Program comprehension is important for programmers and the first dataset evaluated uses source code inspections as a subjective measure of cognitively complexity that degrade program understanding. Predicting the likely location of system failures is important in order to improve a system's reliability. The second dataset uses an objective measure of faults found in system modules in order to predict fault-prone components.

The aim of this research has been to advance the current state of the art in predictive models of software quality by exploring the efficacy of a search-based approach in selecting appropriate metrics subsets for various predictive objectives. Results show that a search-based strategy, such as GA, performs well as a metric selection strategy when used with a linear discriminant analysis classifier. When predicting cognitive complex classes, GA achieved an F-value of 0.845 compared to an F-value of 0.740 using principal component analysis, and 0.750 when using only the CK metrics suite.

By examining the GA chosen metrics with a white box predictive model (decision tree classifier) additional insights into the structural properties of a system that degrade product quality were observed. Source code metrics have been designed for human understanding and program comprehension and predictive models for cognitive complexity perform well with just source code metrics. Models for fault prone modules do not perform as well when using only source code metrics and need additional non-source code information, such module modification history or testing history.

Acknowledgements

This work would not be possible without the loving support and encouragement of my family. A special thank you to Santiago and Athena for unknowingly sacrificing so much of their precious time.

Thank you to Dr. Dean Jin for your guidance and assistance throughout this undertaking. Your flexibility and willingness to accommodate my schedule meant a great deal to me.

Gracias a la vida, que me a dado tanto. Me dio dos niños preciosos con cuales realmente uno aprende lo mas importante de la vida. Me dio la oportunidad para seguir sueños y verlos ser realizado.... gracias.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software engineering has made large strides in recent decades in developing processes and tools that assist organizations in improving the quality of their products. Process and product metrics are used for three basic activities: understanding the process during development, controlling the process, and improving the process and product quality [FP97]. Pfleeger and Atlee present various views on software quality [PA06]; ranging from the metaphysical "transcendental view… we can recognize but not define", the user's value-based idea of quality ("fitness for purpose" and "the amount a customer is willing to pay"), to the product view of quality (its reliability, maintainability, efficiency, etc.). The ISO 9126 model identifies six major characteristics that can be used to assess the quality of a system [ISO91]. The major attributes are: *functionality*, *reliability*, *usability*, *efficiency*, *portability* and *maintainability*. Improving these aspects of the system will enhance the overall quality of the software product. In particular, improving a software system's reliability and maintainability will reduce overall development costs as system maintenance is ongoing and generally consumes a majority of available resources [PA06].

There are four major system maintenance activities: 1) *corrective maintenance*, in which faults not captured during testing are fixed; 2) *adaptive maintenance*, such as integrating new software or hardware sub-systems; 3) *perfective maintenance*, where new user requests are added and efficiency concerns are addressed; 4) *preventive maintenance*, which is aimed at averting potential faults and easing future maintenance tasks. Poor software design and implementation result in problematic components that will require additional corrective, perfective and preventive maintenance costs.

Being able to identify potentially problematic components during development would assist developers and project managers to make the best use of limited resources during maintenance, especially when carrying out mitigating actions such as increased code inspections, refactoring, and more exhaustive testing. Predictive models of software quality have been a focus of research in many empirical studies of software systems. As categorized by Briand *et al.* [BAC+99], the majority of these studies attempt to associate a causal relationship between external measures of quality (such as, for example component faults), with a system's structural metrics obtained from design and/or source code.

Various organizations have adopted object-oriented technologies for product development and numerous studies have been conducted into the formulation of object-oriented metrics to quantify various aspects of a system's design and implementation. As one of the first such studies, the most often cited reference on object-oriented design measures, is the work by Chidamber and Kemerer [CK94]. The CK metrics suite introduces six metrics that attempt to capture the coupling, cohesion, inheritance and complexity in object-oriented designs.

There have been many more object-oriented metrics proposed. Briand *et al.* have identified at least 15 different cohesion and 30 coupling measures in the literature [BDW98, BDW99]. New metrics are still being defined, ranging from static measurements that take into account dependent instance variables for cohesion [CKB04] to dynamic measures that use runtime object interactions as factors in coupling metrics [ABF04].

Many of the proposed product metrics measure similar structural aspects of the system and some of them may be redundant or even irrelevant for a given predictive objective. For example, various complexity measures, such as the Halstead metrics, can be positively correlated with simple size measures such as the number of lines of code. Emam *et al.* have reported on the confounding effect of size on various object-oriented metrics [EBGR01].

Using all the available metrics, some of which are redundant or irrelevant, will have a detrimental impact on a model's predictive accuracy. As well, different projects and predictive objectives may use different metrics for optimal performance. Clearly, selecting an effective metrics set to use with a predictive model is of vital importance and various techniques have been applied to settle on a collection of metrics to use with a predictive model.

The easiest metric selection approach, which is still used by some practitioners, is to apply a “standard” set of metrics, such as the CK metrics suite, without any consideration of the predictive objective. More sophisticated practitioners tend to generate their own subset, drawn from various standard sets and experience. Statistical methods, such as univariate logistic regression and correlation to the predictive objective (dependent variable), have been used by researchers to identify potentially useful metrics to utilize in the final model. However, it does not take into account the discriminatory power of using combined measures. Principal component analysis [Dun89], a multivariate statistical approach that attempts to reduce a large feature space and still capture the original dataset information in terms of variance, has also been used as a data-driven metric selection strategy.

Harman [Har07] has illustrated how search-based optimization is becoming more widely applied to software engineering problems. They have been used to assist with testing, release date planning, and cost estimations, but not in the selection of source code metrics to use with predictive models of software quality. In this work, a search based approach using a genetic algorithm [Gol98] is proposed as a means of heuristically identifying combinations of source code metrics that improve a predictive model’s ability to discover potentially problematic components.

The first aim of this research is to evaluate the efficacy of using source code metrics selected by a search-based metric selection strategy; will the metrics selected by the genetic algorithm improve predictive models of quality? In order to test this hypothesis two non-trivial datasets will be investigated which capture two distinct aspects of product quality.

Software maintenance is a human centered endeavour where developers have to modify existing code, of which the developer is not always the originator. In this case quality should be measured from the programmer’s perspective. In particular, how difficult it is to comprehend a module in order to correctly and efficiently modify the code to carry out future maintenance tasks. The first dataset is a Java application that uses a subjective measure of cognitive complexity for the purpose of maintenance as the predictive objective.

Cognitive complexity of source code is a predictive objective of importance to developers, but for users (major stakeholders in all applications), system faults are even more critical. As such, the second dataset is an Eclipse plug-in that uses faulty or not faulty modules as the predictive objective.

The second aim of this research is to identify the types of metrics that are most useful in certain predictive models. For this goal, the metrics identified as good predictors via the GA metric selection have been further scrutinized with a decision tree classifier to categorize which types of source code metrics have the largest impact on program comprehension, and which static metrics seem to be most useful in predicting fault-prone modules.

System development follows a basic sequence of steps carried out in an iterative fashion: requirements, design and implementation. Since it is desirable to find problems as early as possible in the development cycle, it implies using design metrics instead of source code metrics for all predictive models. However, there are several reasons for choosing source metrics over design metrics for certain types of predictive objectives.

It is known that, due primarily to maintenance tasks, a system will evolve and become more complex over time [LB85]. As well, design changes and source code changes tend to be performed independently of each other, resulting in “design drift” [MNS01]. That is, there are discrepancies between design documents and the actual source code that is executed. Ideally design documents should match the source code, but in reality this is not always the case.

The compilation and linking process is automatic from source code, the last artifact touched by developers. Faults are more likely to be introduced by and bugs not found during testing will be manifest themselves during execution of the binary program. Therefore, for identifying fault prone modules, models based on source code measurements will be a most accurate depiction of what a client will be using, the binary build from source code as modified by developers.

Programmers in charge of maintenance work mostly with source code. Therefore, predictive models of program comprehension will be more accurate if source code measures are used over design metrics, where it is difficult, if not impossible, to measure certain aspects of a module, such as algorithmic complexity and programmer documentation.

This document presents the research carried out for the completion of the thesis and is organized as follows: Chapter 2 is the general background literature review required to formulate the motivation for the research and proposed solution. It includes a discussion on the various object-oriented metrics that can be calculated from source code, a summary of how metrics have been used with predictive models of software quality, a review of the metric selection methods that have been applied to improve predictive models, and a synopsis of search-based techniques have been used in software engineering.

Chapter 3 recaps the salient points of Chapter 2 and establishes the problem statement. In essence, with the large number of metrics available to predictive model developers, there is a need to find effective strategies of metric selection that takes into account the predictive objective and gives the best predictive performance.

Chapter 4 elaborates on the proposed solution and evaluation criteria. A search based selection strategy based on a genetic algorithm will be compared against the well known CK metrics suite and Principal Component Analysis (PCA). Linear Descriminant Analysis (LDA) is used as the predictive model and evaluated using the resulting F-value of N-fold validation.

Chapter 5 investigates an object-oriented dataset on cognitive complexity for the purpose of maintenance by using the proposed solution. Chapter 6 carries out additional analysis on the dataset whose modules have been categorized as being faulty or not faulty using the proposed solution. Chapter 7 concludes with a summary of the research and outlines future work

Chapter 2

Previous Work on Metric Selection

Predictive models of software quality can be built using product metrics such as source code measures. Three basic concepts implicit in these types of predictors need to be elaborated on.

- How are the various metrics calculated?
- How are they validated and used in predictive models?
- How/why are particular metrics chosen as inputs to predictive models?

A general overview of the various source code metrics that have been used with predictors of software quality, with a particular focus on object-oriented measures, is presented in Section 2.1. Section 2.2 summarizes some of the relevant research into empirical validation of object-oriented metrics in developing predictive models of software quality. Section 2.3 describes the various metric selection strategies that have been applied in choosing metrics to be used with predictors. Finally, to illustrate the emergence of search-based techniques in software engineering, Section 2.4 recaps some of the current research carried out in this domain.

2.1 Object-Oriented Metrics

One of the first attempts to measure a system's apparent cognitive complexity was to simply count the lines of code (LOC), not including blank lines and comments. The reasoning being that larger programs are more difficult to fully comprehend and maintain. Additional metrics were developed for procedural programs in an attempt to capture their algorithmic complexity at an operational level on a per function/module basis. The most commonly used metrics are the Halstead [Hal77] and the McCabe Cyclomatic [McC76] complexity metrics.

Halstead metrics attempt to measure computational complexity by taking into account the operators and operands in a segment of code. The various Halstead metrics are calculated from four distinct measures:

- The number of distinct operators.
- The number of distinct operands.
- The total number of operators.
- The total number of operands.

For example, the Halstead *program length* is the sum of the total number of operands and operators, while the Halstead program volume is defined as the program length multiplied by the $\log^2$ of the *program vocabulary* (the sum of the distinct operands and distinct operators). It is easily observed that the Halstead metrics are strongly correlated to size. The more lines of code, the more operands and operators are used and thus larger Halstead values will be computed.

The McCabe Cyclomatic complexity metric is based on the decision structures of the implementation and its aim is to capture the cognitive complexity of a section of code using branching constructs such as ‘for’, ‘do’, ‘while’, ‘switch’, ‘if-else’ and ‘catch’ (for languages that support exception handling). It is assumed that the more branching paths in a segment of code, the harder it is to follow the flow of control, increasing the cognitive burden on developers.

The concept of *coupling*, the amount of interdependency between modules, has long been recognized as an important factor in maintenance tasks. A system that exhibits too much coupling will be difficult to maintain since changes in one component may directly and indirectly influence other parts of the system. In procedural programs, coupling can be measured by *fan-in*, the number of procedures that call a given module, and *fan-out*, the number of procedures called by a given module. Traditional procedural metrics can be grouped into three general categories; measures of size, complexity and coupling.

Object-oriented programming introduces a new paradigm. The various metrics that can be computed may be categorized into several conceptual groupings that industry and the research community see as important to quantify as they are indicators of good system design and implementation. These general categories are *size*, *encapsulation* (data hiding), *inheritance*, *coupling*, *cohesion*, method implementation *complexity* and code *readability* (comments, formatting and meaningful names most commonly identified as desirable code readability aids).

Like in procedural programs, size also influences object-oriented systems. The larger the system, the harder it is to fully understand and maintain. Size measure can be done at the system level, such as the number of classes, packages and overall lines of code. They are also measured at the class level. Usually size is computed directly by LOC in a class. Since different programming languages and styles may result in bloated or condensed LOC, the number of tokens may also be counted. Size can also be done at the method level, such as the mean LOC per method.

One of the advantages of using an object-oriented programming language is encapsulation, the grouping together of related attributes and functionality. Access levels are associated with object’s attributes and methods in order to facilitate the concept of data hiding. Allowing unfettered access to data is considered detrimental to good system design and it is something that should be controlled. Encapsulation and data-hiding metrics are simple count measurements, such as the number of public attributes or the percentage of public methods in a class.

Inheritance, the ability for a class to integrate a parent’s attributes and methods into itself, is a key concept of object-oriented programming. It facilitates code reuse and, with dynamic binding via virtual functions, it supports polymorphism. However, inheritance also introduces added cognitive complexity for a developer, in particular dynamic binding. Subclasses may override and overload methods, altering the behaviour of the parent class. If they have direct access to the parent's attributes, subclasses can modify data and perhaps circumvent data checking code leading to potential run-time faults.

Coupling attempts to measure the degree of interdependency between classes. It is generally agreed that high levels of coupling degrade system quality making a class more difficult to maintain, as changes may have to be propagated across the system. In object-oriented systems there are more ways to couple components than in procedural programs. Briand *et al.* [BDW99] categorized 30 coupling metrics, which include method call coupling (analogous to fan-in and fan-out in procedural programs), parameter list coupling, inheritance coupling, attribute coupling and friendship coupling (for C++).

Cohesion is a difficult concept to measure as it tries to quantify the degree to which the methods and attributes in a class "belong" together to fulfill a particular functionality. The notion of "belonging" is highly subjective and depends on programmer intent, something very challenging to measure from the design or the source code. The basic premise behind measuring cohesion is that high quality modules should exhibit high cohesion, that is, it should provide a well-defined, focused set of services.

One of the ways to quantify cohesion in object-oriented programs is to measure the number of attributes that methods have in common. A class that has many attributes and public methods may be a "god" class and provide far too many diverse services (it is likely to exhibit low cohesion and high coupling). Inheritance and polymorphism introduce additional degrees of freedom for quantifying cohesion. Briand *et al.* [BDW98] have identified at least 15 different ways of measuring cohesion in the literature.

There are many measures of inheritance, such as the number of parents, the number of children, the number of siblings and the number of overridden methods. The main motive for computing inheritance metrics is the belief that classes deep in the inheritance chain have decreased cohesion (as it inherits all of the public interfaces). Coupling can also be increased via overridden methods and dynamic binding. Thus, long inheritance chains have a detrimental impact on product quality due to reduced cohesion and increased coupling.

Most implementation complexity metrics in object-oriented systems are based on the McCabe and Halstead metrics but are applied at the method level. Overall class complexity incorporates the method-based metrics into one measure for the class, such as the weighted method complexity metric from the CK metrics suite.

A final category of metrics, useful for maintenance tasks, are descriptive/readability metrics. Measures of source code comments are straightforward to calculate and can be done at the class or method level. Though difficult to measure, descriptive method and attribute names are invaluable for programmer understanding. Current research into this area uses natural language techniques to build domain specific programmer lexicons [HO07, AGMT07] and translate terse identifiers into meaningful natural language labels [LFB07]. A surrogate metric is simply to count the number of characters in an identifier, on the assumption that the longer the name the more meaningful it is likely to be.

The seminal work by Chidamber and Kemerer [CK94] introduced six design metrics which were theoretically based on the ontology of Bunge and empirically evaluated on two systems using basic descriptive statistical measures such as histogram distributions, median, min and max values. The CK metrics are commonly cited in metrics research, as it was one of the first studies to focus on object-oriented measures of coupling, cohesion, inheritance and complexity.

The following is an abbreviated description of the CK metrics. The weighted methods per class metric (WMC) is a complexity measure based on the number of methods in a class weighted by a functional complexity measure. As the original paper presented these metrics for designs, the complexity measure is not specified. When calculating the WMC in source code, McCabe's complexity measure is commonly used. The depth of inheritance tree (DIT) is based on the assumption that the deeper a class is in the inheritance hierarchy, the harder it is to easily predict its behaviour due to decreased cohesion and increased coupling. Another inheritance metric is the number of children (NOC), where a large number of children may be an indication of the misuse of inheritance. The coupling between objects (CBO) metric measures the connection between classes; it is a count of the methods of one class that use methods or instance variables of another class. The response for a class (RFC) is another coupling metric that counts the "methods that can potentially be executed in response to a message received". This is analogous to fan out.

Cohesion is measured by CK as the **deficit** of cohesion. LCOM (lack of cohesion in methods) counts the number of method pairs that use the same instance variables with the assumption that a class that has too many methods that do not share instance variables is an indication that the class is not cohesive enough, that is, it provides too many diverse services and is an indication of poor class design. This rationale misses the fact that certain types of classes, for example data models, will naturally have many methods that do not access many of the data members, being mostly get/set methods with data checking code. These classes will exhibit low cohesion as measured by LCOM, but are in fact cohesive from a programmer's point of view, since the data members are needed to describe all the metrics in an object. Since its introduction, the CK metrics have been adapted by others to produce new metrics for coupling, cohesion, inheritance, and implementation complexity.

Li [Li98] evaluated the CK metrics and proposed a new set of metrics that refined the concepts of inheritance, implementation complexity, cohesion and coupling for object-oriented systems. The metrics are: NAC, the number of ancestor classes, is an alternative to DIT that takes into account the multiple inheritance capabilities of C++. NDC, the number of descendent classes, is an alternative to NOC and has a broader concept of the influence of the class and includes all subclasses that could be reached through a directed graph. NPM, the number of public methods, is a cohesion type of metric with the rationale that the more methods available to outside clients, the harder to comprehend a class' behaviour. CMC, the class method complexity, is a measure of the internal complexity of a class and includes all methods, public or private. CTA, the coupling through abstract data types, along with the coupling through message passing (CTM), are replacements for the CBO metric introduced in CK.

Harrison *et al.* developed the MOOD set (Metrics for Object-Oriented Design) [HCN98] which specifies metrics for encapsulation, inheritance, coupling and polymorphism, which can be defined at the class level and system level. Method hiding factor (MHF) and attribute hiding factor (AHF) are joint metrics of encapsulation that measure the percentage of a class/system that is concealed from external clients. Method inheritance factor (MIF) and attribute inheritance factor (AIF) measure the percentage of a class' methods and attributes that have been inherited from a super class. The coupling factor (CF) is calculated by considering all pair wise classes that are associated by message passing or semantically (reference to a method or attribute) without considering inheritance coupling. The polymorphism factor (PF) is the ratio of methods that have been redefined from a parent class to the total number of methods in a class.

There have been many more object-oriented metrics proposed over the decades [CL93, EBD99, BEDL99, CKB04, GS06]. A survey published in 2003 catalogued 164 class metrics and 36 method metrics [PV03]. The next section gives an overview of how some of these metrics are validated empirically and used in predictive models of software quality.

2.2 Validation of Metrics and Prediction Models of Quality

There are many ways to compute the various product metrics and there needs to be a way to assess the applicability of the metrics to system development. Basili and Rombach [BR88] introduced the Goal Question Metric (GQM) paradigm to assist with formulation and evaluation of metrics for the planning and analysis of software projects. They advocate that metrics should be developed to answer a specific question and validated against the desired goal. For example, if the goal is to ease maintenance tasks, one question would be how to measure coupling, as high coupling is detrimental to maintenance. Metrics should then be designed to quantify the level of coupling and validated by correlating the metric to the desired maintenance task, such as refactoring. A summary of various investigations of existing object-oriented metrics and their use in predictive models of software quality is presented in order to illustrate the validation methodologies commonly used in empirical software engineering.

Basili *et al.* [BBM96] did a controlled empirical study into the validation of the CK metrics for fault prediction models. Groups of randomly selected undergraduate and graduate students in a Software Engineering course implemented a small C++ application. A team of experts then tested the systems for faults. Univariate logistic regression (fault vs. no-fault) was performed on the combined 180 classes using the CK metrics. Five of the six metrics had a significant relationship with fault-prone classes in the systems implemented by the students; lack of cohesion (LCOM) was the exception. Seeing how difficult it is to properly quantify cohesion, this finding is not surprising.

Multivariate logistic regression analysis was used to develop a fault-prone prediction model with the five metrics that showed a positive correlation. A linear regression model using three non-object-oriented source code metrics was compared against the CK-based model. The procedural code metrics were: *MaxStatNext*, which measures the maximum number of statement nesting in a class; *FunctDef*, which is the number of function declarations; *FunctCall*, which measures the number of function calls. The models were compared using correctness (percentage of classes correctly predicted as having faults) and completeness (percentage of total faults detected). The model that used the five CK design metrics had a correctness of 60% and completeness of 88%, compared to 45.5%

and 83% for the model that used the three non objected-oriented metrics. They concluded that the five CK object-oriented design metrics are better predictors of fault-prone modules than the three non objected-oriented metrics.

Tang *et al.* [TKC99] reported on an empirical study to validate the CK metrics on three subsystems of a client server application developed using C++. System A consisted of 20 classes and 5,600 lines of code, System B had 45 classes and 21,000 lines of code, while System C had 27 classes and 16,000 lines of code. They used faults as the dependent variable divided into three types, object-oriented faults (e.g. misusing parent data members), object management faults (e.g. errors in the object copy method), and "traditional" faults (faults that can occur in non object-oriented systems). They used univariate logistic regression to validate the CK metrics with Systems A, B and C. Their univariate analysis showed that only WMC and RFC were significant predictors of OO faults for the systems evaluated. They suggest that additional metrics may be needed for better fault prediction. They advanced four new metrics to supplement the CK suite: the average method complexity (AMC), which is a size measure in lines of code; two coupling measures, inheritance coupling (IC) to its parent and the coupling between inherited methods and new/redefined methods (CBM). In an attempt to measure the dynamic aspects of a system, the number of object/memory allocations was measured (NOMA). Univariate analysis with Systems A, B and C show that AMC, IC and CBM were significant OO fault indicators for all three systems, while NOMA was significant only for System A and B.

Hypothesizing that the structural properties of a system have an impact on cognitive complexity, and therefore its fault proneness, Emam *et al.* [EMM01] built a predictive model for fault-prone classes in a Java word-processing application using only three design metrics. A faulty class had at least one reported field failure. They started with the six CK metrics suite and a subset of 18 coupling metrics described in [BDW99]. They were only interested in design metrics and reduced the potential metrics set to 10, two from CK (NC and DIT) and eight from [BDW99]. With the aid of descriptive statistics (mean, standard deviation, median, inter-quartile range and the number of observations not equal to zero), four coupling metrics were eliminated from further analysis because too many observations had zero values.

Univariate linear regression was used to identify potentially useful metrics for a multivariate model. They incorporated the number of class attributes (ATTS) in the model to account for the possible confounding effects of size. Four metrics were identified as having significant association with fault-proneness: depth of inheritance tree (DIT); external coupling of class attributes (OCAEC); external coupling of methods (OCMEC); and internal coupling of methods (OCMIC). OCAEC had a large standard deviation and was sensitive to a minority of the observations and was removed from further analysis. OCMIC was removed from the multivariate linear regression model because it had a small change in odds ratio. The three metrics used for building the predictive model were DIT, OCMEC and the size metric ATTS. Leave-one-out cross validation was used on the training set, in version 0.5 of the application, and version 0.6 was used as the test set. The resulting model resulted in a sensitivity of 77% and specificity of 82%. Sensitivity is defined as the percentage of classes correctly identified as fault-prone, and specificity measures the percentage of correctly identified no-fault classes.

Briand *et al.* [BWDP00] performed a detailed analysis of an industrial C++ system consisting of 83 classes and nearly 40,000 source lines. The application had been in development for seven years and six programmers with several years of system development experience worked on the project. The study focused on design metrics, obtaining from the literature 28 coupling measures, 10 cohesion measures, 11 inheritance measures and five size measures. Even though the possible relation between cognitive complexity and system faults was acknowledged, only the objective measure of faults in a class was considered. Descriptive statistics and outlier analysis were performed to eliminate data samples that may bias the predictive model; univariate analysis was performed on each metric to decide if it should be included when building predictive models; multivariate logistic linear regression was used to construct the predictive model; as an additional step, the measures were correlated against the size of the class (measured as the number of methods implemented) in order to study the relationship between size and the metrics.

Three multivariate models, each using different metrics sets, were compared. The first model was constructed using only size measures, the second model using subsets of coupling, cohesion and inheritance measures, and the third model using subsets from all categories of measures. With the small number of size measures, a backward stepwise selection process was used for Model I. The model only used two size metrics due to the highly correlated measures. Model II used forward stepwise selection on the metrics chosen with univariate analysis. Five import coupling measures and one cohesion measure were used. No inheritance metrics were included in Model II. Model III also used forward stepwise selection. One cohesion metric, two size measures, and three import metrics were identified as having the strongest predictive power. Model I resulted in 62% correctness and 80% completeness, Model II had 91% correctness with 97.7% completeness, and Model III resulted in 87% correctness and 96% completeness. Based on the analysis, the authors concluded that coupling is a structural design measure that should be considered when building predictive models of fault-prone classes, method invocation import coupling in particular. They also recommend that export coupling should be measured separately from import coupling.

Emam [Ema00] summarizes a commonly used methodology in research and industry for validating metrics and building predictive models. In this methodology each available metric is individually validated with the dependent variable, but he strongly advises that possible confounding effects, such as size, also be considered in this step. A metric that shows a positive relationship (logistic regression or correlation) to the dependent variable (predictive objective) that is both statistically relevant and whose magnitude of the relationship is significant should be considered when building the predictive model. A critical step for building effective classifiers is choosing the metrics to use and Emam suggests using a statistical approach. The various metric selection techniques used in software engineering are summarized in the next section.

2.3 Metric Selection Strategies in Software Engineering

With the numerous source code metrics available, choosing an effective subset for a particular project and desired objective can be challenging. Some of the approaches used by practitioners are: use a standard set such as the CK metrics suite; design a subset based on personal experience, preference or predictive objective; use univariate statistical methods to identify potentially useful metrics, followed by greedy selection; or use principal component analysis, a multivariate statistical approach.

The easiest metric selection strategy is to use a "standard" set of metrics such as the CK metrics suite. These metrics have been statistically and empirically validated to be useful with various datasets (mostly in defect prediction). Since they attempt to capture some of the key concepts of high quality software, such as coupling, cohesion, complexity and inheritance, practitioners that use these sets assume that they are sufficient for their project and predictive objective. Pai and Dugan [PD07] used the CK metrics, plus a size measure (source lines of code per class) as product metrics into a Bayesian network to compute the probability that a class is fault prone.

Other practitioners tend to generate their own subset (drawn from various standard sets) based on personal experience, preference or predictive objective. Lanza and Marinescu [LM06] devised a metrics "pyramid" to characterize and evaluate a system for potential design problems. They suggest using size and complexity metrics, along with coupling metrics, to form the base of a structural pyramid that should carry the highest weight. Inheritance metrics, the next step in the pyramid, can then be used to further characterize the system. They use the CK metrics for coupling and inheritance as well as some standard procedural measures adapted to OO systems, such as McCabe's Cyclomatic complexity, lines of code, number of packages (for Java based systems) and fan out. These metrics are then used with visualization tools to help project managers target areas of the system that exhibit overly complex designs and need further attention. Alshayeb and Li [AL03] used three CK metrics (WMC, DIT and LCOM) and three metrics from [Li98] (number of local methods, coupling through data abstraction and coupling through message passing) to build multiple linear regression models to predict maintenance effort using two development cycles (an XP iterative process and a long-cycled iterative process). They chose these metrics because "of their wide acceptance among the software engineering community".

Munro [Mun05] used the CK metrics as a starting point to develop simple decision models to identify "bad smells" that are associated with future refactoring efforts. For example, to locate classes that exhibit the *lazy class* bad smell (a class that isn't doing very much and should be eliminated), the following metrics were used: coupling between objects, weighted methods per class, lines of code, number of methods and depth of the inheritance tree. The models were empirically validated with two Java applications.

By far the most common metric selection strategy currently used in empirical software engineering, as outlined by Braind and Wust [BW02], is to use univariate statistical techniques (such as correlation or univariate regression) to identify **potentially** useful metrics from a set and then use greedy selection to build the multivariate model. This approach has been used since the 1990's by Basili *et al.* [BBM96]. Gyimothy *et al.*

[GFS05] reproduced the metric validation strategy outlined in [BBM96] on a C++ open source application (Mozilla). They used the CK metrics suite as well as an additional cohesion metric and LOC. Logistic univariate regression was used to identify metrics that could potentially predict fault-prone classes. They used neural net and decision tree as the predictive model. Forward and backward selection was used to build the multivariate models.

Shatnawi and Li [SL08] used 12 metrics for building a model of fault-proneness. The set consists of five CK metrics (LCOM was omitted as their tool did not collect the revised version), two additional coupling metrics from Li [Li98] (coupling through data abstraction and coupling through message passing), and four metrics from [LK94] (number of attributes, number of operations, number of added methods and number of overridden methods). They analyzed source code from three Eclipse releases. They used univariate binary regression to determine an association between a metric and faults. They then performed Pearson's correlation between the independent variables, as well as variance inflation factor, to eliminate metrics that exhibited high collinearity. Finally, forward stepwise selection was used on the remaining seven metrics to build the multivariate logistic regression predictive model.

In forward stepwise selection, each metric is added to the predictive model, one at a time, and if it improves its performance it is kept, otherwise it is rejected. However, it could be that a rejected metric would have improved performance if combined with a metric not yet included in the model. Backward stepwise elimination starts with all the metrics and each metric is tentatively removed and the model tested. The metric that has the least positive impact on the model's performance is permanently eliminated from the model. Since the metrics are removed one at a time, this approach is sensitive to local minima, as the process stops when model performance is not improved.

Another statistical method used in metric selection is principal component analysis [Dun89]. PCA is a multivariate statistical approach that attempts to reduce a large feature space and still capture the original dataset information in terms of variance. PCA considers all of the available metrics when calculating a principal component (PC). The first PC is the linear combination of the metrics that explains the maximum amount of variance in the normalized dataset. The second and subsequent PCs are also linear combinations with the additional restriction that they are orthogonal to each other. The coefficients (loading factors) in the PC indicate the importance of a metric in explaining the variance. Metrics with high coefficients contribute more towards explaining the variance of the PC and are candidates to be used in the predictive model. Figure 1 shows an example plot of a five metric set after PCA is carried out.

In Figure 2.1, there are five metrics and five PCs. Each PC explains a certain percentage of the variance in the dataset, and each metric has a loading factor for each PC. A common approach when applying PCA for dimensionality reduction is to use only the PCs that explain the majority of the variance, 90% of the total variance for example. In the example shown in Figure 2.1, only the first three PCs would be considered.

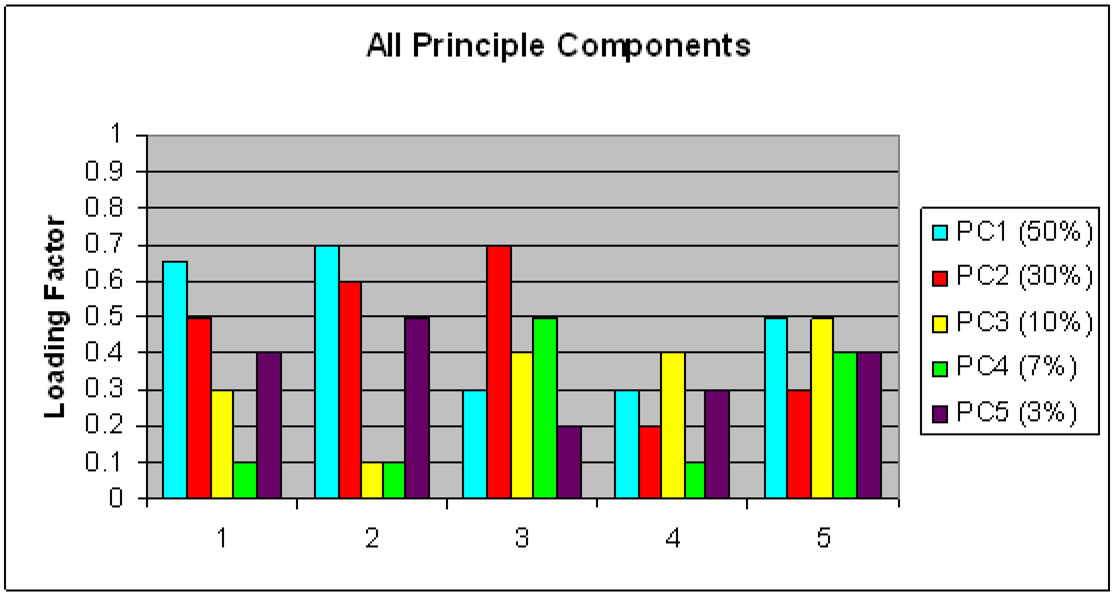


Figure 2.1: PCs and loading factors for five metrics before rotation

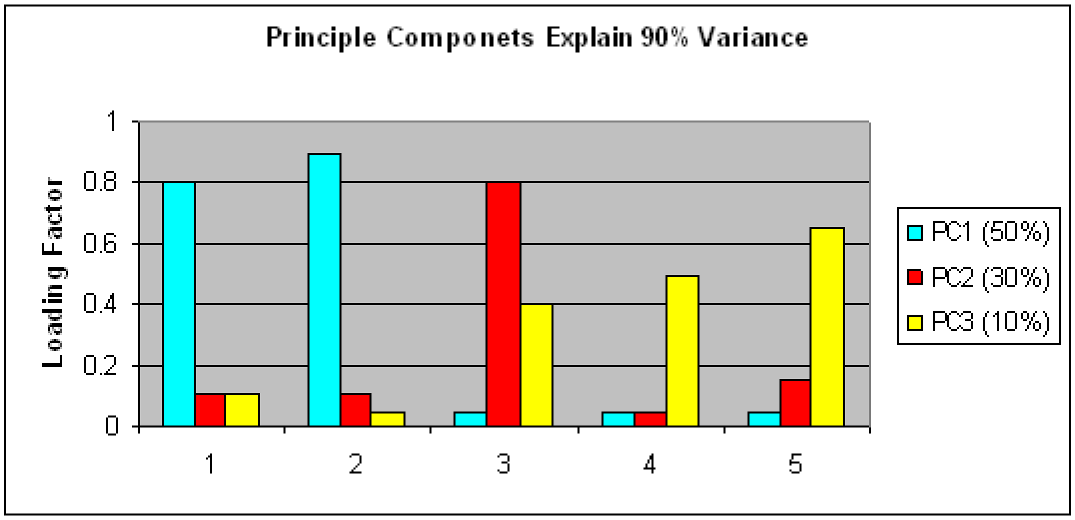


Figure 2.2: Metrics 1, 2 and 3 pass the threshold of 0.7, and will be used with the predictive model

When using PCA for metric selection, the metrics in the PCs must be inspected and chosen to be used with the predictive model. Varimax rotation can be applied to the PCs to enhance the spread between the loading factors of the individual metrics that contribute the most to a particular PC. Metrics with a loading factor above a threshold (0.7 is commonly used by researchers that perform PCA on metric sets) are selected for further analysis. Figure 2.2 illustrates the effects of varimax rotation. In the example, the metrics in the first three PCs (the ones that explain 90% of the variance) are rotated, and three metrics pass the threshold, M1, M2 and M3.

There are three basic ways to use PCA with predictive models. One way is to use the metrics as identified in Figure 1.1 with no further processing. Thwin and Quah [TQ05] followed this approach. They started with the CK metrics, additional coupling metrics from [TKC99] and two size metrics (LOC, number of attributes and methods). They performed PCA analysis and selected the top metric from the three components that explained 90% of the variance. The three metrics were used as inputs into two neural net architectures, a general regression neural net and Ward neural net. They built predictive models for number of faults in a manufacturing system consisting of 97 C++ classes. They also built predictive models for maintenance effort (using the number of modified lines of code as the dependent variable) for a Quality Evaluation System implemented in Class-Ada.

The second approach is to use PCA, as illustrated in Figure 3, as a starting point to identify the metrics to be used in further analysis, such as stepwise selection. Fioravanti and Nesi [FN01] used metrics from a C++ system developed by undergraduate/graduate students as used in [BBM96]. They started with 45 functional metrics, 49 structural metrics, 52 coupling and cohesion metrics, 45 OO size and complexity metrics and 35 cognitive metrics. Using PCA, with a varimax rotation threshold of 0.6, they reduced the set to 68 metrics. They then built various multivariate logistic regression models of 50 metrics, taking out 18 metrics every time (it was not an exhaustive search). From the best subset of 50 metrics they then performed forward stepwise selection to arrive at a model that used seven metrics. The resulting model achieved accuracies slightly better than what was reported in [BBM96].

The third way to use PCA is as a feature reduction technique, also known as a filter approach. This strategy was followed by Nagappan *et al.* [NBZ06] in an empirical study of five Microsoft projects. They obtained 33 metrics for each module in a project. Each module consists of many classes. They used three module metrics (number of classes, number of functions and number of global variables). At the function level 11 metrics were obtained (such as number of lines, number of parameters, fan in, fan out, and McCabe's Cyclomatic complexity) and for each metric they used the maximum and the total as different measures resulting in 22 metrics. Four class level metrics were considered (number of methods, inheritance depth, class coupling and number of subclasses) which were also divided into maximum and total resulting in eight measures. They used Spearman's correlation to determine which metrics in each project are associated with the predictive objective (errors reported from the field). Each project had a slightly different subset when a correlation threshold of 0.5 was used. Project A had 6 metrics, 29 metrics in project B, 33 metrics in project C, project D had one metric (maximum lines of code), and project E had 9 metrics. They concluded that different projects need different metrics.

They wanted to build multivariate predictive models but did not use the metrics identified by correlation analysis as many of the metrics exhibited multicollinearity. They used PCA to overcome this problem. For each project they chose the PCs that explained at least 95% of the variance. They then used these PCs **directly** as the independent variables to a multiple regression model. They found that the predictive model for one project did not work as well for other projects. This technique can improve a predictor, but does not elucidate which structural metrics are important to measure.

2.4 Search Based Computing in Software Engineering

Harman and Jones [HJ01] have made the following observations on software engineering problems:

- There is usually a need to balance competing constraints.
- Occasionally there is a need to cope with inconsistency.
- There are often many potential solutions.
- There is typically no perfect answer…but good ones can be recognized.
- There are sometimes no precise rules for computing the best solution.

These observations are common to many real world disciplines and search-based techniques have been successfully applied to many challenges in traditional engineering domains. In recent years, search-based methods such as hill climbing, simulated annealing, Tabu search, and genetic algorithms have been applied to software engineering problems.

Three issues need to be tackled in order to reformulate an engineering problem into a search-based problem. The representation (encoding) of the solution, the evaluation of the solution's fitness, and the operations needed to form new solutions. The following is a small sampling of the types of areas that different search-based approaches have been used in to solve challenging software engineering problems and arrive at near optimal solutions.

A common software maintenance task is the assignment of human resources to a project. Antoniol *et al.* [ADH05] report on the utilization of three search-based techniques (hill climbing, simulated annealing and a genetic algorithm) to the problem of assigning programming teams to work packages (WP) in a large industrial project addressing Y2K remediation. It consisted of 84 WP and 300 Cobol and JCL files. They implemented two solution encodings: pigeon hole scheduling with 84 entries with each entry being the team assigned to the WP, and an ordering scheme with an 84 element queue, each entry being the queue position of the WP. There were no interdependencies between the WPs. A solution's fitness value was the calculated time to complete all 84 WP.

For hill climbing, the recombinant operation was to assign a randomly chosen WP to a new randomly chosen team. If the new scheduling configuration resulted in a shorter timeline it replaced the current configuration. The process was repeated until the configuration remained unchanged for a given number of iterations. For simulated annealing, the same recombinant operation was used with the same replacement strategy as hill climbing, except that at the beginning of the search, moves to less fit solutions were allowed. As the ‘temperature’ cooled, simulated annealing behaved the same as hill climbing.

For GA the same encoding and fitness function was used. The recombinant operation was different. Parents were chosen roulette wheel style (more fit solutions were more likely to be chosen) and crossover was single point. For the pigeon-hole encoding, GA mutation randomly choose a WP and randomly changed to a new team. For the queue order encoding, GA mutation randomly selected two WPs and exchanged their position. The investigators report that with their case study, hill climbing and GA perform at par while simulating annealing did not fare as well. The search-based solution would have resulted in maintenance completion 50% sooner than the actual project time line.

Testing is a critical phase of development and automating the generation of effective test cases would reduce the time and cost of this activity. Diaz *et al.* [DTB03] present a search-based method based on Tabu search for the automated generation of test cases for dynamic white-box testing of branch coverage. The goal of the search is to obtain maximum branch-test coverage. The program’s control flow graph (nodes are statements and edges are flow of control between statements) is used to encode the solutions. The final solution shows the best tests that are able to reach each node in the flow graph. The problem is tackled by treating each node as a sub-problem, where the aim is to find the test data inputs that reaches the sub-node starting from the parent node’s best test case.

Neighbouring solutions are limited to 4**N*, where *N* is the number of input variables. For each input variable the values are adjusted by (+ -) increments dependent on the type and range of values. The best fitness for a node is the test case that has the largest number of permutations between branches (i.e. most coverage by using the most boundary values when reaching that node).

Tabu search uses adaptive memory (long term and short term memory) to aid in the exploration of the solution space. In this approach, the long term memory stores the worst tests and these are considered “tabu” (not to be explored again). The short term memory stores the best tests for the goal node; further exploration for these nodes is also to be avoided since their neighbourhoods have already been assessed. The evaluation of this approach, as compared to other methods of generating test cases, is not fully given in the short paper other than to say that very high coverage was obtained.

Seng *et al.* [SSB06] carried out a search-based study to automate a refactoring activity. As systems evolve they tend to deviate from original designs, at times degrading the system’s structural quality. Refactoring is the process of changing the configuration of a class to improve its design and interaction with other system components. Refactoring is an important and time consuming task in software maintenance. In this pilot study the authors only consider the movement of a class’s method to another class as the refactoring operation.

A genetic algorithm is used as the search engine. The system is transformed into a suitable class model and the solutions are encoded as the method movements that transformed the original model to the final refactored model. The solutions are not fixed length and can grow as refactoring operations are applied. A solution is modified by using crossover and mutation. In crossover two parents are chosen and all the method movement refactorings from one parent are appended after the first N refactorings from the other parent. Any refactoring that is not valid is removed from the solution. Mutation extends the current refactoring solution by one method movement.

The fitness function is the weighted sum of six structural metrics which includes CK's WMC metric along with two coupling and three cohesion metrics chosen from [BDW98, BDW99]. A set of refactorings that increased cohesion, reduced coupling and complexity is considered a superior solution. The search-based refactoring method was evaluated on JHotDraw, a technical drawing application with 275 classes and 29 KLOC. Two evaluation methods were performed. First, the system was automatically refactored by the prototype and the resulting model manually inspected to see if the refactorings could be "justified". The authors point out that as they were not the developers of JHotDraw they could not for certainty conclude that the refactored design was an improvement. The second validation performed was to randomly move 10 methods in the original source code and verify that the automated refactoring would recover the original design. They performed 10 runs and nine of the ten methods were moved back to the original position in nine of the runs.

Though it appears that no work on search-based metric selection has been carried out in software engineering, an evolutionary approach has been applied to software quality models. Bouktif *et al.* [BKS02] used a genetic algorithm to combine different quality models to form one meta-quality model that is potentially generalizable to different projects. The idea is that each model, derived from a different project or version, is an "expert", and combining the opinion of different experts should result in a more broad-spectrum predictor. The authors used Quinlan's decision trees [Qui93] as the classifier. Quality models for nine different systems were generated using 22 metrics generated by a commercial tool. The predictive outcome for the classifiers was the stability of a class. If the public interface from a previous version was included in the current version, the class was considered stable.

In a decision tree classifier the internal nodes are if-then-else clauses for each metric and the leaf nodes are the classification outcome, stable or not stable. A solution is encoded as a nested vector, where each element consists of the decisions that lead to a leaf node. Crossover is performed by interleaving the subsets of the two parent trees. Mutation is carried out by randomly changing the class label of a leaf node. The fitness function is the correct classification rate of the new decision tree. To evaluate the results they calculated the prediction accuracy with two different techniques: 1) determine the best performer of all the experts with their validation set (a project not used to create any of the classifier), 2) build a new meta-model using the AdaBoost algorithm of combining experts, where each expert is weighted so that the final performance is maximized. They report 57% accuracy using the best single expert, 59% using the AdaBoost meta-model and 68% using the GA driven meta-model.

2.5 Chapter Summary

This chapter described the numerous structural metrics that can be calculated from object-oriented systems. They can be grouped into 7 general areas:

- Complexity, such as cyclomatic complexity (CYCO)
- Coupling, such as CK's coupling between objects (CBO)
- Inheritance, such as CK's depth of inheritance tree (DIT)
- Encapsulation, such as number of public attributes
- Size, such as lines of code (LOC)
- Readability, such as the ratio of comments to code (RCC)
- Cohesion, such as CK's lack of cohesion (LCOM)

Metrics have been selected for predictive models using univariate methods such as correlation followed by greedy algorithms, or multivariate statistical approach such a PCA. This chapter also detailed the area of search-based algorithms and how the various strategies such as hill-climbing, simulated annealing, tabu search and genetic algorithms have been used to find near-optimal solutions to certain types of software engineering research problems. The next chapter will establish the research problem that is the focus of this thesis.

Chapter 3

The Need for Effective Metric Selection

Organizations strive to produce high quality products as effectively as possible and maintenance is a major expense of the software development cycle. Since software development is a human centric activity where developers spend much of their time looking at source code, program comprehension is an important characteristic of a system. It is theorized that structural properties such as size, coupling, cohesion, code functionality complexity, inheritance and program readability (such as meaningful identifier labels) have an impact on the cognitive complexity of a system [Ema02]. That is, these structural properties place a mental burden on developers, inspectors, testers and maintainers in understanding the system, both at the component and overall system level.

As well, a system that is difficult to understand is likely to lead to decreased quality and increased maintenance efforts [BDKZ93]. Therefore, classes that exhibit high cognitive complexity may be indicative of potentially problematic modules that are in need of further attention before they compromise a system's quality [Pig96]. Program comprehension is important for developers, but locating and fixing potential faults before release is also very important for organizations in order to deliver quality products to users.

The ability to identify potentially problematic components (such as high cognitive complexity or fault prone modules) allows for more efficient allocation of resources, decreased costs of development and shortening release dates, all of which lead towards a superior product. Predictive models can be used to classify modules as potentially problematic and in need of mitigating actions. In empirical software engineering, design and source code metrics are used as inputs for the predictive model. However, there exists a large number of structural metrics that capture different aspects of the size, encapsulation, inheritance, coupling, cohesion, and the implementation complexity of systems.

The "curse of dimensionality" is a well-known problem in machine learning [Bel61]. As the number of dimensions in the feature vector grows, it can degrade a classifier's performance, in particular when the size of the training set is small. As well, in large dimensional feature spaces some of the features may be redundant or irrelevant for the predictive objective which degrades a classifier's performance. Therefore, selecting an effective set of product metrics is an important step in building accurate predictive models of software quality.

With the numerous source code metrics available, choosing an effective subset for a particular project and predictive objective can be challenging. One approach still used in current research is to employ a standard collection of metrics such as the CK metrics suite. These metrics have been statistically and empirically validated to be useful with various datasets, but mostly in fault prediction. Since they attempt to capture some of the key concepts indicative of software quality (such as coupling, cohesion, complexity and inheritance) practitioners that use these sets assume that they are sufficient for their project and predictive objective.

Metric subsets can be drawn from various standard sets and customized by considering personal experience, preference or predictive objective. Lanza and Marinescu [LM06] devised a metrics "pyramid" to characterize and evaluate a system for potential design problems. They used the CK metrics for coupling and inheritance as well as some standard procedural measures adapted to OO systems, such as McCabe's cyclomatic complexity, LOC, fan out, and the number of packages (for Java based systems). In their prediction models, Alshayeb and Li [AL03] have used selected metrics from the CK suite (WMC, DIT and LCOM) and from Li [Li98] (number of local methods, coupling through data abstraction and coupling through message passing). They chose these metrics because "of their wide acceptance among the software engineering community".

A commonly used metric selection strategy is to use univariate statistical techniques (such as correlation) to identify "potentially useful" metrics with respect to the predictive objective. Once this set is identified, a greedy selection algorithm is then employed to build the multivariate model, an approach used since the 1990's by Basili *et al.* [BBM96]. In forward stepwise selection, each metric is added to the predictive model, one at a time, and if it improves its performance it is kept, otherwise it is rejected. Backward stepwise elimination starts with all the metrics, each metric is tentatively removed and the model tested. The metric that has the least positive impact on the model's performance is permanently eliminated from the model. A major drawback of the stepwise metric selection strategy is the initial step (the univariate analysis) as it does not take into account the discriminatory power of combined metrics. That is, a metric that may have improved the predictor, when combined with other metrics, will not be included in the final model since it did not pass the univariate analysis

A statistical method that takes into account all of the available metrics at the same time is principal component analysis. In PCA, principal components are linear combinations of all the metrics and account for the variance in the normalized dataset. When applying PCA for dimensionality reduction it is common to use only the PCs that, taken together, explain the majority of the variance in the dataset. The loading factors in the PCs indicate the importance of a metric in explaining the variance. Metrics with high coefficients contribute more towards explaining the variance of the PC and are candidates to be used in the multivariate predictive model.

Setting the loading factor threshold is one limitation of PCA, as it is user dependent. PCA is effective at eliminating redundant metrics but not metrics that are irrelevant for a given predictive objective. When using PCA, it is assumed that only the metrics that explain the majority of the variance will improve predictive models and the same metrics subset can be used for any objective. However, a metrics set that works well in predicting failures may not perform as well when predicting when to refactor a class. This is PCAs major drawback, as it would result in the same metrics set and used with both objectives.

Metrics that exhibit a high variance in values will always be chosen, even if they are not applicable to the predictive objective.

In the absence of theoretical reasons for choosing a particular set of metrics and the observation that predictive models of different projects/objectives may need different metrics [NBZ06] there is a need to find an effective method of metric selection in order to improve the performance of software quality predictors. As well, understanding which metrics are more important in building the predictive models would also assist in providing insights into why certain metrics are best suited for different predictive objectives. It would also help identify what structural properties developers need to consider in order to maintain a high level of product excellence.

3.1 Contribution of This Work

The main contribution of this research is the validation of a search-based approach for the selection of object-oriented metrics to improve predictors of software quality, advancing the current state of the art in software quality improvement models. The ability to select the relevant product metrics for specific predictive objectives not only improves the effectiveness of the models, it also helps in developing project management tools such as real time monitoring and visualization of system status. These tools rely on acquiring measurements on the essential system metrics that best capture the desired objective. As part of the analysis of software projects with different predictive objectives (such as finding classes that show evidence of high cognitive complexity or are fault-prone), the key metrics that are best indicators of such objectives will be identified.

Chapter 4

Proposed Metric Selection Strategy

A search-based selection strategy that takes into account the predictive objective is proposed as a means of identifying combinations of source code metrics that improve predictive models of software quality. There are various search-based algorithms that can be used to find near-optimal solutions to a problem. Genetic algorithms, based on the evolutionary mechanics of selection of the fittest, lend themselves well to a combinatorial metric selection problem. Encoding a solution is straight forward and the search is intuitively advanced by combining the metric subsets present in the solutions that already perform well. Genetic algorithms are based on the evolutionary observation that children whose parents do well to meet an objective carry on combined parental traits and are likely to do as well or better in meeting that objective.

A search-based algorithm will select a metrics subset that gives the best model performance, but in of itself it does not explain why the subset works well, only that that subset works best for the training/test sets, the model used and the given predictive objective. In order to gain a deeper understanding of which metrics have more relevance in predicting the desired outcome, a Decision Tree classifier will be used with the metrics subsets selected by the GA algorithm.

The rest of this section is organized as follows: Section 4.1 presents some of the commonly used search-based strategies for finding near-optimal solutions and points out their various drawbacks when used with metric selection. Section 4.2 will describe the genetic algorithm used for metric selection. Section 4.3 elaborates on how the various metric selection strategies will be evaluated in order to determine their comparative efficacy. Section 4.4 explains the Decision Tree used in this research.

4.1 Search-Based Strategies

In hill climbing the solution space can be conceptualized as a topological landscape where the peaks represent areas of higher fitness [RN03]. In hill climbing, a random starting point is evaluated for its fitness to solve the problem and then the neighbouring solutions are evaluated for their fitness. If a neighbour results in an improved fitness value, it becomes the current solution point. There are two problems with using hill climbing for metric selection. First, it is susceptible to stopping at a local maximum as illustrated in Figure 4.1. That is, no immediate neighbour is found with a higher fitness value though a search further away in metric space may produce a better solution.

Second, metric selection is a combinatorial problem and there is no intuitive way to determine what constitutes a “neighbouring” solution. One could try randomly

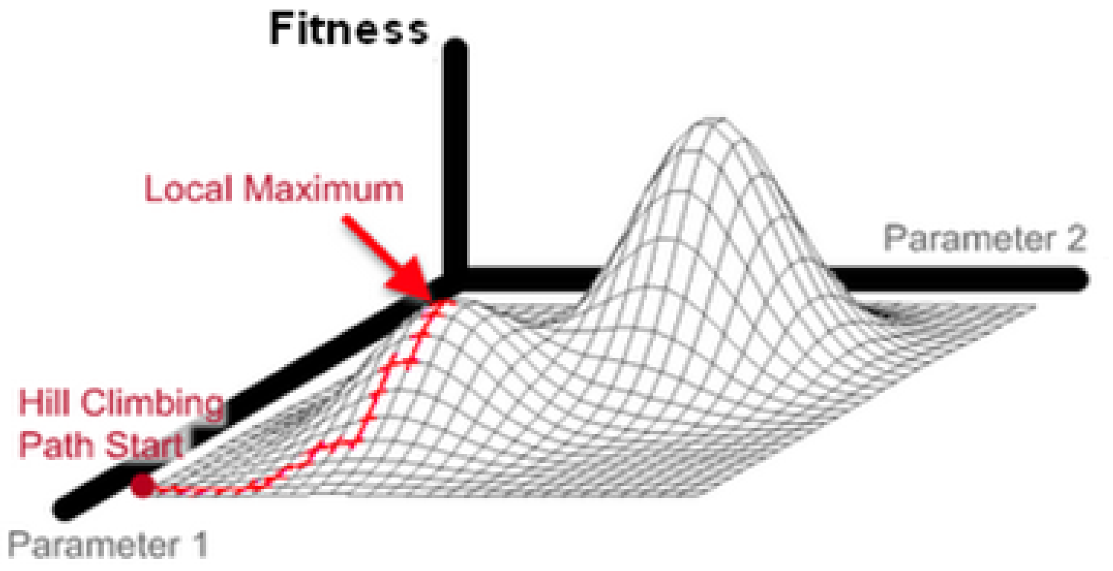


Figure 4.1: Visualization of Hill Climbing and stopping at a Local Maximum.

---

selecting/deselecting one metric at a time, as done by Antoniol *et al.* [ADH05], and calling the new metric set a "neighbour", but in reality that is no different than a random search with short term memory (keep the best combination of metrics found so far). If the optimization problem was a "turn the knob" type of problem such as weather forecasting, where the features are fixed but could range in value (humidity level could be varied up or down as an example) then finding a neighbouring solution would be a slight adjustment to the feature's value. However, such adjustments are not readily applicable to the problem of metric selection. For example, what constitutes as a metric that gives more, or less, coupling?

Simulated annealing attempts to counter the problem of terminating the search in a local optimal solution by using a probability function when deciding to move to a neighbouring solution, even if the solution is an inferior one [RN03]. The probability function is similar to the Maxwell-Boltzmann distribution law for molecular energy in a gas. This law depends on the temperature of the gas. In simulated annealing, initially the “temperature” (T) is set high so the probability of moving to a neighbouring solution is high, even if the solution is an inferior one.

As the search continues T “cools” and the chance of moving to an inferior neighbouring solution decreases. The search will then settle on a near-optimal solution. The chance of stopping in a local optimal solution is less than in hill climbing but it is very dependent on the cooling schedule parameters (stop criterion, initial temperature, and the temperature decrement between successive stages). The same problem of how to advance the search to a “neighbouring” solution is still present for the combinatorial problem of metric selection.

Tabu search is a commonly used search-based algorithm that is not as susceptible to terminating the search in a sub-optimal solution [Glo90]. Unlike hill climbing or simulated annealing it is not limited to advancing the search by traversing the solution “neighbourhood”. Tabu search uses the concept of “memory” to omit moves in the solution space to locations that are considered “taboo”. In Tabu search, a candidate list of new solutions is generated (by any means desired as there is no implied concept of a “solution neighbourhood”) and each candidate is assessed against the current search restrictions in the Tabu list (for example, the solution must have at least three metrics and no more than N/2, or that the solution has been tried before). If the candidate passes the restrictions it is evaluated and if it is an improvement over the current best solution, it supplants it.

In metric selection, coming up with valid Tabu restrictions could be quite elaborate and perhaps artificial. Why restrict the number of metrics to a desired number? With software metrics, why give a priory preference to coupling metrics over complexity for a given objective? A less restrictive combinatorial approach that is not susceptible to terminating the search in a local optimum is needed.

4.2 Metric Selection With A Genetic Algorithm

A genetic algorithm (GA) heuristically searches for an optimal set of solutions to a problem by simulating evolution. They are used in optimization and search problems where a clear analytical solution is not readily available [Gol98]. In GA, a solution (a set of software metrics) is encoded in a gene and a collection of solutions (genes) makes up a population. Genetic algorithms are based on the process of natural selection; over many generations the "fittest" individuals dominate the population. In the context of this study, the fittest individual results in a metrics subset that achieves the best predictive model performance.

The simplest way to encode the solution is with a bit mask as shown in Figure 4.2. A '0' means the corresponding metric is not to be used with the classifier and all the metrics with a corresponding bit set to '1' constitute the metrics sub-set to evaluate with the fitness function (a given predictive model). The N-bit mask vectors of the initial

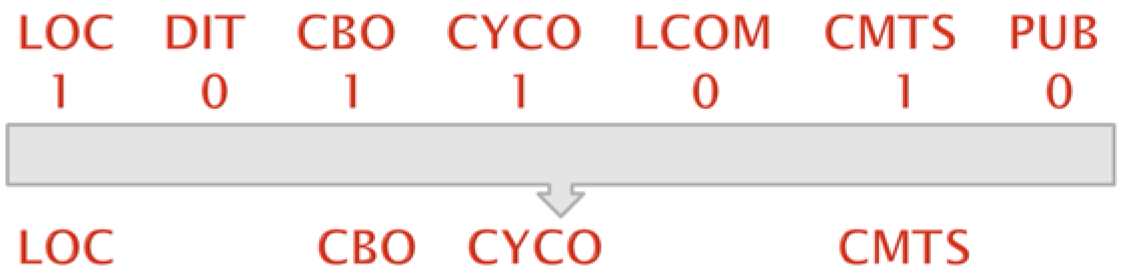


Figure 4.2: Gene encoding a metric subset solution with a bit mask.

population are randomly initialized. A fitness value, indicating how well the encoded solution solves the problem, is associated with each gene. For this research the fittest individual results in the best classifier performance in meeting the objective, that is, to improve the performance of a software quality predictive model.

Combining the genome of selected parents followed by random mutations creates a new population. In this implementation of the GA, a user parameter specified the probability that a bit would flip. The default setting was 5%. Roulette wheel parent selection was used, in which the likelihood of a gene being selected as a parent is based on gene fitness. Superior solutions have a higher probability of being selected to carry on their genome into the next generation.

Merging the genome of two parents at a random crossover point generated a new solution. The bits at the left of the crossover point for one parent and the bits to the right of the crossover point of the second parent are appended as illustrated in Figure 4.3. In this way, the child contains a partial solution from both parents.

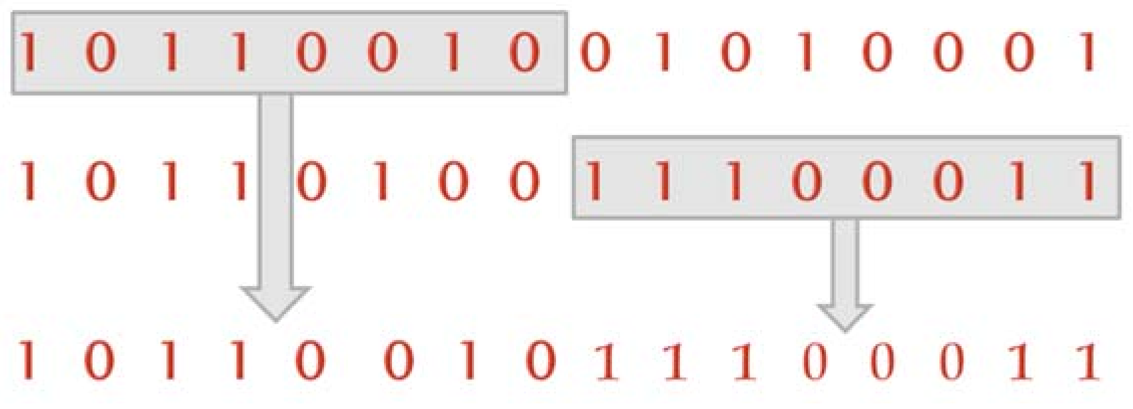


Figure 4.3: Creation of a child gene encoding a new solution using cross-over.

The most computationally intensive aspect of a GA is the calculation of a gene's fitness value. A GA where the fitness function for one gene can be executed independently of other genes in the population lends itself naturally to parallelization.
In a parallel GA implementation [Can00], one process can calculate the fitness function for one gene in a population, while another process does the same for another gene in parallel. The manager process sends a gene to an available worker process until the population is fully evaluated. Each worker process decodes the gene and trains/tests the classifier to obtain a classification rate to be used as the fitness value. Figure 4.4 illustrates the worker/manager parallel GA that has been implemented to search for a near-optimal set of metrics that satisfy a predictive objective.

Using a parallel GA greatly speeds the search for a solution. An evaluation of a parallel GA implemented with MPI resulted in linear performance increase with an efficiency ratio of 0.94 on a Beowulf cluster [VP04]. The increases efficiency allows for larger populations over longer generations.

There are various parameters that are specified by the user: the number of genes; the number of generations; the probability of mutation; and the percent of the population that is kept for next generation (% elite genes). For this study, the number of genes was set to 100, the number of generations to 25, mutation rate to 5% and the percent of elite genes was set to 20%. The stopping condition is commonly a user specified total number of generations to evolve. A working prototype of the above metric selection strategy has been implemented and the tool is presented in the Appendix.

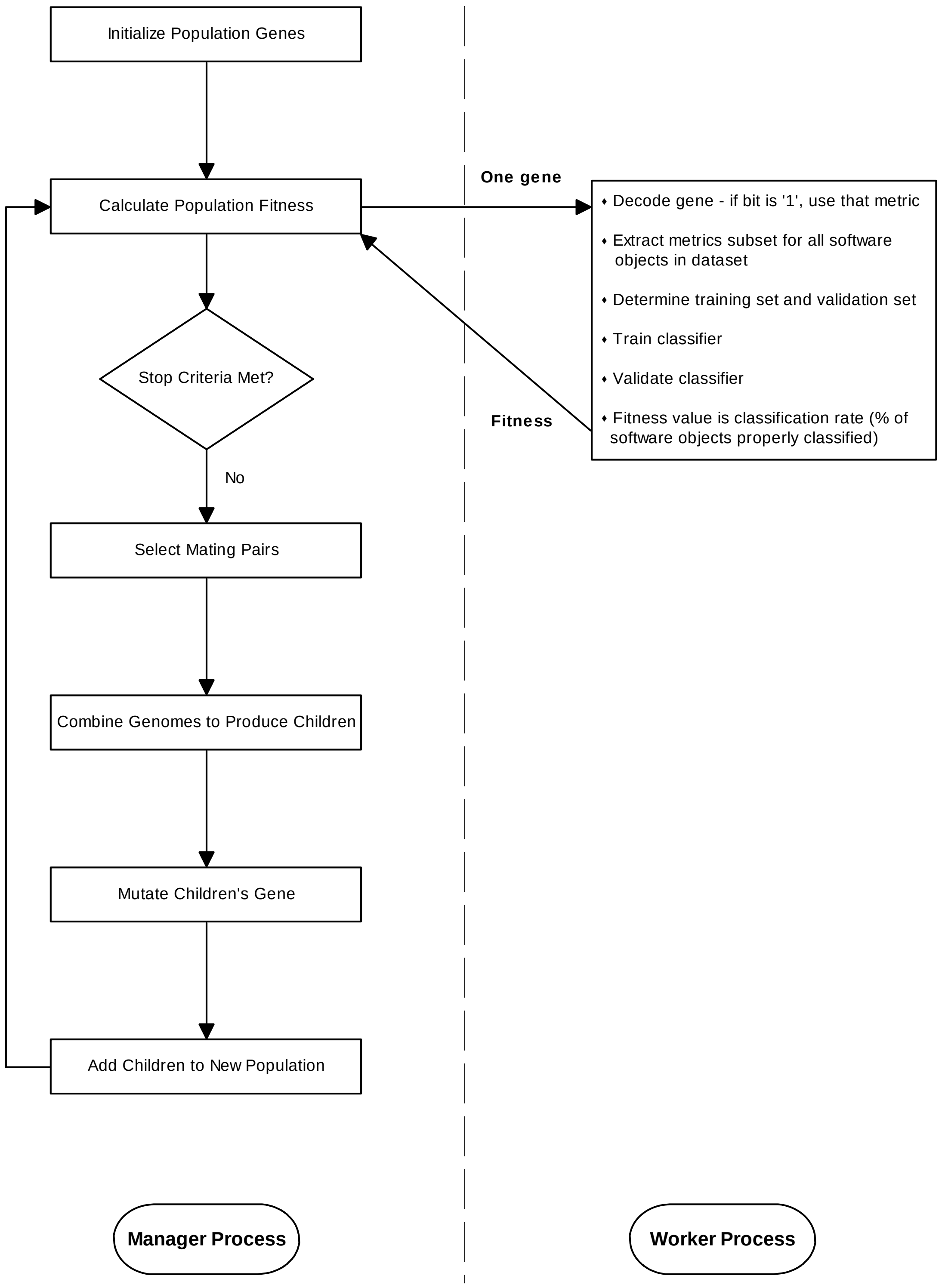

Initialize Population Genes
Calculate Population Fitness
Stop Criteria Met?
No
Select Mating Pairs
Combine Genomes to Produce Children
Mutate Children's Gene
Add Children to New Population
One gene
Fitness
• Decode gene - if bit is '1', use that metric
• Extract metrics subset for all software objects in dataset
• Determine training set and validation set
• Train classifier
• Validate classifier
• Fitness value is classification rate (% of software objects properly classified)
Manager Process
Worker Process

Figure 4.4: Parallel manager/worker genetic algorithm for metric subset selection.

4.3 Evaluation of Metric Selection Strategies

There are many multivariate predictive models available in the literature. In empirical software engineering studies, multivariate linear regression is commonly used, although non regression models are also used. Some examples are random forests [GMCS04], neural nets [KUST07], decision trees [KPB06], Bayesian networks [VG06] and support vector machines [XGL05].

A supervised machine learning algorithm, combined with a linear discriminant analysis (LDA) classifier [DHS00] was utilized in this case study as the multivariate linear model. Figure 4.5 illustrates LDA, which is a conventional supervised classification strategy used to determine linear decision boundaries between groups while taking into account between-group and within-group variances. As Figure 4.5 illustrates, the linear decision boundaries do not always fully separate the groups, leading to misclassifications.

In supervised learning algorithms such as LDA a training set is used to determine the linear decision boundaries that separates the groups. Each data sample in the training set (a software component such as a class) has a group label associated with it, also known as the dependant variable. For example, in software engineering, a common type of problematic component to identify is the one that is likely to be fault prone. Thus, the group labels may be *Fault* (one or more faults) and *No Fault* (no reported faults).

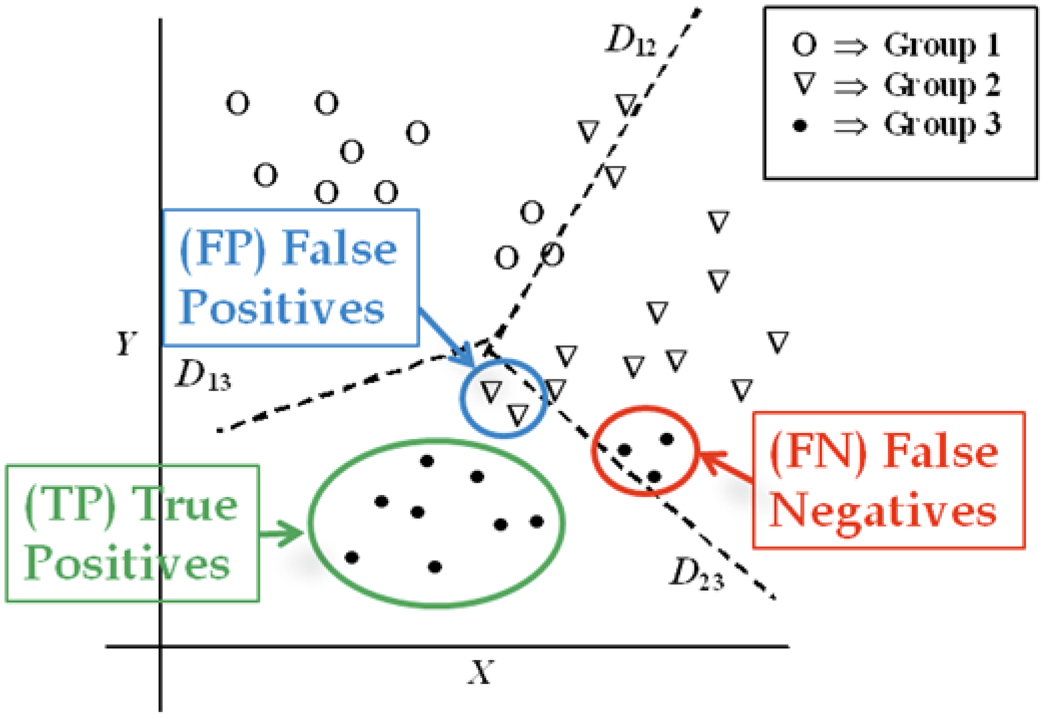


Figure 4.5: LDA decision boundaries for three groups with misclassifications.

Each data sample also has a set of features, the independent variables. In this research the features are the metrics obtained from the source code. With two features, the decision boundaries are lines; with three features they are planes, with four or more features they are referred to as hyper-planes. There is no restriction on the dimensionality of the feature space. However, too many degrees of freedom lead towards the curse of dimensionality. The aim of LDA training is to calculate the coefficients for the hyper-plane boundaries while minimizing the classification error in the training set.

In order to evaluate the classifier the dataset is partitioned into a training set and a validation set. A commonly used approach is to train and test the classifier using N-Fold cross validation. With N-cross validation, the dataset is randomly divided into N subsets. For each subset, training is done on the remaining subsets and testing is performed with the current subset.

In this study, the available datasets will be inspected and the appropriate cross validation method applied. The classification accuracy is the number of software objects in the validation set correctly categorized by the trained LDA classifier. The classification accuracy is a value between 0 and 1 and will be used as a gene's fitness value. Table 4.1 illustrates the confusion matrix for a two class problem.

Table 4.1: Confusion matrix for a two-class problem.

	Actual Negative	**Actual Positive**
Predicted Negative	True Negatives (**TN**)	False Positives (**FP**)
Predicted Positive	False Negative (**FN**)	True Positive (**TP**)

The following values can be calculated from the confusion matrix. For the final evaluation the precision, recall and F-measure will be presented, which can be obtained from the confusion matrix as shown below:

Accuracy $= \frac{TN+TP}{TN+FN+TP+FP}$

Recall (sensitivity) $= \frac{TP}{FN+TP}$

Precision (specificity) $= \frac{TP}{FP+TP}$

F-Measure $= \frac{2\times Recall\times Precision}{Recall+Precision}$

Four metric selection strategies will be evaluated. The baseline for this preliminary work will be the performance when all the available metrics are used. The CK metrics suite will be also utilized as it is commonly used in many empirical studies. PCA will be used as the multivariate statistical selection strategy. The above methods will be compared to metric subsets obtained using the proposed search-based strategy, a genetic algorithm.

The main hypothesis to test is that classification models that use GA driven metrics outperform classifiers that use the CK metrics or subsets obtained from PCA. Using a classifier's classification rate is not enough to compare different models. Since the group labels of the software objects in the dataset are known *apriori*, the number of false positives (Type I error) and false negatives (Type II error) can be calculated. For a two-class problem the sensitivity and specificity can be derived. Since the same predictive model is used for all metric selection strategies, the various metrics sets can then be compared directly and the best performing combination of metrics identified.

In the case where the objective is to identify fault-prone components, if the total number of faults is known, the completeness of the various models can be assessed and compared. In general, the classifier with the highest combination of recall and precision (F-Measure) would be the superior model.

The GA driven metric subsets will be compared with each other and to the PCA metrics in order to determine if they capture the same structural properties of the system. That is, some key metrics may appear in all subsets, and the ones that appear in only one model may in fact be different ways of measuring the same structural attribute. In the latter case, the metrics that are easier to derive from the source code should be used to build predictive models.

4.4 Decision Trees

Linear discriminant analysis (LDA) will be used as the predictive model in the comparative analysis. However, LDA is considered a black-box classifier in that the relative importance of the metrics is not known. A decision tree is proposed as the second step in the prediction process in order to gain additional insights into which source code metrics are more relevant in predicting product quality.

Decisions trees (DT) are considered white-box predictive models. The decision making process is made clear as decision nodes in a tree. At each of the internal nodes one of the features is used as the comparison condition. Data instances that have a certain

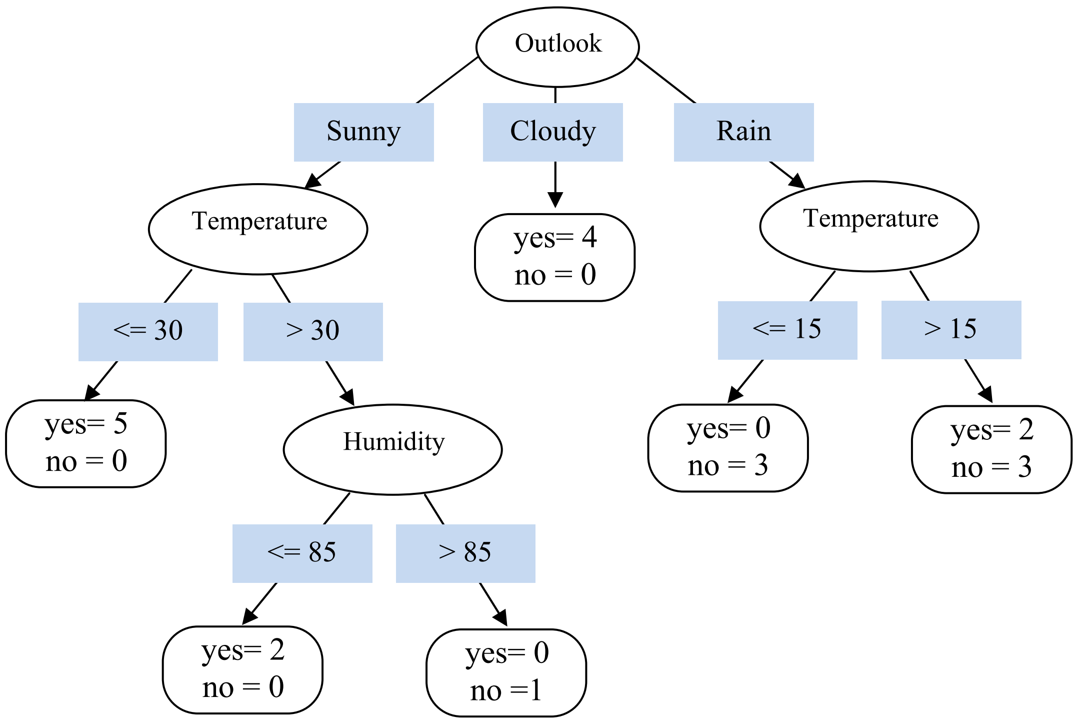


Figure 4.6: Simple decision tree to predict participation in outdoor sporting events.

---

value follow one branch of the tree; the others follow the alternate branches (usually a binary split is used, although it is not necessary). Figure 4.6 illustrates a simple decision tree that predicts when individuals play an outdoor sporting event, such as soccer, depending on three weather conditions: precipitation outlook, temperature and humidity.

Figure 4.6 shows that of the three features, the most important one in this DT is the precipitation outlook (the root of the tree). If the outlook is overcast there is always a game (we have reached a leaf). If the outlook is rain, then one should consider the temperature. For sunny days, one considers the temperature and then the humidity. At each leaf, DT visualizations usually include the number of instances that have been classified to each group. This helps determine how good the prediction will be if that leaf is reached. One of the most important considerations in decision tree algorithms is the splitting process, that is, which feature is used to branch the tree for further analysis.

For this study the C4.5 decision tree classifier was used [WF05]. It uses a greedy algorithm based on information gain using entropy from information theory. At a decision node, each remaining feature is evaluated for how much more information is needed in order to improve the "purity" of subtrees (forming groups of training instances that are as homogenous as possible). In the case of information entropy, the lower its value, the more information is available when using that feature, i.e. the less additional information is needed to form homogenous groups. Let *H(E)* be the entropy for a set *C* of events *E* (the training instances). It is calculated using the following equation:

$$H(E) = -\sum_{c \in C} P(c) \log_2 P(c)$$

Where *P(c)* is the probability an event has occurred (i.e. the training instance belongs to a given group) and is calculated by the ratio of events in group *c* to the total number of events in *C*.

Information gain is calculated by the difference of the total entropy and the entropy for a given feature 'f':

$$G(E, f) = H(E) - H(E, f)$$

With information entropy there is a bias towards features with a large range of values. The larger the range in values, the more branches can be used. For example, with a feature that represents a unique ID number, there could be one branch per value, resulting in perfect classification due to over fitting of the training set.

The gain ratio is used as an attempt to overcome this bias. The set 'V' is the number of instances that can be branched. In the above example the outlook splits into three subtrees, thus the set V may contain the following values (5, 4, 5), if 5 instances had an outlook of sunny, 4 cloudy and 5 rain. Gain ratio takes into account the number and size of subtrees, favouring fewer branches with more instances.

$$GR = \frac{G(E,f)}{-\sum_{v \in V} P(v) \log_2 P(v)}$$

Another potential problem with DT is overly large and complex DT (due to over fitting) which makes the decision making process difficult to fully understand. C4.5 uses pruning in order to minimize over fitting, replacing an entire sub tree by a leaf during the validation step.

C4.5 can be considered a greedy forward-selection classifier. At a decision node each available feature is evaluated for how well it groups the training set. The fitness criterion (information gain) is such that the feature that results in the most homogenous sub trees is chosen. The feature considered the most important in classification is at the root. DT analysis is performed using the metrics obtained from the GA based model in an attempt to gain a deeper understanding as to the structural metrics that help predict potentially problematic object-oriented classes.

4.5 Chapter Summary

In this chapter various search-based strategies were compared and a parallel genetic algorithm was detailed as the proposed metric selection strategy for this research. LDA will be the predictive model used and the recall, precision and F-Measure used to evaluate the selected metric sets (the GA solutions). A decision tree is proposed as the white-box classifier that can be used to determine the relative importance of the selected metrics. Chapters 5 and 6 will analyze two different datasets using the strategies described in Chapter 4.

Chapter 5

Cognitive Complexity Dataset

The source code analyzed for program comprehension and cognitive complexity is a non-trivial Java application used for the analysis of functional magnetic resonance imaging datasets [PVS01]. It consists of 362 classes and 104K source lines of code. Figure 5.1 illustrates the user interface of the application when displaying the results of analyzing functional neuroimages.

Source code inspections are very useful in assessing the quality of implementation and are commonly used in the development process [CLR+02], and for this case study source code inspection was used as a subjective measure of the system's cognitive complexity. Two experienced developers and a Ph.D. student were asked to independently inspect and rank each Java class in the system from Low to High (on a 5-point scale) in terms of difficulty to understand for the purpose of future maintenance.

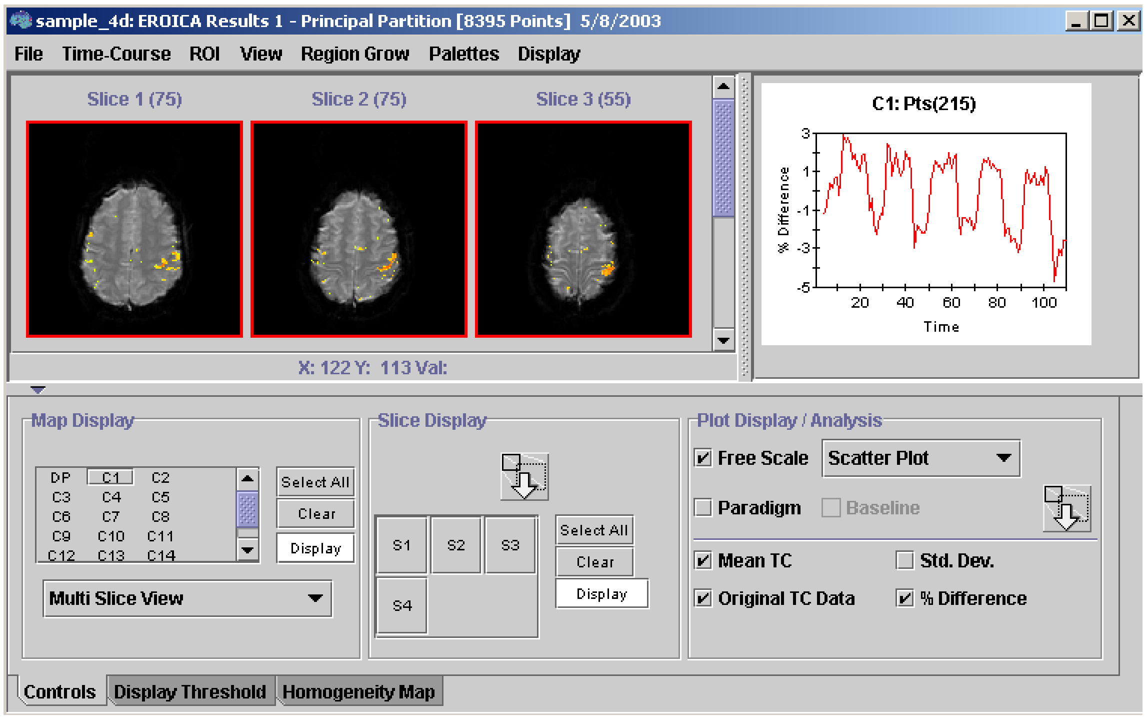


Figure 5.1: Java application for the analysis of neural functional images.

The lead developer had over ten years of OO programming experience with Java and C++ and was responsible for the design, development and maintenance of the entire system. The other developer five years of experience with Java and was involved in the development and maintenance of most, but not all, aspects of the system. The graduate student had three years of experience with Java and C++ and was not involved in the development of the system. He was responsible for the development and implementation of new data analysis algorithms to be added to the application (his area of research).

In order to not influence the inspectors' judgments of cognitive complexity, they were not given any specific instructions on how to rank the classes (focus on lines of code as an example) but were encouraged to use their tacit knowledge and past experiences with OO systems in general and their experience in developing this particular application. The inspectors were not under any time constraint and were free to take as much time as deemed necessary to adequately make the ranking decision.

Software classes given a low ranking were considered easy to understand, use and maintain, while a high ranking implied the class was difficult to fully comprehend and would likely take more effort in order to carry out a maintenance task. The initial grading scale was from 1 (*Low* complexity) to 5 (*High* complexity) to help inspectors make intermediate rankings. For example, a class that is not difficult enough to be ranked 5 (*High*), but also not ranked a 3 (*Medium*), could then be ranked a 4 (*Medium-High*).

Table 5.1: Difference in rankings by inspectors for the 326 Java classes.

Ranking Difference	**0**	**1**	**2**	**3**	**4**	**Kappa Score**
Number	189	113	23	1	0	0.58
Percent	52%	31%	6%	0.4%	0%	

It was observed that some classes were Java interfaces, which have no implemented methods. They had all been ranked as *Low* and were removed from the dataset as most measurements had values of zero. The final dataset consisted of 326 Java classes. The inter-inspector agreement for the three inspectors, the Kappa score, was 0.58 (moderate agreement). Table 5.1 shows the number of classes ranked differently by the three inspectors. A difference of zero indicates identical scores; a difference of one means the developers were off by one grade; for example, an inspector ranked a class 3, another inspector ranked the same class 2 or 4. None of the Java classes were given a completely contradictory ranking; *Low* (rank of 1) by one developer and *High* (rank of 5) by another.

The majority of the agreement (52% of the classes) came in labelling the classes as rank 1. Of the rankings with a difference of 2 (31% of the classes), the majority of the discrepancies were between labels 2 and 3, and between 4 and 5. That is, the inspectors had the most disparity when ranking low-medium vs. medium and medium-high vs. high. For the purpose of classification with the predictive model, the final rankings were 1, 2 and 3 (low, medium and high). The original rankings (1 to 5) were mapped to new ranking labels; low rank if all inspectors agree it was ranked 1, a medium rank if 2 or 3, and a high rank if 4 or 5. The resulting ranked dataset had 115 classes with a low score, 168 labelled as medium and 43 graded as high cognitive complexity.

For each Java class, a total of 63 source code metrics were computed using a commercial source inspection application and an in-house source code parser. The CK metrics suite was part of the final metric set. Tables 5.2 through 5.8 give a brief description of all the metrics calculated for the dataset, grouped into size, complexity, coupling, cohesion, inheritance and code readability metrics.

The final rankings generated three cognitive complexity groups: *Low*, *Medium* and *High*. A predictive model (LDA) was created using all the available metrics in order to establish a baseline. Due to size of the dataset (326 classes) and the small number of classes ranked as *High* cognitive complexity, leave-one-out validation was used. Using the standard N-Fold validation, with N=10, would have resulted in only four test classes, not much of a difference from leave-one-out.

An LDA classifier using the CK metrics was also evaluated to determine if the model would improve with a smaller focused subset commonly used in the literature. A final comparison of classifier performance used principal component analysis as a multivariate statistical metric selection approach and a genetic algorithm as a search-based metric selection strategy.

Section 5.1 summarizes the performance of the LDA classifier when using the various metric selection strategies. Section 5.2 evaluates the metrics identified by the PCA metric selection approach and section 5.3 does the same for the GA based metric selection. Section 5.4 performs white-box analysis via decision trees on the metrics found in the best performing model in an attempt to illuminate which source code metrics best capture cognitive complexity. Finally, section 5.5 discusses the potential threats to validity that should be addressed in future work.

Table 5.2: Size Metrics.

Code	***Metric Description***
ALOC	Average lines of code per method.
ATOK	Average number of tokens per method.
LOC	Number of lines of code.
METH	Number of methods.
MLOC	Median lines of code per method.
MTOK	Median number of tokens per method.
TOK	Number of tokens.

Table 5.3: Implementation Complexity Metrics.

Code	***Metric Description***
ADEC	Average DEC per method.
ATCO	Attribute complexity based on type.
AWDC	Average WDC.
CYCO	Cyclomatic complexity.
DEC	Number of decisions: for, while, if, switch, etc.
HLDF	Halstead computational difficulty.
HLEF	Halstead Effort.
HLON	Halstead number of operands.
HLOR	Halstead number of operators.
HLPL	Halstead Program Length.
HLUN	Halstead number of unique operands.
HLUR	Halstead number unique operators.
HLVC	Halstead Program Vocabulary.
HLVL	Halstead Program Volume.
MAXL	Maximum number of levels of branching.
MAXO	Maximum size operators.
MDEC	Median DEC per method.
MWDC	Median WDC.
OPER	Number of operations.
WDC	Weighted # decisions based on nesting level i: Sum[$i*n_i$]
WMC	[CK] Weighted methods per class using CYCO.
WMC2	Weighted methods per class (public methods).

Table 5.4: Coupling Metrics.

Code	***Metric Description***
CBO	[CK] Coupling between objects.
DAC	Data Abstraction Coupling. Measures the number of instantiations of other classes within the given class. If a class has a local variable that is an object of another class, there is data abstraction coupling
DEMV	Violations of Demeters Law. A class can only send messages to closely-related classes; one of the following: 1. Instance variables of the class. 2. Argument classes to a method. 3. Classes of objects created in a method. 4. Classes of global variables used in a method.
FACE	Number of implemented interfaces.
FNOT	Fan out, the count of other classes called by all the methods.
IMST	Number of import statements.
MAXP	Maximum number of parameters.
MIC	Method Invocation Coupling.
REMM	Number of remote methods, a call to a method that: is not declared in the class itself or a class or interface that the class extends or implements
RFC	[CK] Response for class.
RFO	Response for an object. Response set contains the methods that can be executed in response to a message being received by the object.

Table 5.5: Cohesion Metrics.

Code	***Metric Description***
ATTR	Number of attributes.
CLAS	Number of classes in file (includes inner classes).
CONS	Number of constructors.
INCL	Number of inner classes.
LCOM	[CK] Lack of cohesion of methods.
MEMB	Number of members (attributes and methods).

Table 5.6: Inheritance Metrics.

Code	***Metric Description***
ADDM	Number of added methods.
DIT	[CK] Depth of inheritance.
NOC	[CK] Number of children for a class.
OVRM	Number of overridden methods.
RCR	Code reuse: ratio of overloaded inherited methods to those that are not.
SIBL	Number of siblings for a class.

Table 5.7: Encapsulation/Data Hiding Metrics.

Code	***Metric Description***
PKGM	% package members.
PROM	% protected members.
PRVM	% private members.
PUBM	% public members.

Table 5.8: Code Readability Metrics.

Code	***Metric Description***
MNL1	Maximum method name length.
MNL2	Minimum method name length.
MNL3	Average method name length.
MNL4	Median method name length.
RCC1	Ratio of comment lines of code to total lines of code.
RCC2	Comment to code ratio from Borland.

5.1 Classifier Performance

Table 5.9 shows the model's performance, recall/precision/F-measure of each ranking (*Low*, *Medium* and *High*), for each analysis (all the metrics, CK metrics, PCA and GA derived metrics). The number of Java classes in each ranking and the number of metrics used is also given.

Using all 63 metrics resulted in an F-Measure of 0.758 for the group of interest, the Java classes with high cognitive complexity. Using only the six CK metrics LDA achieved an F-measure of 0.745, while PCA achieved 0.742 predictor performance with 35 metrics that explain 83.3% of the variance. The GA based metrics subset resulted in the best LDA performance with an F-measure of 0.804 with 28 metrics. The best overall performance for this objective with a three label grouping (*Low*, *Medium* and *High* cognitive complexity), was obtained with the GA metrics, with a consistently high F-measure for all three levels of cognitive complexity.

Table 5.9 suggests that identifying the *Medium* complexity classes is a more challenging task, with lower performance (in particular ***recall***) for all metrics subsets. The inspectors also had this problem as the majority of their mismatched rankings were in the medium complexity range. The higher LDA error rate for the *Medium* complexity classes could be due to mislabelled rankings, such as the inspectors not reaching a clear consensus on the ranking or the way the initial 1-5 rankings were reduced to a ranking of 1-3. The poor performance could also be indicative of the type of metrics that were calculated.

Table 5.9: LDA classifier performance for the three ranking levels.

		***Low* (115)**			***Medium* (168)**			***High* (43)**		
	#	***Recall***	***Precision***	***F-Measure***	***Recall***	***Precision***	***F-Measure***	***Recall***	***Precision***	***F-Measure***
All Metrics	63	0.791	0.791	0.791	0.767	0.796	0.794	0.792	0.750	0.758
CK Metrics	6	0.800	0.605	0.689	0.601	0.737	0.662	0.698	0.811	0.745
CK + ALOC	7	0.736	0.622	0.674	0.631	0.702	0.665	0.698	0.811	0.745
PCA Metrics	35	0.801	0.762	0.781	0.750	0.797	0.773	0.767	0.717	0.742
GA Metrics	28	0.852	0.831	0.841	0.810	0.855	0.832	0.860	0.755	0.804

Confusion Matrices (Low, Medium, High rankings)

Table 5.10: All available metrics [63].

	Low	***Medium***	***High***
Low	91	24	0
Medium	24	133	11
High	0	10	33

Table 5.11: CK metrics suite [6].

	Low	***Medium***	***High***
Low	92	23	0
Medium	60	101	7
High	0	13	30

Table 5.12: PCA based metrics [35].

	Low	***Medium***	***High***
Low	93	22	0
Medium	29	126	13
High	0	10	33

Table 5.13: GA based metrics [28].

	Low	***Medium***	***High***
Low	98	17	0
Medium	20	136	12
High	0	6	37

Relatively few measurements of coupling and cohesion metrics were calculated, and perhaps none of the ones that were available captured the subtleties needed to separate a medium complexity class from a low or a high complexity class.

Tables 5.10, 5.11, 5.12 and 5.13 show the confusion matrices for classifying the three complexity groups and each metrics set. The confusion matrices show that none of the *High* complexity classes were misclassified into the *Low* group and vise versa, an indication that these two groups are separable. The confusion matrices also show that most of the incorrect classifications occurred with the *Medium* complexity grouping where most of the incorrect classifications occurred in placing *Low* complexity modules in the *Medium* group and the modules ranked *Medium* into the group ranked *Low*.

We are most interested in locating the problematic classes, the ones that should be considered for corrective actions. Since most of the misclassifications occurred in the medium cognitive complexity group, and correctly identifying high cognitive complexity classes is what we're trying to achieve, the dataset was divided in two, with the classes ranked *Low* or *Medium* grouped together. The results of this grouping are illustrated in Table 5.14 (Recall / Precision / F-Measures) and Tables 5.15 to 5.18 (confusion matrices).

For the group of interest (*High* complexity) the F-Measure performance decreased when using all 63 metrics (0.758 to 0.744) and increased slightly with the CK metrics suite (0.745 to 0.750). Classifier performance with PCA subset selection fell to 0.727 with 24 metrics that explain 50.7% of the variance (using more PCA metrics actually decreased the accuracy even though it increased the variance), while the GA based metrics performance improved the most from 0.800 to 0.854 with 28 metrics. It must be noted that the same PCs were used for the different groupings, *Low* vs. *Medium* vs. *High* and *Low*/*Medium* vs. *High*. PCA does not take into account the predictive objective when calculating the PCs, just the normalized metrics dataset of the Java classes, which was the same for both groupings. For the GA, it was a different set of 28 metrics, and will be elaborated further in section 5.3.

Table 5.14: Performance for *Low*/*Medium* vs. *High* complexity.

		***Low/Medium* (283)**			***High*(43)**		
	#	***Recall***	***Precision***	***F-Measure***	***Recall***	***Precision***	***F-Measure***
All Metrics	63	0.961	0.961	0.961	0.744	0.744	0.744
CK Metrics	6	0.975	0.955	0.965	0.698	0.811	0.750
PCA Metrics	24	0.954	0.961	0.957	0.744	0.711	0.727
GA Metrics	28	0.972	0.982	0.977	0.884	0.826	0.854

Confusion Matrices (Low/Medium and High rankings)

Table 5.15: All available metrics [63].

	Low/Medium	***High***
Low/Medium	272	11
High	11	32

Table 5.16: CK metrics suite [6].

	Low/Medium	***High***
Low/Medium	276	7
High	13	30

Table 5.16: PCA based metrics [24].

	Low/Medium	***High***
Low/Medium	270	13
High	11	32

Table 5.18: GA based metrics [28].

	Low/Medium	***High***
Low/Medium	275	8
High	5	38

In an attempt to improve classifier performance by balancing the dataset, an alternate approach to identifying the high complexity classes a two stage predictive model was built. The first stage differentiates between classes ranked *Low* and *Medium* or *High*. The second stage distinguishes only between *Medium* and *High* complexity classes.

Table 5.19 summarizes the performance. The confusion matrices for the first stage are presented in Tables 5.20 through 5.23.

Table 5.19: Performance for *Low* vs. *Medium/High* complexity.

		***Low* (115)**			***Medium/High*(211)**		
	#	***Recall***	***Precision***	***F-Measure***	***Recall***	***Precision***	***F-Measure***
All Metrics	63	0.765	0.746	0.755	0.858	0.870	0.864
CK Metrics	6	0.791	0.591	0.677	0.701	0.860	0.772
PCA Metrics	32	0.765	0.746	0.755	0.858	0.870	0.864
GA Metrics	23	0.887	0.829	0.857	0.900	0.936	0.918

Confusion Matrices (Low and Medium/High rankings)

Table 5.20: All available metrics [63].

	Low	***Medium/High***
Low	88	27
Medium/High	30	181

Table 5.21: CK metrics suite [6].

	Low	***Medium/High***
Low	91	24
Medium/High	63	148

Table 5.22: PCA based metrics [32].

	Low	***Medium/High***
Low	88	27
Medium/High	30	181

Table 5.23: GA based metrics [23].

	Low	***Medium/High***
Low	102	13
Medium/High	21	190

Again, the best performing model used GA as a metric selection strategy, with an F-Measure of 0.918 for *Medium/High* group using 23 metrics. The PCA based metrics achieved performance of 0.864 with 32 metrics that explain 77.8% of the variance. Using the CK metrics performance was 0.772 while using all 62 metrics LDA achieved and F-Measure of 0.864 for the group of interest. The second model only uses the classes that were ranked as *Medium* or *High*. The results are summarized in Tables 5.24 through 5.28.

Table 5.24: Performance for *Medium* vs. *High* complexity.

		***Medium* (168)**			***High*(43)**		
	#	***Recall***	***Precision***	***F-Measure***	***Recall***	***Precision***	***F-Measure***
All Metrics	62	0.923	0.923	0.923	0.698	0.698	0.698
CK Metrics	6	0.958	0.904	0.931	0.605	0.788	0.684
PCA Metrics	32	0.946	0.924	0.935	0.698	0.769	0.732
GA Metrics	34	0.949	0.969	0.955	0.884	0.792	0.835

Confusion Matrices (Medium and High rankings)

Table 5.25: All available metrics [63].

	Medium	***High***
Medium	155	13
High	13	30

Table 5.26: CK metrics suite [6].

	Medium	***High***
Medium	161	7
High	17	26

Table 5.27: PCA based metrics [32].

	Medium	***High***
Medium	159	9
High	13	30

Table 5.28: GA based metrics [23].

	Medium	***High***
Medium	158	10
High	5	38

New PCs were calculated for the second model, as the second dataset did not include the classes ranked *Low*. The two stage model did not improve upon the *Low*/Medium vs. *High* model. In fact, all of the metric selection strategies had lower F-Measures. The GA metrics model for *Medium* vs. *High* had slightly lower performance (0.836 for *Low/Medium* vs. *High* and 0.835 for *Medium* vs. *High*), but used more metrics, 34 versus 28 for *Low*/*Medium* vs. *High* model. Therefore, the *Low*/*Medium* vs. *High* model should be preferred because it uses less metrics and does not depend on a previous model to remove the *Low* cognitive complexity classes. Sections 5.2 and 5.3 analyze the selected metrics for the *Low/Medium* vs. *High* model in more detail.

5.2 Principal Component Analysis Metrics

There are two ways to use PCA as a means of reducing feature space dimensionality. One is to use the principal components (PCs) directly as inputs, selecting the PCs that explain the majority of the variance in the dataset. In this scenario, as used by Nagapan *et al.* [NBZ06], a large number of metrics are *transformed* to a smaller set of PCs. However, information as to *which* individual metrics are useful for prediction is lost. The aim is solely to improve a classifier's performance by reducing the input dimensionality with data transformation. This strategy is a form of feature reduction, not feature selection.

In PCA driven metric selection, as used by Briand *et al.* [BAC+99], one attempts to identify the individual metrics to use as inputs to the classifier. This is done by first selecting the PCs that contribute to the majority of the variance, and then choosing the metrics whose loading factor is above a user set threshold, usually of 0.7, after varimax rotation is applied. Table 5.29 shows the cumulative variance of the PCs, the corresponding cumulative number of metrics that have a loading factor of 0.7 or higher, and the LDA performance for the *Low/Medium* vs. *High* grouping chosen as the best performer from Section 5.1.

For example, PC1 explains 40.6% of the variance in the dataset and 19 metrics had a loading factor of 0.7 or higher in PC1. These 19 metrics resulted in an F-Measure of 0.690 for predicting high complexity classes. PC2 by itself explains 10.1% of the variance and had 5 metrics with a loading factor above the threshold. Combining PC1 and PC2 accounts for 50.7% of the variance with a total of 24 metrics. Using these 24 metrics resulted in an F-Measure of 0.744 for the high complexity classes. Adding the next principal components did not improve performance and in many cases decreased performance for the group of interest.

Table 5.29: PCs, number of metrics, and LDA F-Measure for *Low/Medium* vs. *High*.

Metric Set	**#**	**Variance**	**Low/Medium**	**High**
PC1	19	40.6%	0.954	0.690
PC2	24	50.7%	0.961	0.744
PC3	26	56.9%	0.961	0.744
PC4	27	61.8%	0.957	0.727
PC5	28	66.1%	0.956	0.713
PC6	29	69.5%	0.954	0.698
PC7	30	72.4%	0.954	0.705
PC8	31	75.2%	0.954	0.705
PC9	32	77.8%	0.954	0.711
PC10	34	79.8%	0.959	0.736
PC12	35	83.3%	0.959	0.742
PC13	36	84.8%	0.957	0.727

Table 5.30 shows metrics found in the first 2 PCs for the *Low*/*Medium* vs. *High* dataset. The effect of size on PCA analysis is evident in the first PC. Most of the metrics in the first PC, which explains 40.6% of the variance, are direct size measures, such as LOC and TOK, or closely influenced by size, such as the Halstead complexity metrics. The rest of the metrics in the first PC are coupling measures. The second principle component is also composed mostly of direct size measures. In effect, PCA shows that size measures account for roughly 50% of the dataset's variance. This observation is not unexpected since the dataset includes many small classes (most of them ranked low complexity) while the majority of the classes ranked as high have substantially more lines of code. However, using many redundant metrics degrades a classifier's performance, which could explain why the PCA metrics set has lower predictive performance than the GA derived metrics set.

For this dataset, with the predictive objective of ranking Java classes in terms of cognitive complexity, the PCA metrics set were dominated by size metrics and the Halstead complexity measures. Since PCA relies heavily on a dataset's variance to identify potentially useful metrics, in this dataset it seems to imply the PCA selection of metrics is heavily influenced by size. Emam *et al.* [Ema02] and others have exposed the confounding effect of size on product metrics. They advise that size measures should be a part of the model, but not to let them dominate as that would degrade the predictor's performance.

Table 5.30: Metrics in PCs for *Low*/*Medium* vs. *High*.

Metric	**Principal Component**
LOC (size)	PC1
TOK (size)	PC1
DEC (complexity)	PC1
WDC (complexity)	PC1
CBO (coupling)	PC1
RFO (coupling)	PC1
CYCO (complexity)	PC1
FNOT (coupling)	PC1
HLEF (complexity)	PC1
HLPL (complexity)	PC1
HLVC (complexity)	PC1
HLVL (complexity)	PC1
HLON (complexity)	PC1
HLUR (complexity)	PC1
HLUN (complexity)	PC1
MIC (coupling)	PC1
REMM (coupling)	PC1
DEMV (coupling)	PC1
WMC (complexity)	PC1
ALOC (size)	PC2
MLOC (size)	PC2
ATOK (size)	PC2
MTOK (size)	PC2
MDEC (complexity)	PC2

5.3 Genetic Algorithm Metrics

As demonstrated in Section 5.1, the *Low/Medium* vs. *High* model should be preferred because it results in best performance for the predictive objective (F-Measure of 0.854 for classes ranked *High*), uses less metrics and does not depend on a previous model. The GA selected metrics used in the *Low/Medium* vs. *High* cognitive complexity model are summarized in Table 5.31, along with the F-Measure.

Table 5.31: F-Measure performance and metrics for best predictive model (*Low/Medium* vs. *High*).

	Low/Medium	***High***
F-Measure:	0.977	0.854

Grouping	***Metrics***
Size	ALOC, ATOK, MLOC
Complexity	ATCO, AWDC, HLDF, HLEF, HLON, HLPL, HLUR, HLVC, HLVL, MAXO, MAXL MDEC, MWDC, WMC [CK]
Coupling	CBO [CK], FACE, MAXP, RFC [CK]
Cohesion	- *none* -
Inheritance	ADDM, DIT [CK], NOC [CK], RCR
Encapsulation	PROM , PRVM
Readability	RCC2

Direct measures of size were present in the best performing model, but it is important to note that the mean and median per method (ALOC, MLOC, ATOK) seem to be more significant than a straight measure such as LOC. Most of the metrics in Table 5.31 are code complexity measures, the majority being the Halstead complexity. The Halstead measures vary greatly as a function of size and facilitate LDA classification, which uses variance to build linear boundaries between groups. Like the direct size measures, other measures of code complexity, such as the number of decisions (DEC), are represented as the mean (ADEC) and the median per method (MDEC), as opposed to a measure for the entire class.

Coupling metrics also played a role in the model with both CK coupling metrics present (CBO, RFC), with the addition of MAXP (coupling via method parameters) and the number of implemented Java interfaces (FACE). Though no questionnaire was filled out by the inspectors, during informal discussions it was noted that coupling was usually one of the deciding factors when ranking a class to a higher cognitive complexity group.

Both CK inheritance metrics were selected by the genetic algorithm, the depth of inheritance (DIT) and number of children (NOC). During the discussions with the inspectors, the number of parent methods that were overridden or overloaded was considered important and GA selected the metrics RCR (ratio of code reuse as the ratio of overloaded inherited methods to those that are not) and ADDM (number of added methods). This seems to suggest that *how* inheritance is used is a contributing factor to cognitive complexity.

Five of the six CK metrics appear in Table 5.31, the exception being LCOM. In fact, none of the cohesion metrics were selected by the GA. Most practitioners define cohesion as “the degree to which the methods and attributes of a class belong together” [BDW98], which suggests that cohesion is a measurement of the intended functionality of a component and not an easy concept to measure from structural metrics alone.

Two encapsulation metrics were present in the GA metrics subset, PRVM and PROM. Both measure the percentage of members that are declared private or protected. It is not clear why these metrics were chosen by the GA, other than being proxies for size measures as larger classes have a larger percentage of protected or private members. Finally, a code readability measure, the ratio of comment lines of code to the total lines of code, RRC2, was selected by the GA as a contributing factor to identifying classes that are labelled as high cognitive complexity. Like cohesion, these types of metrics attempt to measure a programmer's intent, which is difficult to accurately quantify but immensely important to developers.

Search based optimization methods such as genetic algorithms generate many solutions, each solution with a different combination of metrics and its own fitness value (percent correct classification) with respect to the objective function (identifying high complexity Java classes). Table 5.32 shows the classifier performance (F-Measure) and the union of all the metrics used by the top three genes in the *Low/Medium* vs. *High* grouping. The top solution (G1) used 28 metrics with an F-Measure of 0.977 for *Low/Medium* complexity and 0.854 for *High* complexity. The second solution (G2) consists of 30 metrics with slightly lower performance at 0.975 for the *Low/Medium* group and 0.844 for the *High* group. The third solution (G3) used 30 metrics but with lower performance in the *High* complexity group (0.841).

Table 5.32: Metrics in top three genes for *Low/Medium* vs. *High* complexity.

Metric	**G1 [28] 0.977 / 0.854**	**G2 [30] 0.975 / 0.844**	**G3 [30] 0.975 / 0.841**
ADDM (inheritance)	●		●
ADEC (complexity)		●	
ALOC (size)	●		
ALOC (size)			●
ATCO (complexity)	●		
ATOK (size)	●	●	
AWDC (complexity)	●		●
CBO (coupling)	●	●	●
DAC (coupling)		●	
DEC (complexity)		●	
DIT (inheritance)	●		●
FACE (coupling)	●	●	●
FNOT (coupling)			●
HLDF (complexity)	●	●	●
HLEF (complexity)	●		
HLON (complexity)	●	●	
HLPL (complexity)	●	●	●
HLUR (complexity)	●	●	●
HLVC (complexity)	●	●	●
HLVL (complexity)	●	●	●
IMST (coupling)			●
LOC (size)			●

Table 5.32 (continued)

Metric	**G1 [28] 0.977 / 0.854**	**G2 [30] 0.975 / 0.844**	**G3 [30] 0.975 / 0.841**
MAXL (complexity)	●	●	●
MAXO (complexity)	●	●	●
MAXP (coupling)	●	●	●
MDEC (complexity)	●	●	●
MLOC (size)	●		
MNL1 (readability)			●
MNL2 (readability)			●
MNL4 (readability)		●	
MWDC (complexity)	●		
NOC (inheritance)	●	●	●
OVRM (inheritance)		●	●
PKGM (encapsulation)		●	
PROM (encapsulation)	●	●	●
PRVM (encapsulation)	●	●	●
PUBM (encapsulation)		●	
RCC1 (readability)		●	●
RCC2 (readability)	●	●	
RCR (inheritance)	●	●	●
RFC (coupling)	●	●	●
SIBL (inheritance)		●	
TOK (size)			●
WDC (complexity)			●
WMC (complexity)	●	●	●
WMC2 (complexity)		●	

The three solutions used a combined total of 46 metrics with 20 metrics in common. The top two genes have similar performance but only 21 metrics in common. However, since many metrics measure similar attributes it is expected that alternate metrics subsets would have comparable performance. Inspecting the metrics in G1 and G2, both with the highest classification performance for the Java classes ranked *High*, gives additional insight into what could be important metrics for a predictor for cognitive complexity. It is interesting to note that classifier performance with the 20 metrics that are found in G1 and G2 is very close to both solutions; an F-Measure of 0.977 for *Low/Medium* complexity and 0.851 for *High* complexity. Table 6.33 summarizes the intersection of the G1/G2 metrics and model performance.

Table 5.33: F-Measure performance and metrics for the intersect of the G1/G2 solutions.

	Low/Medium	***High***
F-Measure:	0.977	0.851

Grouping	***Metrics***
Complexity	HLDF, HLON, HLPL, HLUR, HLVC, HLVLMDEC, MAXL, MAXO, WMC[CK]
Coupling	CBO[CK], FACE, MAXP, RFC[CK]
Inheritance	NOC[CK], RCR
Encapsulation	PROM, PRVM
Size	ATOK
Readability	RCC2
Cohesion	- *none* -

Both G1 and G2 include a direct size metric, ATOK which may be a better measure of size, as programmers may have more ALOC just by declaring one variable per line, or many variables per line. This would give different ALOC but the same ATOK. Most of the code complexity measures common to G1 and G2 are from the Halstead complexity suite. MDEC and MAXL, complexity based on decisions are present in both G1 and G2, along with WMC which uses McCabe's complexity. The next largest group of metrics is coupling, with four metrics (CBO, RFC, MAXP and FACE). CBO and RFC reflect class coupling via method calls while MAXP measures method coupling via the parameter list. FACE is a weak coupling measure because it could also be considered as a way to implement multiple inheritance in Java applications.

The two inheritance metrics common to G1 and G2, NOC and RCR, capture different aspects of inheritance. NOC is a direct measure of the inheritance tree via class children, while RCR measures how inheritance is used via overloaded methods. No cohesion metrics were present in G1 or G2, but two encapsulation metrics, PRVM and PROM, appeared in the intersection. PRVM and PROM may be considered simple cohesion metrics, as more members in a class indicates that it is has more responsibilities and thus low cohesion. Finally, a code readability/comprehension metric, Borland's comment to code ratio (RRC2), was used by both G1 and G2. Overall, with prediction performance almost at par, and almost 30% fewer metrics, the G1/G2 model should be considered the best performing model, as a model with more metrics tends to over fit the dataset, especially with a small dataset.

Inspecting the metrics in the G1 and G1/G2 predictors, it would appear that complexity and coupling measures are the most important metrics in the model. However, LDA is a black-box model and it is difficult to truly ascertain the relative importance of the various metrics. A decision tree was used on the G1 metrics (superset of G1/G2) in an attempt to further understand which metrics should be included in a predictive model for cognitive complexity.

5.4 Decision Tree Analysis

Comparative analysis of various metric selection strategies (Section 5.1) showed that GA metric selection outperformed PCA and the CK metrics suite in improving a predictor's ability to identify cognitively complex Java classes. A decision tree is proposed as the second step in the prediction process in order to gain additional insights into which source code metrics are more relevant for a predictive objective; object-oriented classes that demonstrate high levels of cognitive complexity which degrade program understanding for the purpose of maintenance.

From Table 5.31, when the 28 metrics from the top performing gene are categorized, the LDA model used three size measures, ten complexity metrics (seven of them Halstead measures), four coupling metrics, two inheritance metrics, two encapsulation measures, and one code readability metric. C4.5 was used with the 28 metrics from the top performing gene to determine the relative importance of the various metrics. Figure 5.2 shows the resulting decision tree.

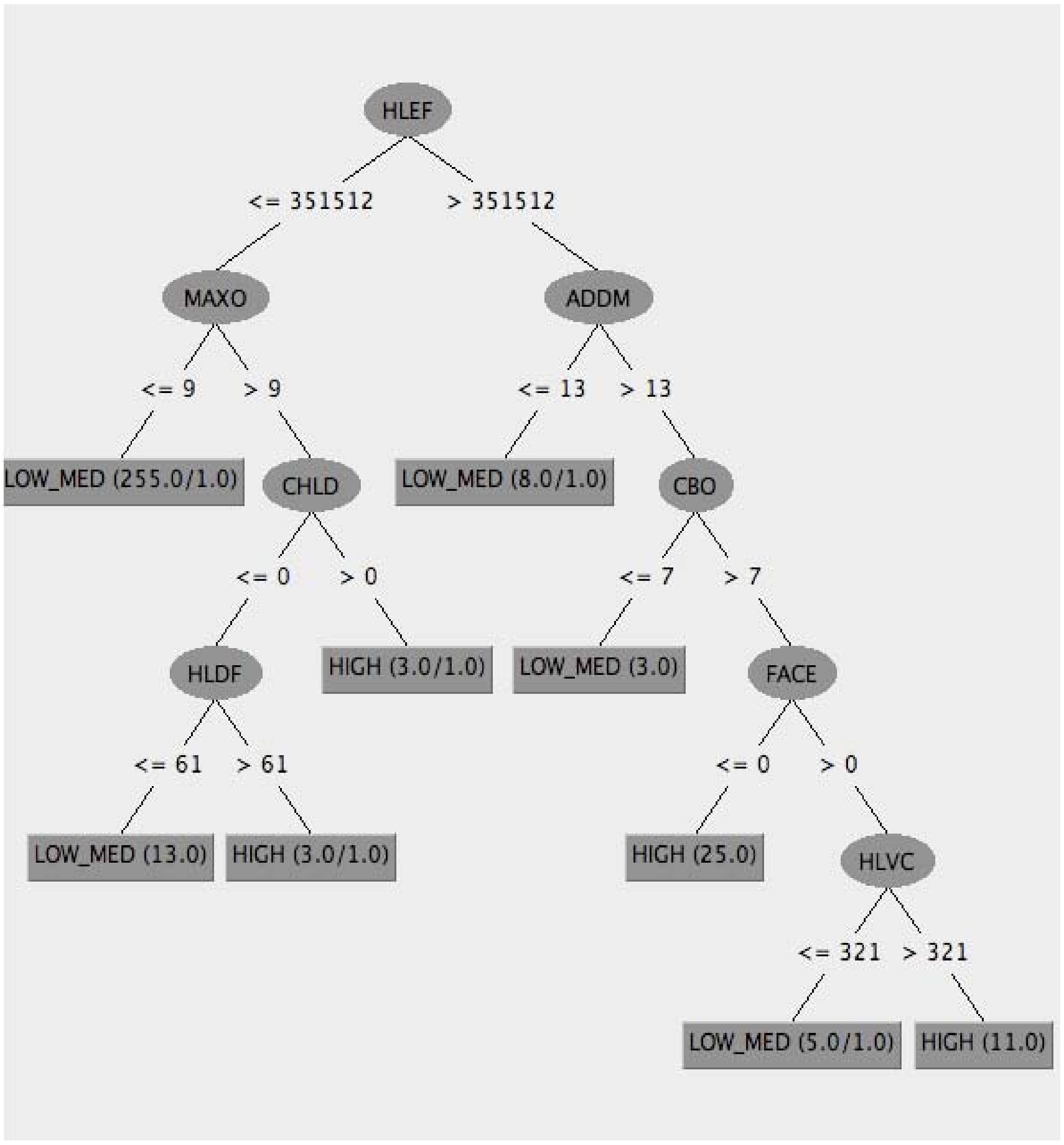


Figure 5.2: Decision tree for the 28 metrics in the top performing GA susbset.

The decision tree shows that C4.5 finds the Halstead Effort metric most useful in separating the training set into two groups, *Low/Medium* vs. *High* cognitive difficulty. It has a large range in values and it combines both a size measure and a complexity measure. Inspecting the homogeneity of the DT groupings using only HLEF, 271 of the 282 *Low/Medium* instances fall below the threshold, and 37 of the 43 *High* instances are found to above the HLEF threshold.

In order to further refine the large group of *Low/Medium* classes, C4.5 used another complexity metric, MAXO. Using only HLEF and MAXO, 255 of the *Low/Medium* training set instances are correctly categorized with only one *High* class in this leaf. An inheritance metric, CHLD (number of children), and another Halstead complexity metric, HLDF, are deemed most useful in distinguishing between Low/Medium and High cognitive complexity classes once HLEF and MAXO are considered.

For training instances that have an HLEF value above the threshold, i.e. most of the High complexity classes, the C4.5 predictive model uses an indirect measure of inheritance, ADDM, the number of added methods by a child class. Again, the classes with a lower metric value are placed in the Low/Medium leaf. Two coupling measures, CBO and FACE, are used next to separate the remaining training instances. Finally another Halstead measure, HLVC, is used to split the node into the two training groups. No direct measure of size was used in the DT model, though it is indirectly measured in the Halstead metrics, which are strongly correlated to size.

Since Halstead metrics seem to have large information gain for the C4.5 algorithm, they were removed from the metrics set and the DT trained again to see what other metrics are considered most important (the root of the DT). The resulting DT is illustrated in Figure 5.3.

Figure 5.3 also shows that two coupling metrics, RFC and CBO, contain the most information for separating the *Low*/*Medium* from the *High* complexity classes. The third metric with the largest information gain is the complexity metric MAXO. The majority of the *Low*/*Medium* instances, 226 of 282, are found in the DT leaf with RFC, CBO and MAX below the threshold values. A coupling metric (FACE) and CK's complexity metric (WMC) are used to separate nine of the *High* complexity training instances from the 45 *Low*/*Medium* data samples that are below the RFC,CBO thresholds and above the MAXO threshold.

Forty percent (17 out of 43) *High* complexity classes are above the RFC coupling metric threshold. To separate another 40% of the *High* complexity classes below the RFC threshold but above the CBO threshold, C4.5 uses the complexity metric MAXO, two inheritance metrics, CHLD and ADDM), the encapsulation metric PRVM, and the size measure, ALOC.

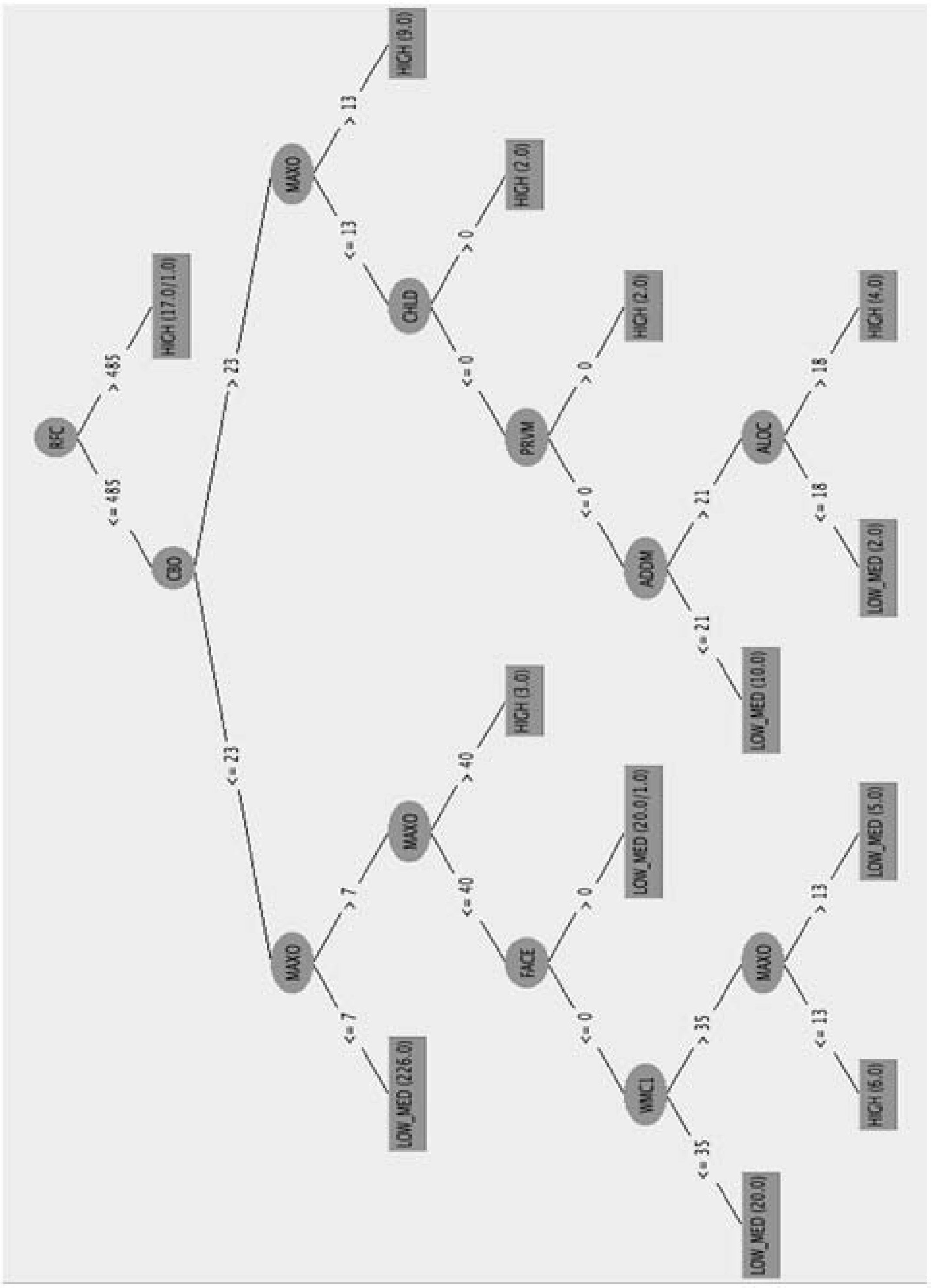


Figure 5.3: Decision tree for the top performing GA subset without any Halstead metrics.

Table 5.34: Decision tree and LDA performance using metrics of best GA solution.

		Low/Medium			***High***		
	#	***Recall***	***Precision***	***F-Measure***	***Recall***	***Precision***	***F-Measure***
C4.5	8	0.965	0.968	0.966	0.791	0.773	0.782
LDA	8	0.975	0.945	0.965	0.698	0.811	0.750
LDA	28	0.972	0.982	0.977	0.884	0.826	0.854

Table 5.35: Decision tree and LDA performance of best GA solution
(without Halstead metrics).

		Low/Medium			***High***		
	#	***Recall***	***Precision***	***F-Measure***	***Recall***	***Precision***	***F-Measure***
C4.5	9	0.943	0.934	0.938	0.558	0.600	0.578
LDA	9	0.972	0.945	0.958	0.628	0.771	0.692
LDA	28	0.972	0.982	0.977	0.884	0.826	0.854

Figures 5.2 and 5.3 show two different decision trees and by comparing their classification performance it can be determined which model is most suitable for the predictive objective. Tables 5.34 and 5.35 summarize the leave-one-out performance (recall, precision and F-measure) for the two DT as well as LDA when using the 28 GA metrics and the 8 and 9 DT metrics.

Table 5.34 shows that for the modules of interest, the Java classes with *High* cognitive complexity, the DT had a better F-Measure (0.782 vs. 0.750) and recall (0.791

vs. 0.698) over LDA for the same eight metrics. However, the DT had a lower precision value, indicating that more problematic components would not be scheduled for mitigating action. For the DT metrics which do not include Halstead measures, LDA performs better than the DT for the same nine metrics with an F-Measure of 0.692 vs. 0.578. The DT with Halstead is a superior prediction model, indicating that for cognitive complexity Halstead complexity measures are important. Table 6.34 also shows that LDA with 28 GA metrics is still the superior model with an F-Measure of 0.854 and recall of 0.884.

Inspecting the DT in Figures 5.2 and 5.3, suggests that complexity metrics (such as Halstead measures), followed by coupling and inheritance contain important structural information to consider when building predictive models. When the GA metrics common to the top two performing solutions are categorized by function (Table 5.33), the findings in the DT analysis are collaborated. Complexity, coupling and inheritance metrics play a critical role in predictive models of cognitive complexity for program comprehension. That the GA based predictive model has more complexity and coupling metrics than the DT and performs significantly better implies that there is no single particular complexity, coupling or inheritance metric that is vital, but rather, it is the combination of the various ways of measuring these three important structural perspectives that provide an optimal predictive model.

5.5 Threats to Validity

There are various threats to validity and the capability to generalize the results to other cognitive complexity datasets. First and most critical is mislabelled data samples, which is detrimental to supervised machine learning algorithms. The rankings used for this dataset are subjective and dependent on the inspector's programming experience, familiarity with the code and ability to carry out inspections. Using examiners with comparable experience and averaging their rankings would alleviate the former, and training in performing inspection would assist with the latter.

In the cognitive complexity case study, three inspectors had 24 classes whose rankings differed by at least two grades. This would be enough to place medium complexity class into the high complexity group and vice versa. As well, the labels were originally assigned from a scale of 1 to 5 but were mapped to a value of 1, 2 or 3. Where to place the boundaries of the mapping could change a class's label from a medium to a high complexity or the reverse.

However, because the impact of mislabelled samples is the same for all four metric selection methods used (all available metrics, CK metrics, PCA and GA), the relative performance of the predictive models should not drastically change with different labels, and it is expected that a search based metric selection strategy would still give the best results.

Second, the dataset is from one particular application and the number of problematic samples is relatively small, only 43 out of a total of 326. This could lead to over fitting, where the classifier is tuned for the particular dataset used in training and not applicable to other datasets that have the same predictive objective. Using larger and more varied datasets should increase the number of problematic modules and augment the generalizability of results.

Thirdly, the type of classifier has an effect on performance and perhaps the metrics chosen. LDA is a simple linear classifier that does not perform as well as non-linear predictive models in large dimensional spaces. Using other supervised classifiers as the GA objective function, neural nets [HKP91] or support vector machines [SS02] for example, may improve the classification results and use a different metrics set.

Finally, the actual metrics that are calculated and available for building predictive models should also be considered. Size and procedural complexity are well understood and fairly easy to compute, and there are only so many different ways of directly measuring the size of a class. Also, there is a strong correlation between the various Halsted complexity metrics and size. Program readability can be indirectly measured via comments and identifier length, though it is hard to measure the *usefulness* of the comments and names used in the code.

Coupling and cohesion is very important for maintenance tasks, and having more metrics to choose from would aid in illuminating the types of measurements that are good predictors of cognitive complexity. There are at least 30 different ways of quantifying coupling [BDW99] and 15 for cohesion [BDW98], and this dataset only used 11 coupling and 6 cohesion metrics to build a predictive model. If the original metrics set contains a poor selection of metrics it will impact negatively on the metric selection subset and overall predictor performance. In such a case, metric selection could result in a larger subset than necessary since it attempts to indirectly capture the discriminatory power of a single superior metric which more closely matches the Goal/Question/Metric paradigm.

5.6 Chapter Summary

The analysis of the functional MRI application's source code illustrate that using a search-based metric selection strategy can improve predictive models of source code quality. Additionally, inspection of the static structural metrics selected in the top performing solutions, along with decision tree analysis, suggest that coupling, inheritance and implementation complexity have a major impact on the cognitive complexity of a system. Cohesion, though an important concept in software engineering, is difficult to properly measure and was not found to have much of an effect on the model's performance. Size metrics were found to be most useful when measured on a per method basis. Chapter 6 carries out the same analysis strategy but with another dataset, Java source code that has been labelled as *Fault* or *No-Fault* in terms of post-release failures.

Chapter 6

Fault Prediction Dataset

This chapter carries out the same analysis methodology as in chapter 5, but with a different predictive objective. The Equinox dataset as mined by [DLR10] was used to assess the efficacy of using a search-based metric selection strategy with fault proneness as the predictive objective. Equinox [Equ10] is Java plug-in for the Eclipse IDE platform. Equinox provides a framework for the Open Services Gateway (OSG) standards. The dataset is one snapshot of the repository and consists of 324 classes with 39.5 KLOC. The dataset has 195 of the classes labelled as having zero faults and 129 determined to contain at least one post release bug.

Upon inspection it was observed that the dataset had 45 classes with zero lines of code (LOC). These modules were removed from the analysis. It was also noted that three of the modules with zero LOC were determined to be faulty by the dataset mining process. The discrepancy between having zero lines of code and yet determined to be buggy could be explained by the fact that Java uses interfaces. Interfaces are basically pure abstract classes; they contain no ***executable*** lines of code and are meant to be implemented by another class which has executable code.

Mapping a bug report to source files is not trivial. One way to do this is to parse repository commit comments were the developer would indicate that the modified file fixed a reported bug (usually referring to a bug number) [MV00, FPG03, CM06]. It could be that in order to fix a bug in a class that implements an interface, a method signature had to be changed, resulting in a change in the interface's method signature. Upon the commit of the two files, the interface and the implementation, two files would be marked as fixing a reported bug. The final dataset consists of 279 modules, 153 *No-Fault* classes and 126 *Fault* classes. Tables 6.1 through 6.6 describe the 17 metrics that were calculated for this dataset by the providers of the dataset [DLR10], the six CK metrics and 11 additional measures they deemed important to measure from personal experience with mining software repositories for predictive models of quality.

A predictive model (LDA) was built using all 17 metrics and evaluated. For this dataset N-Fold could have been used, but to be consistent with the previous dataset, leave-one-out validation was used. An LDA model using six metrics was also evaluated to determine if the predictor would improve with a subset commonly used in the literature, the CK metrics suite. A final comparison of classifier performance used principal component analysis as a multivariate statistical metric selection approach and a genetic algorithm as a search-based metric selection strategy.

Section 6.1 summarizes the performance of the LDA classifier when using the various metric selection strategies. Section 6.2 evaluates the metrics identified by PCA metric selection and Section 6.3 appraises the metrics identified by GA metric selection. Section 6.4 performs white-box analysis via decision trees on the metrics found in the best performing model in an attempt to illuminate which source code characteristics have the best association with prone classes. Finally, Section 6.5 discusses the potential threats to validity that should be addressed in future work.

Table 6.1: Size Metrics.

Code	***Metric Description***
LOC	LOC Number of lines of code.
METH	Number of methods.

Table 6.2: Implementation Complexity Metric.

Code	***Metric Description***
WMC	[CK] Weighted methods per class using CYCO.

Table 6.3: Coupling Metrics.

Code	***Metric Description***
CBO	[CK] Coupling between classes.
FIN	Fan in, count of other classes calling a method.
FOUT	Fan out, count of other classes called by all the methods.
RFC	[CK] Response for class.

Table 6.4: Cohesion Metrics.

Code	***Metric Description***
ATTR	Total number of attributes.
LCOM	[CK] Lack of cohesion of methods.

Table 6.5: Inheritance Metrics.

Code	***Metric Description***
ATRINH	Number of attributes inherited.
DIT	[CK] Depth of inheritance.
METINH	Number of methods inherited.
NOC	[CK] Number of children for a class.

Table 6.6: Encapsulation/Data Hiding Metrics.

Code	***Metric Description***
PUBATR	Number of public attributes.
PRVATR	Number of private attributes.
PUBMET	Number of public methods.
PRVMET	Number of private methods.

6.1 Classifier Performance

Table 6.7 summarizes the model's various performance values (recall, precision, F-Measure) and Tables 6.8 through Table 6.11 show the confusion matrices for classifying the dataset into *No-Fault* and *Fault* groups with each metrics subset (all the metrics, CK metrics, PCA and GA derived metrics). The number of data samples in each group and the number of metrics used is also included in Table 6.7

Table 6.7: Performance for *No-Fault* vs. *Fault*.

		***No-Fault* (153)**			***Fault* (126)**		
	#	***Recall***	***Precision***	***F-Measure***	***Recall***	***Precision***	***F-Measure***
All Metrics	17	0.863	0.667	0.752	0.476	0.741	0.580
CK Metrics	6	0.843	0.655	0.737	0.460	0.707	0.558
PCA Metrics	6	0.791	0.680	0.731	0.548	0.683	0.608
GA Metrics	8	0.895	0.703	0.787	0.540	0.810	0.650

Confusion Matrices (No-Fault and Fault groups)

Table 6.8: All available metrics [17].

	No-Fault	***Fault***
No-Fault	132	21
Fault	66	60

Table 6.9: CK metrics suite [6].

	No-Fault	***Fault***
No-Fault	129	24
Fault	68	58

Table 6.10: PCA based metrics [6].

	No-Fault	***Fault***
No-Fault	121	32
Fault	57	69

Table 6.11: GA based metrics [8].

	No-Fault	***Fault***
No-Fault	137	16
Fault	58	68

Compared to the cognitive complexity dataset, the predictive models performed poorly with the *Fault*/*No-Fault* dataset. There are three possible explanations:

1) Mislabelled classes as *Fault* or *No-Fault*, in particular some classes labelled *No-Fault* may actually contain faults but not yet fixed (most bug data mining relies on bugs being fixed and associating a class with the documented fix).

2) The relatively small number of available metrics do not capture the required structural information to accurately predict fault-prone modules.

3) There are classifiers available, such a Support Vector Machines, and non-linear classifiers such as Neural Networks; thus, the predictive model used, LDA, may not be the best classifier for fault prone datasets, even if it works well with cognitive complexity.

The LDA performance values in Table 6.7 show that using only the CK metrics gives the poorest performance, with an F-measure of 0.558 for the predictive objective, identifying fault-prone classes. Using all 17 metrics resulted in a slightly higher F-measure of 0.580 while PCA achieved 0.608 predictor performance with 6 metrics that explain 85.5% of the variance. Using the GA based metrics subset an F-measure of 0.650 using 8 metrics was obtained for the *Fault* group. The best overall performance for this dataset was obtained with the GA metrics, with higher F-Measures for both *Fault* and *No-Fault* classes.

6.2 Principal Component Analysis Metrics

The same PCA metric selection strategy as used in Section 5.2 was applied to the *Fault*/*No-Fault* dataset. This is done by first selecting the PCs that contribute to the majority of the variance, and then choosing the metrics whose loading factor is above 0.7 after varimax rotation is applied. Table 6.12 shows the cumulative variance of the PCs, the corresponding cumulative number of metrics that have a loading factor of 0.7 or higher, and the LDA performance for *No-Fault* vs. *Fault*.

Table 6.12: PCs with cumulative [metrics] variance and F-Measure.

Metrics	**#**	**Variance**	**No-Fault**	**Fault**
PC1	1	48.9%		
PC2	2	59.8%	0.729	0.541
PC3	3	68.8%	0.744	0.578
PC4	4	75.6%	0.744	0.563
PC5	5	80.8%	0.743	0.567
PC6	6	85.5%	0.731	0.608
PC7	7	89.2%	0.731	0.608
PC8	8	92.6%	0.725	0.599
PC9	9	95.1%	0.729	0.602

Unlike the cognitive complexity dataset with 63 metrics, the 17 metrics in the Equinox dataset the PCs are highly orthogonal. Each PC had one metric that passed the threshold of 0.7. Following the approach from Section 5.2, the metrics in each PC were added to the subset and evaluated with LDA. The PCs that explained 85.5% of the variance, with 6 metrics, had the best F-Measure performance of 0.608 for classifying the *Fault* classes. Including metrics from additional principal components, which explains more of the dataset's variance, decrease classifier performance.

Table 6.13: F-Measure performance and metrics for best PCA based predictive model.

	No Fault	***Fault***
F-Measure:	0.731	0.608

Grouping	***Metrics***
Size	METH
Complexity	WMC[CK]
Coupling	FIN
Cohesion	LCOM[CK]
Inheritance	DIT [CK]
Encapsulation	PRVATR

The six metrics for this predictor are summarized on Table 6.13. One metric from each category was selected with PCA, three of them from the CK metrics suite; WMC, LCOM, DIT. This model has the same number of metrics as the CK metrics suite but has better performance with an F-Measure of 0.608 and recall of 0.548 versus an F-Measure of 0.558 and recall of 0.460 for a model using only the CK metrics. FIN captures part of what CBO measures, namely the classes that use methods of the class. CBO also measures the external classes the current uses in some fashion (attributes or method calls). The PCA model has an indirect size measure with METH. It is not clear how PRVATR contributes to the classification of *Fault* classes as it is not a direct compliment of PUBATR. That is, having a low number of PRVATR does not mean there will be a high number of PUBATR and vice versa.

6.3 Genetic Algorithm Metrics

The GA selected metrics used in the *Fault* vs. *No-Fault* predictive model are summarized in Table 6.14, along with the F-Measure performance. LOC (direct measure of size) and METH (indirect measure of size) were not present in the best performing model. The one complexity measure, WMC, was not present in the GA selected subset and the ATTR cohesion measure was chosen over LCOM. Three coupling and three inheritance metrics were picked by the genetic algorithm, contributing to the majority of the subset. This could be because that eight of the 17 metrics are coupling and inheritance, but since only one of the four encapsulation metrics was selected (PUBATR) it suggests that coupling and inheritance are important for the prediction of fault prone components in this dataset.

Table 6.14: F-Measure performance and metrics for GA based predictive model.

	No Fault	***Fault***
F-Measure:	0.787	0.650

Grouping	***Metrics***
Size	- *none* -
Complexity	- *none* -
Coupling	CBO [CK], FIN, FOUT
Cohesion	ATTR
Inheritance	ATRINH, METINH, NOC [CK]
Encapsulation	PUBATR

It is often useful to see how the top GA solutions differ, as metrics common in the top performing genes would indicate that these metrics are important for the predictor. Table 6.15 shows the classifier performance, the F-Measure for classifying both the *No-Fault* and *Fault* classes, and all the metrics used by the top three genes. In the top three solutions coupling and inheritance metrics are prevalent but with additional encapsulation metrics. Interestingly, in contrast to the cognitive complexity dataset, there is no implementation/algorithmic complexity metric or a direct measure of size is in any of the top genes. This could be that fault prone classes are more dependent on coupling and inheritance than the size of the module or the coding complexity of the methods. However, with only one coding complexity metric to choose from, it cannot be concluded that implementation complexity does not impact on a module’s fault proneness.

Table 6.15: Metrics in top three genes for *No-Fault* vs. *Fault* LDA predictor.

Metric	**G1 [8] 0.787 / 0.650**	**G2 [9] 0.799 / 0.641**	**G3 [9] 0.791 / 0.637**
ATRINH (inheritance)	●	●	●
ATTR (cohesion)	●	●	●
CBO (coupling)	●	●	
FIN (coupling)	●	●	●
FOUT (coupling)	●		●
METINH (inheritance)	●	●	●
NOC (inheritance)	●		
PRVMET (encapsulation)		●	●
PUBATR (encapsulation)	●	●	●
PUBMET (encapsulation)		●	●
RFC (coupling)		●	●

Table 6.16: Performance for best solution, the union and intersect of top genes.

		***No-Fault* (155)**			***Fault* (126)**		
	#	***Recall***	***Precision***	***F-Measure***	***Recall***	***Precision***	***F-Measure***
Best Solution	8	0.895	0.703	0.787	0.540	0.810	0.650
Intersect of Top 3	5	0.889	0.663	0.760	0.452	0.770	0.570
Union Top 3	11	0.889	0.663	0.760	0.476	0.779	0.591
Intersect of G1/G2	6	0.895	0.699	0.785	0.532	0.807	0.641
Intersect of G1/G3	6	0.895	0.695	0.783	0.522	0.805	0.635

Table 6.16 summarizes the performance of the top solution, the five metrics common to the top three genes, the eleven metrics that form the union of the top three genes, and the intersect of the G1/G2 and G1/G3. Both the intersect and the union of G1/G2/G3 have lower performance than the top gene. However, the union slightly outperforms the intersect for classifying the *Fault* classes. The intersect of G1/G2 and G1/G3 perform closer to the top performing solution but with only six metrics. The difference between the intersects of G1/G2 and G1/G3 versus G1/G2/G3 is an additional coupling or inheritance metric. This would again suggest that coupling and inheritance are contributing factors to fault prone classes.

The fact that the union of G1/G2/G3 with 11 metrics has a lower performance suggests that some metrics may not be needed and decrease classifier performance. Both PUBMET and PRVMET could be redundant as there is no reason for the public methods or private attributes (by themselves) to lead fault prone classes, in fact, private attributes are desirable. PUBMET can be considered a weak cohesion metric, as the larger the public API the more services a class provides.

PUBATR is present in the top performing subset and the intersection of G1/G2 and G1/G3. The importance of this metric is not unexpected since having uncontrolled access to attributes exposes a class to unintentional errors as users of the class can bypass useful checks of acceptable values for the attribute.

6.4 Decision Tree Analysis

As with the cognitive complexity dataset, comparative analysis of various metric selection strategies showed that search-based had the best classification performance. Inspection of the metrics chosen by the genetic algorithm suggests the coupling and inheritance are the most important measures in predicting fault prone Java classes. LDA is black-box predictor and a decision tree classifier was used as a white-box classifier to corroborate if the metrics suggested by GA metric selection have the most discriminatory power.

Figure 6.1 illustrates the C.45 decision tree when using the 8 metrics from the best performing GA subset. The root of the tree is the number of attributes (ATTR), a weak proxy for cohesion, with a decision threshold of zero. From an entropy point of view, it is the root because 64 of the 153 *No-Fault* classes have zero attributes, resulting in a homogenous group with a low entropy score with just 5 *Fault* classes that also have no attributes. However, from a software engineering point of view it is not obvious why a simple count of the number of attributes should be considered the most important in discriminating *Fault* vs. *No-Fault* classes. However, upon inspection, the average size of the classes with no attributes is relatively small at 20 LOC, which means that these classes also have something else in common, small size.

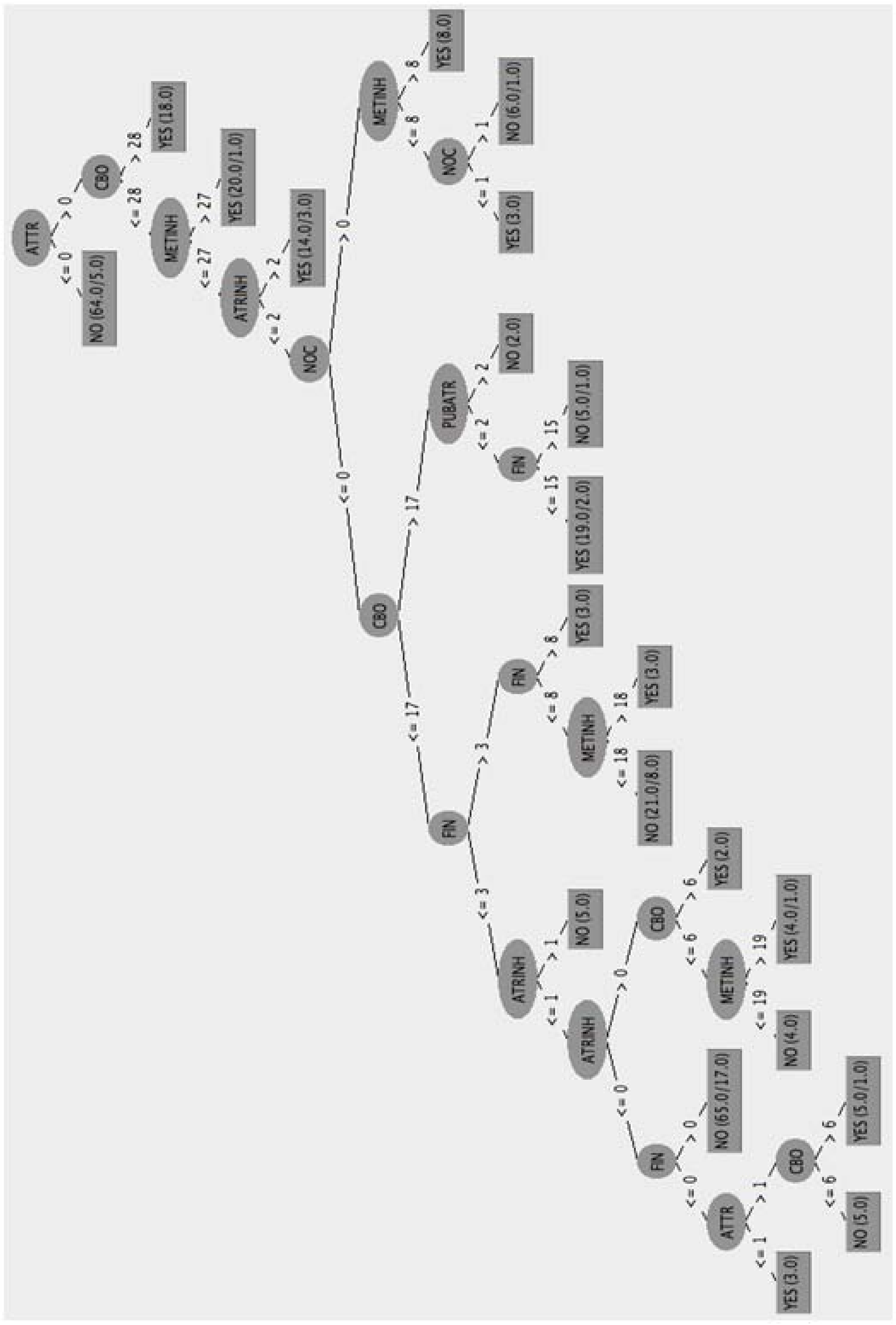


Figure 6.1: Decision tree for the top performing GA subset.

The next metric of importance is CBO, where 18 of the remaining 121 *Fault* classes are correctly classified when a CBO threshold value greater than 28 is used. CBO is followed by three inheritance metrics, METINH, ATRINH and NOC. These three metrics correctly classify 45 of the remaining 101 *Fault* classes. Combined with coupling, the inheritance metrics classify 52% of the *Fault* classes.

After separating the dataset by ATTR, CBO, METINH, ATRINH and NOC, CBO is used again but with a lower threshold of 17, followed by another coupling metric, FIN. The Decision Tree does not use the coupling metric FOUT, resulting in a total of 7 metrics. Compared to the DT for cognitive complex classes (Figure 5.2), the DT for *Fault* vs. *No-Fault* appears to be over trained, that is, the tree seems to be very specific to the dataset with the same metrics being used many times with slightly different thresholds in order to separate a handful of instances. This is especially obvious in the sub-tree after NOC is considered, as shown in Figure 6.2.

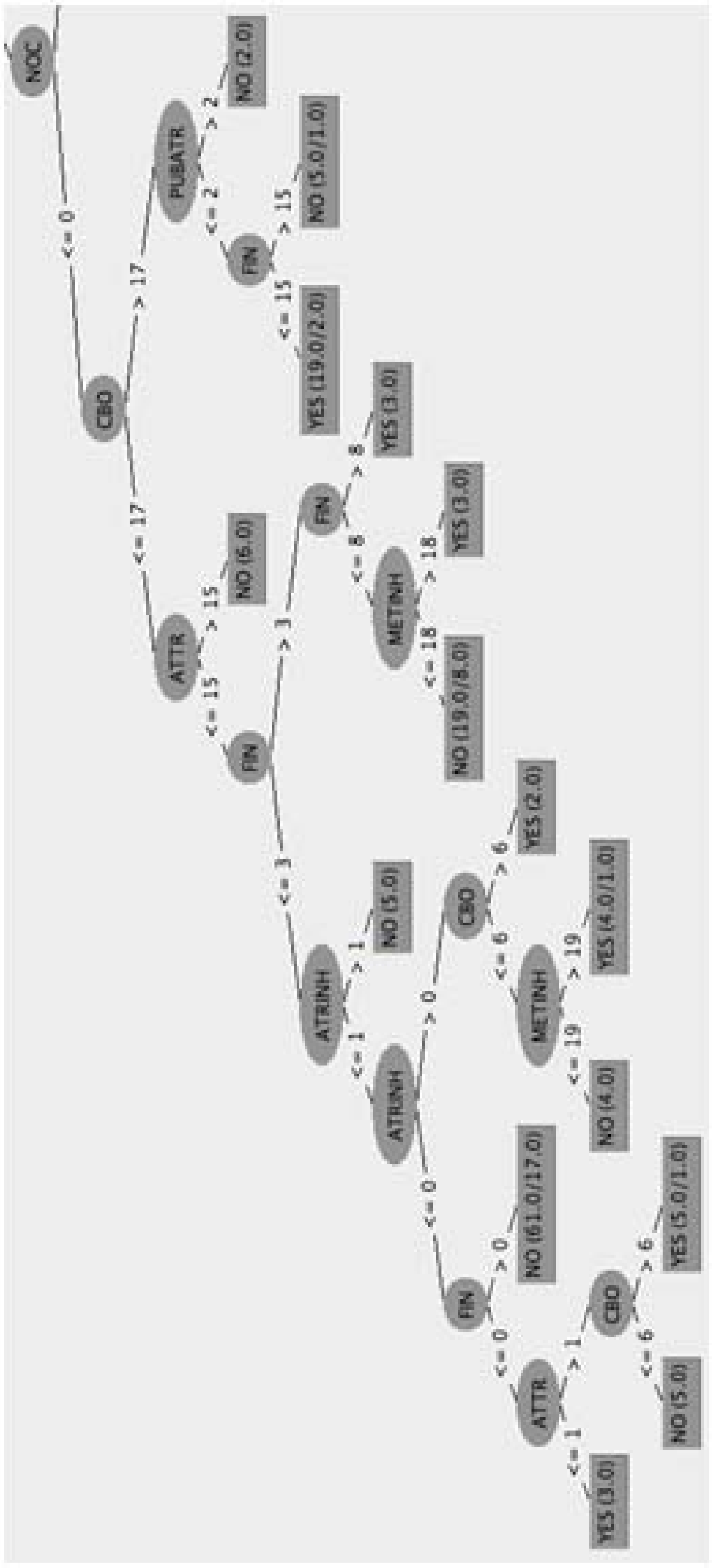


Figure 6.2: Decision sub-tree for the *Fault* vs. *No-Fault* dataset.

Table 6.17: Classifier performance using metrics of best GA solution.

		No-Fault			***Fault***		
	#	***Recall***	***Precision***	***F-Measure***	***Recall***	***Precision***	***F-Measure***
C4.5	7	0.778	0.688	0.730	0.571	0.679	0.621
LDA	7	0.895	0.703	0.787	0.540	0.810	0.650
LDA G1	8	0.895	0.703	0.787	0.540	0.810	0.650

Table 6.17 gives the performance of the DT classifier, LDA with G1 and LDA with the 7 metrics used in the DT. C4.5 has a higher recall for *Fault* classes at 57% compared to 54% for LDA but C4.5 has lower F-Measure with substantially lower precision. When using the seven metrics that the decision tree utilized with the LDA classifier identical performance is achieved, implying that the missing metric, FOUT is a redundant metric when CBO and FIN are used.

Overall the GA metric selection and the decision tree classifier both have coupling and inheritance as important structural metrics to consider when attempting to identify fault prone classes from structural source code measurements. However, when the DT in Figure 6.1 is compared to the cognitive complexity DT in Figure 5.2, it illustrates the over complexity of the DT used to classify fault prone classes. It is an indication that the dataset is not easily separated with the available structural metrics and the DT is likely not generalizable to other datasets.

6.5 Threats to Validity

The threats to validity for the *Fault* vs. *No-Fault* dataset are varied and have a larger impact than with the cognitive complexity dataset. The threats to validity identified in this section, and section 5.5, will be further addressed in the discussion and future work, chapter 8.

Using balanced datasets, where the number of classes are more evenly spread between *Fault* and *No-Fault* samples, should result in improved learning for a classifier. Equinox is more balanced than most fault prediction datasets with 155 *No-Fault* to 126 *Fault*, however the total number of samples (281 files) is very small when compared to a large dataset like Eclipse (over 6,000 files). Ideally the learning dataset should be balanced and large.

Mislabelled data vectors are detrimental to the learning and validation process of classifiers. This dataset, like most other fault prediction datasets, relies upon correct data mining procedures in associating bug reports with source code fixes and a dataset should be used that has the lowest number of potentially mislabelled entries. Another source of mislabelled fault datasets is the fact that bugs go unfixed, and if they are not fixed the matching file is not identified as being buggy. This could be because a user did not report a bug or a reported bug has not been fixed yet. Not much can be done about users not reporting bugs, but in the latter case, a reported bug will be fixed eventually by the maintenance team. Various snapshots of the system should be merged as it will have updated labels when buggy modules are fixed. Improving bug reports should expedite this maintenance task [BPSZ10].

The prediction model for *No-Fault* vs. *Fault* did not perform as well as the cognitive complexity dataset. This could be due to the type of classifier, but LDA was used for both case studies and it performed well for the cognitive complexity predictive objective. A more likely explanation is the lack of structural source code metrics calculated. The previous dataset had 63 metrics, while the bug dataset only 17 with but one implementation complexity metric (WMC) and zero code readability measures. Future datasets should contain as many metrics as possible and let the metric selection process identify the most effective subset.

The suggestion that more metrics should be made available to the subset selection process assumes that structural metrics by themselves are enough to predict fault prone modules. However, there is strong evidence that structural metrics alone are not enough to build accurate predictors of fault-prone components. Hassan suggests that complex code changes is a good predictor of future faults [Has09]. Another study has implied that change coupling, components that are modified during the same maintenance task, is positively correlated to predicting fault-prone components [DLR09].

Finally, software maintenance is a human endeavour; programmers look at code and modify code. This means that human factors are a source of confounding variables, such as developer experience with the programming language, problem domain, development tools, or the company's development process. Other factors that may contribute to the quality of the code are developer's integration with the team, job satisfaction and the pressure of deadlines. Interestingly enough, it has been reported that fixes introduced on Friday or Saturday are more likely to introduce bugs [SZA05].

6.6 Chapter Summary

The analysis on the fault prediction dataset also shows that a search-based metric selection strategy improves predictive models of source code quality. However, the improvement is not as substantial as with the cognitive complexity dataset. This could be because the available metrics, 17 compared to the 63 metrics measured for the cognitive complexity dataset, do not capture enough of the discriminatory properties required to identify fault prone classes from source code. But it could also be that faults are introduced into the final product for other reasons not related to the structural complexities of the system. Human factors, such as deadline pressures and developer experience, could play a larger role than expected. Of the available metrics in this dataset, coupling and inheritance appear to have the largest discriminatory power in identifying fault-prone components. The proposed future work that would help answer the research question, *Is there a connection between a system's structural cognitive complexity and fault prone components*, will be discussed in Chapter 7, *Discussion and Future Work*.

Chapter 7

Discussion and Future Work

Software engineering is a human centric endeavour where the majority of the effort is spent understanding and modifying source code. Predictive models of product quality from source code attempt to identify potentially problematic components that are difficult to understand, maintain, and likely to be fault prone. If we have a better understanding of the source code metrics that are important to capture and measure, we can improve predictive models of software quality. The aim of this research was twofold; first, to advance the state of the art in building better predictive models of software quality and second, to attempt to determine some of the key structural properties that should be monitored in order to maintain a high level of quality in software systems.

One of the most important points to consider when building classifiers is the input metrics. Redundancy and irrelevancy should be removed if possible. With the numerous source code metrics available, choosing an effective subset for a particular project and predictive objective is important and challenging. The current approaches are to use all available metrics; select a subset based on experience or preference; use univariate statistical methods that correlate a metric to the dependent variable, optionally followed by a greedy selection strategy; or use a multivariate statistical approach such as PCA.

This research explored the efficacy of using a search-based metric selection strategy, a genetic algorithm, to improve predictive models of quality from source code measures. Two aspects of quality were explored by using two different Java based applications:

1) Identifying classes with high levels of cognitive complexity that affects program comprehension from a programmer's perspective.
2) Identifying fault-prone classes.

7.1 Cognitive Complexity Dataset

Using LDA as the predictive model, Java classes from a biomedical application were categorized as being *Low*, *Medium* or *High* in cognitive complexity for the purpose of future maintenance. A comparison was done between using all metrics in the dataset, a subset commonly used by other researchers (the CK metrics suite), a statistical multivariate approach using PCA and an evolutionary computational strategy using a genetic algorithm. Comparative analyses demonstrated that the genetic algorithm is able to select a more effective metrics subset.

In the three rank grouping (*Low* vs. *Medium* vs. *High*), using all 63 metrics achieved and F-Measure of 0.758 for high cognitive complexity classes. An LDA model using only the six CK metric had lower performance at 0.745. The PCA derived metrics (a subset of 35) had the lowest performance at 0.742. The best predictive model used a subset of 28 metrics obtained with a search-based metric selection (GA) with a recall of 0.860 and F-Measure of 0.804.

Since identifying high cognitive complexity classes was the goal, the dataset was separated into two groups, *Low/Medium* vs. *High*. With this grouping, the best performing model was again the GA derived metrics set, with recall of 0.884 and F-Measure of 0.854 using a subset of 28 metrics. The majority of the metrics identified by GA were traditional measures of code complexity (mostly Halstead metrics). This could be because they are so highly correlated to size. The GA subset also consisted of direct size measures but as a per-method metric such as mean LOC as oppose to a value for the entire class. Coupling and inheritance metrics also contributed to the best model but cohesion did not seem to be as important. This could be due to the difficulty in accurately measuring cohesion as it depends so much on intended functionality.

One of the appeals of the CK metrics set is that a relatively small set of measures need to be evaluated and understood by developers in order to capture the system's structural complexity and build predictive models. The above analysis of selecting a subset from a larger pool of metrics demonstrates the need to use more metrics than just the CK suite. Search-based metric selection resulted in 28 measures, which is fine for a machine-learning predictive model, but too many for developers to conceptualize effectively. Decision tree analysis (DT) was used as a second stage to determine if there are some general areas of structural measurements that developers need to focus on in order to capture the cognitive complexity of a system.

The DT metrics of importance are size-correlated code complexity measures (Halstead complexity), inheritance and coupling. The GA metric set included a code readability metric and additional coupling measures, suggesting that a developer uses non-coding information and various aspects of system coupling when determining its cognitive complexity. Comparisons of classifier performance between DT with GA derived metrics shows that while the DT performs adequately with fewer metrics (8 out of total of 63), the GA metrics set of 28 outperforms the DT with an F-Measure of 0.854 vs. 0.782 and recall of 0.884 vs. 0.791. This implies that there is not one particular complexity, coupling or inheritance metric that is vital, but rather, it is the combination of the various ways of measuring these important structural properties that provide an optimal predictive model of cognitive complexity. This analysis corroborates what most experienced developers intuitively know, that a system's coupling, code complexity, inheritance and code readability affect cognitive complexity and program comprehension.

7.2 Fault vs. No-Fault Dataset

The same analysis strategy was a used with a *Fault* vs. *No*-Fault dataset, a Java based plug-in for the Eclipse IDE. Bug reports for Equinox were inspected and matched with the source code repository's bug fix information to label classes as *Fault*, while all the other classes were labelled as *No-Fault*.

Using LDA as the classifier with leave one out validation, all 17 metrics achieved a recall of 0.476 and F-Measure of 0.580 for the *Fault* group. Applying the six CK metrics with LDA resulted in a lower performance with a recall of 0.460 and F-Measure of 0.558. The PCA derived metrics, a set of six metrics, had the highest recall at 0.548 and F-Measure 0.608. The best predictive model used a subset of eight metrics selected by the genetic algorithm and attained a recall of 0.540 and F-Measure of 0.650. The search based metric selection strategy resulted in the best performing predictive model for both datasets.

Inspecting the GA selected metrics and the decision tree analysis suggests that like the cognitive complexity dataset, coupling, inheritance and encapsulation are contributing factors in identifying fault prone modules. Another point to consider is that there are fewer metrics in the fault prediction dataset; in particular, there are no Halstead complexity metrics or code readability metrics so it is difficult to infer if all the metrics useful for cognitive complexity would apply for fault prone classes.

Other studies give strong evidence that using only structural measures is not enough for predicting fault prone modules. [RGP07] carried out an empirical study of three projects and found that the time series of non-source code metrics gave the best results (correlation coefficient). The project metrics that demonstrated highest correlation to bug density were: number of authors, commit messages, bug fix lines deleted, author switches and lines added. Such studies strongly suggest that for fault prone module prediction models, process and evolution metrics play an important role.

7.3 Threats to Validity

There are various threats to the validity of this research, some of which will be addressed in future work. One threat to validity is that only LDA was used as the classifier. GA is a wrapper metric selection strategy and needs to build a predictive model for the desired objective in order to obtain fitness values. Different classifiers may not only use different metrics, they are likely to achieve different performance as well. To optimize predictions, the best performing classifier should be used for the type of project being considered. To generalize the applicability of the selected metrics, the initial pool of metrics should be evaluated with different classifiers to investigate what is different in the chosen metrics sets and identify the metrics that are used regardless of classifier type.

The GA algorithm was implemented in a proprietary system where the only classifier available was LDA. Future work will include the implementation of the GA in the statistical framework R [R-Project]. R has coding facilities to implement new algorithms and it also has additional classifiers, such as artificial neural nets and support vector machines, which could be used and compared to LDA.

The remaining threats to validity will be addressed by analyzing the Eclipse dataset for both cognitive complexity and fault prone classes. This dataset has been mined for faulty components by [ZPZ07] but no one has associated cognitive complexity rankings with the classes. By using the same dataset for both objectives we can help answer the research question: *Are components identified as overly complex by developers also fault prone?*

7.4 Future Work – Eclipse Dataset

The Eclipse dataset from [ZPZ07] was not used in this study as the fault-prone objective because it lacked sufficient metrics. It only had four class level metrics with five additional method level metrics and did not include any of the CK metrics. Eclipse is open source and in the future work as many source level metrics as possible will be calculated. This will address the validity threat that the metrics available to the classifier did not capture the subtle differences in structural properties that can better predict fault prone or overly complex modules. By having the same metrics set available to both predictive objectives, comparisons between the different metrics subsets can be made. This will help us answer the research question: *Do different predictive objectives for the same project require different source code metrics?*

The Eclipse project is widely used by many researchers in mining source repositories for bug detection models. This means that new techniques for associating bug reports with source code repositories are being developed and over time the threat of mislabelled data will be reduced as new models can be built with the new labels. Eclipse is also a very active open source project which means bug reports are dealt with relatively quickly. By mining later versions of Eclipse with more bug fixes, classes that were labelled as not faulty will now be correctly marked as having had faults.

Obtaining subjective rankings of cognitive complexity via source code inspections is very time consuming. Most studies that use human subjects for code inspections have a small number of inspectors and a small number of classes, something that can be done in a reasonable amount of time as the inspections are usually voluntary or part of a course in software engineering. Tackling a large project such as Eclipse is daunting and it is not feasible to expect an individual developer to inspect all the classes in Eclipse (over 6,000 files).

Fortunately, the eclipse development community is very active and very large, due to the popularity of the IDE in educational institutes and industry. The fact that it is an open source project also helps in making the source available to a large number of developers who may have a vested interest in participating in the subjective evaluation of the system. A large pool of potential inspectors lessens the amount of effort required from a developer as an individual can inspect as many files as desired. An online system that allows inspectors to evaluate classes on their own time will be deployed.

This system will have various metrics that will address some of the threats to validity. To account for the confounding affects of inspector experience a basic questionnaire will be provided asking questions such as:

1) Experience with object-oriented programming
2) Experience programming with Java.
3) Experience developing with the Eclipse project.
4) Experience with source code inspections.

An important consideration for the system is how to choose classes for inspection. Clearly an individual developer would not be expected to rank every class in Eclipse, yet a large balanced dataset is desired. The files identified as fault prone should be inspected as one research question is to determine if there is a correlation between fault prone classes and its level of cognitive complexity. But to balance the dataset, a file not labelled as faulty must also be inspected. One approach would be to first randomly choose a file from Eclipse. How random should the file selection be? Maximum coverage is desired, thus files with no or few inspections should be preferred. The developer's second file to inspect should be one from the opposite label (if the first file is a faulty file, the next one should not be labelled faulty). Mining the Eclipse dataset for both faults and cognitive complexity would be an important contribution to the empirical software engineering research community.

7.5 Contribution

The ability to use predictive models to identify potentially problematic components would assist developers and project managers to make best use of limited resources when taking mitigating actions such as detailed code inspections, more exhaustive testing, and refactoring. By using search-based metric selection and decision trees to help identify the critical source-level structural metrics to measure, we can develop better tools that can monitor a system as it is developed.

This research has demonstrated the efficacy of using a search-based metric selection strategy in improving predicting models of product quality in software engineering. As well, by using two different predictive objectives, cognitive complexity and fault-prone modules the applicability of structural metrics has been characterized.

Following the Goal-Question-Metric paradigm, the majority of the structural source code metrics are more applicable to cognitive complexity, that is, structural properties that affect human understanding. These metrics may be used indirectly as proxies for predictors of fault-prone modules, but they are not enough. Other non-structural metrics are needed to develop more accurate faulty module predictors.

Empirical studies, from a programmer's point of view, on source code analysis and program comprehension of non-trivial systems will have a significant impact on the development of tools that are user-centric and ease the mental burden on developers. Tools that effectively automate various aspects of source code analysis (identifying overly complex and fault prone modules for example) are important in reducing development and maintenance costs while enhancing product quality as systems become increasingly more complex and ubiquitous in all aspects of modern life.

Appendix - Parallel GA Implementation

A prototype has been developed using Scopira, a C++, open-source, multi-platform framework developed in-house at the Institute for Biodiagnostics, National Research Council Canada, for the development of biomedical data analysis applications [VDJ+07]. A parallel GA for feature subset selection has been implemented using Scopira's built-in Agents Library. Figure A.1 shows the application's main user interface with the monitor panel of a GA run.

Multiple datasets can be loaded and analyzed multiple times. The results are associated with the corresponding datasets in a hierarchical fashion. Figure A.2 shows the GA options the user can set; the number of genes, the number of generations, the percent elite and the probability of mutation. Figure A.3 shows the GA results window. The window has various panels that show different aspects of the results. The "GA Run Log" panel shows the parameters used and all the genes in the population, along with their overall fitness values.

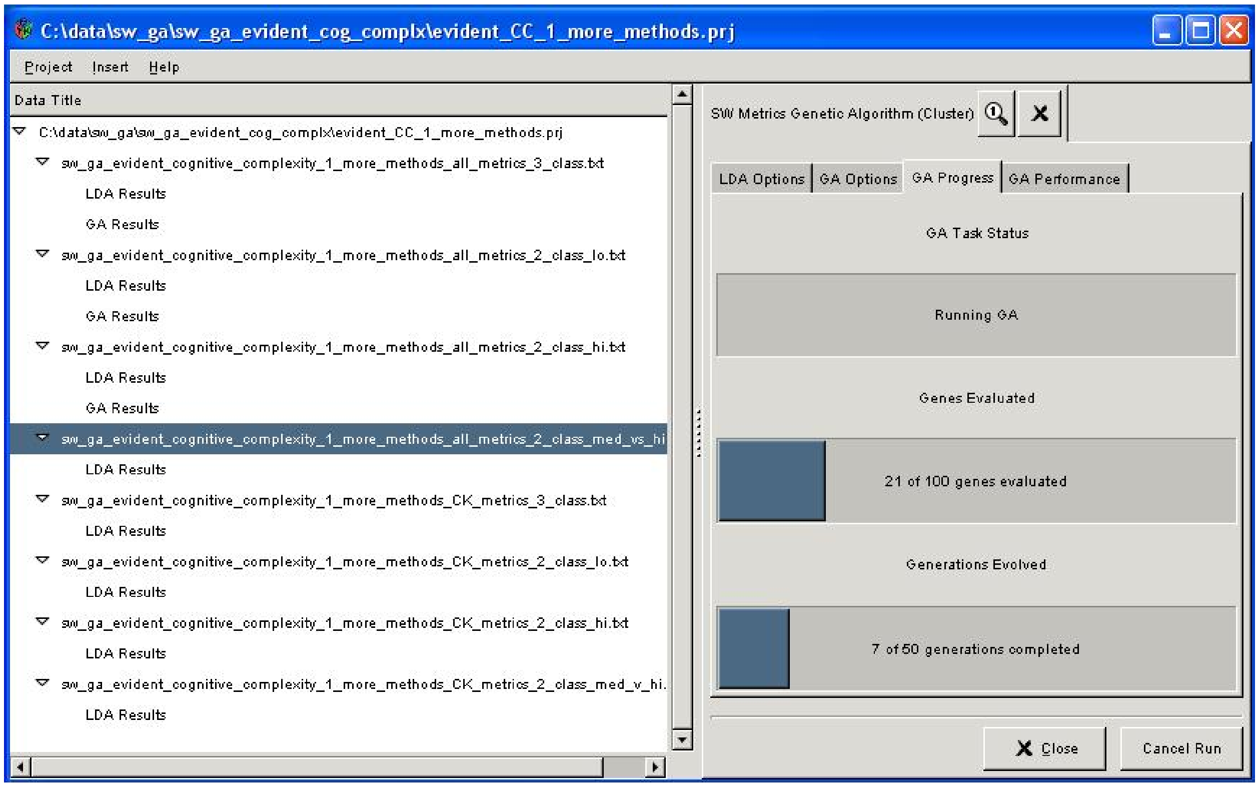
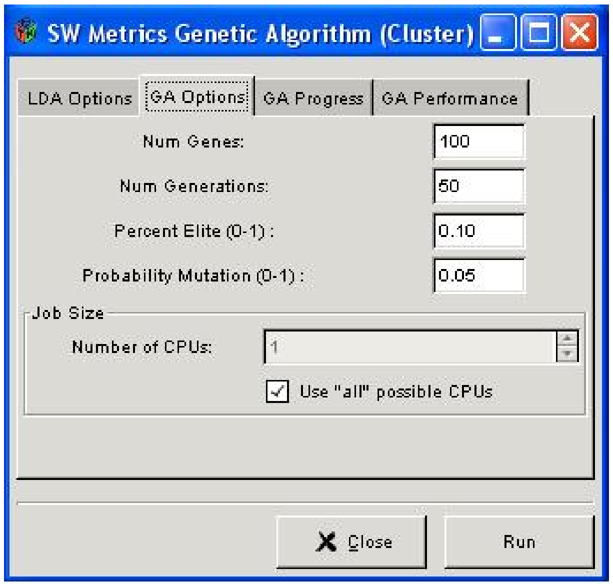
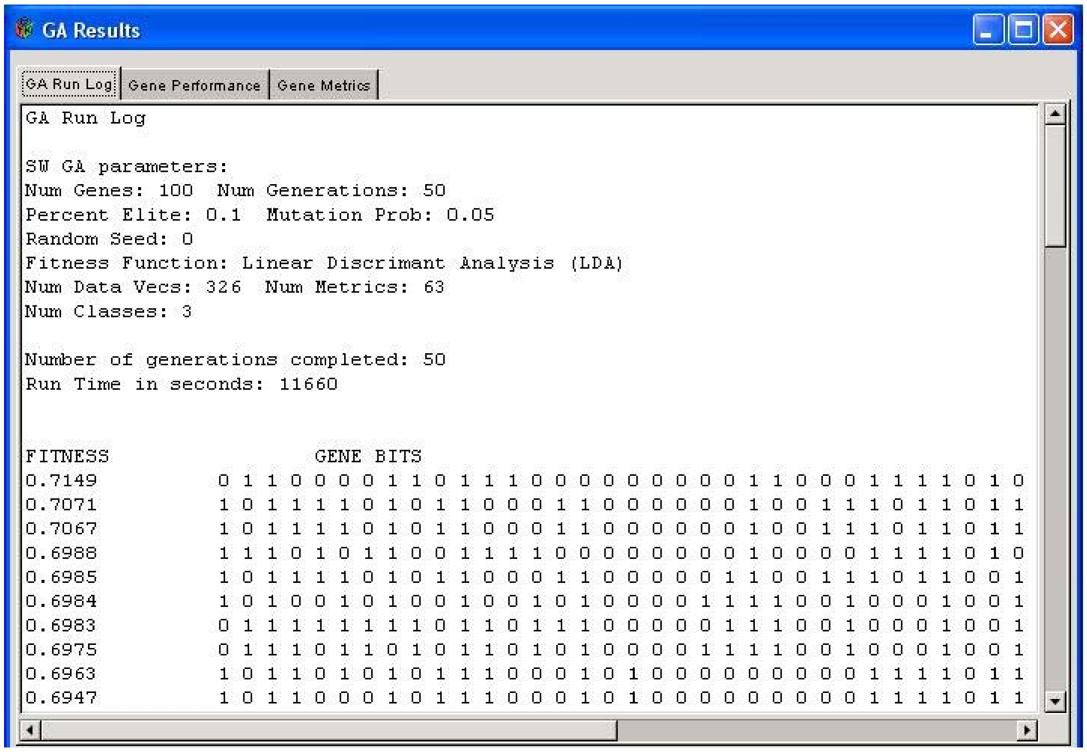

C:\data\sw_ga\sw_ga_evident_cog_complx\evident_CC_1_more_methods.prj
Project Insert Help
Data Title
C:\data\sw_ga\sw_ga_evident_cog_complx\evident_CC_1_more_methods.prj
sw_ga_evident_cognitive_complexity_1_more_methods_all_metrics_3_class.txt
LDA Results
GA Results
sw_ga_evident_cognitive_complexity_1_more_methods_all_metrics_2_class_lo.txt
LDA Results
GA Results
sw_ga_evident_cognitive_complexity_1_more_methods_all_metrics_2_class_hi.txt
LDA Results
GA Results
sw_ga_evident_cognitive_complexity_1_more_methods_all_metrics_2_class_med_vs_hi
LDA Results
sw_ga_evident_cognitive_complexity_1_more_methods_CK_metrics_3_class.txt
LDA Results
sw_ga_evident_cognitive_complexity_1_more_methods_CK_metrics_2_class_lo.txt
LDA Results
sw_ga_evident_cognitive_complexity_1_more_methods_CK_metrics_2_class_hi.txt
LDA Results
sw_ga_evident_cognitive_complexity_1_more_methods_CK_metrics_2_class_med_v_hi.
LDA Results
SW Metrics Genetic Algorithm (Cluster)
LDA Options
GA Options
GA Progress
GA Performance
GA Task Status
Running GA
Genes Evaluated
21 of 100 genes evaluated
Generations Evolved
7 of 50 generations completed
Close
Cancel Run

Figure A.1: Main interface to the GA feature selection prototype for software metrics.



Num Genes: 100
Num Generations: 50
Percent Elite (0-1) : 0.10
Probability Mutation (0-1) : 0.05
Job Size
Number of CPUs: 1
✓ Use "all" possible CPUs

Figure A.2: Parallel genetic algorithm options.

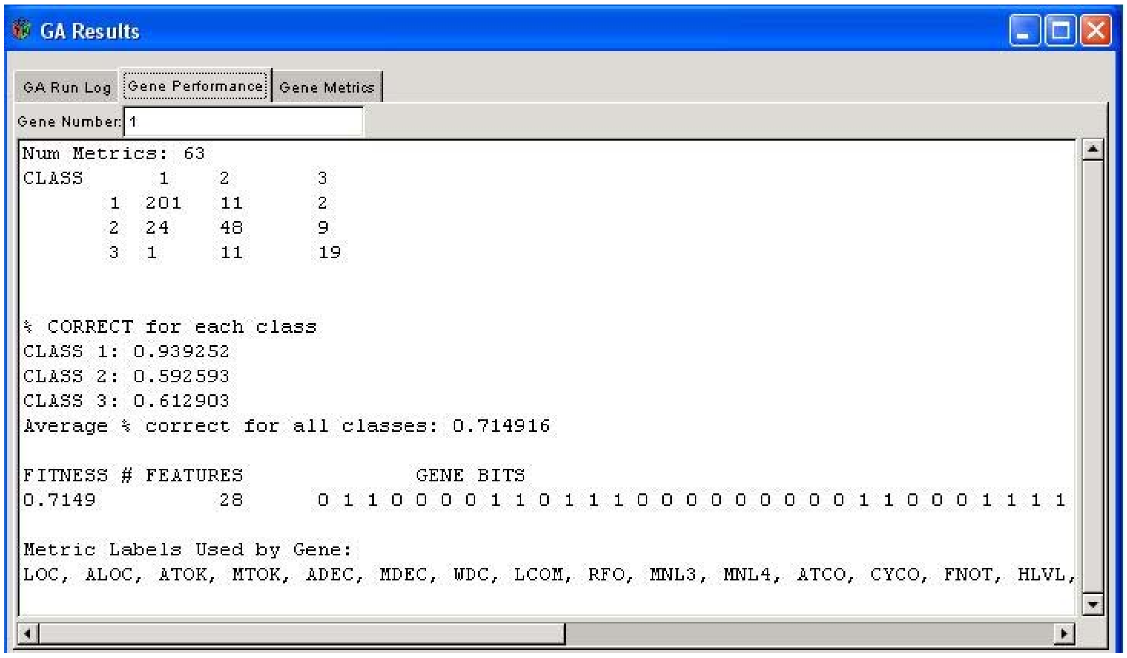


```
GA Run Log

SW GA parameters:
Num Genes: 100  Num Generations: 50
Percent Elite: 0.1  Mutation Prob: 0.05
Random Seed: 0
Fitness Function: Linear Discrimant Analysis (LDA)
Num Data Vecs: 326  Num Metrics: 63
Num Classes: 3

Number of generations completed: 50
Run Time in seconds: 11660


FITNESS              GENE BITS
0.7149        0 1 1 0 0 0 0 1 1 0 1 1 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 1 0 1 0
0.7071        1 0 1 1 1 1 0 1 0 1 1 0 0 0 1 1 0 0 0 0 0 0 1 0 0 1 1 1 0 1 1 0 1 1
0.7067        1 0 1 1 1 1 0 1 0 1 1 0 0 0 1 1 0 0 0 0 0 0 1 0 0 1 1 1 0 1 1 0 1 1
0.6988        1 1 1 0 1 0 1 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 1 0 1 0
0.6985        1 0 1 1 1 1 0 1 0 1 1 0 0 0 1 1 0 0 0 0 0 1 1 0 0 1 1 1 0 1 1 0 0 1
0.6984        1 0 1 0 0 1 0 1 0 0 1 0 0 1 0 1 0 0 0 0 1 1 1 1 0 0 1 0 0 0 1 0 0 1
0.6983        0 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 0 0 0 0 0 1 1 1 0 0 1 0 0 0 1 0 0 1
0.6975        0 1 1 1 0 1 1 0 1 0 1 1 0 1 0 1 0 0 0 0 1 1 1 1 0 0 1 0 0 0 1 0 0 1
0.6963        1 0 1 1 0 1 0 1 0 1 1 1 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1
0.6947        1 0 1 1 0 0 0 1 0 1 1 1 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1
```

Figure A.3: Run log, GA parameters and genes (with overall fitness values).

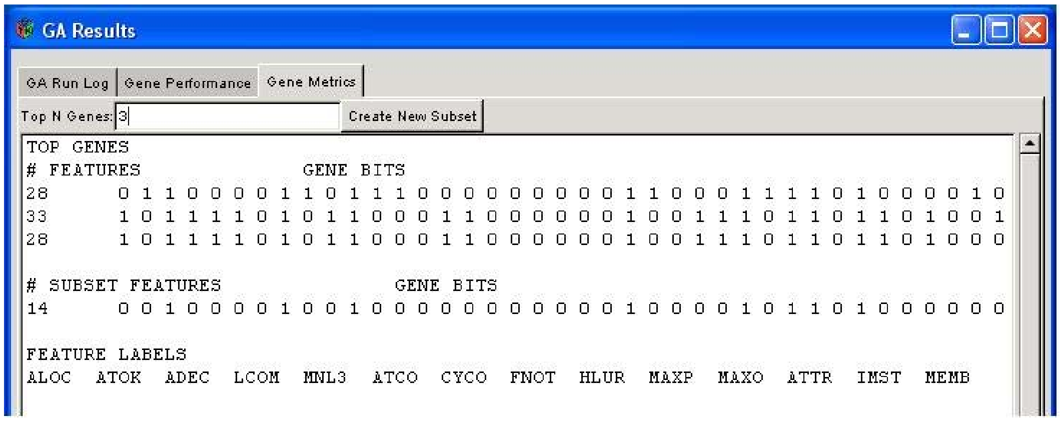
The “Gene Performance” panel, Figure A.4, gives more information on a particular gene. It has the confusion matrix, the encoded bit mask and the corresponding metrics labels. The confusion matrix gives a more detailed performance of the predictive model. The performance of interest is the ability to correctly identify the problematic classes, shown as class three (3) in Figure A.4.

Figure A.5 shows the gene metrics panel where users can see which measures appear in the top N genes in the final population. The intersection of the solutions is also shown, as this gives an indication of the metrics that are common to various solutions.



```
Num Metrics: 63
CLASS       1     2       3
       1  201    11       2
       2  24     48       9
       3  1      11       19


% CORRECT for each class
CLASS 1: 0.939252
CLASS 2: 0.592593
CLASS 3: 0.612903
Average % correct for all classes: 0.714916

FITNESS # FEATURES             GENE BITS
0.7149          28      0 1 1 0 0 0 0 1 1 0 1 1 1 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 1

Metric Labels Used by Gene:
LOC, ALOC, ATOK, MTOK, ADEC, MDEC, WDC, LCOM, RFO, MNL3, MNL4, ATCO, CYCO, FNOT, HLVL,
```

Figure A.4: Detailed performance of a gene; confusion matrix and metric labels.



```
TOP GENES
# FEATURES             GENE BITS
28      0 1 1 0 0 0 0 1 1 0 1 1 1 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 1 0 1 0 0 0 0 1 0
33      1 0 1 1 1 1 0 1 0 1 1 0 0 0 1 1 0 0 0 0 0 0 1 0 0 1 1 1 0 1 1 0 1 1 0 1 0 0 1
28      1 0 1 1 1 1 0 1 0 1 1 0 0 0 1 1 0 0 0 0 0 0 1 0 0 1 1 1 0 1 1 0 1 1 0 1 0 0 0

# SUBSET FEATURES              GENE BITS
14      0 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 1 1 0 1 0 0 0 0 0 0

FEATURE LABELS
ALOC   ATOK   ADEC   LCOM   MNL3   ATCO   CYCO   FNOT   HLUR   MAXP   MAXO   ATTR   IMST   MEMB
```

Figure A.5: Metrics that appear in the top three solutions.

The parallel GA application was used to determine the metrics subset that gave the best LDA performance for two datasets. These metric subsets are then compared to LDA models that use all available metrics, the CK metrics and a PCA derived metrics subset. The dataset with cognitive complexity as the predictive objective was analyzed in Chapter 5. The second dataset, which has fault-prone Java classes as the predictive objective, was analyzed in Chapter 6.

Bibliography

[ADH05] G. Antoniol, M. Di Penta, M. Harman, “Search-Based Techniques Applied to Optimization Planning for a Massive Maintenance Project,” *Proc. IEEE International Conference on Software Maintenance (ICSM)*, pp. 240-249, 2005.

[AGMT07] G. Antoniol, YG. Guehenue, E. Merlo, P. Tonella, “Mining the Lexicon Used by Programmers during Software Evolution,” *Proc. IEEE International Conference on Software Maintenance (ICSM)*, pp. 14-23, 2007.

[AL03] M. Alshayed, W. Li, “An Empirical Validation of Object-Oriented Metrics in Two Different Software Processes,” *IEEE Transactions on Software Engineering*, Vol. 28, No. 11, pp. 1043-1049, 2003.

[ABF04] E. Arisholm, L.C. Briand, A. Foyen, “Dynamic Coupling Measurement for Object-Oriented Software”, *IEEE Transactions on Software Engineering*, Vol. 30, No. 8, pp. 491-506, 2004.

[BPSZ10] S. Breu, R. Premraj, J. Sillito, T. Zimmermann, “Information Needs in Bug Reports: Improving Cooperation Between Developers and Users,” *Proc. ACM Computer Supported Cooperative Work (CSCW)*, Savannah, Georgia, USA, 2010.

[BAC+99] L. Briand, E. Arisholm, S. Counsell, F. Houdek, P. Thevenod-Fosse, "Empirical Studies of Object-Oriented Artifacts, Methods, and Processes: State of the Art and Future Directions", *Empirical Software Engineering*, Vol. 4, No. 4, pp. 387-404, 1999.

[BDKZ93] R.D. Banker, S.M. Datar, C.F. Kemerer, D. Zweig, "Software Complexity and Maintenance Costs," *Communications of the ACM*, Vol. 36, No. 11, pp. 81-94, 1993.

[BEDL99] J. Bansiya, L. Etzkorn, C. Davis, and W. Li, "A Class Cohesion Metric For Object-Oriented Designs," *Journal of Object-Oriented Programming*, vol. 11, pp. 47-52, 1999.

[BBM96] V.R. Basili, L.C. Briand, W.L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators", *IEEE Transactions on Software Engineering*, Vol. 22, No. 10, pp. 751-761, 1996.

[BR88] V.R. Basili, H.D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Transactions on Software Engineering*, Vol. 14, No. 6, pp.758-773, 1988.

[Bel61] R. Bellman, *Adaptive Control Processes: A Guided Tour,* Princeton Press, 1961.

[BKS02] S. Bouktif, N. Kegl, H. Sahraoui, "Combining Software Quality Predictive Models: An Evolutionary Approach," *Proc. of the International Conference on Software Maintenance (ICSM)*, pp. 385-392, 2002.

[BDW98] L.C. Briand, J.W. Daly, J. Wust, “A Unified Framework for Cohesion Measurement in Object-Oriented Systems”, *Empirical Software Engineering*, Vol. 3, pp.65-117, 1998.

[BDW99] L.C. Briand, J.W. Daly, J.K. Wust, “A Unified Framework for Coupling Measurement in Object-Oriented Systems”, *IEEE Transactions on Software Engineering*, Vol. 25, No. 1, pp. 91-121, 1999.

[BWDP00] L.C. Briand, J. Wust, J.W. Daly, D.V. Porter, “Exploring the relationships between design measures and software quality in object-oriented systems,” *The Journal of Systems and Software*, Vol. 51, pp. 245-273, 2000.

[BW02] L.C. Briand, J. Wust, “Empirical Studies of Quality Models in Object-Oriented Systems”, *Advances in Computers*, Vol. 56, pp. 97-166, 2002.

[Can00] E. Cantu-Paz, *Effective and Accurate Parallel Genetic Algorithms,* Kluger Academic Publishers, 2000.

[CKB04] H.S. Chae, Y.R. Kwon, D.H. Bae, “Improving Cohesion Metrics for Classes by Considering Dependent Instance Variables”, *IEEE Transactions on Software Engineering*, Vol. 30, No. 11, pp. 2004.

[CL93] J-Y. Chen, J-F. Lu, “A new metric for object-oriented design,” *Information and Software Technology*, pp. 323-340, 1993.

[CK94] S.R. Chidamber, C.F. Kemerer, “A Metrics Suite for Object Oriented Design”, *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp. 476-493, 1994.

[CLR+02] M. Ciolkowski, O. Laitenberger, D. Rombach, F. Shull, D. Perry, “Software Inspections, Reviews & Walkthroughs,” *Proc. of the 24th IEEE International Conference on Software Engineering (ICSE)*, pp. 641-642, Orlando, Florida, USA, 2002.

[CM06] D. Cubranic, G. C. Murphy, "Hipikat: Recommending pertinent software development artifacts," *Proc. of the 25th IEEE International Conference on Software Engineering (ICSE)*, pp. 408-418, Portland, Oregon, USA, 2003.

[DLR09] M. D'Ambros, M. Lanza, R. Robbes, "On the Relationship Between Change Coupling and Software Defects," *Proc. of Working Conference on Reverse Engineering*, pp.135-144, 2009.

[DLR10] M. D’Ambros, M. Lanza, R. Robbes, “An Extensive Comparison of Bug Prediction Approaches,” *Proc. of MSR (7th IEEE Working Conference on Mining Software Repositories)*, Cape Town, South Africa, 2010.

[DTB03] E. Diaz, J. Tuya, R. Blanco, “Automated Software Testing Using a Metaheuristic Technique Based on Tabu Search,” *Proc. IEEE International Conference on Automated Software Engineering (ASE)*, pp. 310-313, 2003.

[DHS00] C.D. Duda, P.E. Hart, D.G. Stork, *Pattern Classification 2nd Edition*, John Wiley & Sons, 2000.

[Dun89] G. Dunteman, *Principal Component Analysis*, Sage University Press, 1989.

[Ema00] K.E. Emam, “A Methodology for Validating Software Product Metrics,” *NRC ERB*-1076, Institute for Information Technology, National Research Council of Canada, 2000.

[EMM01] K.E. Emam, W. Melo, J.C. Machado, "The prediction of faulty classes using object-oriented design metrics", *The Journal of Systems and Software*, Vol. 56, pp. 63-75, 2001.

[EBGR01] K.E. Emam, S. Benlarbi, N. Goel, S.N. Rai, “The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics,” *IEEE Transactions on Software Engineering*, Vol. 27, No. 7, pp. 630-650, 2001.

[Ema02] K.E. Emam. Object-Oriented Metrics: A Review of Theory and Practice. In *Advances in Software Engineering*, editors, H. Erdogmus, O. Tanir, Springer, 2002.

[EBD99] L. Etzkorn, J. Bansiya, C. Davis, "Design and Complexity Metrics for OO Classes," *Journal of Object-Oriented Programming*, Vol. 12, No. 1, pp. 35-40, 1999.

[Equ10] Equinox Eclipse Plug-In, URL: http://eclipse.org/equinox/, Accessed April 2010.

[FP97] N.E. Fenton, S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach, 2nd Edition*, PWS Publishing Company, 1997.

[FPG03] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," *Proc. International Conference on Software Maintenance (ICSM)*, Amsterdam, Netherlands, 2003.

[Glo90] F. Glover, “Tabu search: A tutorial”, *Interfaces*, Vol. 20, No. 4, pp. 74-94, 1990.

[Gol98] D.E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, 1989.

[GS06] G. Gui, P.D. Scott, "Coupling and Cohesion Measures for Evaluation of Component Reusability," *Proc. of the International Workshop on Mining Software Repositories (MSR)*, pp. 18-21, 2006.

[GMCS04] L. Guo, Y. Ma, B. Cukic, H. Singh, "Robust Prediction of Fault-Proneness by Random Forests," *Proc. of the 15th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 417-428, 2004.

[GFS05] T. Gyimothy, R. Ferenc, I. Siket, "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction," *IEEE Transactions on Software Engineering*, Vol. 31, No. 10, pp. 897-910, 2005.

[Hal77] M.H. Halstead, *Elements of Software Science*, Elsevier North-Holland, 1977.

[Har07] M. Harman, "The Current State and Future of Search Based Software Engineering," *Proc. of the 29th IEEE International Conference on Software Engineering (ICSE)*, Minneapolis, USA, 2007.

[HJ01] M. Harman, B.F. Jones, "Search-based software engineering," *Information and Software Technology*, Vol. 43, pp. 833-839, 2001.

[HCN98] R. Harrison, S.J. Counsell, R.V. Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics," *IEEE Transactions on Software Engineering*, Vol.24, No. 6, pp.491-496, 1998.

[Has09] A. E. Hassan, “Predicting Faults Using the Complexity of Code Changes,” *Proc. of the 31st IEEE International Conference on Software Engineering (ICSE)*, pp 78-88, Vancouver, Canada, 2009.

[HKP91] J. Hertz, A. Krogh, R.G. Palmer, *Introduction to the Theory of Neural Computation*, Addison-Wesley, 1991.

[HO07] E.W. Host, B.M. Ostvold, “The Programmer’s Lexicon, Volume 1: The Verbs,” *Proc. IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 193-202, 2007.

[ISO91] “Information technology – Software product evaluation: Quality characteristics and guidelines for their use”, *ISO IS 9126*, International Standardization Organization, Geneva, 1991.

[KUST07] S. Kanmani, V.R. Uthariaraj, V. Sankaranarayanan, P. Thambidrai, “Object-oriented software fault prediction using neural Networks”, *Information and Software Technology*, Vol. 49, pp. 483-492, 2007.

[KPB06] P. Knab, M. Pinzger, A. Bernstein, “Predicting Defect Densities in Source Code Files with Decision Tree Learners,” *Proc. of the International Workshop on Mining Software Repositories (MSR)*, pp. 119-125, 2006.

[LM06] M. Lanza, R. Marinesur, *Object-Oriented Metrics in Practice*, Springer, 2006.

[LFB07] D. Lawrie, H. Field, D. Binkley, “Extracting Meaning from Abbreviated Identifiers,” *Proc. IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 213-222, 2007.

[LB85] M. Lehman, L. Belady, *Program Evolution: Processes of Software Change*, London Academic Press, London, 1985.

[Li98] W. Li, “Another metric suite for object-oriented programming”, *The Journal of Systems and Software*, Vol. 44, pp. 155-162, 1998.

[LK94] M. Lorenz, J. Kidd, *Object-Orieted Software Metrics*, Prentice Hall, Englewood Cliffs, NJ, 1994.

[MV00] A. Mockus, L.G. Votta, “Identifying Reasons for Software Changes using Historic Databases,” *Proc. International Conference on Software Maintenance*, pp. 120-130, San Jose, USA, 2000.

[McC76] T.J. McCabe, “A complexity Measure”, *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, pp. 308-320, 1976.

[Mun05] M.J. Munro, “Product Metrics for Automatic Identification of ‘Bad Smell’ Design Problems in Java Source-Code,” *Proc. International Software Metrics Symposium*, Como, Italy, 2005.

[MNS01] G. Murphy, D. Notkin, K.J. Sullivan, “Software Reflexion Models: Bridging the Gap between Design and Implementation,” *IEEE Transactions on Software Engineering*, Vol. 27, No. 4, pp. 364-380, 2001.

[NBZ06] N. Nagappan, T. Ball, A. Zeller, “Mining Metrics to Predict Component Failures,” *Proc. of the International Conference on Software Engineering (ICSE),* pp. 452-461, 2006.

[PD07] G.J. Pai, J.B. Dugan, "Empirical Analysis of Software Fault Content and Fault Proneness Using Bayesian Methods," *IEEE Transactions on Software Engineering*, Vol. 33, No. 10, pp. 675-686, 2007.

[PA06] S.L. Pfleeger, J.M. Atlee, *Software Engineering, Theory and Practice 3rd Edition*, Prentice Hall, 2006.

[Pig96] T.M. Pigoski, *Practical Software Maintenance: Best Practices for Managing your Software Investments*, Wiley Computer Publishing, 1996.

[PVS01] N.J. Pizzi, R. Vivanco, R.L. Somorjai, "EvIdent: a functional magnetic resonance image analysis system," *Artificial Intelligence in* Medicine, Vol. 21, pp. 263-269**,** 2001.

[PV03] S. Purao, V. Vaishnavi, "Product Metrics for Object-Oriented Systems," *ACM Computing Surveys*, Vol. 35, No. 2, pp. 190-221, 2003.

[Qui93] J.R. Quinlan, *C4.5 Programs for Machine Learning*, Morgan Kaufmann Publishers, California, USA, 1993.

[R-Project] The R Project for Statistical Computing, URL: http://www.r-project.org/, Accessed March 25, 2010.

[RGP07] J. Ratzinger, H. Gall, M. Pinzger, "Quality Assessment Based on Attribute Series of Software Evolution," *In Proc. Working Conference on Reverse Engineering (WCRE07)*, Vancouver, Canada, 2007.

[RN03] S.J. Russel, P. Norvig, *Artificial Intelligence: A Modern Approach (2nd ed.)*, Upper Saddle River, NJ, Prentice Hal, 2003.

[SL08] R. Shatnawi, W. Li, “The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process,” *The Journal of Systems and Software*, article in press.

[SS02] B. Scholkopf, A.J. Smola, *Learning with Kernels*, MIT Press, 2002.

[SSB06] O. Seng, J. Stammel, D. Burkhart, “Search-Based Determination of Refactoring for Improving the Class Structure of Object-Oriented Systems,” *Proc. Genetic and Evolutionary Computation Conference (GECCO)*, pp. 1909-1916, 2006.

[SZA05] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes? On Fridays," *Proc.of the International Workshop on Mining Software Repositories (MSR)*, St. Louis, Missouri, USA, 2005.

[TKC99] M.H. Tang, M.H. Kao, M.H. Chen, “An Empirical Study on Object-Oriented Metrics”, *Proc. International Software Metrics Symposium*, pp. 242-249, 1999.

[TQ05] M.T. Thwin, T.S. Quah, “Application of neural networks for software quality prediction using object-oriented metrics,” *The Journal of Systems and Software*, Vol. 76, pp. 147-156, 2005.

[VG06] C. Van Koten, A.R. Gray, “An application of Bayesian network for predicting object-oriented software maintainability,” *Information and Software Technology*, Vol. 48, pp. 59-67, 2006.

[VP04] R. Vivanco, N. Pizzi, “Finding Effective Software Metrics to Classify Maintainability Using a Parallel Genetic Algorithm,” In *Proc. Genetic and Evolutionary Computation Conference (GECCO 04)*, Seattle, USA, 2004.

[VDJ+07] R. Vivanco, A.B. Demko, M. Jarmasz, R.L. Somorjai, N.J. Pizzi, “A Pattern Recognition Application Framework for Biomedical Datasets,” *IEEE Engineering in Medicine and Biology Magazine*, pp. 82-85, March/April 2007.

[WF05] I.H. Witten. E. Frank, *Data Mining: Practical machine learning tools and techniques*, 2nd Edition, Morgan Kaufmann, San Francisco, 2005.

[XGL05] F. Xing, P. Guo, M.R. Lyu, “A Novel Method for Early Software Quality Prediction Based on Support Vector Machine,” *Proc. IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pp. 213-222, 2005.

[ZPZ07] T. Zimmermann, R. Premraj, A. Zeller, “Predicting Defects for Eclipse,” *Third International Workshop on Predictor Models in Software Engineering (PROMISE)*, Minneapolis, USA, May 20-26, 2007.