SLICE-BASED COGNITIVE COMPLEXITY METRICS FOR DEFECT PREDICTION

A dissertation submitted

to Kent State University in partial

fulfillment of the requirements for the

degree of Doctor of Philosophy

by

Basma Alqadi

August 2020

Dissertation written by

Basma Alqadi

B.S., King Saud University, Saudi Arabia, 2006

M.S., University of Houston, USA, 2014

Ph.D., Kent State University, USA, 2020

Approved by

Dr. Jonathan I. Maletic , Chair, Doctoral Dissertation Committee

Dr. L. Gwenn Volkert, Members, Doctoral Dissertation Committee

Dr. Kambiz Ghazinour Naini

Dr. Joseph Ortiz

Dr.Robin Selinger

Accepted by

Dr. Javed Khan, Chair, Department of Computer Science

Dr. Mandy Munro-Stasiuk, Interim Dean, College of Arts and Sciences

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First, I would like to show my greatest appreciation to my research advisor, *Prof. Jonathan I. Maletic*, for believing in me, and allowing me the time and freedom to pursue my own interests. His endless patience, unfailing wisdom and excellent coaching are the primary reasons this dissertation was actually completed.

I express my deepest gratefulness toward my wonderful parents, *Suliman* and *Madawi*, who did more than their best to raise, educate and support me. Thank you for keeping me in your thoughts and prayers. None of this would ever have been possible without the enormous support, and patience from the closest soul, my beloved husband, *Khaled*. Thanks to the sweetest kids, sliver of my heart, eyes of glory, in which I see the world through, *Mohammed, Suliman*, *Saud* and *Lena*. My sincere appreciation goes to my sisters and brothers. I am truly lucky and blessed to have them in my life.

I would like to extend my grateful thanks to my colleagues and friends in software engineering development laboratory <SDML>, Computer Science Department, and Kent State University, who helped me in different ways to accomplish my research and dissertation. I wish them luck in their careers. Finally, I greatly thank my dissertation committee for their appreciated services, efforts, and precious time.

Basma Alqadi

August 2020, Kent, Ohio

CHAPTER 1

INTRODUCTION

Given the centrality of software in modern life, defective software code can have negative impact on company stock and brand. Tricentis, a software-testing company analyzed 606 software fails from 314 companies to better understand the business and the financial impact of software failures. The report shows that these software failures affected 3.6 billion people and caused $1.7 trillion in financial losses in 2017 (Tricentis 2017). From a brand value perspective, software defects can affect a customer's confidence in a system or product. For example, there are many stories about how every Microsoft product is released with a list of known issues. Microsoft could have avoided the problems before shipping the product to the customer. However, this would have taken considerable time, and cost a large amount of money and personnel.

Software defect prediction approaches are of tremendous interest both in academia and industry. Early identification of defects helps reduce the costs associated with locating and fixing defects. As expected, the relative costs to find and repair defect increase dramatically as system maturity increases (Pressman 2015). Figure 1.1 illustrates this issue. The industry average cost to correct a defect during code generation is approximately $977 per error. The industry average cost to correct the same defect if it is discovered during system testing is $7,136 per defect while the cost of the same defect in the maintenance

phase is $14,102. For software organization, the cost savings associated with early quality control and assurance activities are compelling.

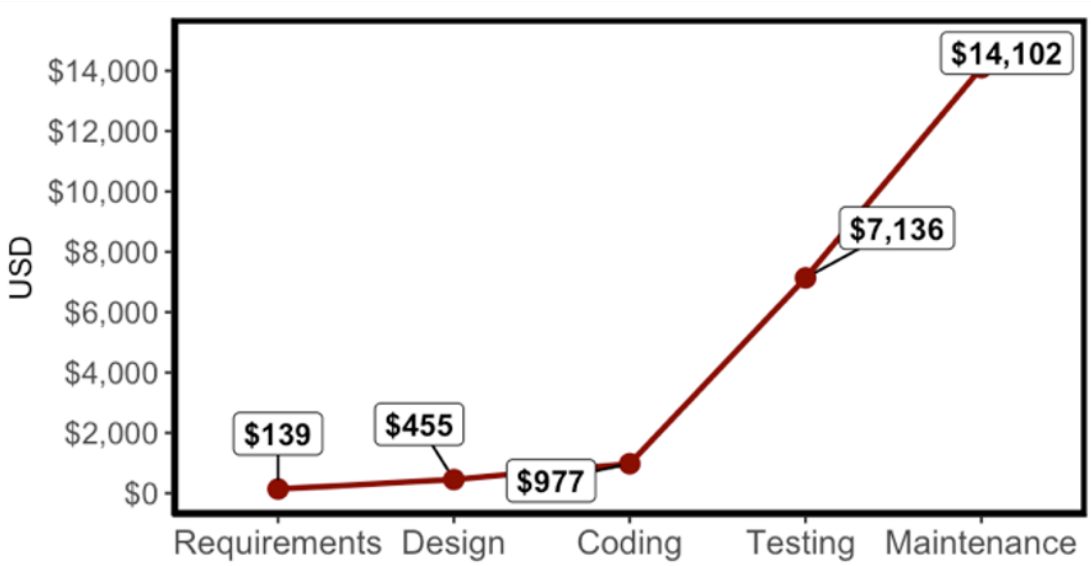


Figure 1.1: Relative cost of correcting defects. *Source* (Pressman 2015)

Therefore, the driving scenario of defect prediction is the limitation of resources for software Quality Assurance (QA), which may include manual code inspections, technical review meetings, and intensive testing. Such resources are always limited by time and by cost, e.g., the deadlines that development teams face to release the product or not enough personnel are available for QA. When managers want to spend resources more effectively, they would typically allocate them on the parts where they expect most defects or at least the most severe ones, which is usually based on their experience of the product and hence make further decisions on testing, inspections, etc.

Defect prediction uses machine-learning algorithms to build models. These models predict the areas of software code where defects are likely to occur. That provides the list

of defect-prone software modules, which can represent a system, a software component (or package), a source code file, a class, a function (or method), and/or a code line according to prediction granularity. Accordingly, QA can effectively allocate limited resources by spending more effort on the modules that are likely to be defective (contains at least one defect). As the size of software projects becomes larger, defect prediction techniques will play an important role to support developers as well as to speed up time to market with more reliable software products.

Models for defect prediction rely on independent and dependent variables. Independent variables are normally software quality metrics collected from software systems. Researchers identified several metrics using different information, such as code metrics and process metrics. Software code metrics (SCMs) used in the models include Lines of Code (LOC) (Fenton and Bieman 2014; Weyuker, Ostrand, and Bell 2010; Nam et al. 2017), Halstead size metrics (Halstead 1977), object oriented metrics (Bird et al. 2009; D'Ambros, Lanza, and Robbes 2010; Khoshgoftaar et al. 1996; Wang, Liu, and Tan 2016) and complexity measures (Turhan et al. 2009; Menzies, Greenwald, and Frank 2007; Mende and Koschke 2010; Zhang et al. 2016). Process metrics include code churn (Nagappan and Ball 2005), revision control histories (Bell, Ostrand, and Weyuker 2006; Graves et al. 2000; Moser, Pedrycz, and Succi 2008; Ostrand, Weyuker, and Bell 2005; Bird et al. 2011; Rahman and Devanbu 2013; Nucci et al. 2018) and number of previous faults identified (Hassan and Holt 2005; Kim et al. 2007; D'Ambros, Lanza, and Robbes 2010). The dependent variables in the models are normally defect variables (e.g. the number of defects predicted in a module, or if a defect has been predicted in the module).

1.1 Motivations

This dissertation is motivated by several factors. First is that existing defect prediction approaches lack metrics to estimate program understandability effort of the source code. It has been shown that software defects are often the result of the incomplete or incorrect comprehension of a program segment (Chen et al. 2018). Therefore, finding sections of code that presents a comprehension challenge to the developer can be the basis for isolating code that has a greater risk of defects. Existing metrics for understandability are often tied to readability or syntactic features of source code such as structural complexity. However, understandability is a cognitive and semantic aspect; a developer can find a piece of code readable, but still difficult to understand (Scalabrino et al. 2017). Much of the research on cognitive models explains how programmers comprehend code using a bottom-up approach (Storey, Wong, and Müller 2000; Storey 2005). The programmer analyzes the source code statement by statement and gradually develop control-flow and data-flow abstractions through the process of chunking (Pennington 1987). Program chunks are grouped together to form larger chunks, until the entire program is understood. In this way a hierarchical semantic representation of the program is built from the bottom-up. Thus, assessing the cognitive complexity of program semantic chunks can be a criterion for characterizing defects for defect prediction. Specifically, in order to make accurate predictions, the metrics need to be discriminative: capable of distinguishing one instance of code region from another of different cognitive complexity.

Second, most of existing metrics, for defect prediction, focus on syntactic aspects of software modules, such as lines of code (LOC), number of declarations, number of

functions, etc. It is known that software systems have well-defined syntax, which can be represented by Abstract Syntax Trees (ASTs) (Hindle et al. 2012) and at the same time have semantics, which is hidden deeply in source code (White et al. 2015). It has been shown that such semantic information is useful for tasks such as code completion, effort estimation and bug detection (Li and Zhou 2005; Nguyen and Nguyen 2015; Alomari, Collard, and Maletic 2014; Tu, Su, and Devanbu 2014; Hindle et al. 2012). Such semantic information should also be useful for characterizing defects to improve defect prediction. Specifically, most existing metrics only focus on single elements and rarely take the interactions between elements into account. However, with the emergence of static and dynamic bug localization techniques, the nature of defects has changed and today most defects in bug databases are of semantic nature (Li et al. 2006).

Third, most of the traditional metrics suffer from being very coarse grained with low capability that measure only a small sub-set of code features. Gray et al. advocate that the coarse grained nature of such metrics prevents machine learning algorithms from effectively differentiate between defective and non-defective modules: if two modules have the same metric values, e.g., LOC, but they have not been labeled the same in terms of their defectiveness, this will obstruct the algorithm's ability to learn (Gray et al. 2011). Gray et al. identify many modules that have identical values across number of metrics but different defectiveness labels. This highlights that commonly used metrics are not sufficient enough for defect prediction.

Lastly, defect predictors based on static code attributes are calculated using static analysis, since they do not require the execution of code. An advantage of static code

attributes is that they can be easy, quickly and automatically collected from the source code, even if no other information is available (Turhan et al. 2009). By contrast, process metrics that are widely introduced and based on information extracted from software archives are among the most expensive ones to collect. These process metrics quantify aspects of software development process such as changes of source code, and ownership of source code files that may be unavailable or hard to characterize especially in new projects and projects without perfect historical records. In practice, employing defect prediction technique should not be expensive in term of time for both data collection and constructing the prediction models themselves (Moser, Pedrycz, and Succi 2008).

To bridge the gap between program understandability and features used for defect prediction, this dissertation proposes a novel set of cognitive complexity metrics by utilizing program slicing to predict defects. Program slicing is a reduction technique that traces the data and control dependencies for determining only those parts of the original program that are relevant to the computation of a given feature of interest (Weiser 1984). Program slicing has been successfully employed for program comprehension during different maintenance tasks such as testing and debugging (Alomari, Collard, and Maletic 2014; Meyers and Binkley 2007; Counsell, Hall, and Bowes 2010). A program slice consists of all the statements that may influence the value of a specific variable at a given program point (Horwitz, Reps, and Binkley 1988; Ferrante, Ottenstein, and Warren 1987). Specifically, we compute program-slicing metrics based on forward, static, non-executable, inter-procedural program slice for each variable in a system and then utilize these features to train a defect prediction model.

Unlike straightforward code metrics based on line counts and statement counts, slice-based cognitive complexity metrics have the potential to consider more insightful code properties based on program behaviors, as captured by program slices and obtained from program analysis and points-to analysis. The slice-based metrics give different weights to each statement based on their significance in the control dependence and flow dependence in the program. For example, a while predicate that encloses multiple statements will contribute more than one control dependence in slice-based cognitive complexity metrics, while in code metrics it typically contributes only one source code line.

Previous work on the computation of program slices is most often been based on the notion of a Program Dependence Graph (PDG) (Ottenstein and Ottenstein 1984) or one of its variants, e.g., a System Dependence Graph (SDG) (Liang and Harrold 1998). Unfortunately, all these approaches suffer from scalability and computational issues due to the fact that building the PDG is complicated in terms of time, space, and data related operations. Consecutively, the use of program slicing approaches in academia and industry has been somewhat limited over years. However, with the emergence of the lightweight and highly scalable slicing tool namely *srcslice* (Alomari et al. 2014; Newman et al. 2016), program slicing can be used to address a number of applications and problems that in practice cannot be (or are extremely costly) addressed with other heavyweight slicing approaches. *srcslice* eliminates the time and effort needed to build the entire PDG of the program by combining a text-based approach with a lightweight static analysis infrastructure that only computes dependence information as needed (aka on-the fly) while

computing the slice for each variable in the program (Alomari et al. 2014; Newman et al. 2016).

In this dissertation, an empirical investigation is performed to determine if cognitive complexity correlates with, and predict defects, on parts of the version history of 10 datasets extracted from 7 open-source systems. That is, to determine their effectiveness in helping practitioners find defects when taking into account the effort needed to test or inspect the code. Like other work on defect prediction, machine learning techniques are used to build regression models from the metric data applied to older versions of a system. This allows evaluation of the prediction models on more recent versions of the system. Additionally, we adapt the most commonly used code metrics and process metrics, including size, structural complexity, Halstead's, line changed, and function changed as the baseline metrics. We first employ correlation analysis to analyze the relationships between slice-based metrics and defect proneness. Then, build multivariate prediction models to investigate the prediction ability of slice-based metrics in defect-proneness (regression models). Finally, we build multivariate prediction models using baseline metrics to examine the effectiveness of slice-based metrics compared to the baseline code and process metrics. In order to obtain comprehensive performance evaluations and ensure that the conclusions that we draw about our models are robust, we evaluate the effectiveness of prediction using the out-of-sample bootstrap validation technique, which has been shown to yield the best balance between the bias and variance (Tantithamthavorn et al. 2017).

1.2 Research Questions

To achieve the goal of this dissertation, we attempt to answer the following two research questions:

RQ1. Do slice-based cognitive complexity metrics significantly correlate to defects?

RQ2. Do slice-based cognitive complexity metrics contribute to the prediction of the probability of defects?

The purpose of these questions is to investigate whether cognitive complexity metrics can effectively lead to significant relationship to defect prediction. These questions are critically important to both software researchers and practitioners, as they help to answer whether slice-based cognitive complexity metrics are of practical value. We choose to use defects as one widely used indicator of software quality and known as a result of comprehension difficulty.

1.3 Contributions

The contributions of this research are relevant for academic and practical activities as follow:

- A new set of slice-based cognitive complexity metrics that capture more fine-grained program properties and pay special attention to interactions between source code elements. These metrics measure static code attributes that can be collected in an easy, quickly and automatically process, even if no other information is available. While others have proposed slicing as a means for defect prediction (Black et al. 2009; Pan, Kim, and Jr 2006; Black et al. 2006), there is no other work

that provides empirical results of the use on realistically sized software systems and indicates evidence of cognitive complexity.

- This work is one of the first that empirically studies the relationship between program understandability and the probability of defects. We validate the correlations between slice-based cognitive complexity metrics and defect-proneness. Results show that most slice-based metrics are statistically related to defect-proneness in an expected direction.
- In a thorough large-scale empirical investigation, we compare slice-based metrics with the most commonly used code and process metrics including size, structural complexity, Halstead's metrics, line changed, and function changed. Results show that slice-based metrics measure essentially different quality information than the baseline code metrics measure and the metrics in general outperform the most commonly used code and process metrics in defect proneness prediction.
- The study provides valuable data in an important area for which there is limited experimental data available. For our analysis, we collect data from industry and made it publicly available for the use of other researchers and practitioners.
- As a practical contribution, we believe that our analysis and the proposed methodology allow the construction of defect predictors even for projects with no local defect data is available.
- Lastly, practitioners can easily adopt the proposed metrics for defect prediction as computing them is scalable to large systems. Generally, program slicing time consuming to compute, however here we take advantage of a lightweight high

scalable and publicly available program slicing approach to compute the necessary information.

1.4 Organization

The following is an overview of the chapters that appear in this dissertation:

CHAPTER 2 provides an overview of what defects are, why they are a problem and how they are currently predicted. It gives the background of common process of software defect prediction that relies on machine learning models.

CHAPTER 3 describes program slicing in more detail and how this relates to the work carried out in this research.

CHAPTER 4 introduces the proposed slice-based cognitive complexity metrics and describes the methodology for the work undertaken in computing the metrics. It discusses the use of *srcSlice* tool to collect analysis data and extract slice-based metrics.

CHAPTER 5 discusses the process used to create the corpus and how we extract defects data, baseline metrics and slice-based metrics.

CHAPTER 6 explains the experimental design and the techniques used to build various statistical prediction models and identify the statistical measures for validating the association between program slicing and defective modules.

CHAPTER 7 provides statistical results to the research questions from applying different models and discusses the potential impact of these results across defect prediction.

CHAPTER 8 discusses the research threats to validity and looks at specific types of validity threats and ways used to avoid them.

CHAPTER 9 concludes this dissertation and highlights future directions of this research.

1.5 Publication Notes

CHAPTER 4 is published at the 35th IEEE International Conference on Software Maintenance and Evolution (ICSME) (Alqadi 2019). CHAPTER 5, CHAPTER 6, and CHAPTER 7 are published at the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) (Alqadi and Maletic 2020).

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter defines the term defect, describes process of defect prediction, and summarizes the methods and metrics have been used for prediction. It highlights the current limitations present in defect prediction and how this can potentially be overcome.

2.1 Software Defect Prediction

IEEE defines a defect within software as “an imperfection in a software product where the product does not meet its requirements or specifications”. Defects appear as the result of errors made during software creation. For a defect to be known as a fault the error must be discovered during software execution. A defect is not known as a fault if it is detected during testing, or inspection before executing the software (“IEEE Standard Classification for Software Anomalies” 2010). Figure 2.1 is a diagram taken from (“IEEE Standard Classification for Software Anomalies” 2010) that presents the relationship between problems, errors, defects and faults as a UML class diagram. The diagram shows that a failure could be the result of problem with the system and a failure could cause one or many problems. A fault is a specific type of defect that is discovered during the software execution and could cause one or more failures.

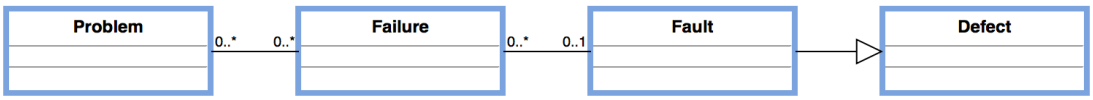


Figure 2.1: A diagram depicting the relationship between problems, failures, faults, and defects ("IEEE Standard Classification for Software Anomalies" 2010).

Within the context of the software process, there is no distinction between fault and defect. Both terms imply a quality shortcoming that is encountered after the software has been delivered to end-users (or to another activity in the software process) (Pressman 2015). These different issues are interdependent and connected to each other. Usually, occurrence of one leads to the introduction of other, which together impacts the functionality of the software. Additionally, in the general consensus within the software engineering community, the point in time that the problem is discovered has no effect on the term used to describe the shortcoming. This means that defects, errors, faults, and bugs are all synonymous. This same nomenclature is followed in this dissertation.

There are many reasons why defects arise in software. A software defect can be the result of inadequate planning and specifications, poor design or coding practice, use of immature technology, or incompatibilities with an underlying level. While some defects are trivial, some other defects can cause major consequences. Software failures can be devastating to company value and reputation. Pressman pointed out in his book "Software Engineering" that the earlier a defect is fixed, the less cost involved in fixing said defect. He shows that once a piece of software makes it into the field, the cost of fixing a defect can be up to 100 times as high as it would have been during the development stage

(Pressman 2015). Defective code can also arise the cost of litigation from irate customers suing suppliers for poorly implemented systems.

Defect prediction is therefore important as it indicates potential artifacts that could contain defects, allowing resources to be assigned to these artifacts of a software system that have a greater propensity to defects. Defect prediction models use software metrics on which to base their decisions. Software metrics are a measure of some property of particular software modules that can represent a system, a software component (or package), a source code file, a class, a function (or method), and/or a code change according to prediction granularity.

2.2 Software Defect Prediction Process

The common process of software defect prediction relies on machine learning models. Figure 2.2 shows a typical prediction process commonly used in the literature (Bacchelli, D'Ambros, and Lanza 2010; Bird et al. 2011; D'Ambros, Lanza, and Robbes 2012; Nam 2015). The key insight behind these models is learning from software evolution history. Most software uses software configuration management (SCM) systems to record the evolution of a software project. Recorded data includes change history, change log messages, and bug fixes that cover years of data and can be a useful resource for learning from previous defects and predicting the new ones.

Software defect prediction relies on three main components; dependent variables, independent variables and a model. The first step in building a prediction model is to collect instances and history information from software archives such as version control systems (commit messages), issue tracking systems, email archives, and so on. Instances can

represent different granularity such as system, a software component (or package), a source code file, a class, a function (or method), or a code change level. Processing the raw data falls into two folds:

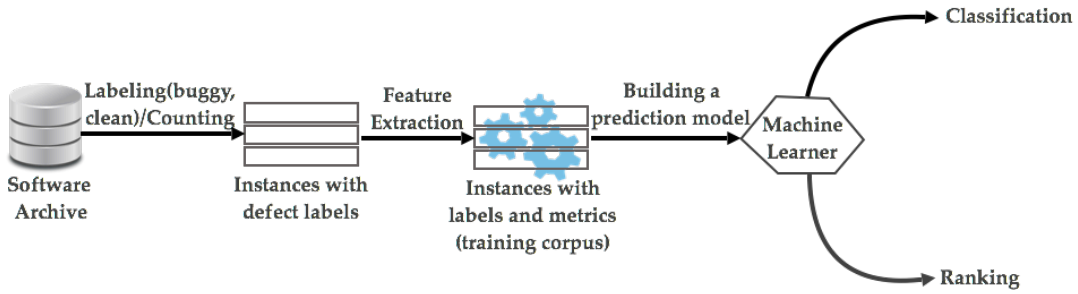


Figure 2.2: Common process of software defect prediction.

1. Labeling instances as buggy/clean or defects count. Defects data are the model dependent variables.
2. Extracting metrics (features) to determine useful patterns in bug fix or occurrence that can be applied for prediction. Metrics are the independent variables, which can describe the software code, how it has changed or who changed it. Independent variables come mostly in two forms, software code metrics; those that can be derived from the software artifact, and process metrics; metrics that measure the change process of software artifact over time.

After generating the corpus, i.e., instances with metrics and labels, preprocessing techniques can be applied which are common in machine learning. Such techniques used in defect prediction studies include feature selection, data normalization, and noise

reduction (Zhang et al. 2014; Nam et al. 2017; Tantithamthavorn et al. 2018). Preprocessing is an optional step and were not applied on all defect prediction studies, e.g., (D'Ambros, Lanza, and Robbes 2010; Zimmermann and Nagappan 2008). The final step is training a prediction model, so the model can predict whether a new instance has a bug or not. The prediction for defect-proneness (buggy/clean) of an instance is based on binary classification, while that for the defects count in an instance is based on ranking.

The model contains the rule(s) or algorithm(s) that predict the dependent variable from the independent variables. These rules can be as simple as the number of independent variables in the model or be as complicated as decision trees and regression techniques. Decision tree creates a graph of decisions based on the chance of an event happening while regression technique seeks to determine best fit of independent value(s) based on a dependent value(s). Defect prediction modeling is an important area of research and the subject of many previous studies. A study by Hall et al. identified over 200 defect prediction studies published and the models/metrics used to carry out defect prediction (Hall et al. 2012).

2.3 Dependent Variables (Defects Data)

In defect prediction, the dependent variables are the variables that indicate whether an artifact is defective or not. The dependent variables can be counts (i.e. the number of defects in an artifact) or categorical (i.e. an artifact is defective or not). A defect can come in two forms - a pre-release or a post-release defect ("IEEE Standard Classification for Software Anomalies" 2010). A pre-release defect is one that is found and fixed during

development and testing before a product is released while a post release defect manifests itself when a customer experiences a failure with the product.

Machine learning models can predict the dependent variables to forecast if a module is defective or not. This result can then be tested to examine the power of the forecast by determining different kinds of statistical measures such as the recall and precision of the model. Precision and recall are measures of relevance of the data used. Precision is a measure of the accuracy of the model used to predict defects (i.e. of all the instances predicted defective, how many are actually defective). Recall is the measure of relevant retrieval of instances (i.e. how many instances are identified by the model as defective out of all those defective instances that should have been returned). Since precision and recall have trade-offs, f-measure, which is a harmonic mean of precision and recall, can be used to compare different prediction models. It is one of the most frequent measures used in defect prediction classification to compare between different classification models.

2.4 Independent Variables (Software Metrics)

Many research studies in a decade have focused on proposing new metrics to build statistical prediction models. Widely studied metrics can be categorized into two kinds: code metrics and process metrics. Sections 2.4.1, 2.4.2, and 2.4.3 detail these independent variables, describing which have been used, why they were used and how effective they have been.

2.4.1 Source Code Metrics (SCM)

Source code metrics (SCM) measure how source code is complex and are directly collected from existing source code. The main assumption of the code metrics is that code with higher complexity can be more bug prone. To measure code complexity, researchers proposed various metrics. These metrics and the studies they appear in are described below.

2.4.1.1 Source Lines of Code (SLOC)

SLOC was introduced as a simple size measure that might represent the complexity of software system and indicate potential defective areas. There are two major types of SLOC measures: physical SLOC (LOC) and logical SLOC (LLOC). LOC measures the size of a software program by counting the number of lines in the text of the program's source code. Specific definitions of physical SLOC measure vary, some studies include comment lines, some include blank lines and others omit one or both of these lines. However, Rosenberg showed that the format of the LOC was irrelevant as they all correlate with each other (Rosenberg 1997). Other studies use logical lines of code (LLOC), but their definitions are tied to specific computer languages, e.g., for C-like programming languages LLOC is the number of statement-terminating semicolons. Unfortunately, SLOC measures are often stated without giving their definition, and LLOC can often be significantly different from LOC.

One of the first defect prediction models proposed by Akiyama was built on SLOC in 1971 (Akiyama 1971). Akiyama built a simple regression model using (SLOC) for determining the number of defects in the system. Using SLOC has some advantages; it is very quick to calculate and easily transferred across different languages. Fenton and

Ohlsson analyzed pre and post release defects of a large communications system (Fenton and Ohlsson 2000). They found that LOC was good at ranking the most fault-prone modules. Zhang confirmed the ranking ability of LOC discovered by Fenton and Ohlsson and showed that LOC can be useful predictors of defects at both package and file level (Zhang 2009). Ostrand et al. proposed a simple LOC based model to predict defect density in a large industrial system (Ostrand, Weyuker, and Bell 2005). Their results reveal that a model based on LOC was a good indicator for predicting defects, with the model finding around 75% of defects.

Bell et al. conducted a case study by using the Ostrand et al. model on a different software system, an automated voice response system (Bell, Ostrand, and Weyuker 2006). In this study, LOC model was not effective as it had been for the other system: 55% versus 75%. Gyimothy et al. found that LOC was a very significant indicator of defects by performing regression analysis on open source web and e-mail suite called Mozilla (Gyimothy, Ferenc, and Siket 2005). Subramanyan and Krishnan also found that LOC was significant when they analyzed a commercial Java/C++ system (Subramanyam and Krishnan 2003). Afterwards, LOC was used in most defect prediction papers to build a model (D'Ambros, Lanza, and Robbes 2012; Hata, Mizuno, and Kikuno 2012; Lessmann et al. 2008; Shihab et al. 2011; Wang, Liu, and Tan 2016; Nam et al. 2017).

Hall et al. reviewed 17 studies that used LOC for building defect prediction models (Hall et al. 2012). While LOC is very easy to collect, a disadvantage is that it measures only one dimension of the code and may only show limited insight into potential sources of defects. Measuring just the size is a coarse-grained feature and does not take into account

the finer detail involved, for example how complex the code is or how the code interacts with the system.

2.4.1.2 Complexity Metrics

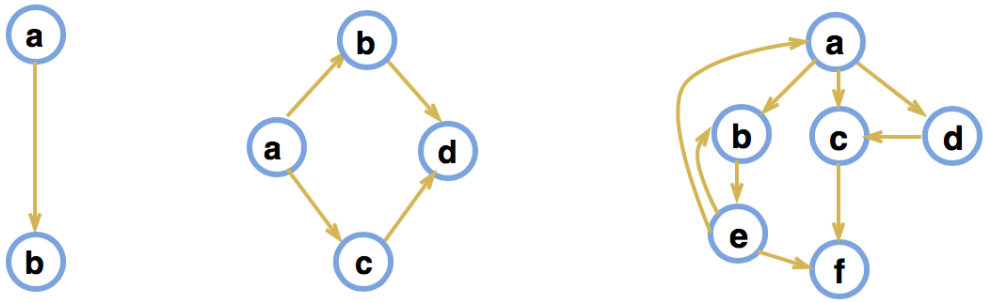
Complexity metrics were introduced to provide a measure of how difficult software code may be to understand and maintain. Halstead metrics and McCabe's complexity measure (Halstead 1977; McCabe 1976) were two of the first complexity measures introduced. McCabe's cyclomatic complexity (CC) is based on the number of decisions in a program (McCabe 1976). CC measures the human comprehension of the code by identifying the number of distinct logical paths through a given unit. As the number of paths increase, it increases the difficulty of testing a module - more test cases are required to cover the various conditional logic paths through the system. High cyclomatic complexity tends to indicate high code complexity and therefore high probability of defects being present.

It is calculated by developing a control flow graph of a particular module. A control flow graph is a representation of all linearly independent paths that could be traversed through a program during its execution (Allen 1970). The nodes are the collection of instructions and edges indicate the direction of flow that which set of instructions is to be executed next. Figure 2.3 shows some examples of control flow graphs in which the nodes represent basic blocks and the edges represent control flow paths. The number of nodes in a program (N), the number of edges (E), and the number of exit node, i.e., the number of disconnected parts of the flow graph (P), are the main components used to calculate the

cyclomatic complexity. Equation 2.1 shows the cyclomatic complexity (V(G)) of any control flow graph (G).

$$V(G) = E - N + 2P \qquad (2.1)$$

Ohlsson and Alberg adopted McCabe's cyclomatic metric to predict defect-prone modules in a telecommunications system and wanted to predict which modules could be faulty before coding had already begun (Ohlsson and Alberg 1996). Ohlsson and Alberg's results showed that the metrics could predict the most fault prone modules in the design phase. Other defect prediction studies (Menzies, Greenwald, and Frank 2007; Moser, Pedrycz, and Succi 2008; Kim et al. 2011; Nam, Pan, and Kim 2013; Nam et al. 2017) also used McCabe's cyclomatic metric to build a prediction model.



(a) A simple control graph (b) If-then-else control flow graph (c) A more complicated control flow graph (As seen in (McCabe 1976))

V(G) = 1 - 2 + 2 = 1 V(G) = 4 - 4 + 2 = 2 V(G) = 9 - 6 + 2 = 5

Figure 2.3 : Examples of control graphs and their calculated complexity scores.

Halstead created a set of metrics that measures how much "information" is in the source code (Halstead 1977). Theses metrics consider the source code as a collection of tokens, which can be classified as either operators or operands, and look at how many tokens are used and how often they are used. By counting the tokens and classifying, which are operators, and which are operands, the following base measures can be collected:

n1= Number of unique operators

n2= Number of unique operands

N1= Total number of operators

N2= Total number of operands

These four measures form the basis of Halstead metrics shown in Table 2.4.1. Halstead metrics have been used popularly in many studies (Menzies, Greenwald, and Frank 2007; Lessmann et al. 2008; Turhan et al. 2009; Zhang et al. 2016). Turhan et al. used McCabe's and Halstead alongside many other metrics to create a defect prediction approach using cross-company and within-company data (Turhan et al. 2009).

There are lots of debates about the usefulness of code metrics as defect predictors (Shepperd and Ince 1994; Fenton and Ohlsson 2000). Contrary, Menzies et al. confirmed that code metrics are useful to build a defect prediction model (Menzies, Greenwald, and Frank 2007). They also showed that how the attributes are used to build models is more important than which particular attributes are used. However, according to Rahman et al.'s study comparing code and process metrics, code metrics is less useful than process metrics because of stagnation of source code metrics (Rahman and Devanbu 2013).

Metric	Equation	Description
Length	$N = N1 + N2$	The total number of operator occurrences and the total number of operand occurrences. A size metric that is an alternative to LOC.
Vocabulary	$n = n1 + n2$	The total number of unique operator and unique operand occurrences. High values indicate harder to read the code and therefore difficult to maintain.
Volume	$V = N \log_2 n$	A size metric that represents the size in bits.
Difficulty	$D = \frac{n1}{2} \times \frac{N2}{n2}$	Measures how difficult to handle the program, thus how error prone it may be.
Level	$L = \frac{1}{D}$	A low-level score increases the program difficulty.
Effort	$E = D \times V$	Measures the amount of mental activity needed to understand the program. The higher the metric the more difficult the code is to maintain.
Content	$C = L - V$	Language independent complexity metric.
Error Estimate	$B = \frac{V}{300}$	This metric aims to predict the number of validation bugs. 300 is the proportion of defects within the system.
Programming Time	$T = \frac{E}{18}$	The time (in minutes) needed to program a particular module. 18 is a constant that reflects the number of decisions a programmer will have to make per second.

Table 2.4.1: Halstead complexity metrics (Halstead 1977).

2.4.1.3 Object-Oriented (OO) Metrics

OO metrics emerged following the introduction of object-oriented programming languages. One of the most popular and highly cited suites for measuring Object-Oriented (OO) characteristics is Chidamber and Kemerer (CK) metrics suite (Chidamber and Kemerer 1994). The authors developed a suite originally consists of 6 metrics calculated for each class and focused on understanding object-oriented design complexity and how complexity can impact on the development process. For the notion of defects prediction,

the higher the complexity of a certain method and/or its class, the higher the potential for errors presents in the certain method and/or its class. The six OO metrics described are outlined below (Chidamber and Kemerer 1994):

1. Weighted Methods per Class (WMC) - WMC is a weighted sum of all the methods defined in a class. The number of methods and the weight of these methods indicate the amount of maintenance needed for the class. As of the inheritance feature of OO, an increase in the number of methods and their weight leads to increase in the impact on the children. Classes with high number of methods are more likely to be application specific thus limiting their reuse potential.
2. Depth of Inheritance Tree (DIT) - The DIT metric is the maximum length from a given class to the root class in the inheritance hierarchy. High DIT means high number of inherited methods thus increases the design complexity and difficulty to predict behavior.
3. Number of Children (NOC) - NOC is the number of immediate subclasses that have inherited from a given class. The greater the number of children, the greater the likelihood of improper abstraction of the parent class.
4. Coupling Between Object classes (CBO) – CBO is a count of the number of other classes to which a given class is coupled. It denotes the dependency of one class on other classes in the design. A large amount of couples reduces the reusability of a class and complicates modifications and testing.

5. Response for a Class (RFC) - This is the count of methods that can be invoked in response to a message received by an object in a given class. A high RFC increase the testing effort and the overall design complexity of the given class.
6. Lack of Cohesion in Methods (LCOM) - LCOM is a count of the number of method pairs whose similarity is zero minus the count of method pairs whose similarity is not zero within a class. The greater the amount of similar methods, the more cohesive is the class. A lack of cohesion increases design complexity, thereby increase the likelihood of errors.

Besides the CK metrics, other object-oriented metrics based on volume and quantity of source code, have been proposed as well (Abreu and Carapuça 1994). As size metrics, D'Ambros et al. identified number of metrics that simply counts the number of instance variables, methods and then build defect prediction models. Table 2.4.2 shows OO metrics proposed by (D'Ambros, Lanza, and Robbes 2010).

Basili et al. investigated OO metrics on eight information management systems written in C++ to see how effective they were as predictors (Basili, Briand, and Melo 1996). Basili et al. concluded that five out of the six CK metrics were useful predictors during the early phase of development. Similar types of analysis have been performed by (Briand, Daly, and Wust 1999; Chidamber, Darcy, and Kemerer 1998; Li and Henry 1993). Each of the studies was performed on industrial C++ projects, except for Li and Henry's study, which was done in Ada. Studies concluded that at least one or more of the metrics

is good at predicting defects. Emam et al. used CK metrics along with metrics from the (Briand, Daly, and Wust 1999) to investigate different defect prediction models on a commercial Java application (Emam, Melo, and Machado 2001). Emam et al. results showed that their model had high accuracy and that the coupling metrics had the strongest association with fault proneness.

Metric	**Description**
FanIn	Number of other classes that reference the class
FanOut	Number of other classes references by the class
NOA	Number of attributes
NOPA	Number of public attributes
NOPRA	Number of private attributes
NOAI	Number of attributes inherited
NOM	Number of methods
NOPM	Number of public methods
NOPRM	Number of private methods
NOMI	Number of methods inherited

Table 2.4.2: Class level OO metrics described in (D'Ambros, Lanza, and Robbes 2010).

Afterward, many defect prediction studies for object-oriented programs have used the OO metrics to build prediction models (Zimmermann and Nagappan 2008; Kamei et al. 2010; Lee et al. 2011; He et al. 2012; Nam et al. 2017). Hall et al. surveyed 42 studies

that utilized OO metrics in defect prediction (Hall et al. 2012). Compared to LOC, the OO metrics measure some finer grained code features and identify more of those features.

2.4.2 Process Metrics

Process metrics are extracted from software archives such as version control systems and issue tracking systems that manage all development histories. Process metrics quantify many aspects of software development process such as changes of source code, ownership of source code files, developer interactions, etc. Usefulness of process metrics for defect prediction has been proved in many studies (Rahman and Devanbu 2013). Main process metrics include:

2.4.2.1 Change Metrics

Change metrics are to measure the extent of changes in the history recorded in version control systems. For example, we can count the number of revisions/bug-fix changes/refactorings of a file and the number of authors editing a file. Graves et al. investigated a telephone switching system that consisted of over 1.5 million LOC (Graves et al. 2000). The authors proposed seven new measures derived from the revision history - number of past faults, number of deltas (i.e. the amount of previous changes), the average age of the code, the development organization, number of developers, how modules are changed together, and a weighted time damp model. The weighted time damp model computes a module's fault potential by adding contributions from each change. A contribution is the level of fault potential, if the change is recent and large then a large fault potential is computed. Graves et al. concluded that the sum of the contributions (the weighted time damp) was the best predictor of faults in a system while the number of

developers and the extent to which a module is connected with another module have no influence on the defect potential (Graves et al. 2000).

Ostrand et al. created a negative binomial regression model that predicts number of faults for each file of a release (Ostrand, Weyuker, and Bell 2005). The model predictors are based on characteristics such as the file size, whether the file was new to the release, or changed or unchanged from the previous release, the age of the file, the number of faults in the previous release, and the programming language. The model was analyzed on 15 different releases of an industrial system. The evaluation showed the model to be very efficient with the top 20% of the files identified as most defective containing at least 84% of the faults (Ostrand, Weyuker, and Bell 2005). Bell et al. used the model described by Ostrand et al. to investigate an automated voice system and attained similar results to the previous study (Bell, Ostrand, and Weyuker 2006). Later, Weyuker et al. tried to generalize Ostrand et al. model to be able to apply to different software systems without extensive statistical modeling expertise and effort (Weyuker, Ostrand, and Bell 2006).

Hassan and Holt presented a top-ten list approach, which validated heuristics about the defect proneness of the most frequently/recently modified areas and the most frequently/recently fixed areas to show the top 10 subsystems that are susceptible to a fault (Hassan and Holt 2005). They used a cache system to track the location of such areas, thus managers can focus testing resources to the subsystems suggested by the list (Hassan and Holt 2005). Kim et al. followed the work of Hassan and Holt and wanted to show that bugs occur in bursts of related faults (Kim et al. 2007). The authors' analysis includes seven different software systems and used a BugCache to hold the locations of the last known

faults and FixCache to hold the locations of where the bug has been fixed. Kim et al. claim that when a fault is fixed in a location, there is a high chance that a bug will appear there in the future (Kim et al. 2007).

Schröter et al. conducted empirical study on 52 Eclipse plug-ins and used information of past and post-release failures to predict future failures at both file and package level (Schröter, Zimmermann, and Zeller 2006). Schröter et al.'s results showed that usage relationships between components can predict failure-prone components, i.e., information of specific use of packages in one failed file/package could be used to predict future failures in another file/package. However, the results on file level were not as good as on package level (Schröter, Zimmermann, and Zeller 2006).

2.4.2.2 Code Churn

Code churn (i.e. the number of modified lines in a file or module per commit) has been researched extensively by a number of researchers. Nagappan and Ball proposed 8 relative code churn metrics measuring the amount of code changes (Nagappan and Ball 2005). For example, one of the metrics is calculated by churned LOC (the accumulative number of added and deleted lines between a base version and a new version of a source file) divided by total LOC. Other metrics consider various normalized changes such as deleted LOC divided by total LOC and the number of changed files in a component divided by files count and so on. In case study, Nagappan and Ball proved that the relative churn metrics are good predictors to explain the defect density of a binary and bug-proneness (Nagappan and Ball 2005).

Moser et al. extracted 18 churn metrics (Table 2.4.3) from three different releases of Eclipse (2.0, 2.1 and 3.0) to conduct a comparative analysis between code and change metrics (Moser, Pedrycz, and Succi 2008). Moser et al.'s change metrics include added and deleted LOC similar to relative code change churn. However, Moser et al.'s change churn metrics did not consider any relativeness by the total LOC and the files count but consider average and maximum values of change churn metrics. Moser et al. results support their hypothesis that change metrics are better predictors than code metrics to predict the presence/absence of bugs in files.

Their conclusion also showed that files with a high number of revisions and files with a high number of bug fixing activities are the best indicators of potential defects while heavily edited files or files committed in a large CVS transaction are less likely to be faulty (Moser, Pedrycz, and Succi 2008).

Metric	Description
REVISIONS	Number of revisions of a file
REFACTORINGS	Number of times a file has been refactored
BUGFIXES	Number of times a file was involved in bug-fixing
AUTHORS	Number of distinct authors that have committed the file into the repository
LOC_ADDED	Sum over all revisions of the lines of code added to a file
MAX_LOC_ADDED	Maximum number of lines of code added for all revisions
AVG_LOC_ADDED	Average number of lines of code added for all revisions
LOC_DELETED	Sum over all revisions of the lines of code deleted from a file
MAX_LOC_DELETED	Maximum number of lines of code deleted for all revisions
AVG_LOC_DELETED	Average number of lines of code deleted for all revisions
CODECHURN	Sum of (added lines of code - deleted lines of code) over all revisions
MAX_CODECHURN	Maximum CODECHURN for all revisions
AVG_CODECHURN	Average CODECHURN for all revisions
MAX_CHANGESET	Maximum number of files committed together to the repository
AVG_CHANGESET	Average number of files committed together to the repository
AGE	Age of the file in weeks

Table 2.4.3: List of change metrics used in (Moser, Pedrycz, and Succi 2008).

2.4.2.3 Change Burst

Change burst which is a sequence of consecutive changes, have been investigated as predictors of defects by looking at code churn over a set number of days with a specified gap size. Sliwerski et al. showed that the larger the change to a file, the more likely that change is going to need fixing in the future (Śliwerski, Zimmermann, and Zeller 2005). The study findings also revealed that it is three times more likely that a fix in an Eclipse project induces a further fix in the future compared to an enhancement. This finding was also observed by Purushothaman and Perry as they showed that nearly 40% of the changes made to correct code introduced a defect into the software (Purushothaman and Perry 2005). Similarly, the authors found that small changes to code were unlikely to introduce fault in the module as a one-line change has less than 4% probability of causing a fault. Nagappan et al. investigated 3,404 Windows Vista binaries exceeding 50 million LOC. Change bursts were shown to have high predictive power in terms of precision and recall (Nagappan et al. 2010).

2.4.2.4 Change Entropy

This metric was investigated by applying Shannon's entropy to capture how changes are complex (Hassan 2009). Hassan measures the complexity of the change process (HCM) by assessing how much modifications are scattered across space and time. The metrics derived from the location of the changes made; scattered changes could be more complex to manage, and thus more likely to induce defects. To validate the HCM, Hassan built statistical linear regression models based on HCM, the number of previous modifications, and previous faults. The evaluations on six open-source projects showed

that prediction models build using HCM outperform those using the other two change metrics (Hassan 2009).

A recent work that considers to what extent developers apply scattered changes in the system is by Di Nucci et al. (Nucci et al. 2015; 2018). The authors exploited the role of structural and semantic scattering of changes performed by a developer in bug prediction. Their findings demonstrate the superiority of the bug prediction model built using scattering metrics with respect to other baseline models including the change entropy by Hassan (Hassan 2009). Moreover, they show that the proposed metrics are orthogonal with respect to other predictors.

2.4.2.5 Code Metrics Churn (CHU) and Code Entropy (HH)

These two metrics are proposed by D'Ambros et al. (D'Ambros, Lanza, and Robbes 2010). The authors conducted an extensive comparisons study of the newly proposed metrics and number of existing bug prediction approaches using source code metrics, change history metrics, past defects and entropy of change metrics. These approaches have been studied and introduced in previous subsections (Nagappan and Ball 2005; Hassan 2009). In contrast to code churn metrics based on the amount of lines (Nagappan and Ball 2005), CHU measures the change in biweekly basis of code metrics such as CK metrics and OO metrics. Thus, CHU captures the extent of changes more precisely than code change churn that computes the amount of changes between a base revision and a new revision.

While change Entropy is computed based on the count of file changes (Hassan 2009), code entropy (HH) is computed based on the count of involved files when a certain

code metric is changed. In the comparison evaluation, D'Ambros et al. concluded that CHU and HH metrics led to good prediction results on all subjects used in their experiments. However, these novel metrics are limited since they require heavy computation resources and data because they track biweekly changes from version control systems (D'Ambros, Lanza, and Robbes 2010).

Metric	**Description**
POP-NOM	The number of e-mails discussing a class.
POP-NOCM	The number of characters in all e-mail discussing a class.
POP-NOT	The number of e-mail threads discussing different topics for a class.
POP-NOMT	The number of e-mails in a thread discussing a class in at least one of e-mails in a thread.
POP-NOA	The number of authors motioning about the same class.

Table 2.4.4: Popularity metrics (Bacchelli, D'Ambros, and Lanza 2010).

2.4.2.6 Popularity Metrics

This group of metrics were proposed by Bacchelli et al. by analyzing e-mail archives by developers in a group mailing list (Bacchelli, D'Ambros, and Lanza 2010). The intuition is that problematic classes are more often discussed in email conversations than classes that have fewer problems. Table 2.4.4lists the popularity metrics. The extracting metrics from e-mail archives is novel but their evaluation of the metrics shows

that popularity metrics themselves did not outperform other code and process metrics (Bacchelli, D'Ambros, and Lanza 2010).

2.4.2.7 Ownership and Authorship Metrics

Ownership and authorship were discussed by Bird et al. who examined the relationship between ownership and quality. They examined the effects of ownership on Windows Vista and Windows 7 (Bird et al. 2011). They proposed four ownership metrics: the number of minor contributors, the number of major contributors, the total number of contributors, and the proportion of ownership for the contributor with the highest proportion of ownership. They concluded that a high ratio of ownership leads to less defects. A similar study by Rahman and Devanbu examined the effects of ownership and experience on quality in several open-source projects, using a fine-grained level of analysis based on fix-inducing code-fragments (Rahman and Devanbu 2011). The interesting finding is that QA should be focused on code files touched by less experienced developers and that a developer's specialized experience in a target file is more important than general experience.

2.4.3 Finer Grained Techniques

2.4.3.1 Text Analysis

Apart from code and process metrics, some defect prediction studies have used less traditional independent variables that focused on finer grained details of the source code. Some of these independent variables have been based on analyzing the text of the code. Mizuno et al. used spam-filtering techniques to create a fault detection technique (Mizuno et al. 2007). Mizuno et al.'s approach considered source code files as text files and used

text-mining techniques used in spam filtering to identify problematic patterns in the text files. This framework is based on the fact that spam e-mails usually include particular patterns of words or sentences. From a viewpoint of source code, similar situation usually occurs in faulty software modules. That is, similar faults may occur in similar contexts. they guessed that faulty software modules have similar pattern of words or sentences like spam e-mail messages. Their result showed that the technique was able to classify more than 75% of modules correctly (Mizuno et al. 2007).

Binkley et al. used an information retrieval based defect prediction technique, known as the QALP score to help identify potential defects (Binkley et al. 2007). The QALP score measures the similarity between a module's comments and its source code using a cosine similarity. The results showed that, when used alongside LOC, the QALP score improves fault prediction (Binkley et al. 2007). Later, Binkley et al. combined the QALP score with two other metrics based on natural language processing of program's identifiers and found them helpful in predicting defects in files (Binkley et al. 2009).

Marcus et al. also proposed an information retrieval technique - latent semantic indexing (LSI) (Marcus, Poshyvanyk, and Ferenc 2008). They used LSI to analyze the text of source code to develop a class cohesion measure called C3. C3 measures how strongly the methods of a class related to each other based on the analysis of the unstructured information embedded in the source code, such as comments and identifiers. The measure is inspired by the mechanisms used to measure textual coherence in cognitive psychology and linguistics. The case study shows that the novel measure captures different aspects of class cohesion compared to any of the existing cohesion measures. Marcus et al. were able

to combine C3 alongside existing structural cohesion metrics to attain better defect prediction than with structural cohesion metrics alone (Marcus, Poshyvanyk, and Ferenc 2008).

2.4.3.2 Code Smells

Abebe et al. used lexicon bad smells (LBS) in conjunction with software structure metrics to improve defects detection (Abebe et al. 2012). Examples of poor-quality lexicon are short terms identifiers (e.g., abbreviation or acronym) and meaningless terms (e.g., foo and bar). Lexicon bad smells for identifiers has been shown to be associated with the introduction of errors (Butler et al. 2009). Taba et al. proposed four anti-pattern metrics (Taba et al. 2013). Antipatterns are specific design and implementation styles that can identify poor system. They are usually introduced in software systems due to the lack of knowledge or experience of developers when solving a particular problem. In their evaluation with two open source projects, they found that design smells can be used to predict faults as files that have design smells tend to have a higher density of faults than other files and anti-pattern metrics could improve prediction performance in terms of f-measure (Taba et al. 2013). A study by Palomba et al. proposed a smell-aware bug prediction model. Their results indicated that the accuracy of a bug prediction model increases by adding the code smell intensity as predictor (Palomba et al. 2016; 2017). Padua and Shang investigated the exception handling anti-patterns and found them to have significant relationship with defects (Pádua and Shang 2018).

2.4.3.3 Network Analysis

Zimmermann and Nagappan presented a study with the Microsoft Windows 2003 server project (Zimmermann and Nagappan 2008). They applied network centrality measures such as centralness, closeness, and betweenness to the static dependency graph of Windows Server 2003 binaries to predict the probability and number of post-release failures. The authors compared their model to models constructed by code and process metrics. In their evaluation, network measure could predict more bug-prone binaries than code and process metrics (Zimmermann and Nagappan 2008).

Petric and Grbac investigated finer grained software code structure that represented with help of graph representations (Figure 2.4) , and subgraph frequencies (Petrić and Grbac 2014). Through an empirical study of more than 30 releases of three open source software systems, Petric and Grbac identified that the same set of sub-graphs of software system is present across the system version, but different sets are present in different software systems. Petric and Grbac were able to find some evidence between certain subgraphs and defects (Petrić and Grbac 2014).

Various researches proposed metrics quantifying other aspects of software engineering in order to model software quality. For example, Shihab et al. (Shihab, Bird, and Zimmermann 2012) consider branching activities; Shang et al. (Shang, Nagappan, and Hassan 2015) investigate logging characteristics, Zhang et al. (Zhang et al. 2012) examine editing patterns, and McIntosh et al. (McIntosh et al. 2016) study code reviews.

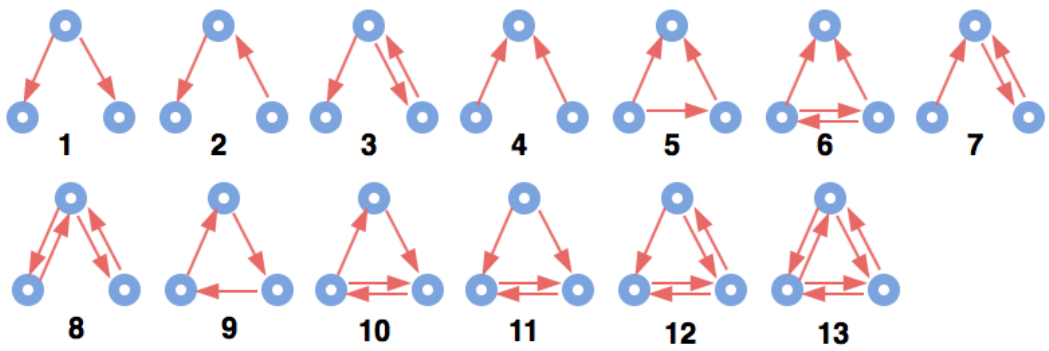


Figure 2.4: The three node subgraphs examined by (Petrić and Grbac 2014).

2.4.4 The Problem with Traditional Independent Variables

Most of the traditional independent variables (as described above) have been extensively used in defect prediction techniques and showed metrics to correlate with the number of defects. Despite all this research, Menzies et al. reported that existing defect prediction models are thought to have reached a predictive performance ceiling and new approaches are needed (Menzies et al. 2010).

One possible reason for the limit power could be that most of the metrics introduced suffer from being very coarse grained with low capability that measure only a small sub-set of code features. Gray et al. advocated that the coarse grained nature of such metrics prevents machine learning algorithms from effectively differentiate between defective and non-defective modules: if two modules have the same metric values, e.g., LOC, but they have not been labeled the same in terms of their defectiveness, this will obstruct the learning algorithm's ability to learn (Gray et al. 2011). Gray et al. identified many modules in the NASA datasets that have identical values across number of metrics but different defectiveness labels. Kim et al. showed that the amount of noise in the data set affects the

predictive power of a technique. Kim et al. highlighted that the current commonly used metrics are not sufficient enough to differentiate modules for defect prediction (Kim et al. 2011).

Another problem with most of existing metrics is that metrics focus on syntactic aspects of software modules, such as LOC, number of declarations, number of functions, etc. It is known that software systems have well-defined syntax, which can be represented by Abstract Syntax Trees (ASTs) (Hindle et al. 2012) and at the same time have semantics, which is hidden deeply in source code (White et al. 2015). It has also been shown that such semantic information is useful for tasks such as code completion, effort estimation and bug detection (Li and Zhou 2005; Nguyen and Nguyen 2015; Alomari, Collard, and Maletic 2014; Tu, Su, and Devanbu 2014; Hindle et al. 2012). This semantic information should also be useful for characterizing defects for improving defect prediction. Specifically, in order to make accurate predictions, the metrics need to be discriminative: capable of distinguishing one instance of code region from another. Additionally, most of these metrics only focus on single elements, but rarely take the interactions between elements into account (Zimmermann and Nagappan 2008). However, with the emergence of static and dynamic bug localization techniques, the nature of defects has changed and today most defects in bug databases are of semantic nature (Li et al. 2006).

CHAPTER 3

PROGRAM SLICING

One of the goals for this research is to investigate the use of slice-based metrics for defect prediction as a mean of semantic view of the source code. Thus, this chapter outlines how program slicing works, how it motivates the work in this dissertation and how it is in favor for defect prediction. In general, slicing techniques are associated with different areas of software engineering including system specifications (Wu and Yi 2004), software architectures (Zhao 1998), and UML and state-based models (Bae and Chae 2008; Korel et al. 2003). The key aspect of slicing in all these areas is when given a particular criterion, all other elements of the domain, whether it is extraneous source code, or architecture specifications or UML models, are eliminated, leaving just that portion that is relevant to some specific element of that domain under study.

Program slicing in particular is the computation of the set of programs statements, the program slice, which may affect or affected by the values computed at some point of interest, referred to as a slicing criterion (Weiser 1984; Horwitz, Reps, and Binkley 1988). Program slicing is a reduction technique that can reduce the total amount of source code to be analyzed to a more manageable level without eliminating relevant pieces. The reason for introducing program slicing by Weiser was to model the behavior that programmers exercised during debugging task (Weiser 1981, 1982; Weiser 1984). It was observed that many expert programmers when debugging start at the location where the fault is identified

and then work backwards to consider what earlier statements might have led to this faulty state. While working backwards, the programmer focuses attention only on statements that could impact the errant line creating a "slice" of the program for analysis. Program slicing supports this technique by eliminating any code statement not affecting the values computed at a specified point in the program.

Weiser defined the slice as an executable program that preserved the behavior of the original program. The algorithm traces the data and control dependencies by solving data-flow equations for determining the direct and indirect relevant variables and statements (Weiser 1981, 1982; Weiser 1984). Based on Weiser algorithm, a static program slice S consists of all statements in program P that could influence the value of variable v at point of interest p. The slice is defined for a slicing criterion – a pair <i, v>, where “i” is a point of interest for slicing typically specified by a location in the program, and “v” is a subset of program’s variables to be observed at statement “i”. The starting point is a specific statement in the program and a variable state at that point in the program. Based on data flow analysis, relevant statements are recursively processed working backwards in the source code to extract the slice.

3.1 Slicing Example

Figure 3.1 shows an example of program (a) and a valid slice (b) of the program with respect to criterion <11, product>. The statements in the backward slice of the output statement write (product) in line 11, are shown in (b). The value of variable product impacted by lines 1, 2, 4, 5, 7, 8, and 9 in the example code.

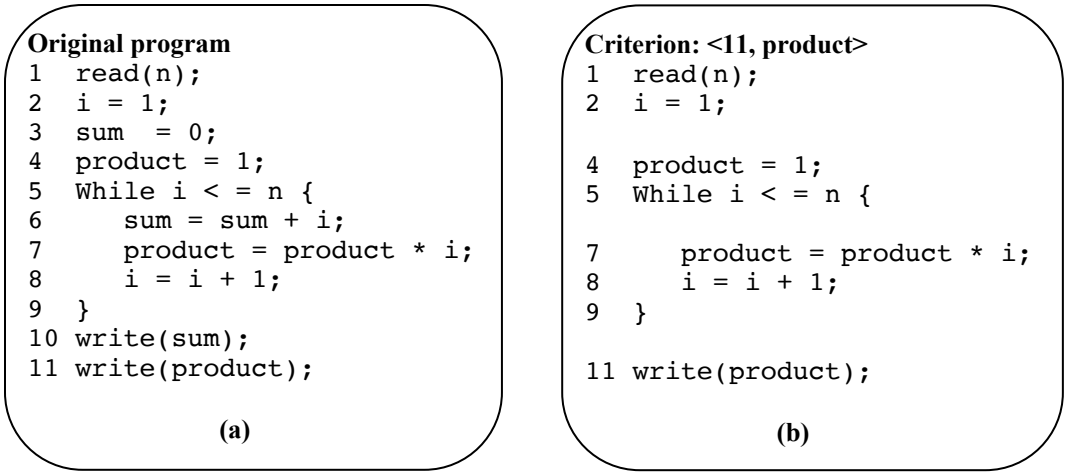


Figure 3.1: (a) An example program (b) A slice of the program w.r.t. criterion<11, product>

3.2 Program Slicing Techniques

Program slicing has motivated a large body of research for different applications in software engineering, and has been proposed to guide programmers during many aspects of the software development life cycle, including software maintenance (Gallagher and Lyle 1991; Feng and Maletic 2006), debugging (Agrawal, Demillo, and Spafford 1993; Weiser and Lyle 1986), program comprehension (Korel and Rilling 1997; 1998; Lucia, Fasolino, and Munro 1999), testing (Binkley 1998; Korel and Rilling 1998; Binkley 1998; Harman and Danicic 1995; Gupta, Harrold, and Soffa 1992), and bug classification (Pan, Kim, and Jr 2006).

These applications require different properties of slices; thus, a number of different slicing definitions have been proposed after Weiser's. Various surveys of the slicing literature (Tip 1994; Lucia 2001; Xu et al. 2005; Silva 2012; Androutsopoulos et al. 2013) covered these definitions in detail. Interestingly, each survey presents the definitions from

a slightly different perspective. These techniques can be broadly distinguished according to the type of slices such as the following:

3.2.1 Static and Dynamic Slicing

Static slice is computed without making assumptions regarding a program's inputs. It includes all statements that potentially affect/affected by the value of a variable at a particular point of interest in the program (Tip 1994; Xu et al. 2005). This captures all possible executions of the value of a variable. Contrary, dynamic slice is a set of statements that affect the value of a variable for one specific input. Dynamic data dependence information is traversed to compute the slices. This information is constructed using an execution trace of the program, thus only the dependencies that occur in a specific execution of the program are taken into account (Tip 1994; Xu et al. 2005). Dynamic slice gives a better understanding of programs and their executions for a particular input that is useful for applications such as debugging and testing (Feng and Maletic 2006; X. Zhang, Gupta, and Gupta 2007).

3.2.2 Direction of Program Traversal

Program slicing can be either backward or forward (Xu et al. 2005). A forward slice contains all the statements and control predicates dependent on the slicing criterion, a statement being "dependent" on the slicing criterion if the values computed at the statement depend on the values computed at the slicing criterion, or if the values computed at the slicing criterion determine the fact if the statement under consideration is executed or not. Thus, a forward slice includes all statements affected by changing the value of the slicing variable. In contrast, a backward slice is computed by gathering statements and control

predicates by way of a backward traversal of the program's control flow graph (CFG) or program dependence graph (PDG), starting at the slicing criterion. Technically, these slices are called backward static slices and contains all the statements in the program that may affect the value of variable (Xu et al. 2005).

3.2.3 Inter-Procedural Versus Intra-Procedural

The slice can be characterized in how it handles slicing across procedure boundaries called inter-procedural slicing, or locally, called intra-procedural slicing slicing (Horwitz, Reps, and Binkley 1988). Weiser (Weiser 1984) introduced inter-procedural program slicing, and extended his previous intra-procedural work proposed in (Weiser 1981).

3.2.4 Executable Slice

A slice is executable if the statements in the slice form a syntactically correct program that can be executed (Xu et al. 2005). Based on slice definition stated earlier, if the slice is computed correctly (safely), the result of running the executable slice produces the same result for variables in V at p for all inputs.

3.3 Program Dependence Analysis

All slicing approaches share a common factor that is they are based on the notion of a Program Dependence Graph (PDG), or one of its variants, e.g., Control-Flow Graph (CFG) and Def/Use Graph, to compute the slice. Statement s2 statically control-depends on s1 if s1 is a conditional statement and can influence whether s2 is executed (Podgurski and Clarke 1990). Statement s2 statically data-depends on s1 if there is a sequence of variable assignments that potentially propagate data from s1 to s2 (Podgurski and Clarke 1990). The Control-Flow Graph (CFG) (Allen 1970) models the static control-flow

between the statements in the program. Statements are represented as nodes. Arcs pointing away from a node represent possible transfers of control to subsequent nodes. A program's entry and exit points are represented by initial and final vertices. So, a program can potentially be executed along paths leading from an initial to a final vertex. The Def/Use Graph extends the CFG and labels every node n by the variables defined and used in n.

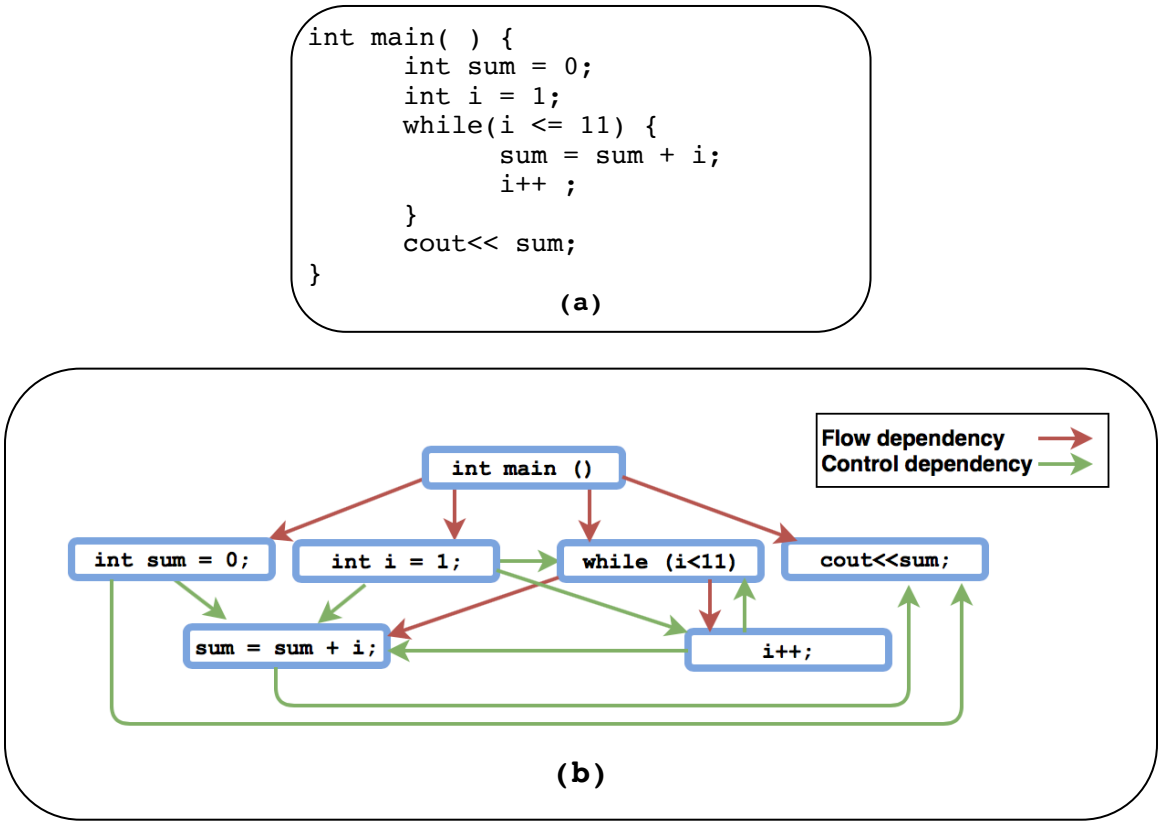


Figure 3.2: (a) An example program. (b) Program Dependency Graph.

In Program Dependence Graph (PDG) (Ottenstein and Ottenstein 1984), every statement s2 is a node that has an outgoing arc to another statement s1 if s2 directly (not transitively) data- or control-depends on s1. A statement s2 syntactically depends on s1 if

in PDG s1 is reachable from s2. An example of a program and its program dependence graph is shown in Figure 3.2. The graph is directed as represented by the arrows pointing from one node to the next. It does not distinguish data- or control-dependence.

3.4 Slice-Based Metrics

The origin of slice-based metrics can be traced back to Weiser, who used backward slicing to describe the concepts of coverage, overlap, and tightness (Weiser 1984). For a given module, Weiser first sliced on every variable where it occurred in the module. Then, Weiser computed Coverage as the ratio of average slice size to program size, Overlap as the average ratio of non-unique to unique statements in each slice, and Tightness as the percentage of statements common in all slices.

Ott and Thuss improved the behavior of slice-based metrics through the use of metric slices on output variables (Ott and Thuss 1993). A metric slice takes into account both the uses and used by data relationships. More specifically, a metric slice with respect to variable v is the union of the backward slice and the forward slice. Ott and Thuss introduced two new metrics for program slicing, supplementing the existing slicing metric introduced by Weiser: MinCoverage and MaxCoverage. MinCoverage and MaxCoverage are respectively the ratio of the size of the smallest slice and the ratio of the size of the largest slice to the module size. Consequently, the slice-based metrics suite proposed by Ott and Thuss consists of five metrics: Coverage, Overlap, Tightness, MinCoverage, and MaxCoverage. Note that these metrics are computed at the statement level, i.e. statements are the basic unit of slicing metrics. Table 3.4.1 summarizes the descriptions of the slice-based metrics introduced by Weiser and ott and thuss (Weiser 1984; Ott and Thuss 1993).

Metric	Description
Coverage	The extent to which the slices cover the module
MaxCoverage	The extent to which the largest slice covers the module
MinCoverage	The extent to which the smallest slice covers the module
Overlap	The extent to which slices are interdependent
Tightness	The extent to which all the slices in the module belong together

Table 3.4.1: Slice-based metrics by (Weiser 1984; Ott and Thuss 1993).

Later, Ott and Bieman used program slicing in the context of tokens rather than statements, in which the number of tokens that are shared by multiple slices are used to represent cohesion (Bieman and Ott 1994). They called such slices data slices. More specifically, a data slice for a variable v is the sequence of all data tokens in the statements that comprise the metric slice of v. Consequently, this leads to five slice-based data-token-level metrics.

Although Weiser introduced program slicing as a comprehension method used by programmers while debugging (Weiser 1981), many slice-based metrics have been developed to quantify the degree of cohesion in a module. Such metrics calculated in which the numbers of statements that are shared by multiple slices represent cohesion (Meyers and Binkley 2007; Black et al. 2006; Bieman and Ott 1994; Meyers and Binkley 2004; Counsell, Hall, and Bowes 2010; Counsell et al. 2010).

3.5 The Use of Slice-Based Metrics for Code Quality

Although slice-based metrics have been proposed for many years, to date little work has been performed to empirically relate them to code quality and program propensity for faults. Meyers and Binkley undertook a large-scale empirical study of five slice-based metrics and analyzed the relations between these metrics and code size metrics (Meyers and Binkley 2004; 2007). They found that slice-based metrics provided a unique view of a program. The research also showed that the same set of metrics could be used to identify degraded modules and guide software reconstruction. However, a major difference between their study and our study is that they did not relate slice-based metrics to defect-proneness nor to cognitive complexity.

Black et al. empirically investigated the ability of two slice-based metrics, Tightness and Overlap, to distinguish between faulty and not-faulty functions (Black et al. 2009). In their study, they combined the nineteen versions of a small program called Barcode to obtain a single data set. Black et al. (Black et al. 2006) had planned to test the hypotheses relating three slice-based metrics (Tightness, Overlap, and Coverage) and defect-proneness. However, they failed to do this due to lack of data. Compared with their work, we perform an in-depth and comprehensive empirical study on the relationships between new and different set of slice-based metrics and defect-proneness. Work by Pan and Kim used C language slicing metrics to compare the classification of defects with code metrics for C++ (Pan, Kim, and Jr 2006). The calculation of their metrics is based on the notion of a Program Dependence Graph (PDG) (Ottenstein and Ottenstein 1984) such as edge count and vertices count (Liang and Harrold 1998).

Yang et al. (Yang et al. 2015) studied the usefulness of Weiser (Weiser 1981) and Ott and Thuss (Ott and Thuss 1993) cohesion metrics in effort aware defect prediction. The metrics leverage program slices with respect to the output variables of a module to quantify the strength of functional relatedness of the elements within the module.

Yet we still know very little about software semantic and their cognitive complexity. Very few studies have investigated the concept of program slicing from an evolutionary viewpoint. To the best of our knowledge, this is the first work that applies program slicing to measure characteristics of cognitive complexity and investigate their relationship to defect propensity from an evolutionary viewpoint. In this dissertation, we use a novel set of slice-based metrics and form a relationship between these slicing metrics and defect propensity through the evolution of the system. In particular, we use forward slicing technique to calculate slice-based metrics at the statement level. The reason for choosing the statement is that previous studies suggested that software metrics at a finer granularity would accordingly have a higher discriminative power and hence may be more useful for fault-proneness prediction.

CHAPTER 4

SLICE-BASED COGNITIVE COMPLEXITY METRICS

For years researchers have devoted their efforts trying to understand how programmers comprehend code and several cognitive model have been proposed (Storey, Wong, and Müller 2000; Storey 2005). Much of the research explains how programmers comprehend complex code using a bottom-up approach (Storey, Wong, and Müller 2000; Storey 2005). The programmer analyzes the source code statement by statement and gradually develops control-flow and data-flow abstractions through the process of chunking (Pennington 1987). Program chunks are grouped together to form larger chunks, until the entire program is understood. In this way a hierarchical semantic representation of the program is built from the bottom-up. Thus, assessing the cognitive complexity of program semantic chunks can be a criterion for characterizing defects for defect prediction. Specifically, in order to make accurate predictions, the metrics need to be discriminative: capable of distinguishing one instance of code region from another of different cognitive complexity.

A study by Siegmund et al. looked at the process of bottom-up program comprehension with (fMRI), a technique used by to understand brain regions activated by cognitive tasks, and found a network of brain areas activated that are related to natural-language comprehension, problem solving, and working memory (Siegmund et al. 2017).

Klemola argues that measuring complexity should reflect attributes of human comprehension since complexity is relative to human cognitive characteristics. They focus on aspects of cognition which involves both short-term and long-term memory. Overloading over a short period of time affects short-term memory (STM) while long term memory (LTM) is affected by the frequency of exposure to a concept over time (Klemola 2000).

Researchers theorize that all information processed for comprehension must at some time occupy short-term memory (STM). For the purposes of natural-language comprehension, the capacity of STM has been measured at 4 concepts (Cowan 2001). This suggests that any code segment that is using more than 4 concepts to make a point unfamiliar to the reader might not be immediately understood.

In coping with these demands and limitations, the programmer must have mental capacity for dealing with large workloads for short periods of time and cognitive mechanisms for locating the code relevant to a particular feature. Program slicing was introduced by Weiser (Weiser 1984) after noticing programmers try to identify program bugs by using slices of the program composed of statements, which affect the computation of interest (Weiser 1982). Thus, slicing process removes from consideration parts of the program that are determined to have no effect upon the semantics of interest in a similar way as it would be perceived by developer during the process of comprehension (Weiser 1982).

A slice is a cognitive chunk of the program that preserves control flow and data flow dependences relevant to a specific point of interest. It is possible to determine the

parts with different behaviors by comparing the slices of two artifacts. With slice granularity, a hierarchical internal semantic representation of the whole program can be measured in addition to a detailed analysis of the comprehension effort required to retrace and inspect particular function. In the following we define four categories of slice-based cognitive complexity measures.

4.1 Definitions of Slice-Based Cognitive Complexity Metrics

4.1.1 sliceCount

The count of slices focuses on the overall cognitive complexity of source code parts. Program segment that has a high number of slices will have high number of features leading to a higher concentration of identifiers, method invocations and relevant control and data dependencies. When there are many possible paths to be taken within a module the time spent tracing references increases and at the same time the use of identifiers must be carefully observed and retained in human memory to arrive at a correct understanding. When time is limited, a program segment with a high value of slice count can be difficult to interpret. *sliceCount* is defined as *number of slices within a module* and more formally as follow:

$$sliceCount(x) = K, \qquad (4.1)$$

where k is the number of slices in x.

4.1.2 sliceSize

This measure provides an indicator of the cognitive effort required to comprehend a particular slice. As stated earlier, program slice consists of all the statements that may influence the values of a variable at a program point (Weiser 1984). It includes program artifacts that are data and control dependent to the function or variable of interest. A study by Alomari et al. shows that the growth of the slice size over time in Linux kernel is related to the maintenance activity being made (Alomari, Collard, and Maletic 2014). An increase in the slice size requires increase in the cognitive effort in analyzing the code related to the slice. Failing in uncovering the causal interactions between components force programmer to make unverified assumptions and eventually introducing defects (Chen et al. 2018; Klemola 2000).

The granularity of the slice for the computation of this metric is relevant. A single slice with highly dense dependencies may be buried in a large block of simple code resulting in a low value for the large block. The single dense slice will be more difficult to correctly interpret than the overall measure would suggest. Consequently, the best use of the metric is to locate system artifacts with high concentrations of identifiers, and dependencies to inspect or refactor. *sliceSize* is the *mean count of statements per slice within a module*. *sliceSize* for a module *x* can be defined as:

$$sliceSize(x) = \sum_{i=1}^{k} S_i/k, \qquad (4.2)$$

where *S* is the number of statements in slice *i*, and *k* is the number of slices in module *x*.

4.1.3 sliceIdentifier

Identifiers play a crucial role in program comprehension, since developers express domain knowledge through the names that they assign to the code entities at different levels (i.e., packages, classes, methods, variables) (Arnaoudova et al. 2010; Enslen et al. 2009; Abebe et al. 2012). Thus, source code lexicon impacts the psychological complexity of a program (Scalabrino et al. 2016; Sharif and Maletic 2010). For the purpose of cognitive complexity metrics, a high identifier density may overload STM and lead to error. The risk of comprehension error has been observed to rise with the increase of general identifier density metric in program code (Klemola 2000; Buse and Weimer 2010) and the increase of concept density in text as well (Kintsch 2005). However, the problem with calculating a general identifier density metric is that it only represents a general view on the system under investigation (Rilling and Klemola 2003). A small block with highly dense identifier may be buried in a large block of simple code resulting in a low value for the large block. Therefore, it is essential to refine identifier density metric to reflect a more realistic assessment based on the development task on hand. Program slicing allows for such refinement, by focusing the metric only on these parts that are relevant with respect to a particular feature or variable. *sliceIdentifier* can be defined as the *mean distinct occurrences of programmer defined labels within a slice in a module*. For a module *x* it can be formally defined as:

$$sliceIdentifier(x) = \sum_{i=1}^{k} SI_i/k, \qquad (4.3)$$

where *SI* is the number of identifiers in slice *i*, and *k* is the number of slices in module *x*.

4.1.4 sliceSpatial

This measures account for the difficulty of reading the source code of a program for understanding, in terms of the lexical distance (measured in lines of code) that the maintainer is required to traverse to follow control and/or data dependencies as they build a mental model (Gold, Mohan, and Layzell 2005; Chhabra and Gupta 2009). This type of complexity was based on the spatial distance between the definition and direct use of various program elements. However, understanding of the use of a program element also requires knowledge of control and data flow in which the program element has been used (Gold, Mohan, and Layzell 2005). More details about the elements are understood through its use in a particular sequence and the use of other artifacts that influence the behavior of the element of interest. Without program slicing, it would be impossible to find all relevant uses that might affect or affected by the element value. The greater the distance in lines of code, the more is the cognitive effort required to understand the purpose and data flow of that slice. If a program element is defined and then used after, (e.g., 500) lines of source code, the element details would be overwritten in the working memory by more recently defined/used elements. Thus, we define *sliceDistance* as the *spatial distance in LOC between the definition and the last use of the slice divided by the module size.*

$$SliceDistance\ (i)\ = Sm_i - Sn_i/q, \qquad (4.4)$$

where *Sm* is the line number of the first statement in slice *i*, *Sn* is the line number of the last statement in slice *i*, and *q* is the module size in LOC. Accordingly, for module *x, sliceSpatial* measured as the *mean of the individual slice distance in x.*

$$sliceSpatial\ (x) = \sum_{i=1}^{k} \frac{sliceDistance(i)}{k}, \qquad (4.5)$$

where *sliceDistance* is the scatter measure of slice *i* measured as in equation (3.4), and *k* is the number of slices in module *x*.

4.2 Extracting Slice-Based Cognitive Complexity Metrics

We use the *srcSlice* tool (Alomari et al. 2014; Newman et al. 2016) to compute the slicing metrics. In the following subsections, we provide an overview description of the theory and implementation of *srcSlice* tool and the computation of proposed slice-based cognitive complexity metrics.

4.2.1 The srcSlice Tool

The *srcSlice* tool (Newman et al. 2016) is a fast and scalable, slicing approach. It has practical means to estimate the source code semantic for very large systems within practical time frames which makes it suitable for this work. Program slicing is typically based on the notion of a Program Dependence Graph (PDG) (Ottenstein and Ottenstein 1984) or one of its variants. Unfortunately, building the PDG is quite costly in terms of computational time and space. As such, slicing approaches generally do not scale well and while there are some (costly) workarounds, generating slices for a very large system can

often take days of computing time. *srcSlice* addresses this limitation by eliminating the time and effort needed to build the entire PDG. As a result, *srcSlice* is very fast and scalable on large systems. It clocks in at about 274K identifiers per minute on Linux Kernel (Newman et al. 2016). The approach was first introduced in (Alomari, Collard, and Maletic 2014), and then evaluated to a total of 18 open source systems through a comparison study to the *CodeSurfer* tool from GrammaTech (Alomari et al. 2014).

The *srcSlice* tool implements a forward, static slicing technique. Forward static program slicing refers to the computation of program points that are affected by other program points (Horwitz, Reps, and Binkley 1988). The forward slice from program point *p* includes all the program points in the forward control flow affected by the computation at *p*. *srcSlice* uses the initial variable declaration as the starting point. It combines a text-based approach with a lightweight static analysis infrastructure that only computes dependence information as needed (aka on-the fly) while computing the slice for each variable in the program. Specifically, *srcSlice* computes a forward, static, non-executable, inter- procedural program slice for each variable in a system.

The tool is enabled by the *srcML* (Collard, Decker, and Maletic 2011; Collard, Maletic, and Robinson 2010) infrastructure (see srcML.org). Source code is first converted to *srcML* and then a stream-oriented approach to compute the slice is performed. *srcML* (SouRce-Code Markup Language) augments source code with abstract syntactic information from the AST to add explicit structure to program source code. This syntactic information is used to identify program dependencies as needed when computing the slice. *srcML* format has been previously used for different maintenance tasks, lightweight fact

extraction (Collard, Decker, and Maletic 2011), pattern matching of complex code (Dragan, Collard, and Maletic 2006), and software artifacts summarization (Abid et al. 2015).

The *srcML* format is supported with a toolkit, including src2srcml and srcml2src, which supports conversion between source code (in multiple languages such as C, C++, and Java) and the format. Then, a system dictionary instead of PDG/SDG represents the program slice is generated by the *srcSlice*. Given a system (in the *srcML* format), *srcSlice* gathers data about every file, function, and variable throughout the system, storing it all in a three-tier dictionary.

4.2.2 Running Example

An example of a slice computed by *srcSlice* on a small program is given in Figure 4.1. The first portion of the figure (a) presents a small program constructed to show how *srcSlice* computes the profile. The second part of the figure (b) is the slice profile for the program in (a). The example is taken from (Newman et al. 2016).

```
1  int fun(int z){
2    z++;
3    return z;
4  }
5  void foo(int &x, int *y){
6    fun(x);
7    y++;
8  }
9  int main(){
10   int abc = 0;
11   int i = 1;
12   while (I <= 10) {
13     foo(abc, &i);
14   }
15   std::cout<<"i:"<<i<<"abc:"<<abc<<std::endl;
16   std::cout<<fun(i);
17   abc = abc + i;
18 }
```

(a)

srcslicetest.cpp,main,i,def{11},use{1,2,5,7,12,13,15,16,17},dvars{abc},pointers{},cfuncs{fun{1},foo{2}}

srcslicetest.cpp,main,abc,def{10,17},use{1,2,5,6,13,15},dvars{},pointers{},cfuncs{fun{1},foo{1}}

srcslicetest.cpp,fun,z,def{1},use{2},dvars{},pointers{},cfuncs{}

srcslicetest.cpp,foo,y,def{5},use{7},dvars{},pointers{i},cfuncs{}

srcslicetest.cpp,foo,x,def{5},use{1,2,6},dvars{},pointers{abc},cfuncs{fun{1}}

(b)

Figure 4.1: (a) Sample source code, (b) System dictionary with all slice profiles for the source code in (a).

4.2.3 Slice Profile

Each entry of the system dictionary is a single slice profile for an identifier, which contains all data gathered about that identifier during the slicing process. The following is a list of that information:

- *File, function, and variable name*– the file/function the variable is in and its name.
- *Def* – the line a variable is defined or redefined on. Def is used to differentiate between variables with the same name but in differing scopes.
- *Use* – the line a variable is used. This refers to a variable's value being used in a computation with no modification to its value. This can be used to construct def-use chains.
- *Slines* – all lines that a variable is defined or used on. This is the union of def and use.
- *Cfunctions* - a list of functions called using the slicing variable.
- *Dvariables* - a list of variables that are data dependent on the slice variable.
- *Pointers* - a list of aliases of the slicing variable. The elements of this list are variables to which the slicing variable is a pointer.

srcSlice produces a system dictionary of all the slice profiles of all variables. It is three-tiered and consists of three maps. On the first level is a map from files to functions, on the second level is a map from functions to variable names, and on the third level is a map from variable names to slice profiles.

4.2.4 Slice-Based Metrics Computation

By using semantic approaches that depend on static-program analysis we can extract facts and other information focusing on semantic aspects of the system. Slicing process removes from consideration parts of the program that are determined to have no effect upon the semantics of interest. It is possible to determine the parts with different behaviors by comparing the slices of two artifacts. The assumption here is that if the slice of artifact x differs from the slice of artifact y, then by the mean of the slicing definition, artifact x potentially exhibits different behavior than artifact y. Thus, program slices have the additional advantage of capturing program behavior, and hence the slice-based metrics are more directly related to the program behavior. Additionally, unlike most other metrics, slice-based metrics are based on program slice information, which is of finer granularity than the measures of many other metrics.

Using the output generated from *srcSlice*, we parse it into a data structure and then calculate the metrics. The first metric is *sliceCount*, it measures the file slices, which is equivalent to the number of paths in the code representation model (e.g., program dependence graph). This simply counts the number of entries in the system dictionary produced by *srcSlice* for each file. Using the information stored in each slice profile, we can easily retrieve the size of the slice for each variable in the system, identifier density, and slice spatial.

sliceSize represents the mean size of a variable slice in a file, measured in number of lines of code. It indicates how much the slice profiles depend on each other by intra-procedural or inter-procedural control or data dependencies. *sliceSize* counts all lines that

a variable is defined or used. This is the union between def and use in the slice profile. Therefore, the *sliceSize* is the ratio of all slice sizes to the *sliceCount*.

Dvars, pointers and cfunc fields in the slice profile capture the distinct identifiers appeared in the slice including variable names and methods invocation. For individual slice, number of identifiers is the count of items in dvars, pointers, and cfunc fields. Accordingly, file *sliceIdentifier* is the mean of all slice identifiers to the *sliceCount*.

sliceSpatial is the extent to which the slice scatter within a file. For an individual slice, we calculate *sliceDistance* as the distance in LOC between the definition and the last use of the slicing variable divided by the file size. As stated earlier, Def and use list all the line numbers in ascending order where the slicing variable is defined or used. Thus, *sliceDistance* is the subtraction of the first use from the last use divided by file size. *sliceSpatial* of the file is then measured as the mean of the individual *sliceDistance*.

We also include *sliceCoverage* metric similar to the one proposed by Weiser (Weiser 1981). However, *srcSlice* is a static slicing technique, which consider subsets of the program with respect to all possible executions/behaviors, while Weiser uses dynamic slicing that is suitable to identify code fragment with respect to one execution (Weiser 1981). We include *sliceCoverage* because of its relation to comprehension as it represents the active portion of the file that the programmer needs to traverse and comprehend (Weiser 1981). By comparing the slice size to the file size, we can measure the *sliceCoverage*, which is the mean slice size relative to file size.

Metric	Description
sliceCount	Number of slices within a file
sliceSize	Average slice size measured in LOC
sliceIdentifier	Average of distinct occurrences of programmer defined labels within a slice
sliceSpatial	Average of spatial distance in LOC between the definition and the last use of the slice divided by the file size
sliceCoverage	Average slice size relative to LOC

Table 4.2.1: Description of file level slice-based metrics.

Table 4.2.1 lists the definitions of these slice-based metrics. Generally, high file-level values of these metrics indicate more logically complex code and potentially more complex behaviors that are difficult to understand, inspect and trace in maintenance activities and hence exhibit more system defects. Note that slice metrics can be calculated at different level of granularity (i.e., system, file, method, and variable), however, due to the file-based nature of Git, a file-level granularity is used in this dissertation.

Table 4.2.2 shows the computations of the slice-based metrics of the running example. In this table, the first rows are the computations for slice-level while the last row shows the computations for file-level.

Granularity level		Metric	Computation	Value
Slice-level	abc	*sliceCount*	1	1
		sliceSize	8	8
		sliceCoverage	8 / 18	0.44
		sliceIdentifier	3	3
		sliceSpatial	16 / 18	0.89
	i	*sliceCount*	1	1
		sliceSize	10	10
		sliceCoverage	10 / 18	0.56
		sliceIdentifier	3	3
		sliceSpatial	16 / 18	0.89
	z	*sliceCount*	1	1
		sliceSize	2	2
		sliceCoverage	2 / 18	0.11
		sliceIdentifier	0	0
		sliceSpatial	1 / 18	0.06
	y	*sliceCount*	1	1
		sliceSize	2	2
		sliceCoverage	2 / 18	0.11
		sliceIdentifier	1	1
		sliceSpatial	2 / 18	0.11
	x	*sliceCount*	1	1
		sliceSize	4	4
		sliceCoverage	4 / 18	0.22
		sliceIdentifier	2	2
		sliceSpatial	5 / 18	0.28
File-level		*sliceCount*	1+1+1+1+1	5
		sliceSize	(8+10+2+2+4) / 5	5.2
		sliceCoverage	(0.44+0.56+0.11+0.11+0.22) / 5	0.29
		sliceIdentifier	(3+3+0+1+2) /5	1.8
		sliceSpatial	(0.89+0.89+0.06+0.11+0.28) / 5	0.45

Table 4.2.2: Slice-based metrics computations of running example.

CHAPTER 5

SOFTWARE DEFECT PREDICTION PROCESS

The common process of software defect prediction relies on machine learning models. The key insight behind these models is learning from software evolution history. Most software uses software configuration management (SCM) systems such as SVN or Git to record the evolution of a software project. Recorded data includes change history, log messages, and bug fixes that cover years of data. This information can be a useful resource for learning from previous defects and predicting the new ones.

Software defect prediction relies on three main components; dependent variables, independent variables and a model. The first step in building the model is to collect instances and history information from software archives. Instances can represent different granularity such as system, a software component (or package), a source code file, or a class. Due to the file-based nature of Git, a file-level granularity is used in this work. Processing the raw data falls into two folds:

1. Labeling instances as defective/non-defective or defects count. Defect data are the model for the dependent variables.
2. Extracting metrics to determine useful patterns in a bug-fix occurrence can be used for prediction. Metrics are the independent variables.

After generating the corpus, i.e., instances with metrics and labels, preprocessing techniques can be applied which are common in machine learning. Such techniques used

in defect prediction studies include feature selection, data normalization, and noise reduction (Zhang et al. 2014; Nam et al. 2017; Tantithamthavorn et al. 2018). Preprocessing is an optional step and are not applied on all defect prediction studies, e.g., (D'Ambros, Lanza, and Robbes 2010; Zimmermann and Nagappan 2008).

The final step is training a prediction model, so the model can predict whether a new instance has a defect or not. The prediction for defect-proneness (defective/non-defective) of an instance is based on binary classification, while that for the defects count in an instance is based on regression (ranking). Figure 5.1 shows the file-level defect prediction process used in this dissertation. Following subsections provide details of our process steps.

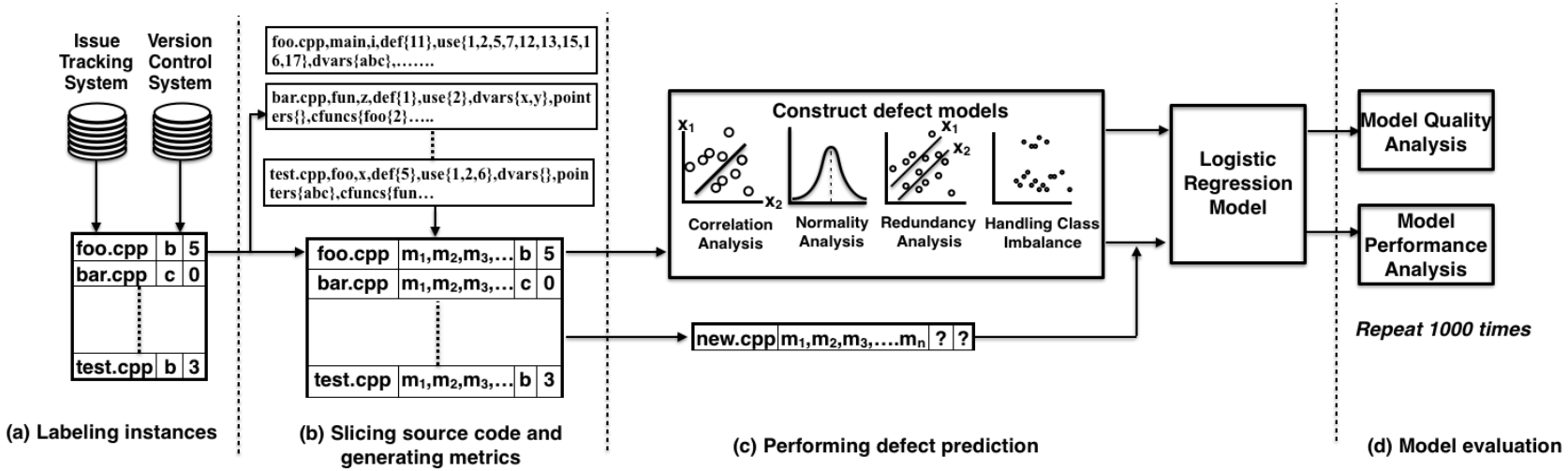


Figure 5.1: Overview of the study design.

5.1 Creating a Labeled Dataset

Source code management systems (SCM) contain a rich version history of every file in software projects. This information includes the full history of commits to each file: timestamps, authorship, change content, and the commit log. Since we will evaluate the performance of bug prediction using program slicing metrics, we need to determine whether a file actually contains bugs and how many times a file is included in a bug fixing task. Typically, bugs are discovered and reported to an issue tracking system such as Bugzilla and later on fixed by the developers. In order to link files with bugs, for each system we download (clone) the repositories from Git, and for each repository *r*, we created a series of patch files $\{P_i\}_{i=1}^{n}$, where *n* is the latest revision number for repository *r* at specific release. Each patch file P_i was responsible for transforming repository *r* from revision r_{i-1} to revision r_i, where r_1 is the initial revision for specific release. By initially setting repository *r* to revision *1* (i.e. the initial revision) and then applying all patches $\{P_i\}_{i=1}^{n}$ in a sequential manner, the revision history for that repository *r* was essentially replayed. Conceptually, this was equivalent to the case of all developers performing their commits sequentially one by one according to their chronological order. To perform bug labeling, we assume that a file has a bug if it is involved in a bug fixing transaction along the whole history for the period under consideration.

The labeling begins with links between bugs reported in the issue tracking system and the specific revision that fixes the bug- we call this a bug-fixing revision. Additionally, we should note that not all bugs are maintained in a bug-tracking database as described by Williams and Hollingsworth (Williams and Hollingsworth 2004). Therefore, we use

several different heuristics to derive our data. Various key words such as “bug”, “fixed” etc. in the git commit log are used to flag bug-fixing transaction. Also, numerical bug ids mentioned in the commit log, are extracted using regular expression and linked back to the issue tracking system’s identifiers. Finally, manual inspection is used to remove spurious linking as much as possible. These heuristic was introduced and used in many bug prediction studies (Śliwerski, Zimmermann, and Zeller 2005; Nagappan, Ball, and Zeller 2006; Bachmann and Bernstein 2009; Rahman and Devanbu 2011; D’Ambros, Lanza, and Robbes 2012).

In cases where the lack of supporting information (e.g., undescriptive ticket and / or commit message) prevents us from classifying a certain commit with satisfactory confidence, that commit is dropped from the dataset. Overall, we dropped 11% (3670), of the defect commits. We repeat this routine until we cover all commits involved in the period under consideration. Further in classification, we excluded any commit for fixing a broken unit test since these are not the ones that matter for the users of a program. Once we know that a transaction contains a fix, we first list files changed in the transaction and then check out the files prior to the fix in order to extract the slicing metrics. Additionally, files are filtered to remove non-source code (e.g., XML, html, log, documentation files, etc.) and unit-test files that are part of the bug-fix commit.

5.2 Cognitive Complexity Metrics

At this point we have a model including source code information over several versions, change history, and defects data. This step is to enrich the model with the metrics we want to evaluate. For each retrieved file, we use the *srcslice* tool (Alomari et al. 2014;

Newman et al. 2016) to compute the slicing metrics at the file level in every repository we had selected to be part of the dataset. To extract slicing metrics, we check out each file of the preceding revision that fixed specific bug (i.e. revision hash~1). In this manner, we can compute slicing metrics prior of fixing the bug to evaluate the nature of defect-prone files. We calculate slice metrics exactly before fixing the bug instead of the common way that calculate the metrics at the end or beginning of the release. The reason for choosing this method is that files could undergo through massive changes through the same release, so metrics could not be representative and thus less accurate. We applied this method to extract all slicing metrics through the revision history as described in Chapter 0.

5.3 Baseline Metrics

In order to quantify the contribution of slice-based metrics, we selected traditional code metrics and process metrics as a control set for providing a comparison. The aim of our study is to investigate five new metrics that demonstrate features of cognitive complexity and then to validate the efficacy of these metrics in defect prediction. To fulfil this, we choose five baseline metrics to perform balanced (5 vs. 5) analysis which avoid the possibility of diluting the effect of the new metrics if compared with large number baseline metrics. This approach in addition ensures that our analysis overcome any overfitting or multidimensionality problem that commonly happened with large number of variables.

Prior research on defect modeling found that product metrics are good indicators of defects (Menzies, Greenwald, and Frank 2007). Similarly, these code metrics are widely used to measure programmer's comprehension effort in research studies related to program

readability and understandability (Scalabrino et al. 2017; A. Rahman 2018). Table 5.3.1 describes the included baseline code metrics, which consist of size metric (i.e., *NLOC*), structural complexity metric (i.e., *McCabe's* (McCabe 1976)), and software science metric (i.e., *Halstead's Program Length* (Halstead 1977)).

The size metric *NLOC* simply counts the non-commentary source lines of code in a function. There is evidence that a larger size function tends to be more defect prone (Basili and Perricone 1984; Moller and Paulish 1993; Fenton and Ohlsson 2000). The structural complexity metric, including the well-known *McCabe's* Cyclomatic complexity metric, assumes that a function with complex control flow structure is likely to be defect-prone (Munson and Khoshgoftaar 1992; Ohlsson and Alberg 1996; Basili, Briand, and Melo 1996; Darcy et al. 2005). *Halstead's* length metric estimate reading complexity based on the counts of tokens, in which a function hard to read is assumed to be hard to understand and defect prone.

Note that we exclude other class-level code metrics such as CK and OO metrics since the analysis of this work is a file-level granularity. We also do not include the other Halstead's metrics because these metrics are fully based on the counts of operators and operands. Consequently they are highly correlated with each other (Farrar and Glauber 1967).

Additionally, process metrics are found to be powerful indicators in defect modeling and show improvement when combined with code metrics (Rahman and Devanbu 2013). Therefore, we include two widely used change metrics in defect prediction, namely *lineChange* the number of lines changed (i.e., added and removed) and

funcChange the number of functions changed within a file. By choosing these five baseline metrics we are able to perform a balanced (5 vs. 5) analysis which ensures that any overfitting or multidimensionality problem that commonly happened with large number of variables is avoided.

Category	**Metric**	**Description**
Size	*NLOC*	Source lines of code in a function (excluding comment lines)
Structural complexity	*CCN*	Cyclomatic complexity
Software science	*Program Length*	Total number of operators and operands of a function
Process	*lineChange*	Average number of lines added and deleted of a function
	funcChange	Average number of functions changed within a file

Table 5.3.1: Description of the baseline metrics.

CHAPTER 6

EXPERIMENTAL DESIGN

In this chapter, we introduce the projects used in the study and the research questions relating slice-based metrics to defect-proneness. Then, we describe the modeling techniques and the data analysis methods.

6.1 Test Systems

We use 10 datasets of 7 open-source projects to investigate the usefulness of cognitive complexity metrics in defect prediction. In selecting the systems, we consider three important criteria:

6.1.1 Different Corpora

To extend the generality of our conclusions, we choose systems from different corpora and domains. The included systems are non-trivial software that are belonging to different problem domains and different programming languages.

6.1.2 Sufficient EPV

Prior studies show that the Events Per Variable (EPV) (i.e., the ratio of the frequency of the least occurring class in the outcome variable to the number of features that are involved in training of a classifier) has a significant influence on the performance of defect classifiers (Tantithamthavorn et al. 2017). In particular, defect classifiers trained with datasets with a low EPV value yield unstable results (Tantithamthavorn et al. 2017; 2016). To ensure the stability of our results, we ensure that included datasets have an EPV

value that is larger than 10 _. Particularly, the systems we select have EPV ranging from 14 to 718 (see Table 6.1.1).

6.1.3 Defect Rate

Since it is unlikely that more software modules have defects than are free of defects, we choose to study datasets that have defective rate ranging from (6%) to (50%). Table 6.1.1 summarizes the details of the projects examined in this dissertation.

Subject	Application type	Prog. lang.	Period	# Revisions	Defect revision rate %	# Instances	Defective instances rate %	EPV
Linux 3.13	Operating system	C	01-19-2014 ~ 03-29-2014	13844	30%	35,397	10%	718
Eclipse 3.1	IDE	Java	06-27-2005 ~ 06-28-2006	2283	29%	1,045	50%	104
Eclipse 3.2			06-29-2006 ~ 06-24-2007	1643	35%	1,122	41%	92
Koffice 2.0	Office suite	C++	05-20-2009 ~ 11-20-2009	1632	32%	4,424	6%	51
Apache HTTP 2.0	Web server	C	04-06-2002 ~ 02-07-2005	5919	19%	266	38%	20
Apache HTTP 2.2	Web server	C	09-11-2012 ~ 11-16-2013	465	15%	402	17%	14
Dolphin 14.11~18.8	File manager	C++	11-08-2014 ~ 09-06-2018	709	21%	327	23%	15
Lucene 3.0	Information retrieval	Java	11-25-2009 ~ 03-29-2011	2696	21%	4599	14%	129
KDE Krita 3.0~ 3.1.3	Graphics editor	C++	05-30-2016 ~ 04-28-2017	2396	31%	5,518	10%	111
KDE Krita 3.1.4~ 4.0			29-04-2017 ~ 08-01-2018	1689	28%	5,166	7%	68
Average	-	-	-	3328	26%	5827	22%	-

Table 6.1.1: Revisions, file instances and % of defective files.

6.2 Correlational Analysis

In order to investigate our first research question (RQ1), we apply correlation coefficient analysis. The correlation coefficient is a bivariate analysis to measure the strength of the relationship between two variables and the direction of this relationship. Thus, we determined the correlation between the number of defects and each slice-based measure.

In statistics, there are several types of correlation coefficients. Widely used types are Pearson correlation, Spearman correlation, Kendall rank correlation, and Point-Biserial correlation. Each one has its own definition and formula. They all calculate the values in the range of -1 to +1, where -1 indicates the strongest negative relationship and +1 indicates the strongest positive relationship (Boddy and Smith 2009). These values can have the following meanings:

- A correlation coefficient of 1 means that for every positive increase of 1 unit in the first variable, there will be a positive increase of 1 unit in the other variable.
- A correlation coefficient of -1 means that for every positive increase of 1 unit in the first variable, there will be a negative increase of 1 unit in the other variable.
- Zero value means that for any positive or negative increase in the first variable, there will be no change in the other variable. This means the two variables are completely unrelated.

In the following subsection, I will discuss the Spearman correlation, as it is the one used in our analyses.

6.2.1 Spearman Rank Correlation

Spearman rank correlation coefficient, also known as Spearman rho or r_s, named after Charles Spearman, is a non-parametric measure of rank correlation between two variables. In contrast to Pearson correlation, the Spearman rank correlation is a robust technique that can be applied even when the association between two variables is non-linear. This correlation is applicable for continuous and discrete ordinal variables (Lehman et al. 2013). Spearman correlation coefficient can be defined as the covariance of the two variables divided by the product of their individual standard deviations. To calculate the Spearman correlation, the following formula is used:

$$r_s = \rho_{rg_X, rg_Y} = \frac{cov(rg_X, rg_Y)}{\sigma_{rg_X}\sigma_{rg_Y}}, \quad (6.1)$$

where:

r_s = Spearman correlation coefficient

ρ = the usual Spearman correlation coefficient, but using ranked variables

rg_X = ranked values of Xi

rg_Y = ranked values of Yi

$cov(rg_X, rg_Y)$ = covariance of the ranked variables

$\sigma_{rg_X}\sigma_{rg_Y}$= standard deviation of the ranked variables

6.2.1.1 Assumptions

Spearman correlation can be used when the association between values is non-linear. Spearman correlation determines the monotonic association between variables rather than linear association. This explains why the assumptions, normality, linearity and homoscedasticity, are not required for Spearman correlation.

6.2.1.2 Statistical significance

The 95% confidence interval (CI) of a Spearman's rank correlation coefficient is computed by bootstrapping with 1,000 replicates (Hervé 2019). The significance (p-values) of the correlation are computed using algorithm AS 89 for n<1290 when exact compute was allowed (Best and Roberts 1975) otherwise Edgeworth series approximation with cutoff modification from the original (Hollander and Wolfe 1999).

6.3 Modeling Techniques

If slice-based cognitive complexity metrics correlate with defects, can we use them to predict defects? This question is essential to answer:

RQ2. Do slice-based cognitive complexity metrics contribute to the prediction of the probability of defects?

Therefore, we build multiple regression models where the number of defects forms the dependent variable in binary classification, representing whether an instance is defective or non-defective. We build separate models for two sets of independent variables:

- **BMM** (baseline metrics model): This set consists of all code and process metrics section 5.3 and Table 5.3.1.

- **SBCCM** (slice-based cognitive complexity model): This set of variables includes the addition of slice-based metrics that were introduced in section 4.1 and Table 4.2.1 to the baseline metrics.

6.4 Model Construction Process

In the following subsections, we present the detail of our model construction process.

6.4.1 Normality Analysis

Regression models expect normality in the outcome and in the predictors. Defect prediction datasets suffer from high skewed data typically do not follow a normal distribution (McIntosh et al. 2016; Shihab, Bird, and Zimmermann 2012) (e.g., defects exist only in a small portion of the files). Therefore, we apply a log transformation log2 (x+1) to reduce the skew and adequate the data to the regression assumption.

6.4.2 Correlation Analysis

Software metrics can be highly correlated to each other (Rajbahadur et al. 2017). Highly correlated metrics (i.e., $|\rho| > 0.7$) can lead to an inflated variance in the estimation of the outcome (Jr 2015). Prior to modeling, we evaluate the correlations among our extracted metrics. We use Spearman pair-wise rank correlation to better account for collinearity between predictors in the data. Afterword, we use Principal Component Analysis (PCA) (Jackson 2003) to build the regression models using sets of principal components (PC), which are independent instead of the actual independent variable (i.e., metrics). Therefore, these components do not suffer from multicollinearity, while at the same time they account for as much sample variance as

possible (i.e., feature selection). We use *prcomp* function from *stats* R package. We include PCs that account for at least 95% of the variance. Across systems, SBCCMs need an average of 80% of the components to account 95% of the data variance, while the BMMs need an average of 67% of the components.

6.4.3 Redundancy Analysis

To ensure principal components of the PCA do not include redundant predictors, the redundancy analysis is performed in an iterative manner in which components are dropped until no components can be predicted with an R^2 or adjusted R^2 higher than 0.9. Hence, we use the *redun* function from *Hmisc* R package and find no redundant PCs in all datasets (Jr and others 2018).

6.4.4 Handling Category Imbalance

Table 6.1.1 shows that our dependent variables are imbalanced, e.g., there are more non-defective instances than defective ones. If left untreated, the models will favor the majority category, since it offers more predictive power. To combat this bias, we use the SMOTE technique (Chawla et al. 2002) (provided by the *DMwR* R package ("DMwR-Package: Functions and Data for the Book 'Data Mining with R' in DMwR: Functions and Data for 'Data Mining with R'" n.d.)) which creates artificial data based on the feature space similarities from the minority modules. The SMOTE technique has been shown to improve AUC and been used in previous defect prediction studies (Tantithamthavorn, Hassan, and Matsumoto 2018).

6.4.5 Binary Logistic Regression

We conduct our experiments using binary logistic regression model. This technique is a standard statistical modeling technique in which the dependent variable can take two different values. It is suitable for building defect prediction models because the files under consideration are divided into two categories: defective and non-defective. Logistic regression predicts likelihoods between 0 and 1, i.e., the likelihood that a file contains at least one defect. The general form of a binary logistic regression is shown in the following equation:

$$logit(\pi) = \log\left(\frac{\pi}{1-\pi}\right) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_\kappa X_\kappa, \quad (6.2)$$

where $\pi = Pr(y = 1)$, y is the dependent variable, β is the regression coefficient and X is the predictor/independent variable. We choose logistic regression over other modeling techniques because it is a widely used technique in defect prediction and it yields the best performance for models that combine both process and code metrics (Rahman and Devanbu 2013). We use the method lrm from the RMS R package (Jr 2018).

6.4.6 Out-of-Sample Bootstrap

In order to ensure that the conclusions that we draw about our models are robust, we use the out-of-sample bootstrap validation technique, which has been shown to yield the best balance between the bias and variance (Tantithamthavorn et al. 2017). Unlike the ordinary bootstrap, the out-of-sample bootstrap technique fits models using the bootstrap samples, but rather than testing the model on the original sample, the model

is instead tested using the rows that do not appear in the bootstrap sample (Tantithamthavorn et al. 2017). Thus, the training and testing corpora do not share overlapping observations. The fraction of this sampling is 1/3 of the size of the original data. The entire bootstrap process is repeated 1000 times with the function validate from the RMS R package (Jr 2018), and the average out-of-sample performance is reported as the performance estimate.

6.5 Model Analysis

6.5.1 Logistic Regression Model Explanatory Power

6.5.1.1 Area under the ROC curve (AUC)

AUC measures the area under the receiver operating characteristic (ROC) curve. The ROC curve is plotted by false positive rate (FP) and true positive rate (TP). Figure 6.1 explains about a typical ROC curve. PF and PD vary based on threshold for prediction probability of each classified instance. By changing the threshold, we can draw a curve as shown in Figure 6.1. the AUC value characterizes the accuracy of the model across all possible cutoff values. When the model gets better, the curve tends to be close to the point of PD=1 and PF=0. Thus, AUC of the perfect model will have "1". For a random model, the curve will be close to the straight line from (0,0) to (1,1) (Menzies, Greenwald, and Frank 2007; Rahman, Posnett, and Devanbu 2012). Other measures such as precision and recall can vary according to prediction threshold values. However, AUC value characterizes the accuracy of the model across all possible cutoff values. In this reason, AUC is a stable measure to compare different prediction models (Rahman, Posnett, and Devanbu 2012). Larger AUC values indicate better performance (Wu and Flach 2005).

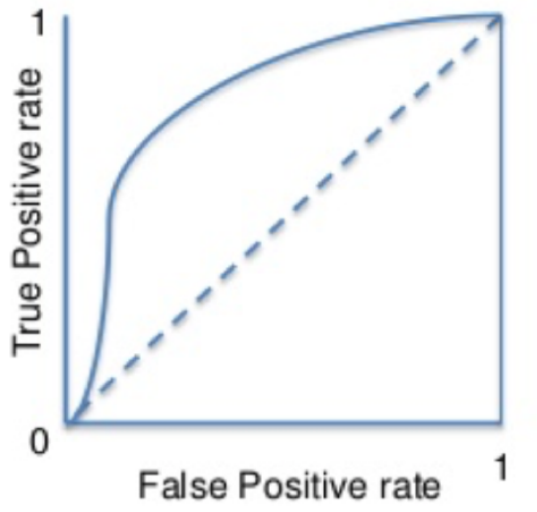


Figure 6.1: A typical ROC curve

6.5.1.2 Nagelkerke R2

Nagelkerke is a specialized R^2 typically used for logistic regression models (Nagelkerke 1991). Nagelkerke R^2 with larger values indicating more variability explained by the model and less unexplained variation—a high Nagelkerke R^2 (provided by the *lrm* method) value indicates good explanative power, but not predictive power.

6.5.1.3 Influential Observations

To identify influential observations in the models, we employ the Cook's distance to measure the effect of removing data point on all the predictors combined. If an observation has a Cook's distance equal to or larger than 1, it is regarded as an influential observation and is hence excluded for the analysis (Belsley, Kuh, and E. Welsch 2005). In addition, we examine observations for any unusual high leverage values, i.e., unusual

combination of predictor values. Any point with leverage value greater than 2.5 the average leverage of a point in the data set is investigated closely.

6.5.2 Logistic Regression Model Prediction Abilities

To assess the models prediction abilities, we use four possible classification outcomes:

- A file is classified as *defective* when it is truly *defective* (true positive, TP)
- It can be classified as *defective* when it is truly *non-defective* (false positive, FP)
- It can be classified as *non-defective* when it is truly *defective* (false negative, FN)
- It can be classified as *non-defective* when it is truly *non-defective* (true negative, TN).

Based on TP, TN, FP, and FN, we calculate precision, recall, and F1-score as follows:

- **Precision:** the proportion of files that are correctly labeled as defective among those labeled as defective.

$$P = TP/(TP + FP) \quad (6.3)$$

- **Recall:** is also known as true positive rate (TPR). Recall measures correctly predicted buggy instances among all buggy instances.

$$R = TP / (TP + FN) \quad (6.4)$$

- **F1-score:** is a harmonic mean that combines both precision and recall. It evaluates if an increase in precision (recall) outweighs a reduction in recall (precision).

$$F = (2 \times P \times R) / (P + R) \qquad (6.5)$$

F1 minimizes the trade-off between precision and recall that can cause difficulties to compare the performance of several prediction models by using only precision or recall alone. This follows the setting used in many software analytics studies (Kim et al. 2011; Nam, Pan, and Kim 2013; Tantithamthavorn et al. 2018). In general, the higher the F1-score is, the better the performance of an approach.

CHAPTER 7

EVALUATION RESULTS

In this chapter, we present our results, broken down by the research questions presented earlier in Section 1.2.

7.1 Research Question 1

Do Slice-Based Cognitive Complexity Metrics Significantly Correlate to Defects?

7.1.1 Data Distribution

To perform correlational analysis, we need first to examine the distribution of our datasets in all systems whether they follow normal distribution, where the data are symmetrically distributed, or not as this will determine the method of for analysis. There are several graphical and numerical tools for assessing normality of a data, we chose histogram visualization to test the distribution of our variables. Figure 7.1, Figure 7.2, Figure 7.3, Figure 7.4, and Figure 7.5 clearly display the asymmetrical distribution of our metrics in most instances. They suffer from substantial right skew indicating more clustered data at the left end of the distribution. Therefore, we choose the Spearman rank correlation coefficient method since it does not require normal distribution and assess relationship between variables using monotone function even if the relation is not linear.

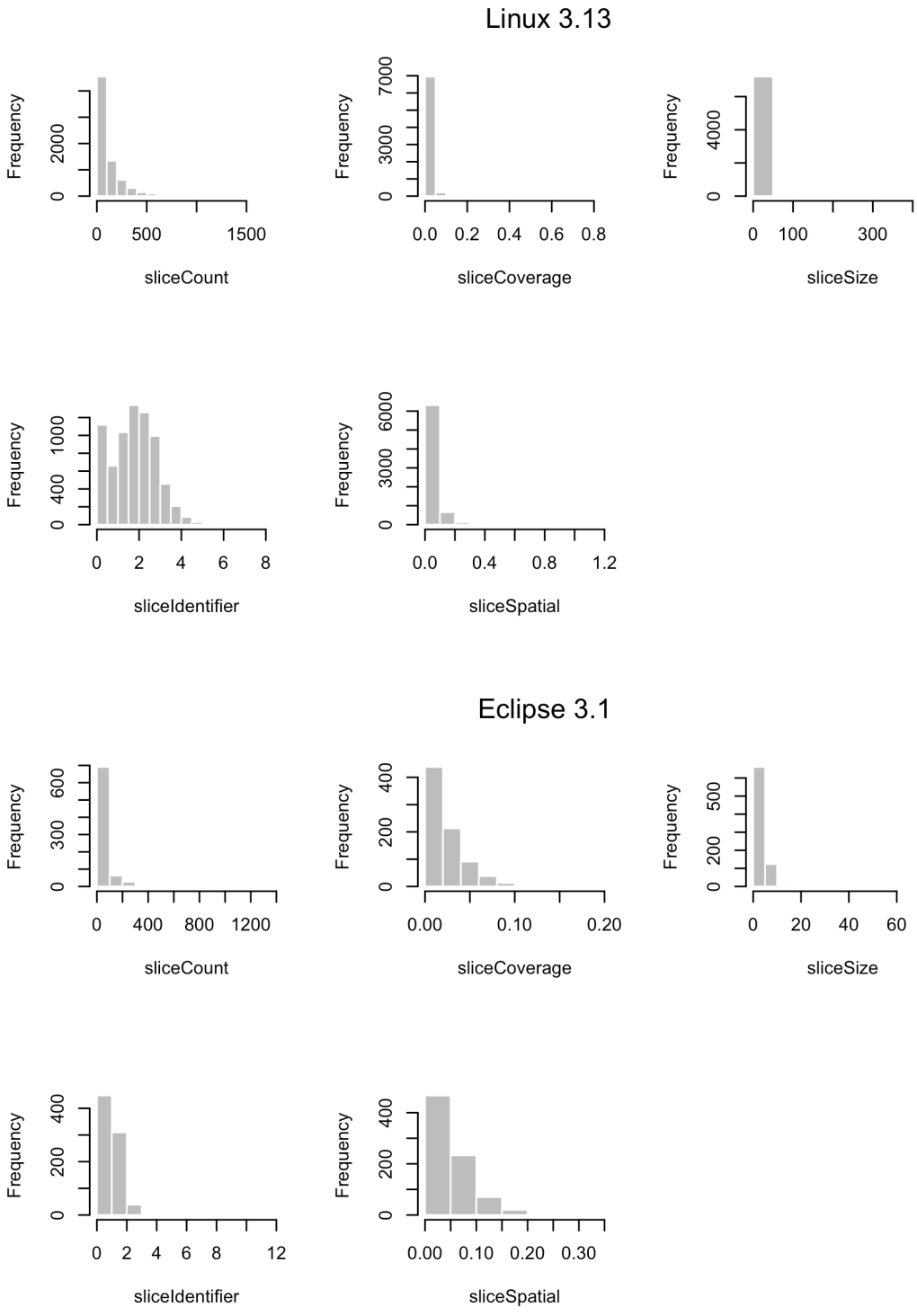


Figure 7.1: Histograms of slice-based metrics in Linux 3.13 and Eclipse 3.1.

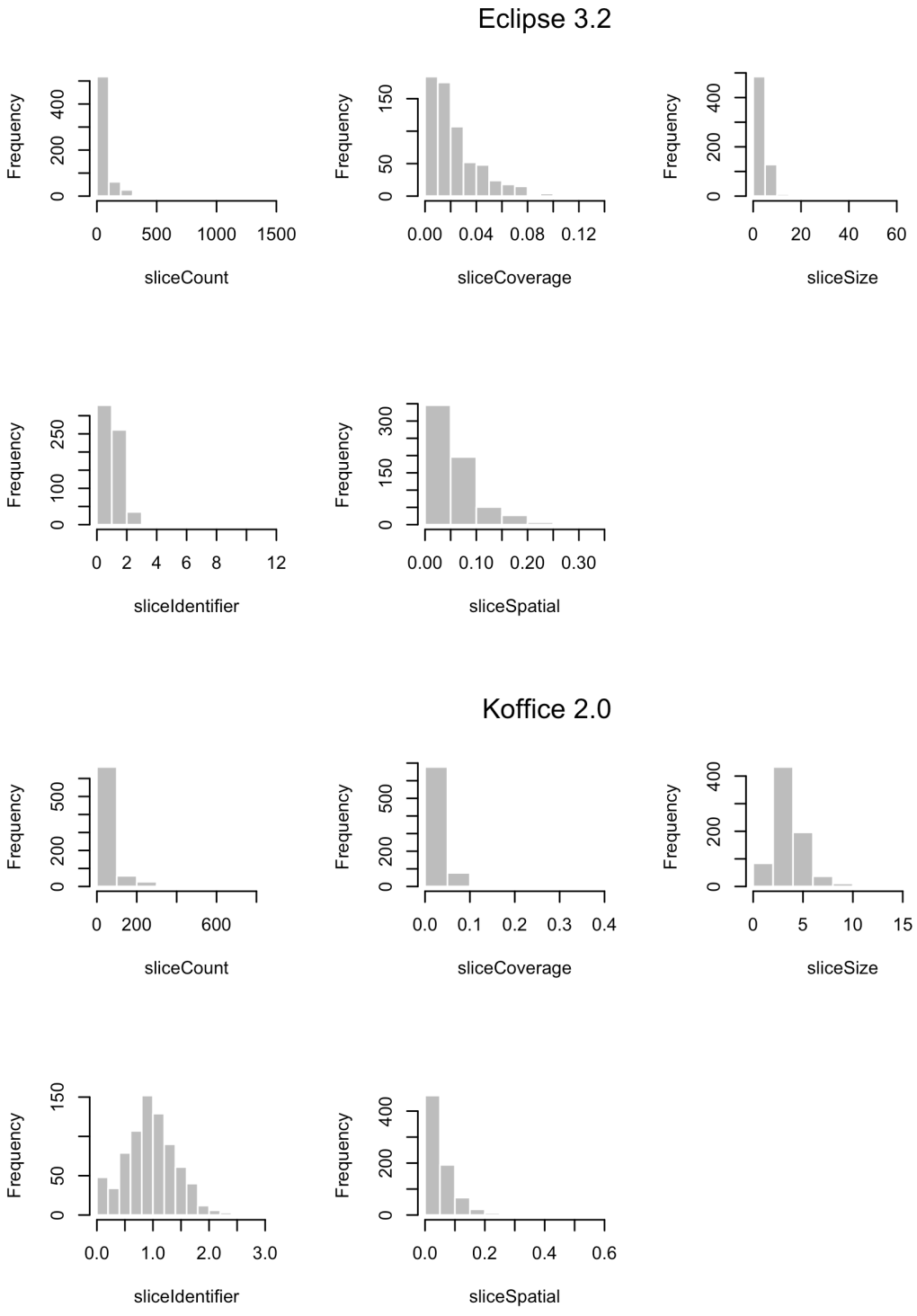


Figure 7.2: Histograms of slice-based metrics in Eclipse 3.2 and Koffice 2.0.

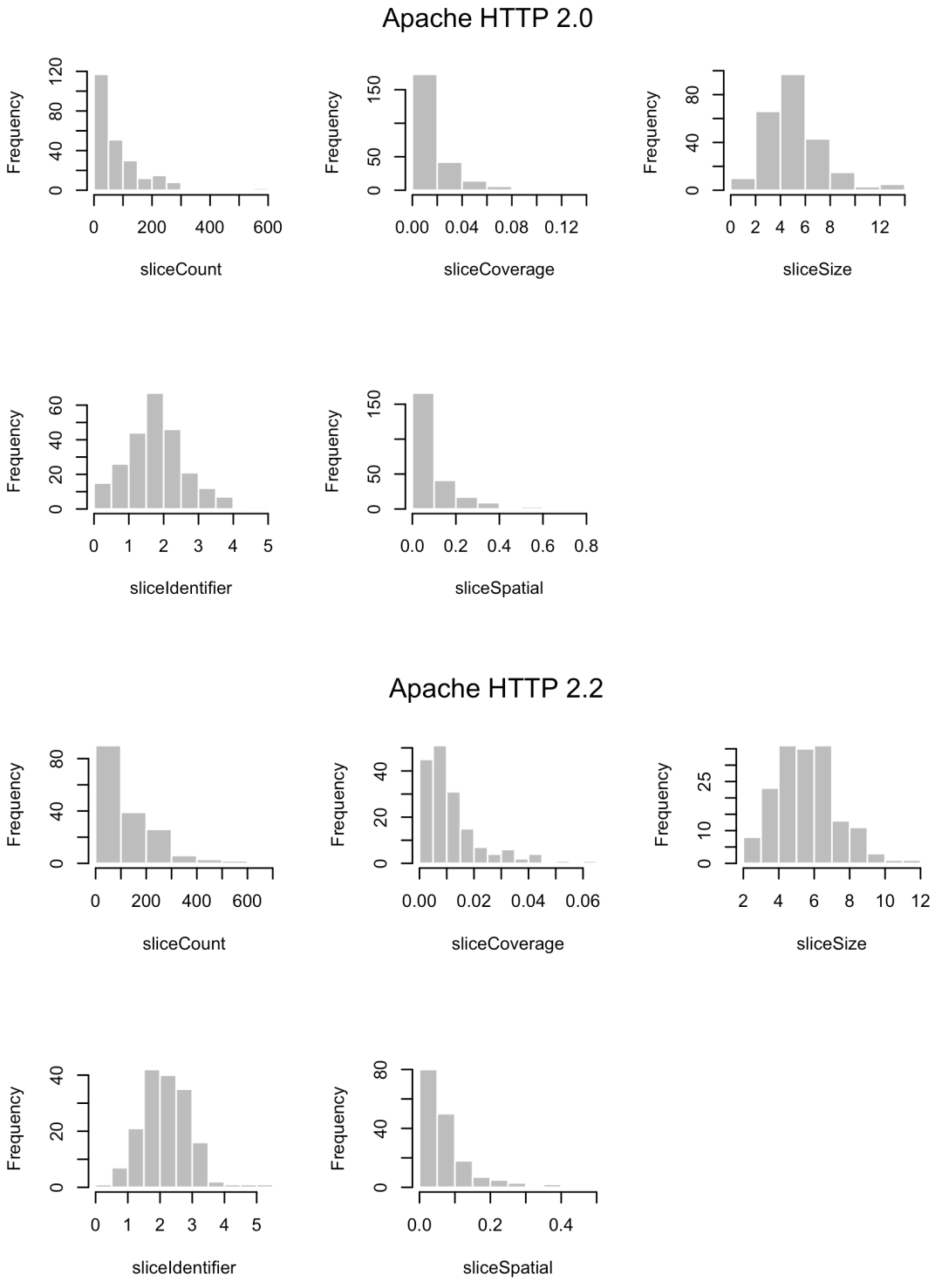


Figure 7.3: Histograms of slice-based metrics in Appache HTTP 2.0 and 2.2.

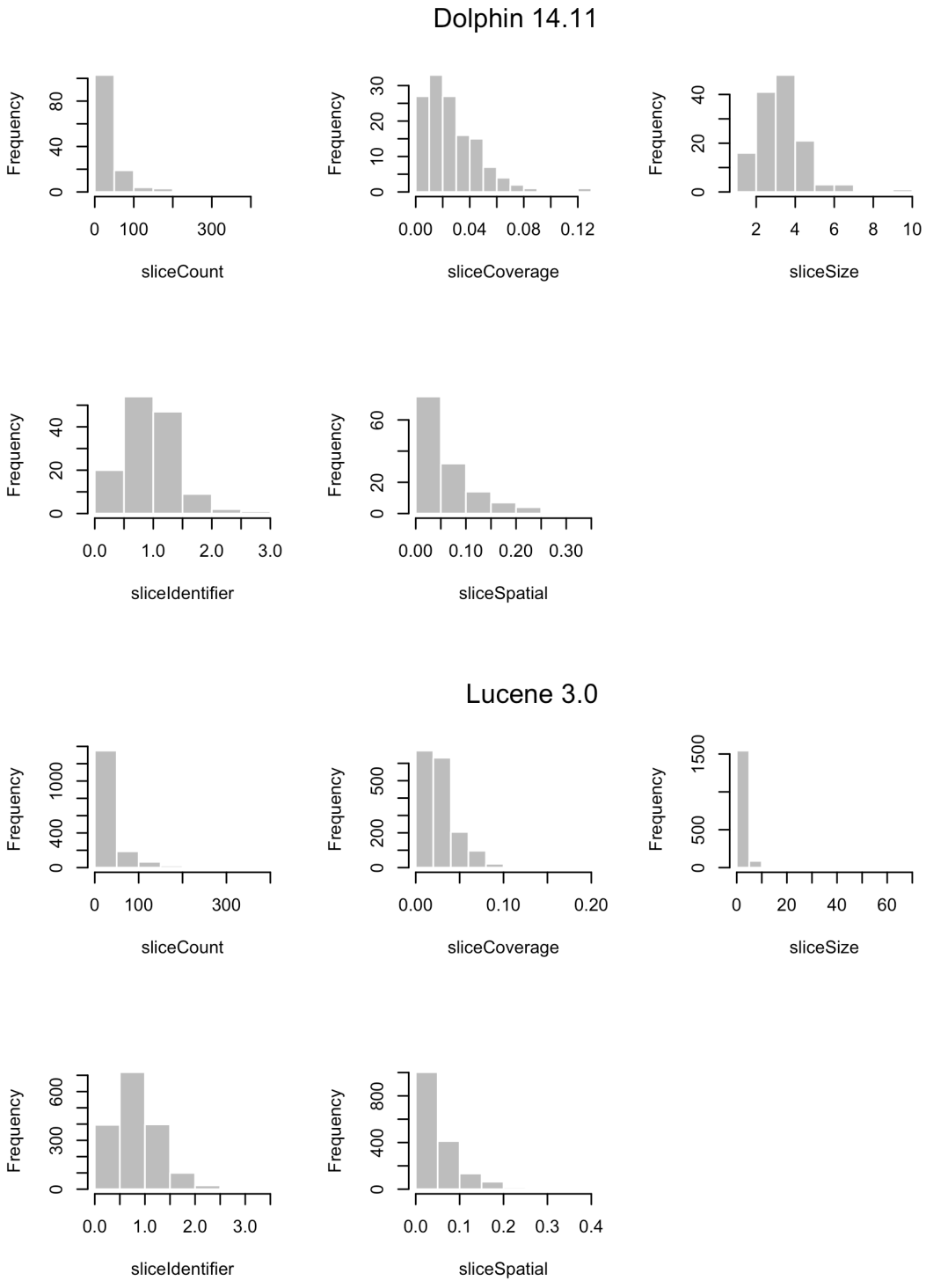


Figure 7.4: Histograms of slice-based metrics in Dolphin 14.11 and Lucene 3.0.

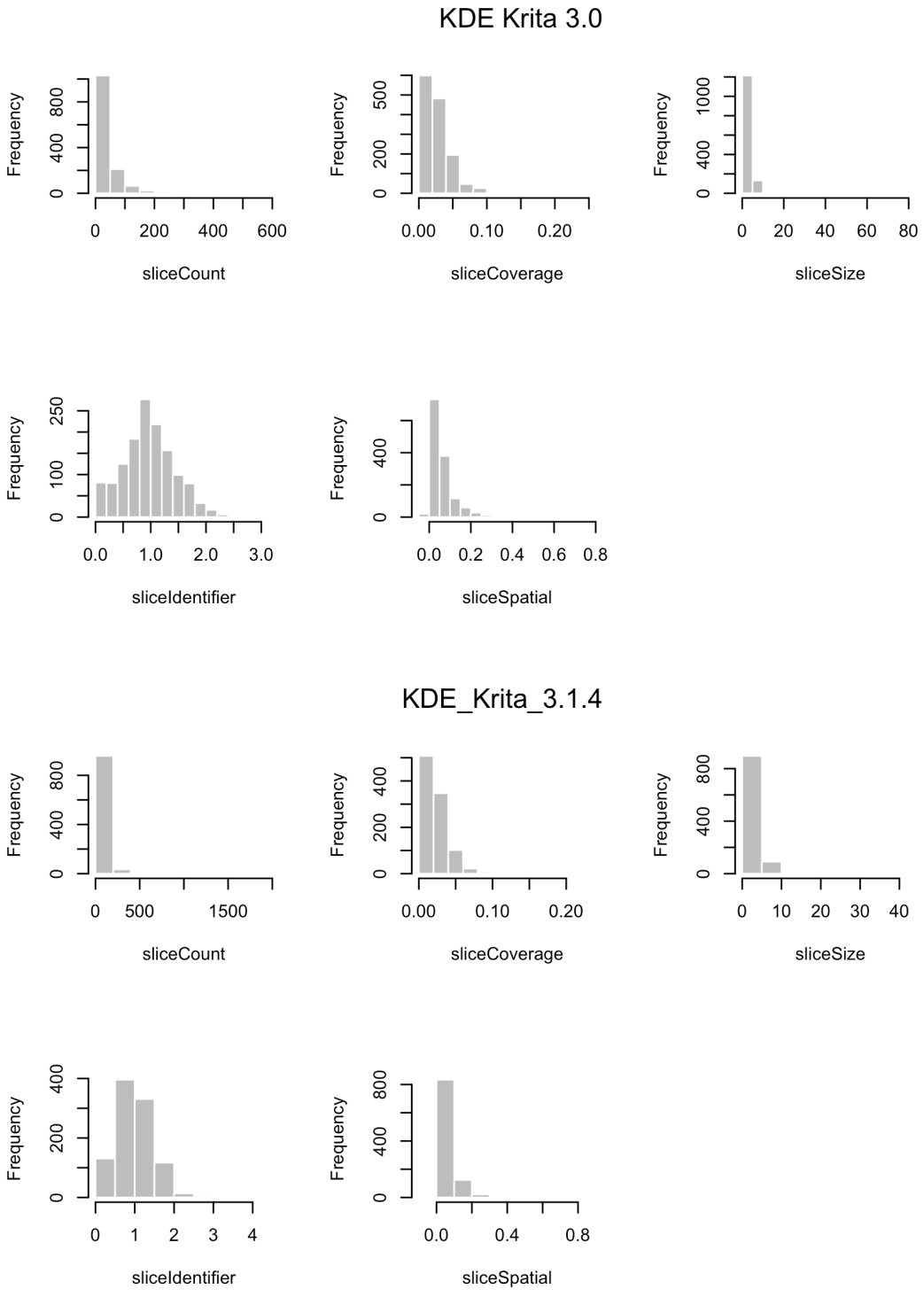


Figure 7.5: Histograms of slice-based metrics in KDE Krita 3.0 and 3.1.4.

7.1.2 Correlation Analysis

Figure 7.6 and Table 7.1.1 show the Spearman correlations for all systems computed by bootstrapping with 1,000 replicates. For clarification, the cells in Figure 7.6 distinguish between the correlation value and correlation intensity shown by the color range. All correlations are significant at 0.95-confidence level (p-value ≤ 0.05) except the correlations that are not bolded. From these results, we can make the following observations.

Most of the investigated slice-based metrics are significantly correlated with defects. In 94% (i.e., 47 out of 50) of the cases, slice-based metrics have p-value ≤ 0.05 and 95% CI that does not include zero. The *SliceCount*, *sliceSize*, and *sliceIdentifier*, show a consistent positive relationship with defect counts across all systems, which means that an increase in the aforementioned metrics leads to an increase in number of defects. This finding suggests that code that is divided into many parts (i.e., higher *sliceCount*), have a higher concentration of method invocations with parameters (i.e., higher *sliceIdentifier*), and have more tracing activity (i.e., higher *sliceSize*) during comprehension process, have a higher probability of defects. The increase of the aforementioned metrics indeed increases the cognitive complexity implying that developers should carefully handle files with high percentage of slice size, slice count and slice identifier.

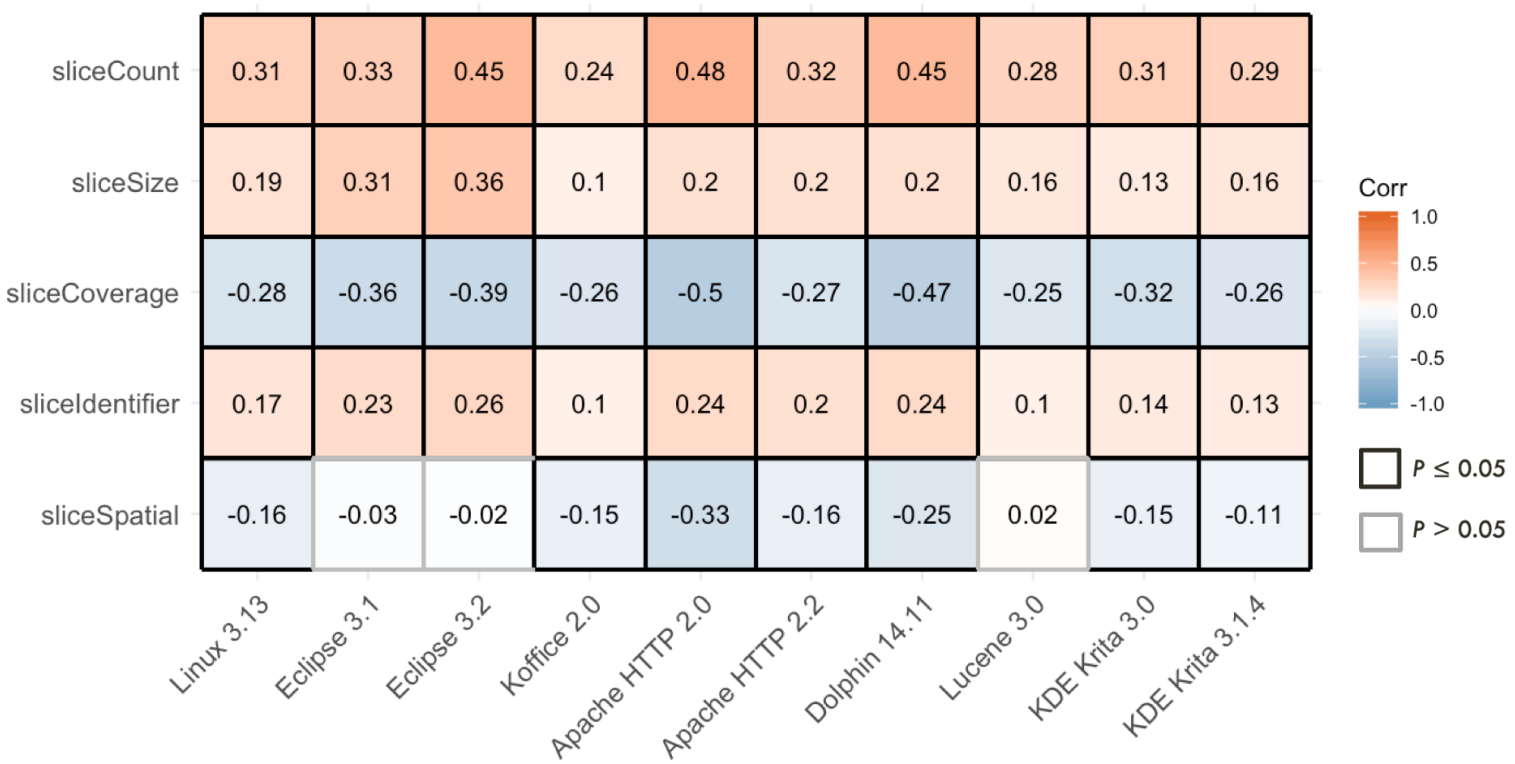


Figure 7.6: Spearman correlation coefficients between bug counts and metrics.

Subject	sliceCount			sliceSize			sliceCoverage			sliceIdentifier			sliceSpatial		
	r_s	95%CI		r_s	95%CI		r_s	95%CI		r_s	95%CI		r_s	95%CI	
		Lower	Upper		Lower	Upper		Lower	Upper		Lower	Upper		Lower	Upper
Linux 3.13	**0.31**	0.29	0.33	**0.19**	0.17	0.21	**-0.28**	-0.30	-0.26	**0.17**	0.15	0.19	**-0.16**	-0.18	-0.14
Eclipse 3.1	**0.33**	0.26	0.4	**0.31**	0.24	0.37	**-0.36**	-0.43	-0.3	**0.23**	0.16	0.30	-0.03	-0.1	0.04
Eclipse 3.2	**0.45**	0.38	0.52	**0.36**	0.29	0.43	**-0.39**	-0.45	0.32	**0.26**	0.19	0.33	-0.02	-0.11	0.06
Koffice 2.0	**0.24**	0.17	0.31	**0.1**	0.02	0.16	**-0.26**	-0.32	-0.19	**0.1**	0.03	0.16	**-0.15**	-0.22	-0.07
Apache HTTP 2.0	**0.48**	0.37	0.58	**0.20**	0.1	0.32	**-0.50**	-0.6	-0.4	**0.24**	0.12	0.36	**-0.33**	-0.44	-0.22
Apache HTTP 2.2	**0.32**	0.18	0.46	**0.2**	0.05	0.34	**-0.27**	-0.4	-0.13	**0.2**	0.05	0.34	**-0.16**	-0.31	-0.0005
Dolphin 14.11	**0.45**	0.27	0.60	**0.2**	0.06	0.35	**-0.47**	-0.61	-0.29	**0.24**	0.09	0.38	**-0.25**	-0.41	-0.08
Lucene 3.0	**0.28**	0.24	0.33	**0.16**	0.11	0.21	**-0.25**	-0.30	-0.20	**0.1**	0.03	0.14	0.02	-0.03	0.07
KDE Krita 3.0	**0.31**	0.26	0.36	**0.13**	0.07	0.18	**-0.32**	-0.37	-0.26	**0.14**	0.1	0.19	**-0.15**	-0.20	-0.1
KDE Krita 3.1	**0.29**	0.23	0.34	**0.16**	0.1	0.21	**-0.26**	-0.3	-0.21	**0.13**	0.07	0.19	**-0.11**	-0.17	-0.04
Average	0.35	0.27	0.42	0.20	0.12	0.28	-0.34	-0.40	-0.19	0.18	0.10	0.26	-0.13	-0.22	-0.05

Table 7.1.1: Spearman correlation coefficient r_s, *p*-value and confidence interval (CI) between defect counts and cognitive complexity metrics. All correlation coefficient values are statistically significant except values not bolded.

Conversely, *sliceCoverage* shows an expected consistent negative relationship across systems, which means an increase in *sliceCoverage* comes with a decrease in number of defects. *sliceCoverage* measures the average sliceSize relative to file LOC (i.e., sliceSize/sliceCount*file LOC). Thus, an increase in the file LOC and *sliceCount* decreases the *sliceCoverage*, leading to more cognitive complexity and eventually defects.

We find that *sliceSpatial* has a negative relationship with the probability of defects. This suggests that being in large and scattered slices do not necessarily result in a high number of defects. While, this correlation is very weak (i.e., -0.12) and all of the insignificant correlation cases (i.e., 3 out of 3) occurred in relation to *sliceSpatial*, one would expect a high slice spatial to increase the cognitive complexity leading to more defects. This might be due to the file level granularity of the analysis by taking the average of *sliceSpatial* within a file. A highly scattered slice might be hidden in a large block of unite code resulting in a low value for the large block. Therefore, *sliceSpatial* might provide a higher and positive correlation with defects if a finer granularity of analysis is used (e.g., at slice-level). However, without further investigation to support this analysis, we cannot claim such argument. For more details about the Spearman correlation and how do they look, the scatterplots for each relationship assessment are highlighted in APPENDIX A.

Overall, the individual correlations of each cognitive complexity metrics indicate a weak/moderate monotonic relationship with number of defects in most of the cases. The metrics with the highest observed correlations are *sliceCount* and *sliceCoverage.* These metrics are more related to defects than finer grained metrics (*sliceIdentifier, sliceSize*, and

sliceSpatial). This finding suggests that handling files with high slice coverage and slice count is more challenging and requires better understanding of the source code by developers. The results in addition suggest that these metrics might have more association if used together in relation to defects which is the analysis to be done in RQ2.

In summary, the characteristics of cognitive complexity captured by slice-based metrics express a statistically significant relationship with the probability of defects, suggesting that slice based cognitive complexity metrics can be used as defects indicators. Slice count, slice size and slice identifier metrics have a consistent and significant positive relationship with the probability of defects across systems, while slice coverage and slice spatial have a significant negative relationship, suggesting that handling files with higher cognitive complexity captured by slice-based metrics is more challenging and requires a better understanding of the source code by developers.

7.2 Research Question 2

Do Slice-Based Cognitive Complexity Metrics Contribute to the Prediction of the Probability of Defects?

To address RQ2, we apply the preprocessing techniques on the metrics as described in sections 6.4.1, 6.4.2, 6.4.3, and 6.4.4. After that, we follow the modeling process described in sections 6.4.5 and 6.4.6 to train multiple regression models for two different sets of predictors:

- SBCCM (slice based cognitive complexity model) and
- BMM (baseline metrics model).

7.2.1 Metrics Preprocessing

7.2.1.1 Normalization of the Data

We use the $\log_2(x+1)$ transformation method to deal with the asymmetrical distribution in the data. It helps in reducing the right skew distribution before performing the modeling. Most variables show a marked correction of the asymmetrical pattern and the rest show a reduction of the right skew distribution.

7.2.1.2 Collinearity and Redundancy Analysis

We perform PCA to deal with collinearity between predictors. APPENDIX B shows the PCA aspects (loadings, communalities, contributions to PCs, eigenvalues, and proportion of variance) of SBCCM and BMM. We include PCs that account for at least 95% of the variance, and on the same time do not suffer from multicollinearity. We after that ensure by the redundancy analysis that the PCs do not include redundant predictors in all datasets. In addition, to ensure that the models are not overfitted with any PCs, we calculate Variance Inflation factor (VIF) for all models. Across all systems, VIFs are < 3.5.

7.2.1.3 Category Balancing

As mentioned before, the dependent variables are clearly imbalanced, and this will lead to bias. Therefore, we use SMOTE technique to create artificial balanced data. Table 7.2.1 highlights the category numbers before and after balancing.

Subject	Before Balancing		After Balancing	
	Buggy	**Clean**	**Buggy**	**Clean**
Linux 3.13	3085	4131	6170	6170
Eclipse 3.1	479	322	644	644
Eclipse 3.2	416	214	428	428
Koffice 2.0	248	518	496	496
Apache HTTP 2.0	99	140	198	198
Apache HTTP 2.2	55	112	110	110
Dolphin 14.11~18.8	57	76	114	114
Lucene 3.0	340	1301	680	680
KDE Krita 3.0~3.1.3	548	814	1096	1096
KDE Krita 3.1.4~4.0	338	658	676	676

Table 7.2.1: The dependent variable counts before and after balancing by SMOTE technique.

7.2.2 Models Explanatory Power

We measure the explanatory power of models from the bootstrap training samples and measure the prediction performance of the models on the bootstrap testing samples. This process is validated by repeating the bootstrapping for 1,000 times.

The generated classifiers are evaluated using the AUC obtained from the training bootstrap samples. Table 7.2.2, Figure 7.7, Figure 7.8, Figure 7.9, Figure 7.10, and Figure 7.11 report the average AUC for each studied dataset. Overall, we find that SBCCM has an average AUC of 0.8 while BMM has an average of 0.71. In addition, ROC curves of

SBCCM where closer to the top left corner than BMM throughout different cut off points. This means SBCCM increases the AUC by 9% on average compared to BMM. In particular, the SBCCM reveals an increase in the AUC values across all systems by up to 16% and not less than 5%. This indicates that the addition of slice-based metrics certainly increases the discriminating power of the models. To measure the significance of the AUC differences between the two models, we use a Wilcoxon signed-rank test (Wilcoxon 1945) since it does not need the data to follow a normal distribution and it tests paired results. The test reveals that the differences between SBCCM and BMM are significant in all datasets.

Figure 7.12 highlights the AUC values distribution of the 1000 iterations of bootstrap of SBCCM and BMM. Both models have a high level of agreement on the computed AUCs, as the boxplots are comparatively short, and whiskers do not stretch over a wider range of values indicating reliable results. We observe a consistent trend of SBCCM outperforming BMM among all systems. SBCCM have a larger median and mean than the BMM. Across all systems, the lower whiskers of SBCCM are larger than BMM median. No sharable Inter-quartile ranges (IQR, i.e., middle box that represents 50% of AUC distribution) across systems. Moreover, all differences are significant using Wilcoxon signed-rank test.

Additionally, Nag. R^2 values of SBCCM in Table 7.2.2 show improvement in all datasets that reaches up to 35% with an average increase of 18% over BMM which indicates an improvement in the fitted model. Thus, SBCCM model has a substantially better classification performance than the BMM.

Subject	BMM		SBCCM	
	AUC	Nag. R^2	AUC (*)	Nag. R^2 (*)
Linux 3.13	0.62	0.04	0.72 **(+10%)**	0.18 **(+14%)**
Eclipse 3.1	0.67	0.15	0.78 **(+11%)**	0.30 **(+15%)**
Eclipse 3.2	0.73	0.20	0.79 **(+6%)**	0.32 **(+12%)**
Koffice 2.0	0.75	0.23	0.80 **(+5%)**	0.34 **(+11%)**
Apache HTTP 2.0	0.73	0.20	0.87 **(+14%)**	0.49 **(+29%)**
Apache HTTP 2.2	0.74	0.23	0.82 **(+8%)**	0.41 **(+18%)**
Dolphin 14.11~18.8	0.71	0.16	0.87 **(+16%)**	0.51 **(+35%)**
Lucene 3.0	0.69	0.14	0.78 **(+9%)**	0.30 **(+16%)**
KDE Krita 3.0~3.1.3	0.70	0.15	0.76 **(+6%)**	0.26 **(+11%)**
KDE Krita 3.1.4~4.0	0.74	0.21	0.80 **(+6%)**	0.35 **(+14%)**
Average	0.71	0.17	0.80 **(+9%)**	0.35 **(+18)**

* The values between ( ) represent the improvement between BMM and SBCCM.

Table 7.2.2: Logistic regression models average AUC, Nagelkerke R^2 values across systems using 1000 bootstrap validation (Bold font highlights the best performance).

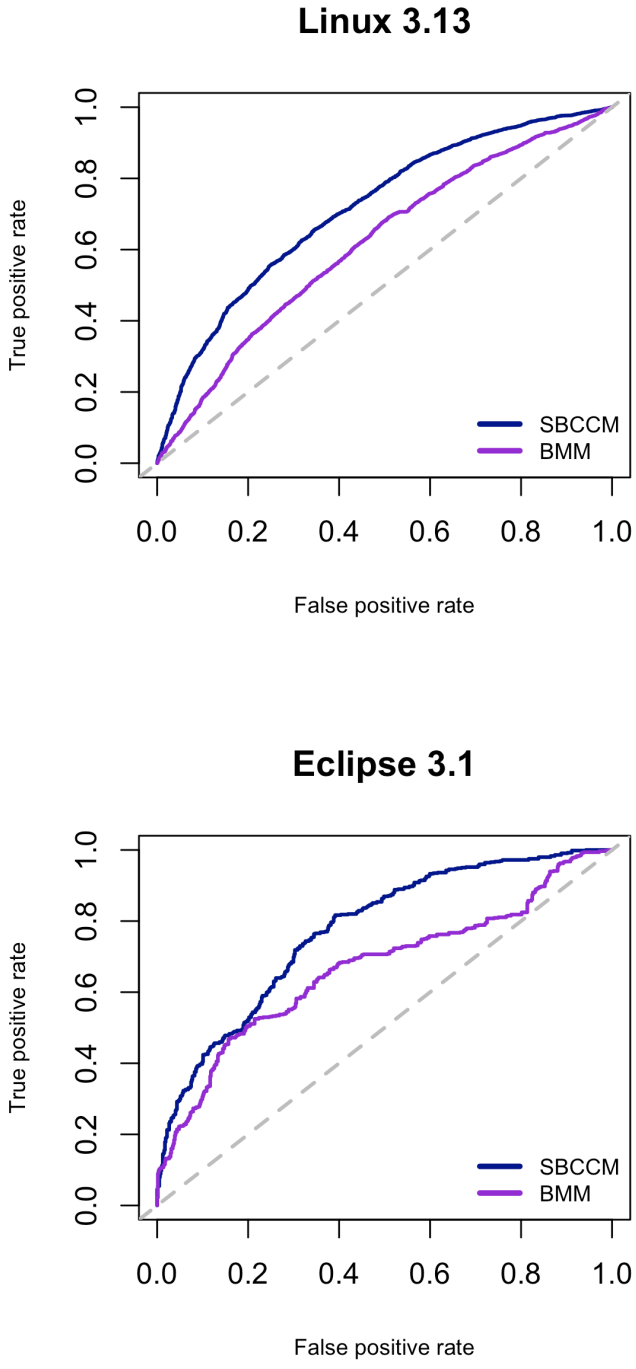


Figure 7.7: ROC curves comparing the models of SBCCM and BMM in Linux 3.13 and Eclipse 3.1

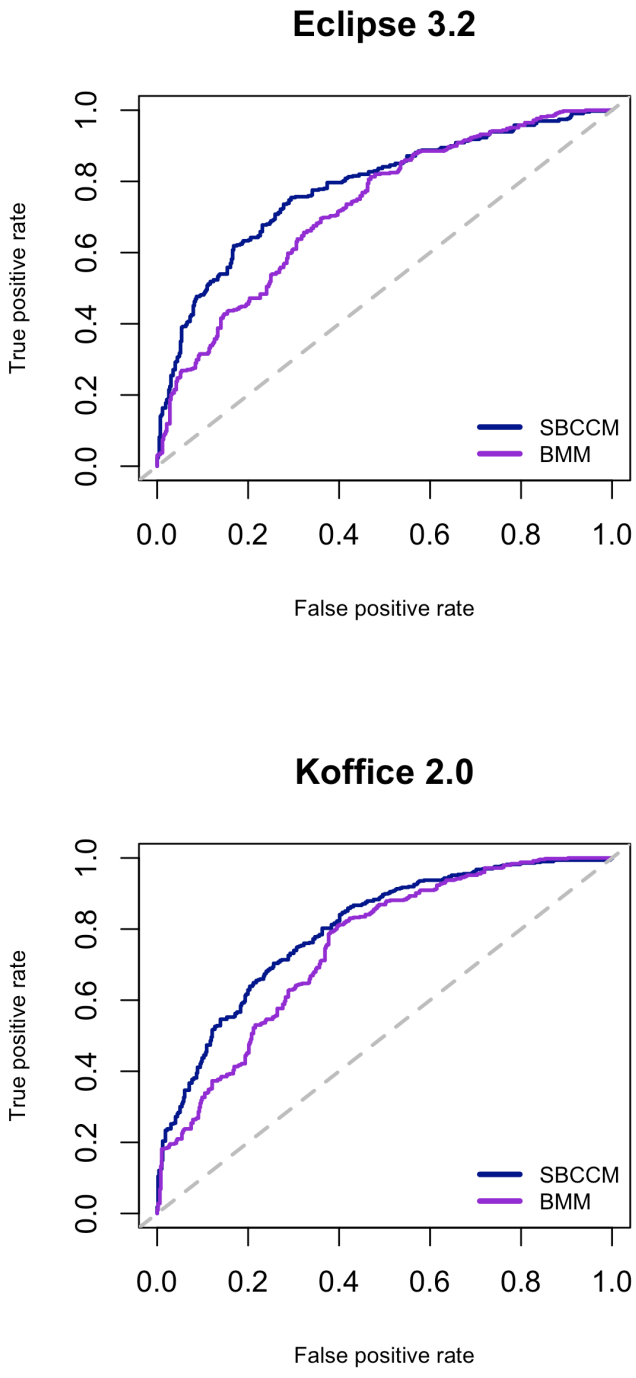


Figure 7.8: ROC curves comparing the models of SBCCM and BMM in Eclipse 3.1 and Koffice 2.0.

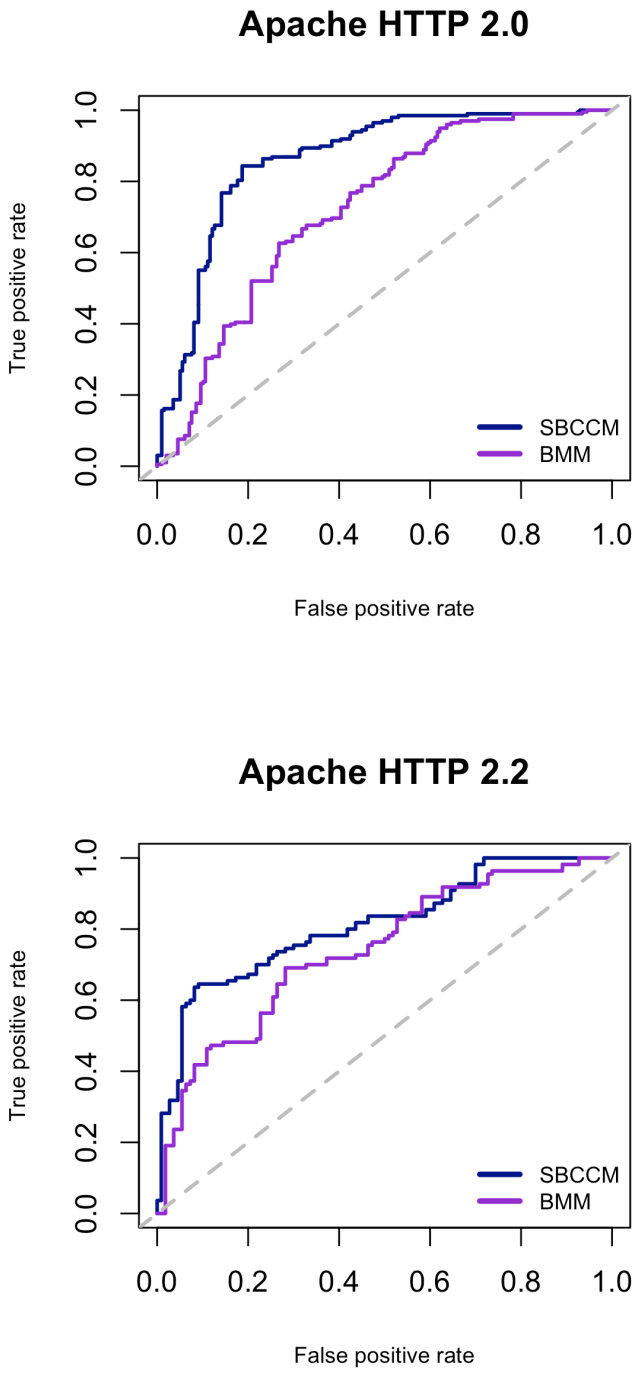


Figure 7.9: ROC curves comparing the models of SBCCM and BMM in Appache HTTP 2.0 and 2.2.

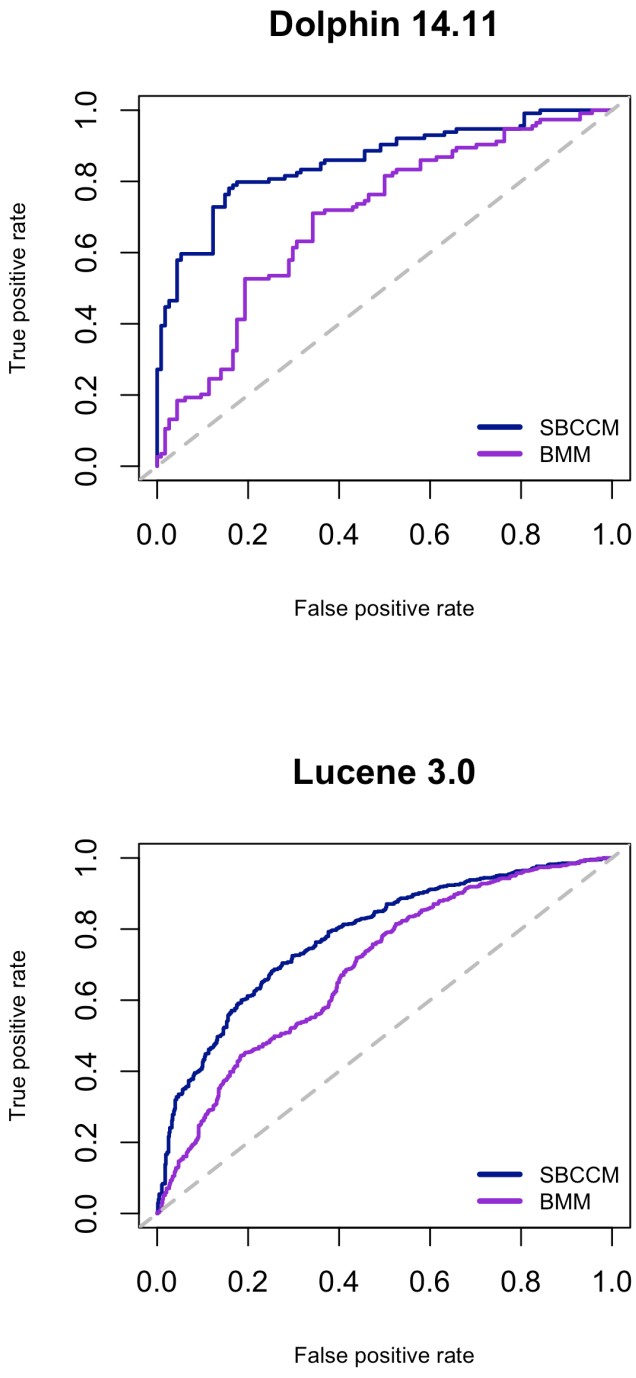


Figure 7.10: ROC curves comparing the models of SBCCM and BMM in Dolphin 14.11 and Lucene 3.0.

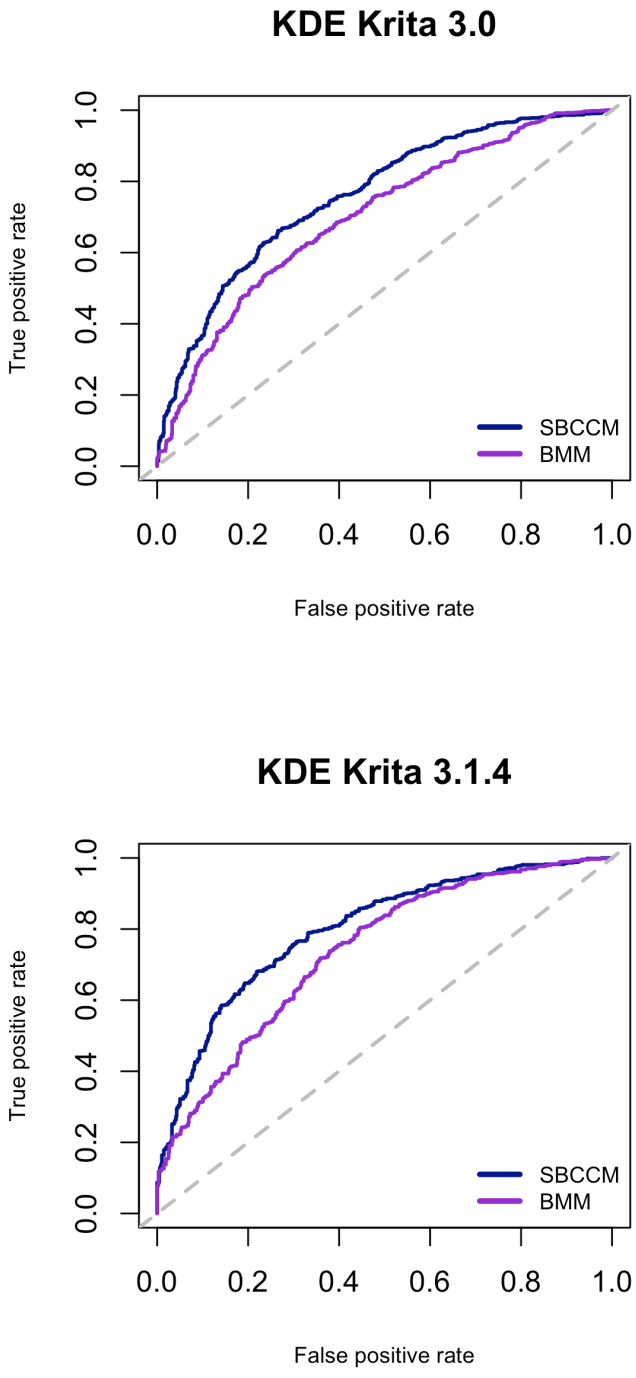


Figure 7.11: ROC curves comparing the models of SBCCM and BMM in KDE Krita 3.0 and 3.1.4.

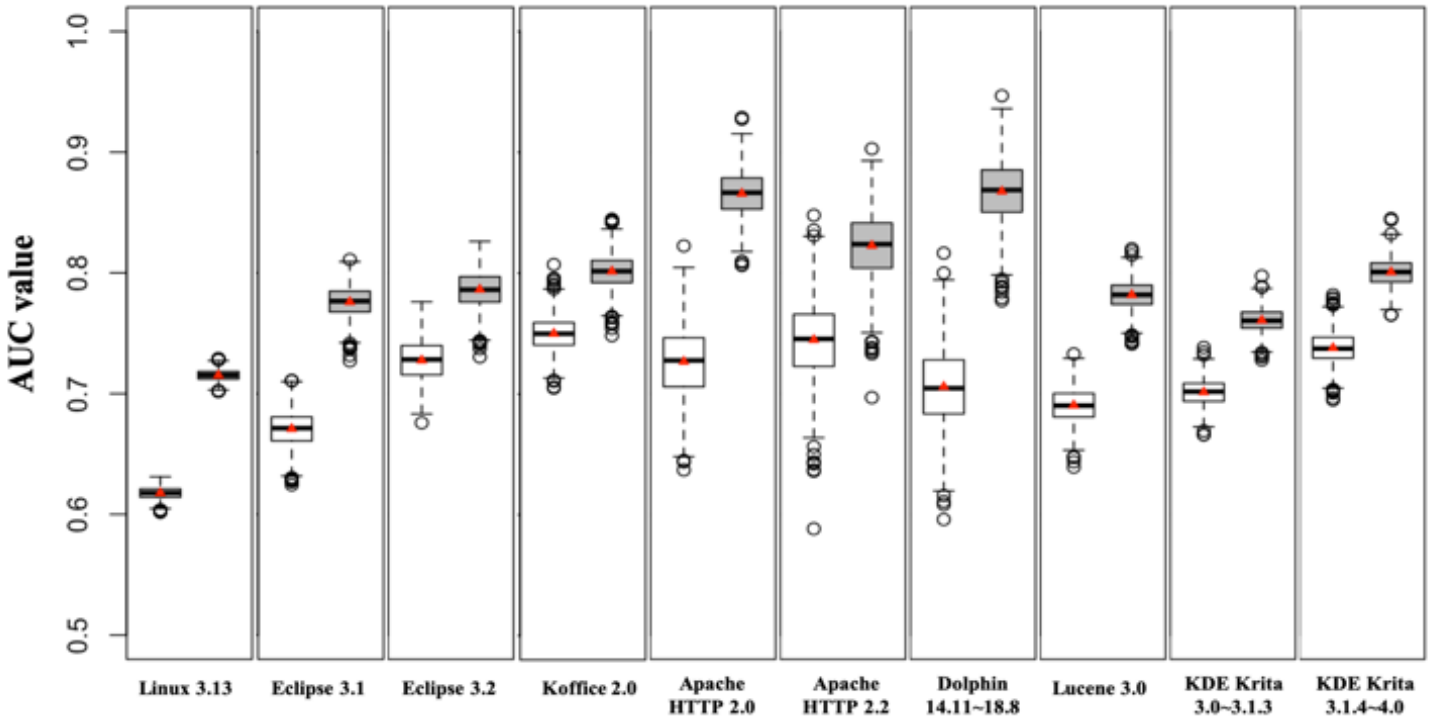


Figure 7.12: Logistic regression models AUC distribution of the 1000 out of sample bootstrap (Gray boxplots are SBCCM models, white boxplots are BMM and red triangles indicate mean values).

7.2.3 Models Prediction Power

In order to test the logistic regression model's prediction ability, we compute precision, recall and F-measure using the out-of-sample bootstrap validation (1,000 times) and report them in Table 7.2.3. The comparison between SBCCM and BMM shows that SBCCM is having higher F-measure in all 10 datasets. SBCCM achieves an improvement in F-measure up to 14% and not less than 6% (average 9%) over the BMM. Figure 7.13 shows the F-measure values distribution of the 1000 iterations of out-of-sample bootstrap validation for all datasets of SBCCM and BMM. In all systems, SBCCM has a larger median and mean than the BMM. In addition, no sharable IQR across systems.

To measure the significance of the differences, we use a Wilcoxon signed-rank test. The results reveal that the differences are significant ($p < 0.001$) across all datasets. The higher F-measure values for SBCCM include an increase in both recall (average 12%, range 6%-17%) and precision (average 6%, range 2%-13%) in all systems.

In summary, the addition of slice based cognitive complexity metrics significantly improves AUC and Nag. R^2 measures across all systems. Across all systems, slice-based cognitive complexity features significantly improve defect classification F1, recall and precision.

Subject	BMM			SBCCM		
	Recall	Precision	F-1	Recall (*)	Precision (*)	F-1(*)
Linux 3.13	48	60	54	65 **(+17%)**	65 **(+5%)**	65 **(+11%)**
Eclipse 3.1	54	67	60	65 **(+11%)**	69 **(+2%)**	67 **(+7%)**
Eclipse 3.2	61	67	64	67 **(+6%)**	73 **(+6%)**	70 **(+6%)**
Koffice 2.0	61	68	64	69 **(+8%)**	72 **(+4%)**	70 **(+6%)**
Apache HTTP 2.0	68	65	66	81 **(+13%)**	78 **(+13%)**	79 **(+13%)**
Apache HTTP 2.2	61	67	63	71 **(+10%)**	70 **(+3%)**	70 **(+7%)**
Dolphin 14.11~18.8	61	64	62	75 **(+14%)**	77 **(+13%)**	76 **(+14%)**
Lucene 3.0	58	62	60	71 **(+13%)**	70 **(+8%)**	71 **(+11%)**
KDE Krita 3.0~3.1.3	58	67	62	66 **(+8%)**	70 **(+3%)**	68 **(+6%)**
KDE Krita 3.1.4~4.0	55	68	61	70 **(+15%)**	73 **(+5%)**	72 **(+11%)**
Average	58	66	62	70 **(+12%)**	72 **(+6%)**	71 **(+9%)**

* The values between ( ) represent the improvement between BMM and SBCCMs

Table 7.2.3: Logistic regression models recall, precision and F1 values across systems using 1000 bootstrap validation (Bold font highlights the best performance).

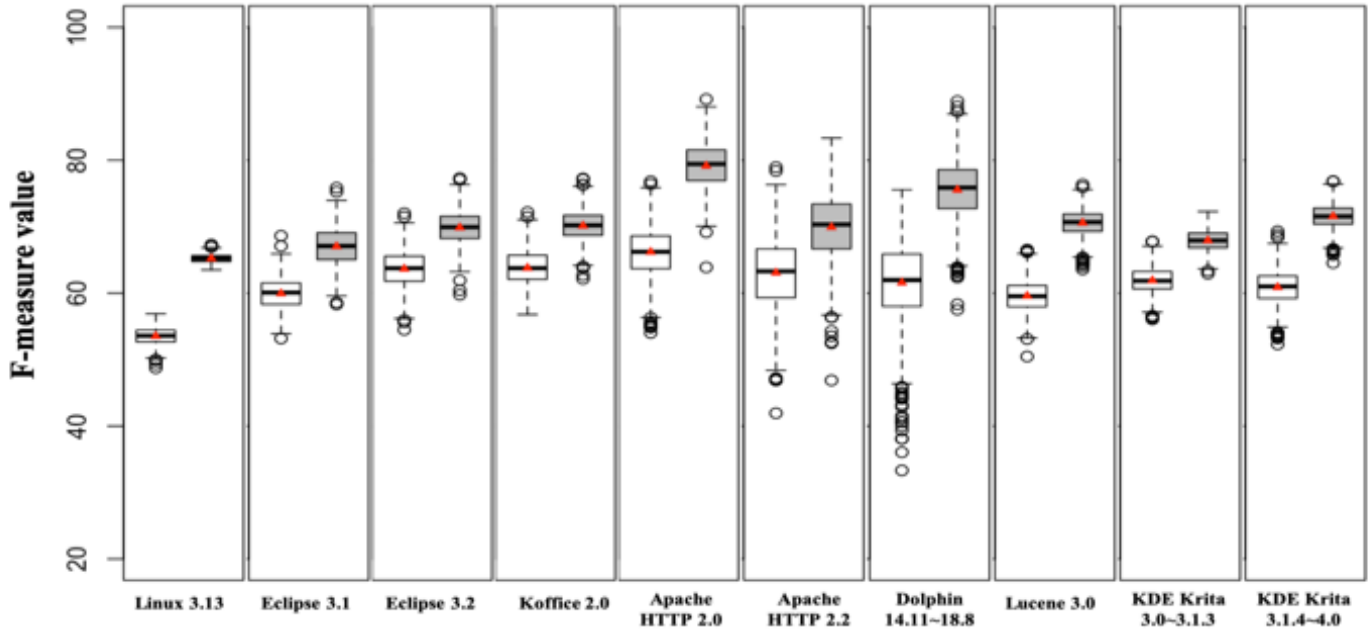


Figure 7.13: Logistic regression models F-measure distribution of the 1000 out of sample bootstrap (Gray boxplots are SBCCM models, white boxplots are BMM and red triangles indicate mean values).

7.3 Applying the Cognitive Complexity Measures During Software Inspections

For very large software systems, inspecting and comprehending all code and its dependencies to identify possible defects is an expensive, time consuming, and often unrealistic approach. An as-needed reading approach has to be adopted to deal with the possibly large amounts of de-localized information. The aim of any inspection approach has to be to limit the number of dependencies that have to be analyzed and comprehended.

Through the combination of slicing with cognitive complexity metrics, the inspection process can be further enhanced by focusing a developer's attention on only the parts that are relevant with respect to a particular function/variable. The criteria for choosing a function or variable for slicing include, for example, a function with high slice identifier, or a variable whose computation involves code with a high slice spatial. One might choose to inspect a code slice either prior to performing maintenance or as part of code verification prior to the release of a product.

Additionally, identifying the most difficult to comprehend program elements can be a valuable aid in deciding schedules and choosing appropriate programmers for a project. It is not always feasible to improve on the cognitive complexity of a difficult program element, as some domains are inherently complex. In such situations, we should allocate enough time for individuals to understand the material with minimal error. Additionally, programmer experience plays a larger role in such situations and the programmer's familiarity with the application domain and type of implementation should be considered when setting time constraints.

Moreover, program slicing allows for fine-grained cognitive complexity driven inspection of the source code, by focusing and prioritizing the preventive maintenance activity on these fine-grained parts of the system that might require additional comprehension and maintenance effort during future system maintenance.

Testing of large software systems and their executions is difficult and a time-consuming task. A possible application is that we can apply slice-based metrics and defect prediction results to prioritize or select test cases. In regression testing, executing all test suites for regression testing is very costly so that many prioritization and selection approaches for test cases have been proposed (Yoo and Harman 2012). Since defect prediction results provide bug-prone software artifacts and their ranks, it might be possible to use the results for test case prioritization and selection.

CHAPTER 8

THREATS TO VALIDITY

Like any empirical study design, experimental design choices may impact the results of our study. However, we perform a highly controlled experiment to ensure that our results are robust. Below, we discuss threats that may impact the results of our study.

8.1 Construct Validity

We use an automated bug linking process, which may introduce false positives in the linked set. This may arise because undetected defects in the considered interval are labeled clean, as defective commits are detected and fixed in 100-300days (Kim and Whitehead 2006; Tan et al. 2015). To overcome such issue, we cover a large span of time period an average of 20 months across systems.

The conclusions are based on a rule-of thumb EPV value that is suggested by (Peduzzi et al. 1996) and (Tantithamthavorn et al. 2017), who argue that, EPV has a significant influence on the performance of defect classifiers. In particular, defect classifiers trained with datasets with a low EPV value yield unstable results (Tantithamthavorn et al. 2017; 2016). To ensure the stability of our results, we ensure that included datasets have an EPV value that is larger than 10. Particularly, the systems we select have EPV ranging from 14 to 718.

The slicing process of *srcSlice* is performed using the *srcML* (Collard, Decker, and Maletic 2011; Collard, Maletic, and Robinson 2010) format for source code which provides

direct access to abstract syntactic information. While this approach is inter procedural and highly scalable, it might not match the accuracy of generating a complete PDG/SDG. However, a previous study (Alomari et al. 2014) compared *srcSlice's* accuracy with a heavyweight slicing tool and shows that *srcSlice* produces reliable accuracy given its speed and lightweight approach.

Previous research (Arisholm and Briand 2006; Bettenburg and Hassan 2013; Bland and Altman 1996) suggested mitigating the skewness distribution of defect datasets is necessary. Indeed, Jiang et al. (Jiang, Cukic, and Menzies 2008) point out that log transformation rarely affects the performance of defect prediction models. Thus, we suspect that the use of log transformation poses a threat to the validity of our conclusions. However, applying other choices of data transformation techniques may yield different results.

8.2 External Validity

We studied 10 datasets that represent varying application domains and with different characteristics (defect rates, size, language, #files, etc.) to make our dataset general and representative. However, it is unclear how well they generalize to closed source software, which may have different behavior. Our approach only requires software metrics that can be computed in a standard way by publicly available tools and all our data will be made publicly available. Replication using closed source systems may prove fruitful.

8.3 Internal Validity

We validate our model stability using out-of-sample bootstrap which been recently shown to provide the least bias and most stable performance estimates across measures in defect prediction (Tantithamthavorn et al. 2017). While 100 repetition found to be sufficient (Tantithamthavorn et al. 2017), we repeated the experiment 1,000 times to ensure that the results converge, and found consistent result.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

9.1 Conclusions

We empirically examine the usefulness of cognitive complexity slice-based metrics in the context of defect prediction. The findings from an evaluation of 10 datasets covers parts of the version histories of open source systems show that 94% of the investigated metrics are statistically significant in relation to defects. Cognitive complexity metrics have significant impact on defect classification measured by AUC, R^2, F1, recall and precision.

Slice-based metrics allows for fine-grained cognitive complexity driven inspection of the source code, by focusing and prioritizing the preventive maintenance activity on these parts of the code that require additional comprehension and maintenance effort during future maintenance. The approach can also be practically applied, as the slicing approach used is scalable to large system. Running it on the largest system (Linux) takes less than 10 minutes on a typical desktop machine.

Future effort will be devoted to investigating the metrics performance in a cross-project prediction and to replicate the analyses on closed source software which might exhibit different behaviors. Furthermore, we will provide a replication package, which includes data for both the defects and metrics used for our experiment, to allow other researchers to compare our results.

9.2 Future Work

There are several directions for the future of this research. Some directions involve the following:

9.2.1 Cross-Project Prediction

Most of prediction models are trained on historical data. Since most new projects don't have historical data, there is interest in cross-project prediction: using data from one project to predict defects in another. However, most experiments in cross-project defect prediction report poor performance, using the standard measures of precision, recall and F-score. It's been argued that these IR-based measures, while broadly applicable, are not as well suited for the QA settings in which defect prediction models are used. These measures are taken at specific threshold. However, in practice, QA processes choose from a range of time-and-cost vs quality tradeoffs: how many files should we inspect? Thus, measures based on a variety of tradeoffs, viz., 5%, 10% or 20% of files tested/inspected would be more suitable (Rahman, Posnett, and Devanbu 2012). A model that works well upon inspecting 80% of SLOC, may not work as well when inspecting only 20% of SLOC.

Therefore, we want to investigate cross-project defect prediction using slice-based metrics from this perspective. Since slice-based metrics are of a finer granularity and can capture more detailed view of the modules, we hypothesize that cross-project prediction performance is no worse than within-project performance, and substantially better than state of the art cross-project prediction models.

9.2.2 Churn of Slice-Based Metrics

Using *churn of slice-based metrics* can have some potential to predict defects. These measure code churn as deltas of slice-based metrics instead of line-based code churn. The intuition is that higher-level metrics may better model code churn than simple metrics like addition and deletion of lines of code. We sample the history of the source code and compute the deltas of slice metrics for each consecutive pair of samples. For each slice-based metric, we create a matrix where the rows are the files, the columns are the sampled versions, and each cell is the value of the metric for the given file at the given version.

9.2.3 Enhance Reliability

Other directions involve building *Module-order models* for targeting reliability enhancement. The goal is to target reliability enhancement activities to those modules that are most likely to have defects. Previous research including our classification models have focused on classification models to identify *defect-prone* and *not defect-prone* modules. Such models require that *defect-prone* be defined as a class before modeling, usually via a threshold on the number of defects. However, due to resource constraints that limit the amount of reliability enhancement effort, software development managers often cannot choose an appropriate threshold at the time of modeling. In such cases, with a predicted rank-order in hand, one can select as many modules from the top of the list for enhancement for as long as resources allow (Khoshgoftaar and Allen 2003).

9.2.4 Varimax Transformation

Another idea is to use varimax transformation as a post processing technique for PCA. In this work, we use PCA to best account for multicollinearity. Software metrics can

be highly correlated to each other (Rajbahadur et al. 2017) and highly correlated metrics (i.e., $|\rho| > 0.7$) can lead to an inflated variance in the estimation of the outcome.

Ortiz et al. apply varimax transformation for PCA (VPCA) in paleoclimate and remote sensing studies to address multicollinearity and show improvement in terms of R^2 from 0.7 to over 0.9 (Ortiz et al. 2019; Avouris and Ortiz 2019; Judice et al. 2020). In their work, the values that they use to conduct the VPCA are the derivative of a reflectance spectra as a function of wavelength, so the variables are wavelength bands. The PCA is based on the decomposition of the correlation matrix from the derivative transform of the reflectance values as a function of wavelength. In analogy to this work, these wavelengths would be the slice-based model variables or the baseline model variables. Then, we will run the logistic regression modeling against known bugs in each of the systems and compare these results against PCA results shown in APPENDIX B .

APPENDIX A

SCATTERPLOTS FOR THE RELATIONSHIPS BETWEEN SLICE-BASED METRICS AND DEFECT COUNTS

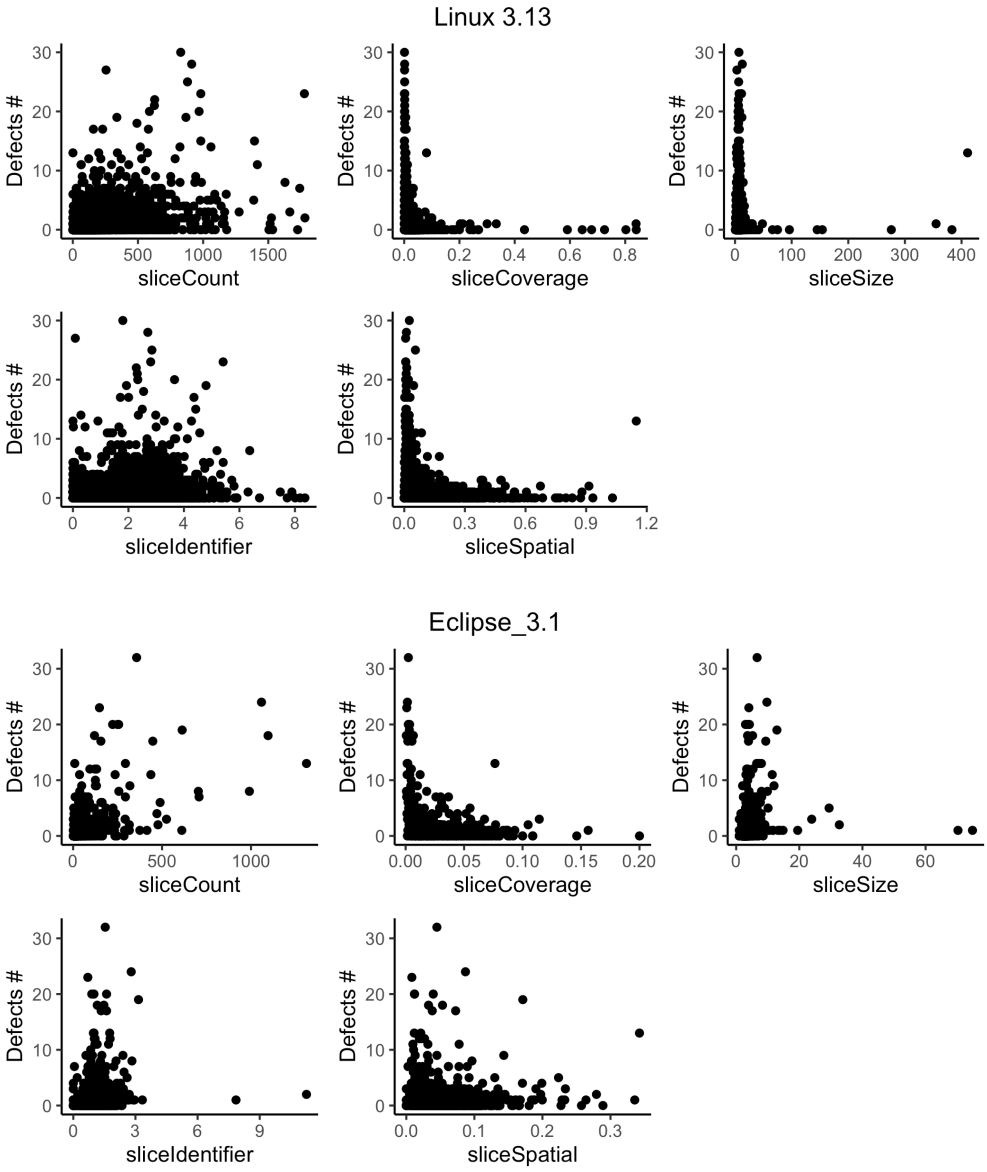


Figure 9.1: Scatterplots for the relationships between slice-based metrics and defect counts in Linux 3.13 and Eclipse 3.1.

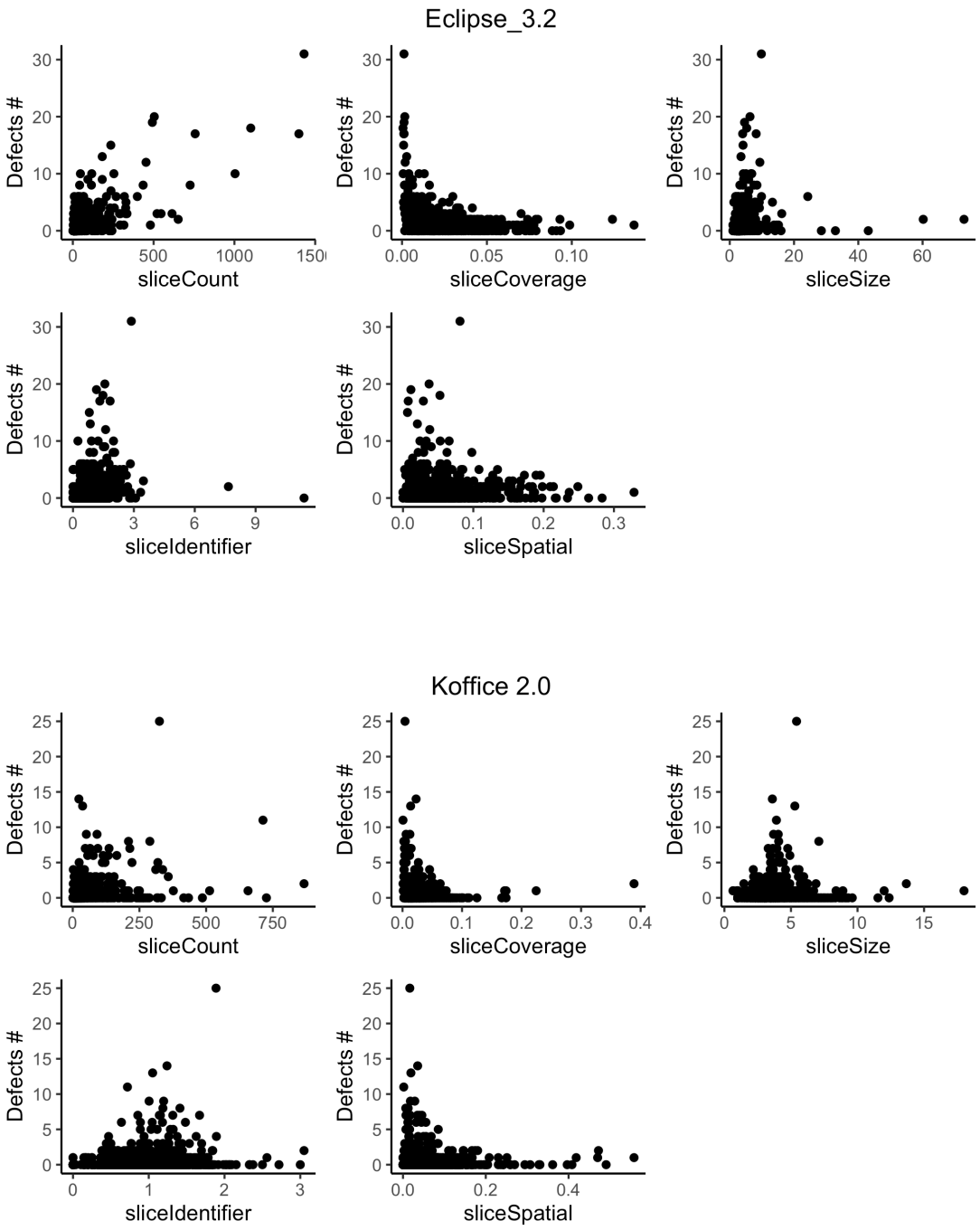


Figure 9.2: Scatterplots for the relationships between slice-based metrics and defect counts in Eclipse 3.2 and Koffice 2.0.

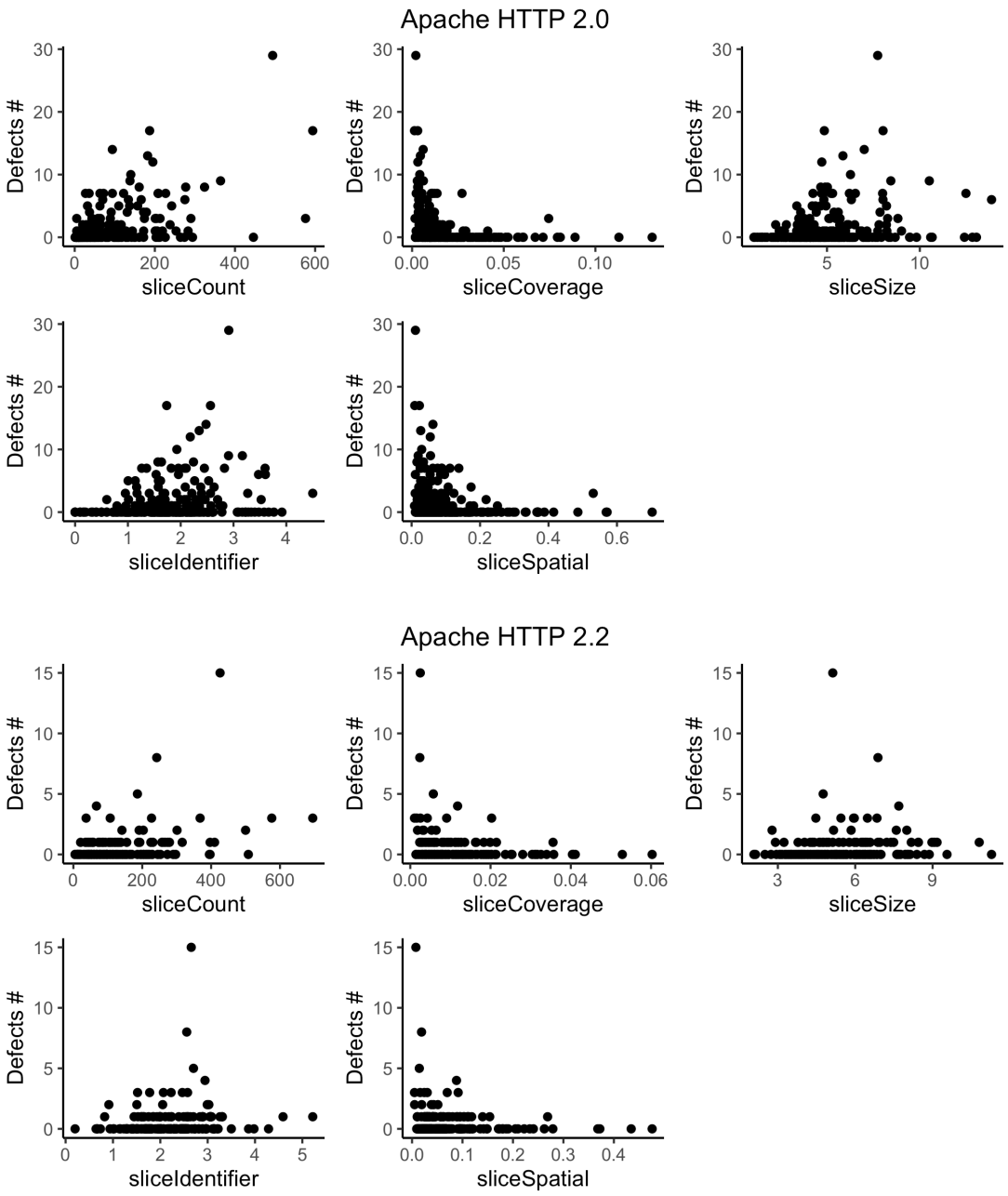


Figure 9.3: Scatterplots for the relationships between slice-based metrics and defect counts in Apache HTTP 2.0 and 2.2.

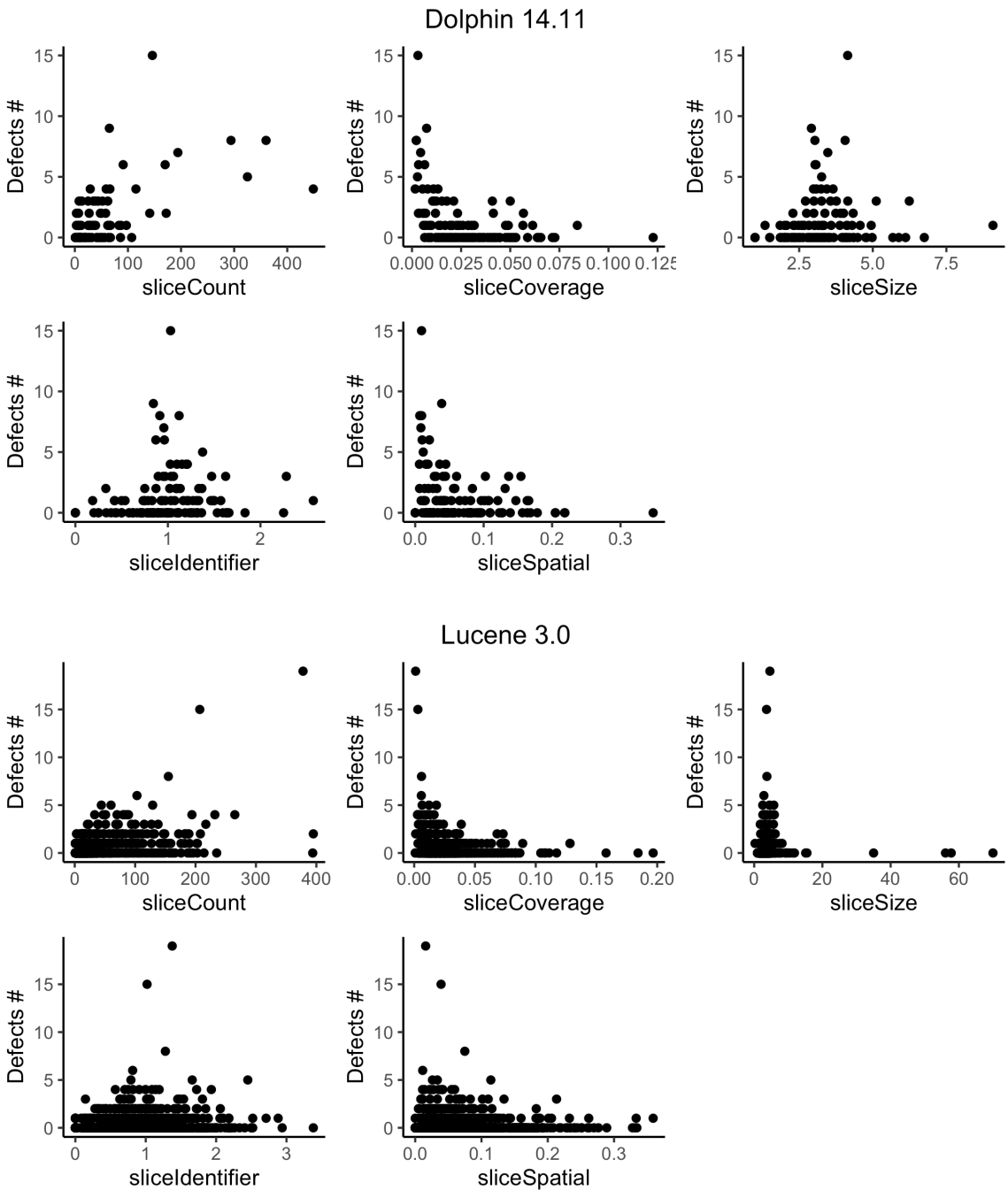


Figure 9.4: Scatterplots for the relationships between slice-based metrics and defect counts in Dolphin 14.11 and Lucene 3.0.

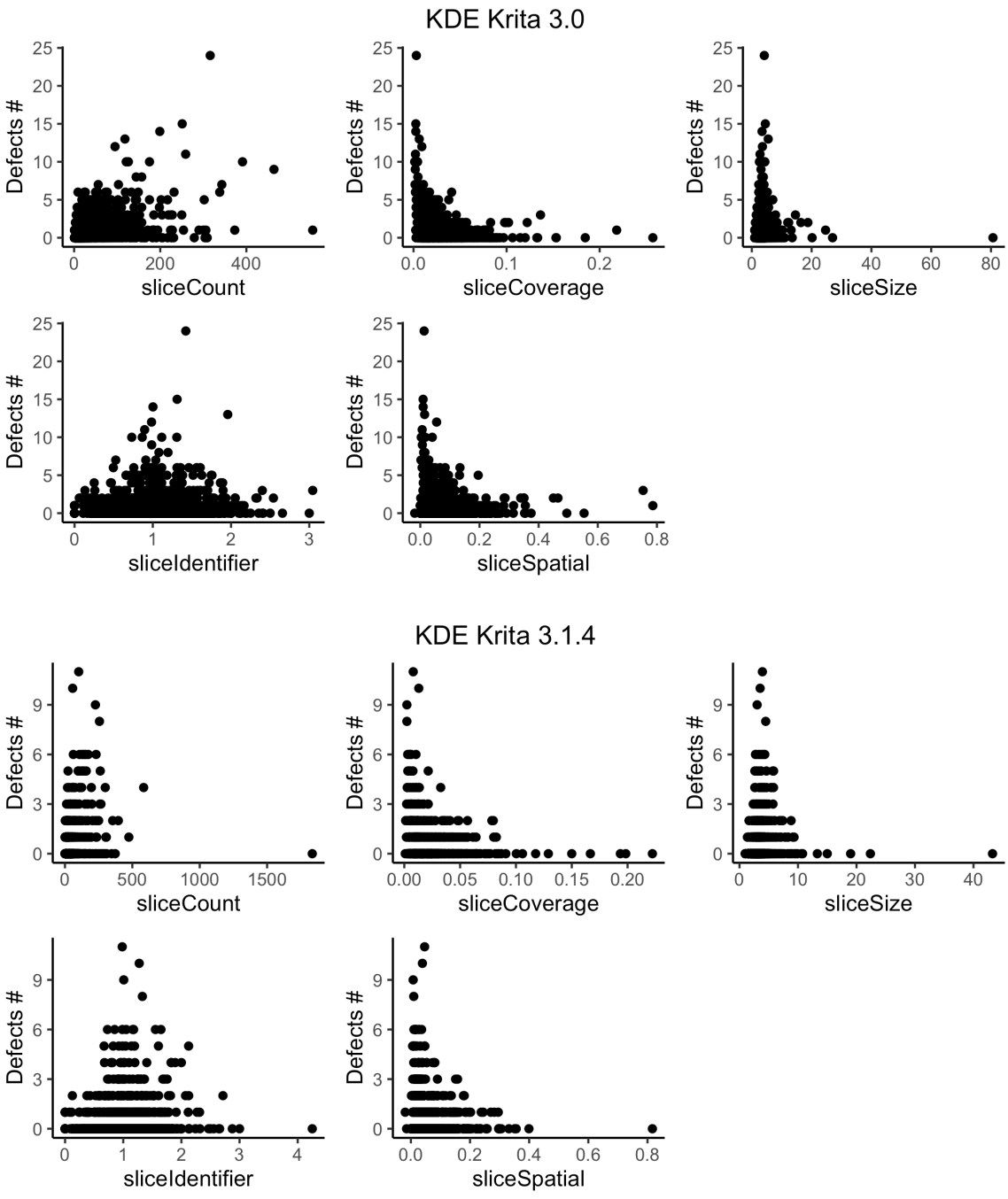


Figure 9.5: : Scatterplots for the relationships between slice-based metrics and defect counts in KDE Krita 3.0 and 3.1.4.

APPENDIX B

Results of the principal component analysis (PCA)

Model	Aspect	Metric	PC1	PC2	PC3	PC4	PC5
SBCCM	**Loadings**	sliceCount	0.52	-0.36	0.18	-0.12	0.74
		sliceCoverage	-0.32	0.59	-0.33	-0.39	0.53
		sliceSize	0.54	0.27	0.12	-0.68	-0.39
		sliceIdentifier	0.54	0.20	-0.71	0.41	-0.04
		sliceSpatial	0.20	0.64	0.58	0.45	0.10
	Communalities	sliceCount	0.66	0.22	0.01	0.00	0.10
		sliceCoverage	0.25	0.61	0.04	0.05	0.05
		sliceSize	0.70	0.12	0.01	0.14	0.03
		sliceIdentifier	0.69	0.07	0.19	0.05	0.00
		sliceSpatial	0.09	0.72	0.13	0.06	0.00
	Contribution to the PCs	sliceCount	27.51	12.74	3.216	1.386	55.14
		sliceCoverage	10.54	34.85	11.14	15.17	28.3
		sliceSize	29.36	7.066	1.373	46.85	15.35
		sliceIdentifier	28.74	4.134	50.3	16.62	0.202
		sliceSpatial	3.845	41.21	33.97	19.98	1.005
	EigenValue		2.40	1.74	0.38	0.30	0.19
	Proportion of Variance %		0.48	0.35	0.08	0.06	0.04
	Cumulative Proportion %		0.48	0.84	0.91	0.97	1.00
BMM	**Loadings**	NLOC	0.50	0.29	-0.04	0.29	-0.76
		CCN	0.49	0.29	0.08	-0.81	0.12
		Program Length	0.49	0.32	-0.05	0.50	0.64
		lineChange	0.37	-0.60	0.70	0.08	0.00
		funcChange	0.37	-0.61	-0.70	-0.07	0.02
	Communalities	NLOC	0.87	0.10	0.00	0.01	0.01
		CCN	0.85	0.10	0.00	0.05	0.00
		Program Length	0.85	0.12	0.00	0.02	0.01
		lineChange	0.49	0.44	0.08	0.00	0.00
		funcChange	0.48	0.45	0.08	0.00	0.00
	Contribution to the PCs	NLOC	24.77	8.62	0.19	8.48	57.93
		CCN	23.97	8.46	0.72	65.46	1.39
		Program Length	23.98	10.23	0.24	24.91	40.63
		lineChange	13.81	35.80	49.68	0.70	0.00
		funcChange	13.47	36.88	49.17	0.44	0.04
	EigenValue		3.53	1.22	0.15	0.08	0.02
	Proportion of Variance %		0.71	0.24	0.03	0.200	0.005
	Cumulative Proportion %		0.71	0.95	0.98	0.995	100.00

Table 9.2.1: PCA aspects of Eclipse 3.1.

Model	Aspect	Metric	PC1	PC2	PC3	PC4	PC5
SBCCM	**Loadings**	sliceCount	0.53	-0.35	0.07	0.04	0.77
		sliceCoverage	-0.34	0.58	-0.22	-0.46	0.54
		sliceSize	0.53	0.28	0.50	-0.57	-0.26
		sliceIdentifier	0.54	0.22	-0.79	0.01	-0.21
		sliceSpatial	0.18	0.65	0.29	0.67	0.11
	Communalities	sliceCount	0.68	0.22	0.00	0.00	0.10
		sliceCoverage	0.27	0.60	0.02	0.06	0.05
		sliceSize	0.67	0.14	0.09	0.10	0.01
		sliceIdentifier	0.69	0.09	0.21	0.00	0.01
		sliceSpatial	0.07	0.76	0.03	0.13	0.00
	Contribution to the PCs	sliceCount	28.54	12.16	0.472	0.188	58.65
		sliceCoverage	11.24	33.31	4.754	21.28	29.42
		sliceSize	28.1	7.65	24.67	33.04	6.545
		sliceIdentifier	29.01	4.867	61.88	0.021	4.224
		sliceSpatial	3.113	42.02	8.23	45.47	1.17
	EigenValue		2.38	1.82	0.35	0.29	0.17
	Proportion of Variance %		0.48	0.36	0.07	0.06	0.03
	Cumulative Proportion %		0.48	0.84	0.91	0.97	1.00
BMM	**Loadings**	NLOC	0.55	-0.20	0.04	-0.23	0.78
		CCN	0.54	-0.19	-0.02	0.80	-0.18
		Program Length	0.54	-0.22	0.02	-0.55	-0.60
		lineChange	0.26	0.65	-0.71	-0.04	0.01
		funcChange	0.23	0.67	0.70	0.01	-0.03
	Communalities	NLOC	0.91	0.07	0.00	0.00	0.01
		CCN	0.89	0.06	0.00	0.05	0.00
		Program Length	0.88	0.08	0.00	0.02	0.01
		lineChange	0.21	0.71	0.07	0.00	0.00
		funcChange	0.17	0.76	0.07	0.00	0.00
	Contribution to the PCs	NLOC	29.87	3.92	0.17	5.46	60.59
		CCN	29.00	3.46	0.05	64.12	3.38
		Program Length	28.86	4.89	0.05	30.22	35.97
		lineChange	6.87	42.47	50.47	0.19	0.00
		funcChange	5.40	45.25	49.26	0.02	0.06
	EigenValue		3.06	1.68	0.15	0.08	0.02
	Proportion of Variance %		0.61	0.34	0.03	0.016	0.005
	Cumulative Proportion %		0.61	0.95	0.98	0.995	1.00

Table 9.2.2: PCA aspects of Eclipse 3.2.

Model	Aspect	Metric	PC1	PC2	PC3	PC4	PC5
SBCCM	**Loadings**	sliceCount	0.59	-0.10	0.28	-0.63	0.40
		sliceCoverage	-0.34	0.52	0.73	0.09	0.26
		sliceSize	0.43	0.50	0.12	-0.08	-0.73
		sliceIdentifier	0.53	0.33	-0.22	0.62	0.43
		sliceSpatial	-0.25	0.60	-0.57	-0.45	0.22
	Communalities	sliceCount	0.83	0.02	0.03	0.10	0.03
		sliceCoverage	0.28	0.50	0.21	0.00	0.01
		sliceSize	0.44	0.46	0.01	0.00	0.09
		sliceIdentifier	0.66	0.20	0.02	0.10	0.03
		sliceSpatial	0.15	0.67	0.13	0.05	0.01
	Contribution to the PCs	sliceCount	35.06	1.03	7.97	39.88	16.06
		sliceCoverage	11.74	27.27	53.24	0.78	6.97
		sliceSize	18.85	24.97	1.50	0.72	53.97
		sliceIdentifier	27.98	10.74	4.67	38.26	18.35
		sliceSpatial	6.36	35.99	32.63	20.36	4.66
	EigenValue		2.35	1.85	0.39	0.25	0.16
	Proportion of Variance %		0.47	0.37	0.08	0.05	0.03
	Cumulative Proportion %		0.47	0.84	0.92	0.97	1.00
BMM	**Loadings**	NLOC	0.59	-0.03	-0.30	-0.02	-0.75
		CCN	0.56	-0.04	0.80	0.16	0.12
		Program Length	0.58	-0.04	-0.48	-0.10	0.65
		lineChange	0.06	0.70	0.13	-0.70	-0.02
		funcChange	0.03	0.71	-0.13	0.69	0.03
	Communalities	NLOC	0.97	0.00	0.01	0.00	0.01
		CCN	0.89	0.00	0.10	0.00	0.00
		Program Length	0.95	0.00	0.04	0.00	0.01
		lineChange	0.01	0.91	0.00	0.07	0.00
		funcChange	0.00	0.92	0.00	0.07	0.00
	Contribution to the PCs	NLOC	34.48	0.08	8.78	0.03	56.62
		CCN	31.48	0.12	64.45	2.53	1.42
		Program Length	33.62	0.18	23.33	1.03	41.83
		lineChange	0.33	49.61	1.71	48.32	0.03
		funcChange	0.09	50.00	1.73	48.08	0.10
	EigenValue		2.83	1.84	0.16	0.15	0.02
	Proportion of Variance %		0.56	0.37	0.03	0.03	0.01
	Cumulative Proportion %		0.56	0.93	0.96	0.99	1.00

Table 9.2.3: PCA aspects of Linux 3.13.

Model	Aspect	Metric	PC1	PC2	PC3	PC4	PC5
SBCCM	Loadings	sliceCount	-0.62	0.17	-0.33	-0.17	0.67
		sliceCoverage	0.39	-0.52	0.01	0.45	0.61
		sliceSize	-0.46	-0.43	-0.46	0.46	-0.42
		sliceIdentifier	-0.50	-0.33	0.80	0.01	0.02
		sliceSpatial	0.09	-0.63	-0.19	-0.74	-0.05
	Communalities	sliceCount	0.83	0.06	0.04	0.01	0.07
		sliceCoverage	0.33	0.57	0.00	0.05	0.06
		sliceSize	0.45	0.39	0.08	0.05	0.03
		sliceIdentifier	0.54	0.22	0.23	0.00	0.00
		sliceSpatial	0.02	0.84	0.01	0.13	0.00
	Contribution to the PCs	sliceCount	38.18	2.87	10.90	2.97	45.07
		sliceCoverage	15.07	27.36	0.01	20.36	37.20
		sliceSize	20.85	18.90	21.23	21.54	17.48
		sliceIdentifier	25.15	10.67	64.13	0.01	0.03
		sliceSpatial	0.74	40.20	3.73	55.12	0.21
	EigenValue		2.16	2.08	0.37	0.24	0.15
	Proportion of Variance %		0.43	0.42	0.07	0.05	0.03
	Cumulative Proportion %		0.43	0.85	0.92	0.97	1.00
BMM	Loadings	NLOC	-0.55	0.24	-0.29	0.01	-0.74
		CCN	-0.47	0.27	0.83	0.01	0.11
		Program Length	-0.54	0.23	-0.47	-0.03	0.66
		lineChange	-0.30	-0.64	0.05	-0.70	-0.02
		funcChange	-0.30	-0.64	0.02	0.71	0.03
	Communalities	NLOC	0.86	0.09	0.03	0.00	0.02
		CCN	0.63	0.12	0.25	0.00	0.00
		Program Length	0.82	0.09	0.08	0.00	0.01
		lineChange	0.25	0.69	0.00	0.07	0.00
		funcChange	0.26	0.67	0.00	0.07	0.00
	Contribution to the PCs	NLOC	30.53	5.68	8.49	0.01	55.29
		CCN	22.33	7.24	69.17	0.00	1.25
		Program Length	29.06	5.40	22.06	0.10	43.38
		lineChange	8.78	41.28	0.22	49.70	0.02
		funcChange	9.30	40.40	0.05	50.19	0.06
	EigenValue		2.82	1.66	0.36	0.13	0.03
	Proportion of Variance %		0.56	0.33	0.07	0.03	0.01
	Cumulative Proportion %		0.56	0.89	0.96	0.99	1.00

Table 9.2.4: PCA aspects of Koffice 2.0.

Model	Aspect	Metric	PC1	PC2	PC3	PC4	PC5
SBCCM	**Loadings**	sliceCount	0.59	-0.07	0.30	-0.29	0.69
		sliceCoverage	-0.52	0.34	-0.47	-0.22	0.59
		sliceSize	0.34	0.57	-0.19	-0.60	-0.40
		sliceIdentifier	0.41	0.49	-0.28	0.71	0.12
		sliceSpatial	-0.32	0.56	0.76	0.09	0.03
	Communalities	sliceCount	0.88	0.01	0.03	0.02	0.07
		sliceCoverage	0.67	0.21	0.06	0.01	0.05
		sliceSize	0.29	0.59	0.01	0.08	0.02
		sliceIdentifier	0.42	0.45	0.02	0.11	0.00
		sliceSpatial	0.25	0.58	0.16	0.00	0.00
	Contribution to the PCs	sliceCount	35.03	0.49	9.25	8.28	46.95
		sliceCoverage	26.66	11.31	21.80	4.98	35.25
		sliceSize	11.56	32.33	3.68	36.17	16.26
		sliceIdentifier	16.60	24.25	7.92	49.80	1.43
		sliceSpatial	10.14	31.62	57.35	0.77	0.11
	EigenValue		2.51	1.84	0.28	0.23	0.14
	Proportion of Variance %		0.50	0.37	0.06	0.05	0.03
	Cumulative Proportion %		0.50	0.87	0.93	0.98	1.00
BMM	**Loadings**	NLOC	0.55	-0.19	0.01	-0.27	0.77
		CCN	0.54	-0.18	0.05	0.81	-0.15
		Program Length	0.55	-0.18	0.03	-0.53	-0.62
		lineChange	0.20	0.69	0.70	-0.02	0.01
		funcChange	0.26	0.65	-0.71	0.01	-0.01
	Communalities	NLOC	0.92	0.06	0.00	0.01	0.01
		CCN	0.89	0.06	0.00	0.05	0.00
		Program Length	0.91	0.06	0.00	0.02	0.01
		lineChange	0.12	0.83	0.05	0.00	0.00
		funcChange	0.20	0.75	0.05	0.00	0.00
	Contribution to the PCs	NLOC	30.28	3.59	0.01	7.32	58.81
		CCN	29.21	3.34	0.22	65.01	2.22
		Program Length	29.96	3.40	0.08	27.63	38.94
		lineChange	3.93	47.22	48.81	0.02	0.02
		funcChange	6.62	42.45	50.89	0.02	0.01
	EigenValue		3.04	1.76	0.10	0.08	0.02
	Proportion of Variance %		0.61	0.35	0.02	0.016	0.003
	Cumulative Proportion %		0.61	0.96	0.98	0.996	1.00

Table 9.2.5: PCA aspects of Apache HTTP 2.0.

Model	Aspect	Metric	PC1	PC2	PC3	PC4	PC5
SBCCM	**Loadings**	sliceCount	0.59	0.06	-0.28	-0.32	-0.68
		sliceCoverage	-0.59	0.09	0.23	0.26	-0.72
		sliceSize	0.15	0.66	-0.39	0.62	0.06
		sliceIdentifier	0.12	0.67	0.64	-0.35	0.07
		sliceSpatial	-0.51	0.32	-0.56	-0.57	0.08
	Communalities	sliceCount	0.88	0.01	0.02	0.03	0.07
		sliceCoverage	0.88	0.02	0.01	0.02	0.08
		sliceSize	0.06	0.80	0.04	0.10	0.00
		sliceIdentifier	0.03	0.82	0.12	0.03	0.00
		sliceSpatial	0.65	0.19	0.09	0.08	0.00
	Contribution to the PCs	sliceCount	35.19	0.41	7.78	10.38	46.23
		sliceCoverage	35.17	0.88	5.19	6.50	52.26
		sliceSize	2.31	43.92	14.84	38.58	0.35
		sliceIdentifier	1.40	44.63	41.26	12.24	0.47
		sliceSpatial	25.92	10.16	30.93	32.29	0.69
	EigenValue		2.49	1.83	0.28	0.25	0.15
	Proportion of Variance %		0.50	0.37	0.06	0.05	0.03
	Cumulative Proportion %		0.50	0.87	0.93	0.98	1.00
BMM	**Loadings**	NLOC	0.53	-0.25	0.15	-0.22	-0.77
		CCN	0.52	-0.24	-0.31	0.74	0.16
		Program Length	0.52	-0.25	0.13	-0.52	0.62
		lineChange	0.29	0.65	-0.66	-0.25	-0.06
		funcChange	0.31	0.63	0.66	0.25	0.06
	Communalities	NLOC	0.89	0.10	0.00	0.00	0.01
		CCN	0.85	0.10	0.01	0.04	0.00
		Program Length	0.87	0.10	0.00	0.02	0.01
		lineChange	0.27	0.68	0.04	0.00	0.00
		funcChange	0.30	0.65	0.04	0.00	0.00
	Contribution to the PCs	NLOC	27.83	6.02	2.20	5.03	58.92
		CCN	26.71	5.96	9.38	55.50	2.45
		Program Length	27.40	6.06	1.63	27.08	37.83
		lineChange	8.51	42.02	43.00	6.09	0.38
		funcChange	9.55	39.93	43.79	6.30	0.42
	EigenValue		3.19	1.63	0.09	0.08	0.02
	Proportion of Variance %		0.64	0.32	0.019	0.015	0.003
	Cumulative Proportion %		0.64	0.96	0.979	0.994	1.00

Table 9.2.6: PCA aspects of Apache HTTP 2.2.

Model	Aspect	Metric	PC1	PC2	PC3	PC4	PC5
SBCCM	**Loadings**	sliceCount	0.20	-0.63	0.39	0.24	0.60
		sliceCoverage	-0.49	0.42	0.11	-0.34	0.68
		sliceSize	-0.46	-0.43	0.45	-0.50	-0.39
		sliceIdentifier	-0.38	-0.49	-0.77	-0.04	0.13
		sliceSpatial	-0.60	0.09	0.18	0.76	-0.14
	Communalities	sliceCount	0.09	0.80	0.04	0.01	0.05
		sliceCoverage	0.55	0.36	0.00	0.03	0.06
		sliceSize	0.49	0.37	0.06	0.06	0.02
		sliceIdentifier	0.34	0.48	0.18	0.00	0.00
		sliceSpatial	0.85	0.01	0.01	0.13	0.00
	Contribution to the PCs	sliceCount	4.02	39.42	14.93	5.94	35.69
		sliceCoverage	23.69	17.82	1.12	11.52	45.84
		sliceSize	21.21	18.32	20.67	24.93	14.87
		sliceIdentifier	14.55	23.71	59.87	0.14	1.74
		sliceSpatial	36.52	0.72	3.42	57.47	1.86
	EigenValue		2.31	2.04	0.30	0.22	0.13
	Proportion of Variance %		0.46	0.41	0.06	0.04	0.03
	Cumulative Proportion %		0.46	0.87	0.93	0.97	1.00
BMM	**Loadings**	NLOC	0.51	0.26	-0.31	0.11	-0.75
		CCN	0.31	0.66	0.66	0.01	0.17
		Program Length	0.50	0.17	-0.57	-0.04	0.63
		lineChange	0.46	-0.44	0.25	-0.73	-0.06
		funcChange	0.43	-0.52	0.28	0.68	0.09
	Communalities	NLOC	0.87	0.08	0.03	0.00	0.01
		CCN	0.33	0.52	0.15	0.00	0.00
		Program Length	0.84	0.03	0.11	0.00	0.01
		lineChange	0.70	0.24	0.02	0.04	0.00
		funcChange	0.62	0.32	0.03	0.03	0.00
	Contribution to the PCs	NLOC	25.89	6.89	9.50	1.23	56.49
		CCN	9.68	43.74	43.61	0.01	2.96
		Program Length	25.10	2.92	32.50	0.12	39.36
		lineChange	20.95	19.67	6.39	52.69	0.31
		funcChange	18.38	26.79	8.00	45.95	0.87
	EigenValue		3.36	1.20	0.35	0.07	0.03
	Proportion of Variance %		0.67	0.24	0.070	0.01	0.01
	Cumulative Proportion %		0.67	0.91	0.980	0.99	1.00

Table 9.2.7: PCA aspects of Dolphin 14.11~18.8.

Model	Aspect	Metric	PC1	PC2	PC3	PC4	PC5
SBCCM	**Loadings**	sliceCount	-0.53	0.36	-0.28	0.05	-0.72
		sliceCoverage	0.35	-0.61	0.12	-0.33	-0.62
		sliceSize	-0.50	-0.34	-0.38	-0.63	0.31
		sliceIdentifier	-0.52	-0.11	0.84	-0.04	0.00
		sliceSpatial	-0.28	-0.62	-0.22	0.70	0.03
	Communalities	sliceCount	0.69	0.21	0.03	0.00	0.07
		sliceCoverage	0.30	0.62	0.01	0.03	0.05
		sliceSize	0.62	0.19	0.06	0.12	0.01
		sliceIdentifier	0.68	0.02	0.30	0.00	0.00
		sliceSpatial	0.19	0.64	0.02	0.14	0.00
	Contribution to the PCs	sliceCount	27.78	12.63	7.77	0.24	51.57
		sliceCoverage	11.95	36.66	1.53	10.92	38.95
		sliceSize	25.07	11.24	14.52	39.79	9.38
		sliceIdentifier	27.44	1.16	71.27	0.13	0.00
		sliceSpatial	7.75	38.32	4.91	48.92	0.10
	EigenValue		2.48	1.68	0.42	0.29	0.13
	Proportion of Variance %		0.50	0.33	0.08	0.06	0.03
	Cumulative Proportion %		0.50	0.83	0.91	0.97	1.00
BMM	**Loadings**	NLOC	0.58	-0.10	0.08	0.16	0.79
		CCN	0.55	-0.16	-0.19	-0.76	-0.25
		Program Length	0.57	-0.08	0.12	0.59	-0.56
		lineChange	0.14	0.69	-0.70	0.13	0.03
		funcChange	0.13	0.69	0.68	-0.21	-0.03
	Communalities	NLOC	0.96	0.02	0.00	0.00	0.02
		CCN	0.87	0.04	0.01	0.08	0.00
		Program Length	0.93	0.01	0.00	0.05	0.01
		lineChange	0.06	0.85	0.09	0.00	0.00
		funcChange	0.05	0.85	0.09	0.01	0.00
	Contribution to the PCs	NLOC	33.41	1.03	0.60	2.52	62.45
		CCN	30.36	2.47	3.74	57.05	6.38
		Program Length	32.44	0.63	1.49	34.46	30.98
		lineChange	1.99	47.73	48.57	1.64	0.08
		funcChange	1.80	48.14	45.61	4.34	0.11
	EigenValue		2.86	1.77	0.20	0.14	0.03
	Proportion of Variance %		0.57	0.35	0.04	0.03	0.01
	Cumulative Proportion %		0.57	0.92	0.96	0.99	1.00

Table 9.2.8: PCA aspects of Lucene 3.0.

Model	Aspect	Metric	PC1	PC2	PC3	PC4	PC5
SBCCM	**Loadings**	sliceCount	-0.32	-0.57	-0.17	-0.44	0.59
		sliceCoverage	0.59	0.27	-0.10	0.22	0.72
		sliceSize	0.39	-0.48	-0.69	0.21	-0.29
		sliceIdentifier	0.18	-0.60	0.64	0.45	0.03
		sliceSpatial	0.61	-0.08	0.27	-0.72	-0.21
	Communalities	sliceCount	0.23	0.68	0.01	0.05	0.03
		sliceCoverage	0.78	0.16	0.00	0.01	0.05
		sliceSize	0.34	0.49	0.15	0.01	0.01
		sliceIdentifier	0.07	0.75	0.13	0.05	0.00
		sliceSpatial	0.82	0.01	0.02	0.14	0.00
	Contribution to the PCs	sliceCount	10.09	32.62	2.79	19.29	35.21
		sliceCoverage	34.83	7.49	0.91	5.06	51.71
		sliceSize	15.22	23.41	48.17	4.52	8.68
		sliceIdentifier	3.11	35.84	41.09	19.87	0.08
		sliceSpatial	36.74	0.64	7.04	51.25	4.32
	EigenValue		2.23	2.08	0.32	0.27	0.10
	Proportion of Variance %		0.45	0.42	0.06	0.05	0.02
	Cumulative Proportion %		0.45	0.87	0.93	0.98	1.00
BMM	**Loadings**	NLOC	-0.53	0.29	-0.30	0.03	-0.74
		CCN	-0.44	0.29	0.84	-0.01	0.09
		Program Length	-0.52	0.29	-0.44	0.01	0.67
		lineChange	-0.35	-0.62	0.04	0.70	0.01
		funcChange	-0.36	-0.60	0.01	-0.71	0.00
	Communalities	NLOC	0.81	0.13	0.04	0.00	0.02
		CCN	0.56	0.13	0.31	0.00	0.00
		Program Length	0.77	0.13	0.09	0.00	0.01
		lineChange	0.35	0.58	0.00	0.07	0.00
		funcChange	0.38	0.55	0.00	0.07	0.00
	Contribution to the PCs	NLOC	28.34	8.53	8.91	0.06	54.16
		CCN	19.53	8.41	71.20	0.02	0.85
		Program Length	26.85	8.43	19.72	0.02	44.98
		lineChange	12.06	38.47	0.17	49.30	0.02
		funcChange	13.23	36.16	0.00	50.60	0.00
	EigenValue		2.88	1.52	0.44	0.14	0.03
	Proportion of Variance %		0.57	0.30	0.09	0.03	0.01
	Cumulative Proportion %		0.57	0.87	0.96	0.99	1.00

Table 9.2.9: PCA aspect KDE Krita 3.0~3.1.3.

Model	Aspect	Metric	PC1	PC2	PC3	PC4	PC5
SBCCM	**Loadings**	sliceCount	0.62	-0.15	0.17	0.33	-0.67
		sliceCoverage	-0.46	0.46	0.00	-0.32	-0.69
		sliceSize	0.41	0.47	0.63	-0.41	0.23
		sliceIdentifier	0.47	0.40	-0.76	-0.22	0.05
		sliceSpatial	-0.11	0.62	0.05	0.76	0.14
	Communalities	sliceCount	0.85	0.05	0.01	0.03	0.06
		sliceCoverage	0.47	0.44	0.00	0.03	0.07
		sliceSize	0.36	0.46	0.13	0.04	0.01
		sliceIdentifier	0.47	0.33	0.19	0.01	0.00
		sliceSpatial	0.03	0.81	0.00	0.15	0.00
	Contribution to the PCs	sliceCount	39.06	2.17	2.84	10.81	45.11
		sliceCoverage	21.49	20.97	0.00	10.27	47.27
		sliceSize	16.53	22.05	39.35	16.63	5.44
		sliceIdentifier	21.62	15.75	57.58	4.81	0.23
		sliceSpatial	1.29	39.06	0.22	57.48	1.96
	EigenValue		2.19	2.08	0.32	0.27	0.14
	Proportion of Variance %		0.44	0.42	0.06	0.05	0.02
	Cumulative Proportion %		0.45	0.87	0.93	0.98	1.00
BMM	**Loadings**	NLOC	-0.58	0.18	-0.30	0.04	-0.74
		CCN	-0.48	0.26	0.83	-0.09	0.10
		Program Length	-0.57	0.16	-0.45	0.08	0.67
		lineChange	-0.22	-0.67	0.15	0.69	-0.01
		funcChange	-0.26	-0.65	-0.02	-0.71	0.01
	Communalities	NLOC	0.89	0.06	0.03	0.00	0.02
		CCN	0.61	0.12	0.27	0.00	0.00
		Program Length	0.86	0.05	0.08	0.00	0.01
		lineChange	0.13	0.80	0.01	0.06	0.00
		funcChange	0.18	0.76	0.00	0.06	0.00
	Contribution to the PCs	NLOC	33.35	3.14	8.93	0.17	54.41
		CCN	22.88	6.65	68.82	0.73	0.91
		Program Length	32.23	2.57	19.88	0.66	44.65
		lineChange	4.94	44.90	2.33	47.83	0.01
		funcChange	6.60	42.73	0.04	50.62	0.02
	EigenValue		2.67	1.79	0.39	0.12	0.03
	Proportion of Variance %		0.53	0.36	0.08	0.02	0.01
	Cumulative Proportion %		0.53	0.89	0.97	0.99	1.00

Table 9.2.10: PCA aspects of KDE Krita 3.1.4~4.0.

REFERENCES

Abebe, S. L., V. Arnaoudova, P. Tonella, G. Antoniol, and Y. G. Guéhéneuc. 2012. "Can Lexicon Bad Smells Improve Fault Prediction?" In 2012 19th Working Conference on Reverse Engineering, 235–44. https://doi.org/10.1109/WCRE.2012.33.

Abid, N. J., N. Dragan, M. L. Collard, and J. I. Maletic. 2015. "Using Stereotypes in the Automatic Generation of Natural Language Summaries for C++ Methods." In 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), 561–65. https://doi.org/10.1109/ICSM.2015.7332514.

Abreu, Fernando Brito e, and Rogério Carapuça. 1994. "Candidate Metrics for Object-Oriented Software within a Taxonomy Framework." Journal of Systems and Software 26 (1): 87–96. https://doi.org/10.1016/0164-1212(94)90099-X.

Agrawal, Hiralal, Richard A. Demillo, and Eugene H. Spafford. 1993. "Debugging with Dynamic Slicing and Backtracking." Softw. Pract. Exper. 23 (6): 589–616. https://doi.org/10.1002/spe.4380230603.

Akiyama, F. 1971. "An Example of Software System Debugging." In International Federation of Information Processing Societies Congress, 353–359.

Allen, Frances E. 1970. "Control Flow Analysis." In Proceedings of a Symposium on Compiler Optimization, 1–19. New York, NY, USA: ACM. https://doi.org/10.1145/800028.808479.

Alomari, H. W., M. L. Collard, and J. I. Maletic. 2014. “A Slice-Based Estimation Approach for Maintenance Effort.” In 2014 IEEE International Conference on Software Maintenance and Evolution, 81–90. https://doi.org/10.1109/ICSME.2014.30.

Alomari, H. W., M. L. Collard, J. I. Maletic, N. Alhindawi, and O. Meqdadi. 2014. “SrcSlice: Very Efficient and Scalable Forward Static Slicing: SRCSLICE: VERY EFFICIENT AND SCALABLE FORWARD STATIC SLICING.” Journal of Software: Evolution and Process 26 (11): 931–61. https://doi.org/10.1002/smr.1651.

Alqadi, Basma. 2019. “The Relationship Between Cognitive Complexity and the Probability of Defects.” In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 600–604. https://doi.org/10.1109/ICSME.2019.00095.

Alqadi, Basma S., and Jonathan I. Maletic. 2020. “Slice-Based Cognitive Complexity Metrics for Defect Prediction.” In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 411–22. https://doi.org/10.1109/SANER48275.2020.9054836.

Androutsopoulos, Kelly, David Clark, Mark Harman, Jens Krinke, and Laurence Tratt. 2013. “State-Based Model Slicing: A Survey.” ACM Comput. Surv. 45 (4): 53:1–53:36. https://doi.org/10.1145/2501654.2501667.

Arisholm, Erik, and Lionel C. Briand. 2006. “Predicting Fault-Prone Components in a Java Legacy System.” In Proceedings of the 2006 ACM/IEEE International

Symposium on Empirical Software Engineering, 8–17. ISESE ’06. Rio de Janeiro, Brazil: Association for Computing Machinery. https://doi.org/10.1145/1159733.1159738.

Arnaoudova, V., L. Eshkevari, R. Oliveto, Y. G. Guéhéneuc, and G. Antoniol. 2010. “Physical and Conceptual Identifier Dispersion: Measures and Relation to Fault Proneness.” In 2010 IEEE International Conference on Software Maintenance, 1–5. https://doi.org/10.1109/ICSM.2010.5609748.

Avouris, Dulcinea M., and Joseph D. Ortiz. 2019. “Validation of 2015 Lake Erie MODIS Image Spectral Decomposition Using Visible Derivative Spectroscopy and Field Campaign Data.” *Journal of Great Lakes Research* 45 (3): 466–79. https://doi.org/10.1016/j.jglr.2019.02.005.

Bacchelli, Alberto, Marco D’Ambros, and Michele Lanza. 2010. “Are Popular Classes More Defect Prone?” In Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering, 59–73. FASE’10. Berlin, Heidelberg: Springer-Verlag. https://doi.org/10.1007/978-3-642-12029-9_5.

Bachmann, Adrian, and Abraham Bernstein. 2009. “Software Process Data Quality and Characteristics: A Historical View on Open and Closed Source Projects.” In Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops, 119–128. IWPSE-Evol ’09. New York, NY, USA: ACM. https://doi.org/10.1145/1595808.1595830.

Bae, Jung Ho, and Heung Seok Chae. 2008. “UMLSlicer: A Tool for Modularizing the UML Metamodel Using Slicing.” In 2008 8th IEEE International Conference on Computer and Information Technology, 772–77. https://doi.org/10.1109/CIT.2008.4594772.

Basili, V. R., L. C. Briand, and W. L. Melo. 1996. “A Validation of Object-Oriented Design Metrics as Quality Indicators.” IEEE Transactions on Software Engineering 22 (10): 751–61. https://doi.org/10.1109/32.544352.

Basili, Victor R., and Barry T. Perricone. 1984. “Software Errors and Complexity: An Empirical Investigation0.” Commun. ACM 27 (1): 42–52. https://doi.org/10.1145/69605.2085.

Bell, Robert M., Thomas J. Ostrand, and Elaine J. Weyuker. 2006. “Looking for Bugs in All the Right Places.” In Proceedings of the 2006 International Symposium on Software Testing and Analysis, 61–72. ISSTA ’06. New York, NY, USA: ACM. https://doi.org/10.1145/1146238.1146246.

Belsley, David, Edwin Kuh, and Roy E. Welsch. 2005. “Detecting Influential Observations and Outliers.” In , 6–84. https://doi.org/10.1002/0471725153.ch2.

Best, D. J., and D. E. Roberts. 1975. “Algorithm AS 89: The Upper Tail Probabilities of Spearman’s Rho.” Journal of the Royal Statistical Society. Series C (Applied Statistics) 24 (3): 377–79. https://doi.org/10.2307/2347111.

Bettenburg, Nicolas, and Ahmed E. Hassan. 2013. “Studying the Impact of Social Interactions on Software Quality.” Empirical Software Engineering 18 (2): 375–431. https://doi.org/10.1007/s10664-012-9205-0.

Bieman, J. M., and L. M. Ott. 1994. “Measuring Functional Cohesion.” IEEE Trans. Softw. Eng. 20 (8): 644–657. https://doi.org/10.1109/32.310673.

Binkley, D., H. Feild, D. Lawrie, and M. Pighin. 2007. “Software Fault Prediction Using Language Processing.” In Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007), 99–110. https://doi.org/10.1109/TAIC.PART.2007.10.

Binkley, David. 1998. “The Application of Program Slicing to Regression Testing.” Information and Software Technology 40 (11–12): 583–94. https://doi.org/10.1016/S0950-5849(98)00085-8.

Binkley, David, Henry Feild, Dawn Lawrie, and Maurizio Pighin. 2009. “Increasing Diversity: Natural Language Measures for Software Fault Prediction.” J. Syst. Softw. 82 (11): 1793–1803. https://doi.org/10.1016/j.jss.2009.06.036.

Bird, C., N. Nagappan, H. Gall, B. Murphy, and P. Devanbu. 2009. “Putting It All Together: Using Socio-Technical Networks to Predict Failures.” In 2009 20th International Symposium on Software Reliability Engineering, 109–19. https://doi.org/10.1109/ISSRE.2009.17.

Bird, Christian, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 2011. “Don’T Touch My Code!: Examining the Effects of Ownership on Software Quality.” In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, 4–14. ESEC/FSE ’11. New York, NY, USA: ACM. https://doi.org/10.1145/2025113.2025119.

Black, S., S. Counsell, T. Hall, and D. Bowes. 2009. “Fault Analysis in OSS Based on Program Slicing Metrics.” In 2009 35th Euromicro Conference on Software Engineering and Advanced Applications, 3–10. https://doi.org/10.1109/SEAA.2009.94.

Black, Sue, Steve Counsell, Tracy Hall, and Paul Wernick. 2006. “Using Program Slicing to Identify Faults in Software.” In Beyond Program Slicing, edited by David W. Binkley, Mark Harman, and Jens Krinke. Dagstuhl Seminar Proceedings. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. http://drops.dagstuhl.de/opus/volltexte/2006/587.

Bland, J. M., and D. G. Altman. 1996. “Transformations, Means, and Confidence Intervals.” BMJ (Clinical Research Ed.) 312 (7038): 1079. https://doi.org/10.1136/bmj.312.7038.1079.

Boddy, Richard, and Gordon Smith. 2009. Statistical Methods in Practice: For Scientists and Technologists. 1 edition. Chichester, U.K: Wiley.

Briand, L. C., J. W. Daly, and J. K. Wust. 1999. “A Unified Framework for Coupling Measurement in Object-Oriented Systems.” IEEE Transactions on Software Engineering 25 (1): 91–121. https://doi.org/10.1109/32.748920.

Buse, R. P. L., and W. R. Weimer. 2010. “Learning a Metric for Code Readability.” IEEE Transactions on Software Engineering 36 (4): 546–58. https://doi.org/10.1109/TSE.2009.70.

Butler, S., M. Wermelinger, Y. Yu, and H. Sharp. 2009. “Relating Identifier Naming Flaws and Code Quality: An Empirical Study.” In 2009 16th Working Conference on Reverse Engineering, 31–35. https://doi.org/10.1109/WCRE.2009.50.

C. Williams, Chadd, and Jeffrey K. Hollingsworth. 2004. “Bug Driven Bug Finders,” January.

Chawla, N. V., K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. 2002. “SMOTE: Synthetic Minority Over-Sampling Technique.” Journal of Artificial Intelligence Research 16 (June): 321–57. https://doi.org/10.1613/jair.953.

Chen, C., S. Lin, M. Shoga, Q. Wang, and B. Boehm. 2018. “How Do Defects Hurt Qualities? An Empirical Study on Characterizing a Software Maintainability Ontology in Open Source Software.” In 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), 226–37. https://doi.org/10.1109/QRS.2018.00036.

Chhabra, Jitender Kumar, and Varun Gupta. 2009. “Evaluation of Object-Oriented Spatial Complexity Measures.” SIGSOFT Softw. Eng. Notes 34 (3): 1–5. https://doi.org/10.1145/1527202.1527208.

Chidamber, S. R., D. P. Darcy, and C. F. Kemerer. 1998. “Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis.” IEEE Transactions on Software Engineering 24 (8): 629–39. https://doi.org/10.1109/32.707698.

Chidamber, S. R., and C. F. Kemerer. 1994. “A Metrics Suite for Object Oriented Design.” IEEE Transactions on Software Engineering 20 (6): 476–93. https://doi.org/10.1109/32.295895.

Collard, M. L., M. J. Decker, and J. I. Maletic. 2011. “Lightweight Transformation and Fact Extraction with the SrcML Toolkit.” In Proceedings of the 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation, 173–184. SCAM ’11. Washington, DC, USA: IEEE Computer Society. https://doi.org/10.1109/SCAM.2011.19.

Collard, M. L., J. I. Maletic, and B. P. Robinson. 2010. “A Lightweight Transformational Approach to Support Large Scale Adaptive Changes.” In 2010 IEEE International Conference on Software Maintenance, 1–10. https://doi.org/10.1109/ICSM.2010.5609719.

Counsell, S., T. Hall, and D. Bowes. 2010. “A Theoretical and Empirical Analysis of Three Slice-Based Metrics for Cohesion.” International Journal of Software Engineering and Knowledge Engineering 20 (05): 609–36. https://doi.org/10.1142/S0218194010004888.

Counsell, S., T. Hall, E. Nasseri, and D. Bowes. 2010. “An Analysis of the ‘Inconclusive’’ Change Report Category in OSS Assisted by a Program Slicing Metric.’” In 2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications, 283–86. https://doi.org/10.1109/SEAA.2010.17.

Cowan, N. 2001. “The Magical Number 4 in Short-Term Memory: A Reconsideration of Mental Storage Capacity.” The Behavioral and Brain Sciences 24 (1): 87–114; discussion 114-185.

D’Ambros, M., M. Lanza, and R. Robbes. 2010. “An Extensive Comparison of Bug Prediction Approaches.” In 2010 7th IEEE Working Conference on Mining

Software Repositories (MSR 2010), 31–41. https://doi.org/10.1109/MSR.2010.5463279.

D'Ambros, Marco, Michele Lanza, and Romain Robbes. 2012. "Evaluating Defect Prediction Approaches: A Benchmark and an Extensive Comparison." Empirical Softw. Engg. 17 (4–5): 531–577. https://doi.org/10.1007/s10664-011-9173-9.

Darcy, David P., Chris F. Kemerer, Sandra A. Slaughter, and James E. Tomayko. 2005. "The Structural Complexity of Software: An Experimental Test." IEEE Trans. Softw. Eng. 31 (11): 982–995. https://doi.org/10.1109/TSE.2005.130.

De Lucia, Andrea, Anna Fasolino, and Malcolm Munro. 1999. "Understanding Function Behaviors through Program Slicing," August.

"DMwR-Package: Functions and Data for the Book 'Data Mining with R' in DMwR: Functions and Data for 'Data Mining with R.'" n.d. Accessed January 14, 2019. https://rdrr.io/cran/DMwR/man/DMwR-package.html.

Dragan, N., M. L. Collard, and J. I. Maletic. 2006. "Reverse Engineering Method Stereotypes." In 2006 22nd IEEE International Conference on Software Maintenance, 24–34. https://doi.org/10.1109/ICSM.2006.54.

Emam, Khaled El, Walcelio Melo, and Javam C. Machado. 2001. "The Prediction of Faulty Classes Using Object-Oriented Design Metrics." J. Syst. Softw. 56 (1): 63–75. https://doi.org/10.1016/S0164-1212(00)00086-8.

Enslen, E., E. Hill, L. Pollock, and K. Vijay-Shanker. 2009. "Mining Source Code to Automatically Split Identifiers for Software Analysis." In 2009 6th IEEE

International Working Conference on Mining Software Repositories, 71–80. https://doi.org/10.1109/MSR.2009.5069482.

Farrar, Donald E., and Robert R. Glauber. 1967. “Multicollinearity in Regression Analysis: The Problem Revisited.” The Review of Economics and Statistics 49 (1): 92–107. https://doi.org/10.2307/1937887.

Feng, Tie, and J. I. Maletic. 2006. “Using Dynamic Slicing to Analyze Change Impact on Role Type Based Component Composition Model.” In 5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering,Software Architecture and Reuse (ICIS-COMSAR’06), 103–8. https://doi.org/10.1109/ICIS-COMSAR.2006.86.

Fenton, N. E., and N. Ohlsson. 2000. “Quantitative Analysis of Faults and Failures in a Complex Software System.” IEEE Transactions on Software Engineering 26 (8): 797–814. https://doi.org/10.1109/32.879815.

Fenton, Norman, and James Bieman. 2014. Software Metrics: A Rigorous and Practical Approach, Third Edition. CRC Press.

Ferrante, Jeanne, Karl J. Ottenstein, and Joe D. Warren. 1987. “The Program Dependence Graph and Its Use in Optimization.” ACM Trans. Program. Lang. Syst. 9 (3): 319–349. https://doi.org/10.1145/24039.24041.

Gallagher, K. B., and J. R. Lyle. 1991. “Using Program Slicing in Software Maintenance.” IEEE Transactions on Software Engineering 17 (8): 751–61. https://doi.org/10.1109/32.83912.

Gold, N. E., A. M. Mohan, and P. J. Layzell. 2005. “Spatial Complexity Metrics: An Investigation of Utility.” IEEE Transactions on Software Engineering 31 (3): 203–12. https://doi.org/10.1109/TSE.2005.39.

Graves, T. L., A. F. Karr, J. S. Marron, and H. Siy. 2000. “Predicting Fault Incidence Using Software Change History.” IEEE Transactions on Software Engineering 26 (7): 653–61. https://doi.org/10.1109/32.859533.

Gray, D., D. Bowes, N. Davey, Y. Sun, and B. Christianson. 2011. “The Misuse of the NASA Metrics Data Program Data Sets for Automated Software Defect Prediction.” In 15th Annual Conference on Evaluation Assessment in Software Engineering (EASE 2011), 96–103. https://doi.org/10.1049/ic.2011.0012.

Gupta, R., M. J. Harrold, and M. L. Soffa. 1992. “An Approach to Regression Testing Using Slicing.” In Proceedings Conference on Software Maintenance 1992, 299–308. https://doi.org/10.1109/ICSM.1992.242531.

Gyimothy, T., R. Ferenc, and I. Siket. 2005. “Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction.” IEEE Transactions on Software Engineering 31 (10): 897–910. https://doi.org/10.1109/TSE.2005.112.

Hall, T., S. Beecham, D. Bowes, D. Gray, and S. Counsell. 2012. “A Systematic Literature Review on Fault Prediction Performance in Software Engineering.” IEEE Transactions on Software Engineering 38 (6): 1276–1304. https://doi.org/10.1109/TSE.2011.103.

Halstead, Maurice H. 1977. Elements of Software Science (Operating and Programming Systems Series). New York, NY, USA: Elsevier Science Inc.

Harman, Mark, and Sebastian Danicic. 1995. “Using Program Slicing to Simplify Testing.” Software Testing, Verification and Reliability 5 (September). https://doi.org/10.1002/stvr.4370050303.

Hassan, A. E., and R. C. Holt. 2005. “The Top Ten List: Dynamic Fault Prediction.” In 21st IEEE International Conference on Software Maintenance (ICSM’05), 263–72. https://doi.org/10.1109/ICSM.2005.91.

Hassan, Ahmed E. 2009. “Predicting Faults Using the Complexity of Code Changes.” In Proceedings of the 31st International Conference on Software Engineering, 78–88. ICSE ’09. Washington, DC, USA: IEEE Computer Society. https://doi.org/10.1109/ICSE.2009.5070510.

Hata, Hideaki, Osamu Mizuno, and Tohru Kikuno. 2012. “Bug Prediction Based on Fine-Grained Module Histories.” In , 200–210. IEEE. https://doi.org/10.1109/ICSE.2012.6227193.

He, Zhimin, Fengdi Shu, Ye Yang, Mingshu Li, and Qing Wang. 2012. “An Investigation on the Feasibility of Cross-Project Defect Prediction.” Automated Software Engineering 19 (2): 167–99. https://doi.org/10.1007/s10515-011-0090-3.

Hervé, Maxime. 2019. RVAideMemoire: Testing and Plotting Procedures for Biostatistics (version 0.9-71). https://CRAN.R-project.org/package=RVAideMemoire.

Hindle, Abram, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. “On the Naturalness of Software.” In Proceedings of the 34th International

Conference on Software Engineering, 837–847. ICSE ’12. Piscataway, NJ, USA: IEEE Press. http://dl.acm.org/citation.cfm?id=2337223.2337322.

Hollander, Myles, and Douglas Wolfe. 1999. Nonparametric Statistical Methods, 2nd Edition. Wiley-Interscience. http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0471190454.

Horwitz, S., T. Reps, and D. Binkley. 1988. “Interprocedural Slicing Using Dependence Graphs.” In Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, 35–46. PLDI ’88. New York, NY, USA: ACM. https://doi.org/10.1145/53990.53994.

“IEEE Standard Classification for Software Anomalies.” 2010. IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993), January, 1–23. https://doi.org/10.1109/IEEESTD.2010.5399061.

Jackson, J. Edward. 2003. A User’s Guide to Principal Components. Hoboken, N.J: Wiley-Interscience.

Jiang, Yue, Bojan Cukic, and Tim Menzies. 2008. “Can Data Transformation Help in the Detection of Fault-Prone Modules?” In Proceedings of the 2008 Workshop on Defects in Large Software Systems, 16–20. DEFECTS ’08. Seattle, Washington: Association for Computing Machinery. https://doi.org/10.1145/1390817.1390822.

Jr, Frank E. Harrell. 2015. Regression Modeling Strategies: With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis. 2nd ed. 2015 edition. Cham Heidelberg New York: Springer.

———. 2018. Rms: Regression Modeling Strategies (version 5.1-2). https://CRAN.R-project.org/package=rms.

Jr, Frank E. Harrell, and with contributions from Charles Dupont and many others. 2018. Hmisc: Harrell Miscellaneous (version 4.1-1). https://CRAN.R-project.org/package=Hmisc.

Judice, Taylor, Edith Widder, Warren Falls, Dulcinea Avouris, Dominic Cristiano, and Joseph Ortiz. 2020. "Field-Validated Detection of Aureoumbra Lagunensis Brown Tide Blooms in the Indian River Lagoon, Florida Using Sentinel-3A OLCI and Ground-Based Hyperspectral Spectroradiometers." Preprint. Earth and Space Science Open Archive. Earth and Space Science Open Archive. World. May 3, 2020. https://doi.org/10.1002/essoar.10501382.2.

Kamei, Y., S. Matsumoto, A. Monden, K. i Matsumoto, B. Adams, and A. E. Hassan. 2010. "Revisiting Common Bug Prediction Findings Using Effort-Aware Models." In 2010 IEEE International Conference on Software Maintenance, 1–10. https://doi.org/10.1109/ICSM.2010.5609530.

Khoshgoftaar, T. M., E. B. Allen, K. S. Kalaichelvan, and N. Goel. 1996. "Early Quality Prediction: A Case Study in Telecommunications." IEEE Software 13 (1): 65–71. https://doi.org/10.1109/52.476287.

Khoshgoftaar, Taghi M., and Edward B. Allen. 2003. "Ordering Fault-Prone Software Modules." Software Quality Journal 11 (1): 19–37. https://doi.org/10.1023/A:1023632027907.

Kim, S., T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. 2007. “Predicting Faults from Cached History.” In 29th International Conference on Software Engineering (ICSE’07), 489–98. https://doi.org/10.1109/ICSE.2007.66.

Kim, Sunghun, and E. James Whitehead Jr. 2006. “How Long Did It Take to Fix Bugs?” In Proceedings of the 2006 International Workshop on Mining Software Repositories, 173–174. MSR ’06. New York, NY, USA: ACM. https://doi.org/10.1145/1137983.1138027.

Kim, Sunghun, Hongyu Zhang, Rongxin Wu, and Liang Gong. 2011. “Dealing with Noise in Defect Prediction.” In , 481. ACM Press. https://doi.org/10.1145/1985793.1985859.

Kintsch, Walter. 2005. “Toward a Model of Text Comprehension and Production.” In .

Klemola, Tom. 2000. “A Cognitive Model for Complexity Metrics.” In 4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (ECOOP 2000), Sophia Antipolis, 12–16.

Korel, B., and J. Rilling. 1997. “Dynamic Program Slicing in Understanding of Program Execution.” In , Fifth International Workshop on Program Comprehension, 1997. IWPC ’97. Proceedings, 80–89. https://doi.org/10.1109/WPC.1997.601269.

———. 1998. “Program Slicing in Understanding of Large Programs.” In , 6th International Workshop on Program Comprehension, 1998. IWPC ’98. Proceedings, 145–52. https://doi.org/10.1109/WPC.1998.693339.

Korel, B., I. Singh, L. Tahat, and B. Vaysburg. 2003. “Slicing of State-Based Models.” In International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings., 34–43. https://doi.org/10.1109/ICSM.2003.1235404.

Lee, Taek, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. 2011. “Micro Interaction Metrics for Defect Prediction.” In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, 311–321. ESEC/FSE ’11. New York, NY, USA: ACM. https://doi.org/10.1145/2025113.2025156.

Lehman, Ann, Norm O’Rourke, Larry Hatcher, and Edward Stepanski. 2013. JMP for Basic Univariate and Multivariate Statistics: Methods for Researchers and Social Scientists, Second Edition. SAS Institute.

Lessmann, S., B. Baesens, C. Mues, and S. Pietsch. 2008. “Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings.” IEEE Transactions on Software Engineering 34 (4): 485–96. https://doi.org/10.1109/TSE.2008.35.

Li, Wei, and Sallie Henry. 1993. “Object-Oriented Metrics That Predict Maintainability.” J. Syst. Softw. 23 (2): 111–122. https://doi.org/10.1016/0164-1212(93)90077-B.

Li, Zhenmin, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. 2006. “Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software.” In Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, 25–33.

ASID ’06. New York, NY, USA: ACM. https://doi.org/10.1145/1181309.1181314.

Li, Zhenmin, and Yuanyuan Zhou. 2005. “PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code.” In Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 306–315. ESEC/FSE-13. New York, NY, USA: ACM. https://doi.org/10.1145/1081706.1081755.

Liang, D., and M. J. Harrold. 1998. “Slicing Objects Using System Dependence Graphs.” In Proceedings of the International Conference on Software Maintenance, 358–. ICSM ’98. Washington, DC, USA: IEEE Computer Society. http://dl.acm.org/citation.cfm?id=850947.853342.

Lucia, A. De. 2001. “Program Slicing: Methods and Applications.” In Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation, 142–49. https://doi.org/10.1109/SCAM.2001.972675.

Marcus, A., D. Poshyvanyk, and R. Ferenc. 2008. “Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems.” IEEE Transactions on Software Engineering 34 (2): 287–300. https://doi.org/10.1109/TSE.2007.70768.

McCabe, T. J. 1976. “A Complexity Measure.” IEEE Transactions on Software Engineering SE-2 (4): 308–20. https://doi.org/10.1109/TSE.1976.233837.

McIntosh, Shane, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2016. “An Empirical Study of the Impact of Modern Code Review Practices on Software

Quality." Empirical Software Engineering 21 (5): 2146–89. https://doi.org/10.1007/s10664-015-9381-9.

Mende, Thilo, and Rainer Koschke. 2010. "Effort-Aware Defect Prediction Models." In Software Maintenance and Reengineering (CSMR), 2010 14th European Conference On, 107–116. IEEE.

Menzies, T., J. Greenwald, and A. Frank. 2007. "Data Mining Static Code Attributes to Learn Defect Predictors." IEEE Transactions on Software Engineering 33 (1): 2–13. https://doi.org/10.1109/TSE.2007.256941.

Menzies, Tim, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. 2010. "Defect Prediction from Static Code Features: Current Results, Limitations, New Approaches." Automated Software Engineering 17 (4): 375–407. https://doi.org/10.1007/s10515-010-0069-5.

Meyers, T. M., and D. Binkley. 2004. "Slice-Based Cohesion Metrics and Software Intervention." In 11th Working Conference on Reverse Engineering, 256–65. https://doi.org/10.1109/WCRE.2004.34.

———. 2007. "An Empirical Study of Slice-Based Cohesion and Coupling Metrics." ACM Trans. Softw. Eng. Methodol. 17 (1): 2:1–2:27. https://doi.org/10.1145/1314493.1314495.

Mizuno, O., S. Ikami, S. Nakaichi, and T. Kikuno. 2007. "Spam Filter Based Approach for Finding Fault-Prone Software Modules." In Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007), 4–4. https://doi.org/10.1109/MSR.2007.29.

Moller, K. H., and D. J. Paulish. 1993. “An Empirical Investigation of Software Fault Distribution.” In [1993] Proceedings First International Software Metrics Symposium, 82–90. https://doi.org/10.1109/METRIC.1993.263798.

Moser, R., W. Pedrycz, and G. Succi. 2008. “A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction.” In 2008 ACM/IEEE 30th International Conference on Software Engineering, 181–90. https://doi.org/10.1145/1368088.1368114.

Moser, Raimund, Witold Pedrycz, and Giancarlo Succi. 2008. “A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction.” In Proceedings of the 30th International Conference on Software Engineering, 181–190. ICSE ’08. New York, NY, USA: ACM. https://doi.org/10.1145/1368088.1368114.

Munson, J. C., and T. M. Khoshgoftaar. 1992. “The Detection of Fault-Prone Programs.” IEEE Transactions on Software Engineering 18 (5): 423–33. https://doi.org/10.1109/32.135775.

Nagappan, N., and T. Ball. 2005. “Use of Relative Code Churn Measures to Predict System Defect Density.” In Proceedings of the 27th International Conference on Software Engineering, 284–292. ICSE ’05. New York, NY, USA: ACM. https://doi.org/10.1145/1062455.1062514.

Nagappan, N., A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. 2010. “Change Bursts as Defect Predictors.” In 2010 IEEE 21st International Symposium on

Software Reliability Engineering, 309–18. https://doi.org/10.1109/ISSRE.2010.25.

Nagappan, Nachiappan, Thomas Ball, and Andreas Zeller. 2006. “Mining Metrics to Predict Component Failures.” In , 452. ACM Press. https://doi.org/10.1145/1134285.1134349.

Nagelkerke, N. J. D. 1991. “A Note on a General Definition of the Coefficient of Determination.” Biometrika 78 (3): 691–92. https://doi.org/10.1093/biomet/78.3.691.

Nam, J., W. Fu, S. Kim, T. Menzies, and L. Tan. 2017. “Heterogeneous Defect Prediction.” IEEE Transactions on Software Engineering, 1–1. https://doi.org/10.1109/TSE.2017.2720603.

Nam, J., S. J. Pan, and S. Kim. 2013. “Transfer Defect Learning.” In 2013 35th International Conference on Software Engineering (ICSE), 382–91. https://doi.org/10.1109/ICSE.2013.6606584.

Nam, Jaechang. 2015. “Software Defect Prediction on Unlabeled Datasets.” 2015.

Newman, Christian D., Tessandra Sage, Michael L. Collard, Hakam W. Alomari, and Jonathan I. Maletic. 2016. “SrcSlice: A Tool for Efficient Static Forward Slicing.” In Proceedings of the 38th International Conference on Software Engineering Companion, 621–624. ICSE ’16. New York, NY, USA: ACM. https://doi.org/10.1145/2889160.2889173.

Nguyen, Anh Tuan, and Tien N. Nguyen. 2015. “Graph-Based Statistical Language Model for Code.” In Proceedings of the 37th International Conference on

Software Engineering - Volume 1, 858–868. ICSE '15. Piscataway, NJ, USA: IEEE Press. http://dl.acm.org/citation.cfm?id=2818754.2818858.

Nucci, D. Di, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia. 2018. "A Developer Centered Bug Prediction Model." IEEE Transactions on Software Engineering 44 (1): 5–24. https://doi.org/10.1109/TSE.2017.2659747.

Nucci, D. Di, F. Palomba, S. Siravo, G. Bavota, R. Oliveto, and A. De Lucia. 2015. "On the Role of Developer's Scattered Changes in Bug Prediction." In 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), 241–50. https://doi.org/10.1109/ICSM.2015.7332470.

Ohlsson, N., and H. Alberg. 1996. "Predicting Fault-Prone Software Modules in Telephone Switches." IEEE Transactions on Software Engineering 22 (12): 886–94. https://doi.org/10.1109/32.553637.

Ortiz, Joseph D., Dulci M. Avouris, Stephen J. Schiller, Jeffrey C. Luvall, John D. Lekki, Roger P. Tokars, Robert C. Anderson, Robert Shuchman, Michael Sayers, and Richard Becker. 2019. "Evaluating Visible Derivative Spectroscopy by Varimax-Rotated, Principal Component Analysis of Aerial Hyperspectral Images from the Western Basin of Lake Erie." *Journal of Great Lakes Research* 45 (3): 522–35. https://doi.org/10.1016/j.jglr.2019.03.005.

Ostrand, T. J., E. J. Weyuker, and R. M. Bell. 2005. "Predicting the Location and Number of Faults in Large Software Systems." IEEE Transactions on Software Engineering 31 (4): 340–55. https://doi.org/10.1109/TSE.2005.49.

Ott, L. M., and J. J. Thuss. 1993. “Slice Based Metrics for Estimating Cohesion.” In [1993] Proceedings First International Software Metrics Symposium, 71–81. https://doi.org/10.1109/METRIC.1993.263799.

Ottenstein, Karl J., and Linda M. Ottenstein. 1984. “The Program Dependence Graph in a Software Development Environment.” In Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 177–184. SDE 1. New York, NY, USA: ACM. https://doi.org/10.1145/800020.808263.

Pádua, Guilherme B. de, and Weiyi Shang. 2018. “Studying the Relationship Between Exception Handling Practices and Post-Release Defects.” In Proceedings of the 15th International Conference on Mining Software Repositories, 564–575. MSR ’18. New York, NY, USA: ACM. https://doi.org/10.1145/3196398.3196435.

Palomba, F., M. Zanoni, F. A. Fontana, A. D. Lucia, and R. Oliveto. 2016. “Smells Like Teen Spirit: Improving Bug Prediction Performance Using the Intensity of Code Smells.” In 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), 244–55. https://doi.org/10.1109/ICSME.2016.27.

Palomba, F., M. Zanoni, F. Arcelli Fontana, A. De Lucia, and R. Oliveto. 2017. “Toward a Smell-Aware Bug Prediction Model.” IEEE Transactions on Software Engineering, 1–1. https://doi.org/10.1109/TSE.2017.2770122.

Pan, K., S. Kim, and E. J. Whitehead Jr. 2006. “Bug Classification Using Program Slicing Metrics.” In 2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation, 31–42. https://doi.org/10.1109/SCAM.2006.6.

Peduzzi, P., J. Concato, E. Kemper, T. R. Holford, and A. R. Feinstein. 1996. “A Simulation Study of the Number of Events per Variable in Logistic Regression Analysis.” Journal of Clinical Epidemiology 49 (12): 1373–79. https://doi.org/10.1016/s0895-4356(96)00236-3.

Pennington, Nancy. 1987. “Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs.” Cognitive Psychology 19 (3): 295–341. https://doi.org/10.1016/0010-0285(87)90007-7.

Petrić, Jean, and Tihana Galinac Grbac. 2014. “Software Structure Evolution and Relation to System Defectiveness.” In Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, 34:1–34:10. EASE ’14. New York, NY, USA: ACM. https://doi.org/10.1145/2601248.2601287.

Podgurski, A., and L. A. Clarke. 1990. “A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance.” IEEE Transactions on Software Engineering 16 (9): 965–79. https://doi.org/10.1109/32.58784.

Pressman, Roger S. 2015. Software Engineering: A Practitioner’s Approach. Eighth edition. New York, NY: McGraw-Hill Education.

Purushothaman, R., and D. E. Perry. 2005. “Toward Understanding the Rhetoric of Small Source Code Changes.” IEEE Transactions on Software Engineering 31 (6): 511–26. https://doi.org/10.1109/TSE.2005.74.

Rahman, Akond. 2018. “Comprehension Effort and Programming Activities: Related? Or Not Related?” In Proceedings of the 15th International Conference on Mining Software Repositories, 66–69. MSR ’18. New York, NY, USA: ACM. https://doi.org/10.1145/3196398.3196470.

Rahman, F., and P. Devanbu. 2013. “How, and Why, Process Metrics Are Better.” In 2013 35th International Conference on Software Engineering (ICSE), 432–41. https://doi.org/10.1109/ICSE.2013.6606589.

Rahman, Foyzur, and Premkumar Devanbu. 2011. “Ownership, Experience and Defects: A Fine-Grained Study of Authorship.” In Proceedings of the 33rd International Conference on Software Engineering, 491–500. ICSE ’11. New York, NY, USA: ACM. https://doi.org/10.1145/1985793.1985860.

Rahman, Foyzur, Daryl Posnett, and Premkumar Devanbu. 2012. “Recalling the ‘Imprecision’ of Cross-Project Defect Prediction.” In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, 61:1–61:11. FSE ’12. New York, NY, USA: ACM. https://doi.org/10.1145/2393596.2393669.

Rajbahadur, Gopi Krishnan, Shaowei Wang, Yasutaka Kamei, and Ahmed E. Hassan. 2017. “The Impact of Using Regression Models to Build Defect Classifiers.” In Proceedings of the 14th International Conference on Mining Software Repositories, 135–145. MSR ’17. Piscataway, NJ, USA: IEEE Press. https://doi.org/10.1109/MSR.2017.4.

Rilling, Juergen, and Tuomas Klemola. 2003. “Identifying Comprehension Bottlenecks Using Program Slicing and Cognitive Complexity Metrics.” In Proceedings of the 11th IEEE International Workshop on Program Comprehension, 115–. IWPC ’03. Washington, DC, USA: IEEE Computer Society. http://dl.acm.org/citation.cfm?id=851042.857047.

Rosenberg, J. 1997. “Some Misconceptions about Lines of Code.” In Proceedings Fourth International Software Metrics Symposium, 137–42. https://doi.org/10.1109/METRIC.1997.637174.

Scalabrino, S., M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto. 2016. “Improving Code Readability Models with Textual Features.” In 2016 IEEE 24th International Conference on Program Comprehension (ICPC), 1–10. https://doi.org/10.1109/ICPC.2016.7503707.

Scalabrino, Simone, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. 2017. “Automatically Assessing Code Understandability: How Far Are We?” In Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, 417–427. ASE 2017. Piscataway, NJ, USA: IEEE Press. http://dl.acm.org/citation.cfm?id=3155562.3155617.

Schröter, Adrian, Thomas Zimmermann, and Andreas Zeller. 2006. “Predicting Component Failures at Design Time.” In Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, 18–27. ISESE ’06. New York, NY, USA: ACM. https://doi.org/10.1145/1159733.1159739.

Shang, Weiyi, Meiyappan Nagappan, and Ahmed E. Hassan. 2015. “Studying the Relationship between Logging Characteristics and the Code Quality of Platform Software.” Empirical Software Engineering 20 (1): 1–27. https://doi.org/10.1007/s10664-013-9274-8.

Sharif, Bonita, and Jonathan I. Maletic. 2010. “An Eye Tracking Study on CamelCase and Under_Score Identifier Styles.” In Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension, 196–205. ICPC ’10. Washington, DC, USA: IEEE Computer Society. https://doi.org/10.1109/ICPC.2010.41.

Shepperd, Martin, and D.C. Ince. 1994. “A Critique of Three Metrics.” Journal of Systems and Software 26 (September): 197–210. https://doi.org/10.1016/0164-1212(94)90011-6.

Shihab, Emad, Christian Bird, and Thomas Zimmermann. 2012. “The Effect of Branching Strategies on Software Quality.” In Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, 301–310. ESEM ’12. New York, NY, USA: ACM. https://doi.org/10.1145/2372251.2372305.

Shihab, Emad, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2011. “High-Impact Defects: A Study of Breakage and Surprise Defects.” In , 300. ACM Press. https://doi.org/10.1145/2025113.2025155.

Siegmund, Janet, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. 2017.

"Measuring Neural Efficiency of Program Comprehension." In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 140–150. ESEC/FSE 2017. New York, NY, USA: ACM. https://doi.org/10.1145/3106237.3106268.

Silva, Josep. 2012. "A Vocabulary of Program Slicing-Based Techniques." ACM Comput. Surv. 44 (3): 12:1–12:41. https://doi.org/10.1145/2187671.2187674.

Śliwerski, Jacek, Thomas Zimmermann, and Andreas Zeller. 2005. "When Do Changes Induce Fixes?" In Proceedings of the 2005 International Workshop on Mining Software Repositories, 1–5. MSR '05. New York, NY, USA: ACM. https://doi.org/10.1145/1082983.1083147.

Storey, M.-. 2005. "Theories, Methods and Tools in Program Comprehension: Past, Present and Future." In 13th International Workshop on Program Comprehension (IWPC'05), 181–91. https://doi.org/10.1109/WPC.2005.38.

Storey, M. -A. D., K. Wong, and H. A. Müller. 2000. "How Do Program Understanding Tools Affect How Programmers Understand Programs?" Science of Computer Programming 36 (2): 183–207. https://doi.org/10.1016/S0167-6423(99)00036-2.

Subramanyam, R., and M. S. Krishnan. 2003. "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects." IEEE Transactions on Software Engineering 29 (4): 297–310. https://doi.org/10.1109/TSE.2003.1191795.

Taba, S. E. S., F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan. 2013. “Predicting Bugs Using Antipatterns.” In 2013 IEEE International Conference on Software Maintenance, 270–79. https://doi.org/10.1109/ICSM.2013.38.

Tan, Ming, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. “Online Defect Prediction for Imbalanced Data.” In Proceedings of the 37th International Conference on Software Engineering - Volume 2, 99–108. ICSE ’15. Piscataway, NJ, USA: IEEE Press. http://dl.acm.org/citation.cfm?id=2819009.2819026.

Tantithamthavorn, C., A. E. Hassan, and K. Matsumoto. 2018. “The Impact of Class Rebalancing Techniques on the Performance and Interpretation of Defect Prediction Models.” IEEE Transactions on Software Engineering, January, 1. https://doi.org/10.1109/TSE.2018.2876537.

Tantithamthavorn, C., S. McIntosh, A. E. Hassan, and K. Matsumoto. 2016. “Automated Parameter Optimization of Classification Techniques for Defect Prediction Models.” In 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), 321–32. https://doi.org/10.1145/2884781.2884857.

———. 2017. “An Empirical Comparison of Model Validation Techniques for Defect Prediction Models.” IEEE Transactions on Software Engineering 43 (1): 1–18. https://doi.org/10.1109/TSE.2016.2584050.

———. 2018. “The Impact of Automated Parameter Optimization on Defect Prediction Models.” IEEE Transactions on Software Engineering, 1–1. https://doi.org/10.1109/TSE.2018.2794977.

Tip, Frank. 1994. “A Survey of Program Slicing Techniques.” Amsterdam, The Netherlands, The Netherlands: CWI (Centre for Mathematics and Computer Science).

Tricentis. 2017. “Software Fail Watch: 2016 in Review.” Software Testing Tools for Continuous Testing | Tricentis (blog). January 4, 2017. https://www.tricentis.com/resource-assets/software-fail-watch-2016/.

Tu, Zhaopeng, Zhendong Su, and Premkumar Devanbu. 2014. “On the Localness of Software.” In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 269–280. FSE 2014. New York, NY, USA: ACM. https://doi.org/10.1145/2635868.2635875.

Turhan, Burak, Tim Menzies, Ayşe B. Bener, and Justin Di Stefano. 2009. “On the Relative Value of Cross-Company and within-Company Data for Defect Prediction.” Empirical Software Engineering 14 (5): 540–78. https://doi.org/10.1007/s10664-008-9103-7.

Wang, Song, Taiyue Liu, and Lin Tan. 2016. “Automatically Learning Semantic Features for Defect Prediction.” In , 297–308. ACM Press. https://doi.org/10.1145/2884781.2884804.

Weiser, M. 1984. “Program Slicing.” IEEE Transactions on Software Engineering SE-10 (4): 352–57. https://doi.org/10.1109/TSE.1984.5010248.

Weiser, Mark. 1981. “Program Slicing.” In Proceedings of the 5th International Conference on Software Engineering, 439–449. ICSE ’81. Piscataway, NJ, USA: IEEE Press. http://dl.acm.org/citation.cfm?id=800078.802557.

———. 1982. “Programmers Use Slices When Debugging.” Commun. ACM 25 (7): 446–452. https://doi.org/10.1145/358557.358577.

Weiser, Mark, and Jim Lyle. 1986. “Experiments on Slicing-Based Debugging Aids.” In Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers, 187–197. Norwood, NJ, USA: Ablex Publishing Corp. http://dl.acm.org/citation.cfm?id=21842.28894.

Weyuker, Elaine J., Thomas J. Ostrand, and Robert M. Bell. 2006. “Adapting a Fault Prediction Model to Allow Widespread Usage.” In Proceedings of the Second International Promise Workshop, 1.

———. 2010. “Comparing the Effectiveness of Several Modeling Methods for Fault Prediction.” Empirical Softw. Engg. 15 (3): 277–295. https://doi.org/10.1007/s10664-009-9111-2.

White, Martin, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. “Toward Deep Learning Software Repositories.” In Proceedings of the 12th Working Conference on Mining Software Repositories, 334–345. MSR ’15. Piscataway, NJ, USA: IEEE Press. http://dl.acm.org/citation.cfm?id=2820518.2820559.

Wilcoxon, Frank. 1945. “Individual Comparisons by Ranking Methods.” Biometrics Bulletin 1 (6): 80–83. https://doi.org/10.2307/3001968.

Wu, Fangjun, and Tong Yi. 2004. “Slicing Z Specifications.” SIGPLAN Not. 39 (8): 39–48. https://doi.org/10.1145/1026474.1026481.

Wu, Shaomin, and Peter Flach. 2005. A Scored AUC Metric for Classifier Evaluation and Selection.

Xu, Baowen, J. Qian, X. Zhang, Zhongqiang Wu, and Lin Chen. 2005. "A Brief Survey of Program Slicing." SIGSOFT Softw. Eng. Notes 30 (2): 1–36. https://doi.org/10.1145/1050849.1050865.

Yang, Y., Y. Zhou, H. Lu, L. Chen, Z. Chen, B. Xu, H. Leung, and Z. Zhang. 2015. "Are Slice-Based Cohesion Metrics Actually Useful in Effort-Aware Post-Release Fault-Proneness Prediction? An Empirical Study." IEEE Transactions on Software Engineering 41 (4): 331–57. https://doi.org/10.1109/TSE.2014.2370048.

Yoo, S., and M. Harman. 2012. "Regression Testing Minimization, Selection and Prioritization: A Survey." Software Testing, Verification & Reliability 22 (2): 67–120. https://doi.org/10.1002/stv.430.

Zhang, F., F. Khomh, Y. Zou, and A. E. Hassan. 2012. "An Empirical Study of the Effect of File Editing Patterns on Software Quality." In 2012 19th Working Conference on Reverse Engineering, 456–65. https://doi.org/10.1109/WCRE.2012.55.

Zhang, Feng, Audris Mockus, Iman Keivanloo, and Ying Zou. 2014. "Towards Building a Universal Defect Prediction Model." In Proceedings of the 11th Working Conference on Mining Software Repositories, 182–191. MSR 2014. New York, NY, USA: ACM. https://doi.org/10.1145/2597073.2597078.

Zhang, Feng, Quan Zheng, Ying Zou, and Ahmed E. Hassan. 2016. "Cross-Project Defect Prediction Using a Connectivity-Based Unsupervised Classifier." In Proceedings of the 38th International Conference on Software Engineering, 309–

320. ICSE '16. New York, NY, USA: ACM. https://doi.org/10.1145/2884781.2884839.

Zhang, H. 2009. "An Investigation of the Relationships between Lines of Code and Defects." In 2009 IEEE International Conference on Software Maintenance, 274–83. https://doi.org/10.1109/ICSM.2009.5306304.

Zhang, Xiangyu, Neelam Gupta, and Rajiv Gupta. 2007. "A Study of Effectiveness of Dynamic Slicing in Locating Real Faults." Empirical Softw. Engg. 12 (2): 143–160. https://doi.org/10.1007/s10664-006-9007-3.

Zhao, Jianjun. 1998. "Applying Slicing Technique to Software Architectures." In Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No.98EX193), 87–98. https://doi.org/10.1109/ICECCS.1998.706659.

Zimmermann, T., and N. Nagappan. 2008. "Predicting Defects Using Network Analysis on Dependency Graphs." In 2008 ACM/IEEE 30th International Conference on Software Engineering, 531–40. https://doi.org/10.1145/1368088.1368161.