Evaluating the Likelihood of Bug Inducing Commits Using Metrics Trend Analysis

Parul Parul,
Supervisor: Kontogiannis, Konstantinos, The University of Western Ontario
A thesis submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science
© Parul Parul 2023

Follow this and additional works at: https://ir.lib.uwo.ca/etd
Part of the Software Engineering Commons

Recommended Citation
https://ir.lib.uwo.ca/etd/9391

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact wlswadmin@uwo.ca.
Abstract

Continuous software engineering principles advocate a release-small, release-often process model, where new functionality is added to a system, in small increments and very frequently. In such a process model, every time a change is introduced it is important to identify as early as possible, whether the system has entered a state where faults are more likely to occur. In this paper, we present a method that is based on process, quality, and source code metrics to evaluate the likelihood that an imminent bug-inducing commit is highly probable. More specifically, the method analyzes the correlations and the rate of change of selected metrics. The findings from the technical debt dataset extracted data from SonarQube indicate that before bug-inducing commits, metrics that otherwise are not correlated, suddenly exhibit a high correlation or a high rate of change. This metric behavior can then be used for assessing an impending bug-inducing commit. The technique is programing language agnostic, based on metrics that can be extracted without the use of specialized parsers and can be applied to forewarn developers that a file, or a collection of files, has entered a state where faults are highly probable.

Keywords: Fault Assessment, Fault prediction, process metrics, quality metrics, quantitative analysis
Summary for Lay Audience

The primary focus of the software development process is to produce high-quality software at every stage. To achieve this, it is important to continuously assess and improve the quality of the software. One way to accomplish this is through software quality prediction, which involves evaluating the quality of the software regularly and identifying any potential quality issues early on. This process is also known as bug prediction, as it aims to identify and address bugs or defects in the software.

Bug prediction is vital for improving the overall quality of the software and can also help to reduce the cost and time required for testing. This thesis proposes techniques for evaluating the likelihood of the fault-inducing commit based on process, quality, and source code metrics. Our approach is based on the calculation of correlation and the angle of the least square regression line of different metrics. By examining the values, we aim to assess that faults are highly probable in the upcoming commits.
Acknowledgement

I would like to express my deepest gratitude to my supervisor Dr. Kostas Kontogiannis, for his invaluable guidance and support throughout this research project. This endeavor would not have been possible without his encouragement and immense knowledge. At many stages during this research project, I benefited from his invaluable advice. His positive outlook and confidence in my research inspired me.

I would like to extend my sincere thanks to IBM (International Business Machines Corporation) for giving me the opportunity to do part of my research on their premises through my graduate program.

Besides, I am thankful to my family for their encouragement and support throughout my studies. I am also grateful to all my friends, faculty members, and other members of the Department for their help, support, and valuable discussion throughout this dissertation work. I thank all members who helped me directly or indirectly in bringing this report to this present form.
Contents

Abstract i

Summary for Lay Audience ii

Acknowledgement iii

List of Figures viii

List of Tables ix

List of Appendices xi

1 INTRODUCTION 1
  1.1 Preface ................................................. 1
  1.2 Problem Area and Motivation .......................... 2
  1.3 Thesis Contribution and Scope ........................ 3
  1.4 Thesis Outline ........................................ 5

2 Background And Related Work 6
  2.1 Background ............................................. 6
  2.1.1 Software Defect Prediction ......................... 6
  2.1.2 Software Metrics ................................... 7
        Quality Metrics ....................................... 8
        Static Code Metrics ................................... 9
        Process Metrics ....................................... 11
        Technical Debt Metrics .............................. 12
  2.1.3 Machine Learning .................................... 15
5.3 Statistical Analysis of Experiments .......................... 61
5.4 Limitations ....................................................... 63

6 Conclusion ......................................................... 64
6.1 Future Work ..................................................... 65

Bibliography .......................................................... 66

A SQL Queries for pre-processing the Technical Debt Dataset 76
A.1 Determining the files that are common in fixing and inducing commits .. 76
A.1.1 Created a new table for inducing commit and fixing commit, along with
their respective dates. ........................................... 76
A.1.2 Created a table having commit hash, file and date information ........... 77
A.1.3 Table to get the files of inducing commits .................................. 78
A.1.4 Table to get the files of Fixing commits ..................................... 79
A.1.5 Table to get the intersection of files of fixing and inducing commits to
determine which files had issue .................................... 81
A.2 Determining the relevant Commits for Fault Inducing and Fixing Commits .. 82
A.2.1 Created a table having required SONAR MEASURES with respect to
COMMIT_HASH .................................................. 82
A.2.2 Created a table to store all the data groupwise (Groups are based on
the fault inducing and fault fixing commits, and all the commits that
occurred before fault fixing) ..................................... 86
Create a table to get all the commits for the group .......................... 86
Specify if the commit is fault inducing or fixing or before inducing or
after inducing and before fixing .................................. 89
Add the Pair_Number (group_Number) for each inducing and fixing
commit .............................................................. 91
Add commit measures to the table .................................. 92
A.3 Final Dataset Preparation ........................................ 95

B Data extraction and processing Algorithms .......................... 96
B.1 Algorithm to determine the low correlation metrics .......................... 96
B.2 Algorithm to calculate the correlation of 5 and 10 commits before bug inducing commit .............................................................. 100
B.3 Algorithm to calculate the Slope of 5 and 10 commits before bug inducing commit 106

Curriculum Vitae 113
List of Figures

2.1 Entity Relationship Diagram of the Technical debt dataset . . . . . . . . . . . . 23

3.1 Filtering false positive bug-inducing commits reported by SZZ algorithm . . 37

3.2 Schematic of relevant commits $PC_{i,5}$ . . . $PC_{i,1}$ occurring prior to a bug inducing commit $BIC_i$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

3.3 Schematic of relevant commits $PC_{i,10}$ . . . $PC_{i,1}$ occurring prior to a bug inducing commit $BIC_i$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

3.4 Example of metrics values in commits prior to a bug-inducing commit (commit 17) and between the bug-inducing and the bug-fixing commit (commit 18-21) . 40

4.1 Correlation of NCLOC and New SQALE Debt Ratio for five commits before bug inducing commit of one of the pairs . . . . . . . . . . . . . . . . . . . . . . 48

4.2 Schematic of relevant commits in segment $PC_{i,15}$. . . $PC_{i,6}$ and relevant commits in segment $PC_{i,5}$ . . . $PC_{i,1}$ occurring prior to a bug-inducing commit $BIC_i$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

4.3 Example of the rate of change (slope) of metric New SQALE Debt Ratio in the segment of 15 - 6 commits prior to bug-inducing commit #16 and the segment of 5 commits prior to the bug-inducing commit #16 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

A.1 Sample group details . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
# List of Tables

2.1 Confusion Matrix ........................................... 19

3.1 Sample data of table ZALLPROJECTSFAULT INDUCING_COMMITS .......................... 32
3.2 Sample data of table ZALLPROJECTS_GIT_COMMITS_CHANGES .......................... 33
3.3 Sample data of table ZALLPROJECTS_FILES_INDUCING_COMMITS ....................... 34
3.4 Sample data of table ZALLPROJECTS_FILES_FIXING_COMMITS .......................... 35
3.5 Sample data of table ZALLPROJECTS_FILES_INDUCING_INTERSECTION_FIXING_COMMITS ........................................... 36
3.6 SONAR METRICS considered for analysis ................................................................. 42
3.7 Sample data of ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_INDUCING for one pair ........................................... 43
3.8 Project Statistics ........................................................................................................ 43
3.9 Correlation of six metrics with New SQALE Debt Ratio ........................................ 43

4.1 Correlation matrix for project “httpcomponents-client” ......................................... 46

5.1 Behavior of correlations of the six metrics with New SQALE Debt Ratio in five relevant commits before a bug-inducing commit ........................................... 57
5.2 Behavior of correlations of the six metrics with New SQALE Debt Ratio in ten relevant commits before a bug-inducing commit ........................................... 58
5.3 Difference of rate of change in the periods 15-6 and 5-1 commits before bug inducing commit, of the raw values of the seven metrics considered ........................................... 59
5.4 Rate of Change Prediction Probabilities .................................................................... 60
5.5 Chi-square test on the correlation of 5 relevant commits ........................................ 62
5.6 Chi-square test on the correlation of 10 relevant commits ........................................ 62
5.7 Chi-square test on the rate of change of metrics . . . . . . . . . . . . 62
List of Appendices

Appendix A SQL Queries for pre-processing the Technical Debt Dataset . . . . . . . . . 76
Appendix B Data extraction and processing Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . 96
Chapter 1

INTRODUCTION

1.1 Preface

Software systems constantly evolve. Some systems evolve in an incremental or iterative fashion by introducing new versions or new releases respectively. In contrast, other systems (see Netflix, eBay, Amazon) evolve in what is referred to as continuous cycles, where new features are added, deployed, and released on a daily basis. Early detection of error-proneness is an important issue in continuous integration and continuous deployment (CI/CD) so overwhelming the testing process can be prevented. Agile process models that support continuous integration and deployment require a shift-left approach where quality assurance and risk analysis are applied as early as possible in the software life cycle. “Shift-left” is the practice of moving testing, quality, and performance evaluation earlier in the software life cycle. Shift-left testing helps teams anticipate changes that arise during the development process that can affect performance or other delivery processes [31]. In a continuous development environment where new source code is introduced, deployed, and released at a fast pace, it is important for the developers to be able to assess whether their system has entered a state where there is a risk of an imminent bug-inducing commit. In this thesis, we take a quantitative approach, and our hypothesis is that the changes in the behavior of certain process metrics, quality metrics, and source code metrics can serve as an indicator for an impending bug-inducing commit.

In this study, we present the methods that use metrics like NCLOC, New Sqale debt ratio,
Violations, etc. of the commits, obtained from the Technical Debt dataset [76] to explore the probing of the bug-inducing commit. We use the term error or bug or defect in this thesis to indicate a mistake in the computer program that causes deviation from its specified observable and expected behavior.

The analysis is performed on 22 selected projects of the dataset [76]. We pre-processed the dataset to get the relevant information before performing the analysis. Finally, we analyzed the rate of change and the correlations of selected metrics, and how the former parameters change a few commits before the bug-inducing commits

1.2 Problem Area and Motivation

**Problem Area:** Software Analysis for the identification and assessment of fault proneness of a software system

**Motivation:** In any software application, it is almost certain that the bugs would be introduced at the development or maintenance phase. It is no surprise that usually a significant number of resources are dedicated to fixing bugs and enhancing the quality of code.

Moreover, the cost of fixing the bug in later stages, could be more as compared to the development phase. Hence, it is necessary to find the bugs as early as possible. Most of the faults are generally detected during the testing phase. It has been noted that testing operations often take up between 45% and 75% of the total development time. [45].

By predicting the bugs, we may reduce the time consumed during the testing phase. Through this study, we hope to devise techniques that forewarn developers and testers about the likelihood of bugs being introduced in shortly upcoming commits. Developers can improve the quality of the code by reducing the time spent on debugging and addressing errors with the help of this. Finding the faulty commit promptly will not only save plenty of time but also a lot of resources.

Additionally, it may be advantageous to predict software flaws in advance, given the current demand for software systems to be released in shorter time frames and the application of the continuous engineering method to DevOps.
1.3 Thesis Contribution and Scope

With this study, we hope to alert developers and testers about the probability of impending faulty commits so they may plan ahead and prioritize test cases to reduce risk.

1.3 Thesis Contribution and Scope

Estimating the possibility of bugs getting induced in the upcoming commits has been the area where the software engineering community has paid significant attention over the past few years. We have seen major research activity on fault proneness prediction frameworks that mostly utilize machine learning. These frameworks rely on classification models which are trained on large data sets so that a file or a module can be classified as fault-prone or not. These systems fall into two main categories:

- Fault proneness prediction systems that are based on source code metrics [72],[90],[32]
- Systems that are based on process metrics [17],[94],[98]

Research in this area indicates that process metrics-based frameworks slightly outperform the accuracy of the source code metrics-based frameworks [56]. However, a major drawback of machine learning systems is:

- The trained models can quickly be out-of-tune when frequent source code changes occur and require the models to be re-trained.
- There is no specific learning technique that performs best for all the datasets.

In this thesis, we present an approach that is based on the quantitative analysis of a collection of processes, quality, and simple-to-extract source code metrics. The focus of this thesis is to devise fault assessment techniques, as a result, the developer and testers can be alerted regarding the risk of bug induction in a particular commit. Our hypothesis is that metrics that do not correlate or do not exhibit any common pattern behavior throughout the system’s lifetime, start behaving differently just before an impending bug-inducing commit. The premise is that this change of behavior indicates that the system enters an unsafe state, which can serve as a risk indicator to the developers that any source code change, they make from this point on has a
higher potential of introducing a fault. For our work, we utilized the SonarQube Technical Debt dataset [76]. This dataset provides a wealth of information including:

- Commit information details such as time stamps and unique commit Ids.
- A list of refactoring applied in each commit.
- Code quality data in the form of technical debt-related metrics such as anti-patterns, style violations, and code smells.
- Classification of whether a commit is a bug-inducing or bug-fixing commit.
- A list of source code size-related metrics, and source code complexity metrics.

Overall, the focus of this thesis is:

- Explore and select a set of metrics suitable for developing techniques to assess bug induction.
- Analyze and report whether the change in the correlation of selected metrics can be used as an indicator of an imminent bug-inducing commit.
- Investigate and report whether the rate of change of selected metrics can be used as an indicator of an imminent bug-inducing commit.
- Analyze and report whether the combination of correlation change and rate of metrics values change can be used as an indicator of an imminent bug-inducing commit.

Our approach was first to thoroughly examine the correlation of these metrics across a system’s lifetime. Out of 29 available metrics, we have identified 7 metrics that exhibit low pair-wise Pearson correlation with the New SQALE Debt Ratio metric across the lifetime of all twenty-two open-source projects we have examined. These metrics are the number of code style violations, number of code smells, non-comment Lines of Code, number of functions, number of classes, total lines, and class-level cyclomatic complexity.

Second, for every bug-inducing commit we examined how each of these seven pairs of correlation metrics behaves within a period of five relevant commits prior to the bug-inducing commit.
A relevant commit is a commit that includes at least one of the files appearing in the intersection of files contained in the bug-inducing commit and the corresponding bug-fixing commit.

Third, we monitored the difference in the rate of change of all metrics in consecutive rolling windows of fifteen commits, each split into two segments a 10 commits-wide segment followed by a 5 commits-wide segment, and we examined how these rates differ in these two segments leading to a bug-inducing commit.

The results from these analyses indicate that the correlations of the monitored metrics as well as the rate of change exhibit different behavior prior to an imminent bug-inducing commit compared to a normal period of the system’s lifetime where no bug-inducing commits are recorded. In this respect, we can say that these changes in metrics behavior are very strong indicators of an imminent bug-inducing commit.

1.4 Thesis Outline

The remainder of this thesis is organized as follows:

Chapter 2 presents relevant background information that is useful in placing this work in context and related work describing different techniques of fault prediction.

Chapter 3 discusses the dataset and the pre-processing of the dataset.

Chapter 4 explores the metrics analysis for evaluating the likelihood of bug-inducing commits.

Chapter 5 presents the experimental results of how changes in metric behavior serve as indicators for imminent bug-inducing commits, the statistical analysis of the experiments using the Chi-Square Test, and the limitations

Finally, Chapter 6 concludes the paper along with pointers for future research.
Chapter 2

Background And Related Work

2.1 Background

2.1.1 Software Defect Prediction

A software defect is a bug or fault in the source code that prevents the software to behave in the desired manner, which is specified by the requirement documents. Every software has bugs, which need to be reduced. It has been observed that a critical fault can completely shut down the application [13]. Thus, eliminating the defects at the early stage is necessary to minimize their impact and enhance the quality of the software. By predicting the defects, we can advance the prompt removal of defects. Moreover, it is much more cost-efficient as compared to software testing.

Software Defect Prediction (SDP) is a standardized way of forecasting the defect proneness of the software component. Once the module is predicted to be fault-prone, the developers can take essential steps to avoid the bugs, and testers would know which module requires extensive testing. SDP resolves one of the most exhaustive parts of software development which is detecting and fixing bugs.

Although fault prediction can be propitious, it is not always straightforward to achieve [12]. A significant body of research has been conducted in the past four decades to predict bugs. Unfortunately, to date, there is no single direct methodology for bug forecasting. As a result,
researchers and software scientists utilize various combinations of metrics to achieve the desired result. Upon selecting the metrics, the next step is to choose the appropriate technique to achieve the defect prediction [79]. In the related literature, there is a wealth of approaches for defect prediction and error proneness identification. Two widely used defect prediction techniques are regression and classification. The main purpose of regression techniques is to estimate the number of software defects in a software component. In contrast, classification techniques aim at tagging a software module as faulty or not. It has been shown that classification models can be trained on defect data from earlier versions of the system under scrutiny.

2.1.2 Software Metrics

In this era of Continuous engineering where new features are added in small increments and very frequently. The strict deadlines and the need to develop more functionalities often result in neglecting the defects. High software quality is crucial to organizations. Enhancing the quality of software has been a significant research topic in the software engineering field. Metrics can be utilized to quantify the development and maintenance of software resulting in improved systems [65].

Software metrics are the standards of quantitative measures for software characteristics. The quality of the code can be determined by tracking and analyzing the metrics. Metrics are the numeric values of the software that can be used to predict bugs [64].

Software metrics are categorized into process, project, and product metrics. Process metrics measure the properties of software development and maintenance processes. It helps in predicting the size of the final system and determining whether a project on running according to the schedule.[81]. Project metrics determine the characteristics and execution of the project such as size, complexity, performance, and design features. These metrics are utilized by the project managers to decide the project workflow [63]. Product metrics measure the product properties at any stage in Software Development Lifecycle (SDLC) such as source code and design documents. Some metrics belong to multiple categories for example software quality metrics are associated with the process and product metrics. Code metrics are one of the major types that help to improve and compare the system components. It is further subdivided based
on measuring the specific part of the software Dynamic code metrics and Static code metrics [83].

Dynamic code metrics are accumulated upon the execution of the program to examine the behavior of the system by comparing actual and expected responses. On the other hand, Static code metrics are obtained before the execution of the code [53]. They are suitable for checking attributes of the code, for example, the complexity and lines of code[83].

Careful choice of metrics can improvise the defect prediction performance. Researchers have mostly focused on four broad classes of metrics for defect prediction quality metrics, process metrics, static code metrics, and technical debt metrics [68]. In the next section, we briefly discuss these types.

**Quality Metrics**

IEEE has published a standard for the software quality metrics methodology [5]. As per that report, “*Software quality is the degree to which software possesses a desired combination of attributes. This desired combination of attributes shall be clearly defined; otherwise, assessment of quality is left to intuition*”. Defined collections of software quality attributes specify the software quality for the projects. To measure the software quality attributes an appropriate set of software metrics should be identified.

Software quality metrics evaluate the software to ascertain that the software quality requirements are fulfilled. Software metrics provide the quantitative analysis of software quality, as a result, we do not need to rely on subjective evaluations. Software metrics, however, cannot take the place of human judgment in software evaluation. The adoption of software metrics within an organization is anticipated to have a positive impact by making quality more visible.

A valid predictive metric must demonstrate a strong correlation with the quality factors it represents. This is the same as precisely describing the quality state(s) of a process or a product. Here states refer to the various conditions or levels that the quality factor can exhibit. For example, a product may have different states of quality, such as "high quality," "medium quality," or "low quality.” Quality factors are the specific aspects or characteristics of a product or process
that contribute to its overall quality. Examples of quality factors could include performance, reliability, usability, security, maintainability, and so on. A metric could be valid in relation to some validity criteria but invalid in relation to others. The following criteria have been recommended in a report on IEEE standards for software quality metrics [5]:

**Correlation**: A certain threshold must be exceeded when the variation in the quality factor values for a product or process is explained by the variance in the associated metric values.

**Tracking**: A change in the value of a quality factor for a given product or process must be associated with an equivalent change in the metric value if the metric is related to that factor.

**Consistency**: The appropriate metric values must be ranked in the same order as the quality factor values for the associated products or processes.

**Predictability**: If a metric is used to anticipate a product or process’ quality factor, it must do so with a certain degree of precision.

**Discriminative Power**: The ability to distinguish between high-quality and low-quality goods or processes is a requirement for a metric.

**Reliability**: For a defined percentage of the metric’s applications, a metric must exhibit the aforementioned correlation, tracking, consistency, predictability, and discriminative power features.

**Static Code Metrics**

Static code metrics are assessments of potential quality-related software features. These features include – size, complexity, etc. Size is measured by lines of code (LOC) counts and complexity is measured by the linearly independent path. Through the parsing of source code, static code metrics are computed, and their recording can be carried out automatically. Therefore, regardless of their size, it is computationally feasible to calculate the metrics of whole software systems. [29]

Since engineers have a thorough understanding of the system’s vulnerabilities, they can readily identify defective modules and make predictions about the entire system based on metrics.
Because static code metrics are simpler and more commonly utilized, they constitute a reliable option for identifying software flaws.[83] Below are a few listed static code metrics that can be used to predict the bugs [definitions from [83]]

**Weighted methods per class (WMC):** Returns the number of methods of a given class. Depth of inheritance tree (DIT): Provides the number of levels of inheritance beginning with the Object class.

**Number of children (NOC):** Provides the number of descendants of the specific Class.

**Coupling between Object class (CBO):** Calculates the number of classes that are associated with a certain class by inheritance, method calls, arguments, field declarations, and other connections.

**Response for a class (RFC):** Returns the total number of class methods that have been called, and the methods that are included in class methods’ bodies.

**Lines of code (LOC):** Represents the overall count of class fields, methods, and code written in the methods.

**Number of public methods (NPM):** Returns the total number of methods with access modifiers set to public.

**Data access metric (DAM):** Returns the ratio of the number of public/protected attributes to the overall number of all class attributes.

**Measure of aggregation (MOA):** Provides the number of classes that are defined by a user.

**Cohesion among methods of class (CAM):** Provides the number of methods that are related to each other within the class based on the method’s parameter list.

**Inheritance coupling (IC):** Returns the number of base classes to which an examined class is connected.

**Coupling between methods (CBM):** This shows how many inherited methods are connected to altered or new methods in the class.
Average method complexity (AMC): Provides the total number of the average size of methods within some class.

McCabe’s cyclomatic complexity (CC): Identifies the number of different paths within the method plus one, computing all edges, nodes, and related components of the graph. The formula for defining CC is as follows:

\[
CC = E - N + P
\]

where variable E defines the number of edges, N defines nodes, and P defines the connected components of the graph.

Process Metrics

Modern software quality management now focuses on process management. Many software process quality models and standards for software quality assurance have been developed since the 1970s [34], including Spice [23], ISO/IEC12007 [6], CMM [66], and Bootstrap [88]. All the aforementioned models revolve around implementing software process metrics to promote ongoing software process improvement. Process metrics are therefore essential to the management and evaluation of software process capacity [95]. Process metrics are measurements of the software development process, such as the overall length of time spent developing the product, the average level of programming staff expertise, or the type of methodology used [49].

The process metrics that influence the precision of defect prediction have been the subject of extensive research. The most popular metrics among these are NR, NDC, NML, and NDPV. (Definitions after [40], [55]):

Number of Revisions (NR). Measures the count of changes made to a specific class during the development of the investigation’s release.

Number of Distinct Committers (NDC). Measures the count of unique authors—typically developers—who made changes to the class during the development of the software system under investigation.

Number of Modified Lines (NML). The total number of lines of source code that were added
or removed from the class constitutes the NML metric’s value. Every revision made during the development of the studied release of a software system is taken into account. According to the CVS version control system, changing a line of source code entails first removing the previous version and then adding a new one.

**Number of Defects in Previous Version (NDPV).** counts the number of bugs that were fixed in a class during the development of a software system’s prior release. When comparing Source Code Metrics and Process Metrics on a broad scale, Majumder et al. [78] used four different statistical models for prediction across a set of 700 GitHub projects including 722,471 commits. Their findings showed that process measurements excelled over source code metrics when models were designed and tested using both types of metrics.

### Technical Debt Metrics

Software developer Ward Cunningham introduced the term ”technical debt” in 1992. At WyCash, he proposed the metaphor to convey to non-technical stakeholders why budgeting resources for refactoring was necessary [21]. Technical debt (TD) is defined as unfinished or immature artifacts that are present in the software development life cycle and result in greater costs and subpar quality, according to Seaman and Guo [77]. These artifacts might cause rapid development. Low quality does, however, have a tendency to incur costs over time because of the maintenance efforts required for adjustments [58]. We are able to evaluate the risk connected to the software components thanks to our expertise of TD.

There are 13 different categories of technical debt, according to the Institute’s ”Towards an Ontology of Terms on Technical Debt” [11]

### Architecture Debt

Problems with project architecture, such as a breach of modularity, which may have an impact on architectural requirements, are referred to as Architecture Debt (performance, robustness, among others). Typically, this form of debt cannot be settled with straightforward code modifications, necessitating more comprehensive development efforts.
2.1. **Background**

**Build Debt**

Build Debt refers to build-related problems that unduly increase the difficulty and processing time of this task. The customer may see a lot of useless code during the build process of a project. Additionally, the build process will become needlessly slow if it must run ill-defined dependencies. When this happens, a debt buildup might be seen.

**Code Debt**

Code debt is the term used to describe issues with the source code that can make it harder to maintain since they make the code less manageable. Usually, this debt can be determined by looking at the project’s source code and taking into account problems with poor coding skills.

**Defect Debt**

Software projects may have known and unknowing flaws. Defect debt consists of known defects that the CCB agrees should be corrected but must be postponed owing to conflicting priorities and limited resources. Known defects are typically detected by testing activities or by the user and reported on bug track systems. Judgments made by the CCB to put off fixing defects might cause a product to accrue a lot of technical debt, making it more difficult to correct them later.

**Design Debt**

Design debt is debt that can be found by analyzing the source code and spotting the usage of techniques that clearly violated the principles of good object-oriented design (e.g., very large or tightly coupled classes).

**Documentation Debt**

Documentation Debt refers to the issues with software project documentation, which can be found by looking for any type of missing, insufficient, or incomplete documentation. Inadequate documentation falls short of some quality standards for software projects even when it currently functions well in the system.

**Infrastructure Debt**
Infrastructure debt is the term for infrastructure problems that, if they exist in the software organization, may cause some development activities to be delayed or hampered. Delaying an update or infrastructure repair are some examples of this type of debt.

**People Debt**

People’s concerns that exist in the software organization might impede or postpone some development efforts, which is referred to as “people debt.” Expertise concentrated on very few people as a result of postponed hiring and/or training is an example of this type of debt.

**Process Debt**

Process debt is a term used to describe ineffective processes, such as the process that was designed to handle some tasks may be no longer appropriate.

**Requirement Debt**

Trade-offs made regarding the requirements the development team needs to implement or how to achieve them are referred to as requirements debt. This sort of debt includes, among other things, requirements that are partially implemented, implemented but not in all circumstances, and implemented but in a fashion that doesn’t completely satisfy all the non-functional requirements (e.g. security, performance, etc.).

**Service Debt**

Business or technical goals may demand the replacement of web services. A TD that needs to be managed, cleared, and converted from liability to value-added may be introduced by the substitute. Technical debt can cover several dimensions, which are related to the selection, composition, and operation of the service.

**Test Automation Debt**

The effort required to automate tests of previously developed functionality in order to support continuous integration and quicken development cycles is known as test automation debt.

**Test Debt**
2.1. Background

Issues discovered during testing that may compromise the efficacy of those efforts are referred to as test debt. Planned tests that were not executed or known defects in the test suite (e.g. low code coverage) are examples of this type of debt.

2.1.3 Machine Learning

With the use of machine learning (ML), which is a form of artificial intelligence (AI), software programs can predict outcomes more accurately without having to be explicitly instructed to do so. In order to forecast new output values, machine learning algorithms use existing data as input. The way in which a prediction-making algorithm learns to improve its accuracy is a common way to classify traditional machine learning [3]. There are four fundamental strategies: reinforcement learning, semi-supervised learning, unsupervised learning, and supervised learning.

**Supervised learning**

For this sort of machine learning, data scientists describe the variables they want the computer to look for correlations between and provide the algorithms with labeled training data. The algorithm’s input and output are both described.

**Unsupervised learning**

Algorithms used in this sort of machine learning are trained on unlabeled data. The algorithm searches through data sets in search of any significant relationships. Both the input data that algorithms use to train and the predictions or suggestions they produce are predefined.

**Semi-supervised learning**

Semi-supervised learning is a machine learning approach that combines aspects of both supervised and unsupervised learning. In this method, the algorithm is provided with a large amount of labeled training data by data scientists, but it also has the freedom to explore and analyze the unlabeled data to gain additional insights about the dataset.

By having access to both labeled and unlabeled data, the algorithm can leverage the labeled
data to learn the relationships between input features and their corresponding labels, similar to supervised learning. However, what sets semi-supervised learning apart is its ability to autonomously explore the unlabeled data and extract valuable information from it without explicit target or class information.

**Reinforcement learning**

Reinforcement learning is frequently used by data scientists to train a system to finish a multi-step process with well-defined criteria. An algorithm is programmed by data scientists to fulfill a goal, and they provide it with positive or negative feedback as it determines how to do so. However, the algorithm typically chooses the course of action on its own.

Below are some of the well-known algorithms for defect prediction.

**Logistic Regression (LR):** Logistic regression is a supervised classification algorithm whereby the target variable $O$ (i.e. output), can take on values in the interval $[0, 1]$ representing the probability for a given set of input features $I$ to belong to class 1 or 0.

It can be used to predict the likelihood of defects in software development processes. By classifying the data into two categories, such as "defect" or "no defect."

Logistic regression works by fitting a mathematical model to the data that can be used to predict the probability that a given instance belongs to one of the two categories. The model is trained on a dataset of labeled examples, where the input variables (such as process or quality metrics) are used to predict the output label (defect or no defect).

Once the model is trained, it can be used to predict the likelihood of defects in new instances of data. The predicted probability can then be thresholded to generate a binary classification, such as "defect" or "no defect."

**Random Forest (RF):** RF is an ensemble type of learning method used for both classification and regression problems. Random forest works by constructing a large number of decision trees, each of which is trained on a subset of the data. The decision trees are trained to predict the output label (defect or no defect) based on the input variables (such as process or quality
The predictions from the individual decision trees are then combined to generate a final prediction.

**Support Vector Machine (SVM):** SVM is a discriminative classifier formally defined by a separating hyperplane. In SVMs, given a labeled training dataset whereby each data item is marked as belonging to either of two categories, the algorithm outputs an optimal hyperplane, which classifies new unseen data in one of these two categories.

**k-Nearest Neighbors (k-NN):** k-NN is a non-parametric method that can be used for both classification and regression problems. In both cases, the input consists of the k closest training examples in a feature space. The output depends on whether k-NN is used for classification or regression. In classification, the output is to categorize an input to one of the equivalence classes. In regression, the output is to assign a value to the input, usually the average of the values of its closest k-neighbors.

**Neural Networks (NN):** Neural Networks are nonlinear predictive structures that consist of interconnected processing elements called neurons that work together in parallel within a network to produce output, often simulating an unknown function or phenomenon. It is a type of model that is used to perform tasks such as classification, regression, and clustering. Neural networks are inspired by the structure and function of the brain and are composed of multiple neurons that are designed to process and transmit information.

To perform defect prediction using a neural network, you would need to gather data about the product or system, including information about its design, materials, manufacturing process, and any past defects that have been identified. This data would then be used to train the neural network, which would learn to identify patterns and relationships in the data that are indicative of defects.

Once the neural network has been trained, it can be used to make predictions about the likelihood of defects in new products or systems based on their characteristics. These predictions can be used to identify potential issues early on, allowing manufacturers to take preventive measures to avoid defects and improve the reliability of their products.
There are many different types of neural networks that can be used for defect prediction, including feed-forward neural networks, convolutional neural networks, and recurrent neural networks, among others. The choice of neural network architecture will depend on the specific characteristics of the data and the requirements of the task.

**Multi-layer Perceptron (MLP):** MLPs refer to a class of feedforward artificial neural networks (ANN). An MLP is comprised of a directed graph of multiple layers of nodes that are fully connected to the nodes of the next layer. For training purposes, MLP utilizes a supervised learning technique defined as backpropagation.

Apart from supervised learning, unsupervised techniques have also been used in several research papers. Such unsupervised learning techniques are outlined below.

**Self-Organizing Maps (SOM):** SOM is a type of artificial neural network (ANN) that is trained using unsupervised learning to produce a low-dimensional, discretized representation of the input space of the training samples, called a map, and is, therefore, a method to do dimensionality reduction.

**K-Means Clustering (KMeans):** K-means clustering is a method of vector quantization, that aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster.

Both these techniques can be used to partition data into two classes without having to use labeled data. Such techniques have been investigated by Yang et al. [96] to compare their efficacy in performing Just-in-Time Defect Prediction.

### 2.1.4 Evaluating Classification Models

After a classification model has been chosen and developed, it is important to evaluate its performance using a testing dataset that includes data points that were not used in the training process.

There are four different possibilities that could occur while making classification predictions.
True Positive (TP): When the model correctly predicts the positive class for an observation, and the observation actually belongs to the positive class.

True Negative (TN): When the model correctly predicts the negative class for an observation, and the observation truly belongs to the negative class.

False Positive (FP): When the model incorrectly predicts the positive class for an observation, but the observation actually belongs to the negative class.

False Negative (FN): When the model incorrectly predicts the negative class for an observation, but the observation truly belongs to the positive class.

These four outcomes are often plotted on a confusion matrix.

<table>
<thead>
<tr>
<th>Predicted Values</th>
<th>Positive</th>
<th>Negative</th>
<th>TruePositive(TP)</th>
<th>FalsePositive(FP)</th>
<th>FalseNegative(FN)</th>
<th>TrueNegative(TN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negative</td>
<td>FalseNegative(FN)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Confusion Matrix

Accuracy, precision, and recall are the three primary measures used to assess the efficacy of a classification algorithm.

**Accuracy**

The percentage of accurate predictions made using the test data is known as accuracy. It is simple to compute i.e. By dividing the number of accurate predictions by the total number of predictions.

\[
Accuracy = \frac{TP + TN}{TP + TN + FP + FN}
\]  \hspace{1cm} (2.2)

**Precision**

The metric used to evaluate how accurately a classification was made is called precision. This equation, intuitively, represents the proportion of true positive events to all positive predicted events. The precision increases with the fraction, therefore the model’s ability to correctly
classify the positive class is enhanced.

\[ \text{Precision} = \frac{TP}{TP + FP} \]  

(2.3)

Recall

Recall reveals the proportion of accurately detected positive cases among all positive cases.

\[ \text{Recall} = \frac{TP}{TP + FN} \]  

(2.4)

2.1.5 Chi-Square Test

The chi-square test is a statistical method used to determine if there is a significant difference between observed frequencies and expected frequencies in a dataset. It is commonly used to compare categorical data or count data.

The test involves comparing the observed data with the expected data, which is calculated based on the null hypothesis. The null hypothesis states that there is no significant difference between the observed and expected frequencies, and any observed differences are due to chance.

To calculate the chi-square test statistic, the observed data is compared to the expected data, and the differences between them are squared, divided by the expected data, and summed. This results in a single value that represents the degree of difference between the observed and expected frequencies.

The chi-square test statistic is then compared to a critical value from the chi-square distribution table, based on the degrees of freedom and the desired level of significance. If the chi-square test statistic is greater than the critical value, the null hypothesis is rejected, and it is concluded that there is a significant difference between the observed and expected frequencies.

The Formula for Chi-Square is -

\[ \chi^2 = \sum \frac{(O_i - E_i)^2}{E_i} \]

where:
2.1. Background

- $\chi^2$ is the chi-square test statistic
- $O_i$ is the observed value(s)
- $E_i$ is the expected value(s)

The degrees of freedom ($c$) for the chi-square test depends on the number of categories being compared and can be calculated as $c = (r - 1)(c - 1)$, where $r$ is the number of rows and $c$ is the number of columns in the contingency table.

2.1.6 SonarQube

A self-managed, automatic code review tool called SonarQube [2] methodically aids in producing clean code. SonarQube, a key component of our Sonar system, integrates into the current workflow and finds flaws in the code to support ongoing code inspections of the projects. To make sure the code complies with high-quality standards, the tool analyses more than 30 different programming languages and interfaces with the CI pipeline and DevOps platform.

It is among the most popular open-source static code analysis tools for determining static quality. It can be carried out on-site or using the free cloud-based service available at sonarcloud.io. SonarQube determines a variety of metrics, including the count of lines of code and code complexity, and assesses the code’s compliance with a predefined list of ”coding rules.” When a coding rule is violated in the examined source code or a metric exceeds a specified threshold (also known as a ”quality gate”), SonarQube creates an ”issue”. Calculating the remediation cost and the Technical Debt requires taking into account the amount of time required to fix the issues (remediation effort). Each rule is categorized as being linked to the code’s Security, Maintainability, or Reliability. Reliability rules, which are also referred to as ”bugs,” produce TD issues that represent a problem in the code that will eventually be reflected in a bug. Code smells, or maintainability rules, are regarded as ”maintenance-related concerns” in the code that reduce readability and modifiability. It is significant to note that the phrase ”code smells” used in SonarQube refers to a different set of guidelines rather than the well-known code smells identified by Fowler et al. [76] [27]. SonarQube asserts that while Security concerns may contain some false positives, Reliability, and Maintainability rules are expected to have zero
false-positive issues [76].

Additionally, SonarQube categorizes these rules into five severity levels [2].

BLOCKER: High-impact bug that could affect how the application behaves in production. A memory leak or an open JDBC connection, for instance, are BLOCKERS that need to be repaired right away.

CRITICAL: Either a bug that has a low chance of influencing how the application behaves in production or a bug that exposes a security issue. SQL injection or an empty catch block would be a Critical problem. The code has to be reviewed right now.

MAJOR: A fault in the product’s quality could seriously hinder the developer’s efficiency. Examples of Major errors include exposed code, duplicate blocks, and unused parameters.

MINOR: A defect in quality that might have a small negative effect on the developer’s productivity. Line length should be kept to a minimum, and "switch" statements should include at least three cases. These are two examples of minor faults.

INFO: It is merely a finding and neither a bug nor a quality issue. SonarQube also advises analyzing blockers and significant bugs very away.

For the most popular programming languages, such as Java, Python, C++, and JavaScript, SonarQube includes various sets of rules. More than 500 rules for Java are included in SonarQube version 7.5. The Technical Debt Dataset’s (discussed in section 2.1.7) file ”sonar rules.csv” contains the whole list of rules.

2.1.7 Technical Debt Dataset

The Technical Debt Dataset [74] is a curated dataset that includes measurement information from five tools used on every change made to 31 projects from the Apache Software Foundation. This dataset’s purpose is to give researchers access to a common data set so they may compare their findings.

The dataset was created by extracting the data of projects and performing various tool analyses
on it. The GitHub repositories of the projects were cloned to gather the data, which was then used to classify refactorings using Refactoring Miner [91], collect commit information from the git log using PyDriller [82], and extract defect information from issues in Jira. Then, technical debt items were examined with SonarQube, and code smells and anti-patterns were examined with Ptidej [30], respectively. Additionally, by using the implementation of the SZZ method [69], the fault-causing and -fixing commits were found [76]. Entity relationship of the dataset is mentioned in figure 2.1 [76]

This dataset has recently been used by the authors for several publications [20], [50], [51], [73], and [75].

The dataset is simple to obtain because it offers two different methods for accessing the data, a set of CSV files and the SQLite database file, to help with data queries [76].

Figure 2.1: Entity Relationship Diagram of the Technical debt dataset
2.1.8 SZZ Algorithm

The SZZ algorithm, introduced by Sliwerski, Zimmermann, and Zeller [37] at MSR 2005, is widely acknowledged for its effectiveness in identifying bug-inducing commits given a bug-fixing commit. The algorithm operates by analyzing version control data to determine the most recent change (commit) made to each line of source code that was modified in the bug-fixing commit. This is achieved by leveraging the annotation/blame feature of versioning systems. These identified commits are considered as triggers for the bug-fixing commit, indicating their potential role in introducing the bug. [28]

The SZZ algorithm has been extensively utilized in defect prediction techniques [7][8][48][9] and empirical studies investigating bug introduction circumstances [15][54]. However, it is important to acknowledge the false positive limitation when utilizing the SZZ algorithm for defect prediction. False positives can arise due to factors such as code refactoring, where the code structure is modified without altering its functionality. In such cases, the algorithm may mistakenly associate refactoring changes with the introduction of defects. For instance, the algorithm can treat changes to code comments and whitespaces as equivalent to other modifications. Consequently, if a comment is modified in the bug-fixing commit, the most recent change to that comment is incorrectly identified as a bug-inducing commit, even if it does not impact the program’s behavior.

It is essential to be aware of this false positive limitation when utilizing the SZZ algorithm for defect prediction.

2.2 Related Work

2.2.1 Repository Mining

Many different research teams have investigated the benefits of mining software repositories, whether these repositories maintain the change logs of software systems, as is the case for GitHub, or contain complimentary information as is the case for repositories like Bugzilla and Jira.
2.2. Related Work

Mining GitHub

It allows analysts to extract data directly from GitHub repositories and analyze it to identify patterns that are indicative of defects in the software. In [41], Kalliamvakou et al. investigate the quality of data mined from GitHub repositories and provide researchers with a set of recommendations on how to approach such datasets. Additionally, they provide a list of the perils of using GitHub, that were identified during their research. In [35], H. Inayat et al. used repository mining to extract data from GitHub projects, including information on commit history, code changes, and developer activity. They used this data to predict defects in the software development process using machine learning algorithms.

Bug to Bug-Fixing-Commit Link Recovery

Bug-to-bug fixing commit link recovery involves analyzing version control systems to identify the relationship between bugs and the commits that fix them. In [14], Bachmann et al. tackle the problem of link recovery for bug-fixing commits, addressing the threat to validity stemming from using imprecise link recovery techniques. Their method incorporates the extraction of ground truth and the use of this for evaluating a link recovery tool. In [87], Tantithamthavorn et al. investigate the effects of data mislabelling on the overall performance of automated Defect Prediction models. In [99] T. Zimmerman et al. identified the relationship between bugs and the commits that fix them in a version control system. They used this information, along with the process and quality metrics, to predict defects in the software development process.

2.2.2 Machine Learning

A variety of methods have been proposed and assessed for addressing the software bug prediction problem. These methods include decision trees [44], neural networks [39],[89], Naive Bayes [59][92][67], support vector machines [29], Bayesian networks [62] and Random Forests [16]. These have been applied using either Source Code or Process Metrics.

Source Code Metrics Approaches

Deciding whether a component has a high likelihood of being defective or not, has proven to
have a strong correlation with a number of software metrics. Identifying and measuring software metrics is vital for various reasons, including estimating program execution, the required effort for processes, the number of defects during software development, measuring the effectiveness of software processes, and controlling software project processes [72] [90]. Various software metrics have been commonly used for defect prediction, including lines of code (LOC) metrics, McCabe metrics, Halstead metrics, and object-oriented software metrics. Hence, the automated prediction of defective components from extracted software metrics evolved as a very active research area [32]. In [60], Nagappan aims to find the best code metric to predict bugs. The conclusion of this work is that complexity metrics can successfully predict post-release defects, but there is no single set of metrics that is applicable to all systems. Hassan et al. have investigated the impact of different aspects of the modeling process on the end results and the interpretation of the generated models [85][84][46][38][86]. Shakhovska et al. investigate the use of unsupervised learning for performing Defect Prediction utilizing SOMs and Hierarchical Clustering on data from the Promise software engineering research repository [78][80]. In [10] Al-Mamun et al. used static code analysis to extract data from the source code of a software program, including information on code complexity, code coverage, and code maintainability. They used this data, along with machine learning algorithms, to predict defects in the software. The authors found that static code analysis and machine learning were effective at predicting defects in the software and that they had a high accuracy rate. They also found that certain types of data, such as code complexity and code maintainability, were more useful for predicting defects than others.

**Process Metrics Approaches**

In [17], Venkata et al. compared different machine learning models for identifying faulty software modules and they found that there is no particular learning technique that performs the best for all datasets. In [94], Wang and Yao aim to find bugs without decreasing the overall performance of the model. In this process, they find that imbalanced distribution between classes in bug prediction is the root cause of its learning difficulty. Similarly, in [98], Zimmermann et al. propose an approach to predict bugs in cross-language systems. The work examined a large number of such systems and concluded that only 3.4 of the systems had precision and
2.2. Related Work

Recall prediction levels above 75. The authors also tested the influence of several factors on the success of cross-language prediction and concluded that there was no single factor that led to such successful predictions. The authors used decision trees, trained the model, and estimated precision, recall, and accuracy before attempting a prediction across systems. Lastly, in [33], Hassan discusses how frequent source code “commits” in the repository negatively affect the quality of the software system, meaning that the more changes are incurred to a file, the higher the chance that the file will contain critical errors. Furthermore, the author in [33] presents a model which can be used to quantify the overall system complexity using historical code-change data, instead of plain source code features.

Combination of Metrics Approaches

There have been several studies that have examined the use of a combination of static code metrics and process metrics, along with machine learning, for defect prediction in software development processes. O. Chubato et al. [19] investigated by extracting static code metrics and process metrics information. They used this data to predict defects in the software development process using machine learning techniques and found that the combination of static code metrics and process metrics, along with machine learning, was effective at predicting defects in the software development process, and that it had a high accuracy rate. In [56], Majumder et al. performed a large-scale comparison between Source Code Metrics and Process Metrics utilizing four different statistical models for prediction over a collection of 700 GitHub projects comprising 722,471 commits. Their results indicated that when models were trained and tested on source code metrics and process metrics, the process metrics outperformed the source code metrics.

2.2.3 Technical Debt

Technical debt has been the subject of several studies on defect prediction in software development. In [36], M.R. Islam et al. observed that technical debt was significantly correlated with defects in the software development process and that it was a significant predictor of defects. T.C.T. Nguyen et al. [61] examined the relationship between technical debt and defects in software development processes. Their study suggests that technical debt can be an important
factor in defect prediction in software development processes. By analyzing data about the
technical debt in a software program and identifying patterns or trends that are indicative of
defects, it is possible to improve the quality of the software and reduce the number of defects.

2.2.4 Statistical Process Control

Statistical process control (SPC) is a technique used to monitor and control processes in manu-
facturing and other industries, and it has also been applied in the field of software development
for defect prediction. SPC involves collecting data about a process and analyzing it to identify
patterns or trends that may indicate a problem or defect in the process.

SPC has been found to be effective at identifying potential defects in software development
processes. In a study by S.R. Chidamber et al. [18], SPC was used to predict defects in a
software development process, and the authors found that it was effective at identifying potential
defects and could be used to improve the quality of the software. G.M. Kapfhammer et al.
[43] presents a study that compares the effectiveness of different defect prediction models for
identifying potential defects in software systems. The study involved collecting data on various
characteristics of a software development process, such as the number of defects identified, the
number of lines of code, and the number of developers working on the project. The authors then
used several different defect prediction models to analyze the data and identify potential defects.
The models included a statistical process control (SPC) model, a decision tree model, and a
neural network model. The authors compared the performance of these models based on their
accuracy in predicting defects and their ability to identify the root cause of defects. The results
of the study showed that the SPC model had the highest accuracy rate for defect prediction,
followed by the decision tree model and the neural network model. The authors also found that
the SPC model was better at identifying the root cause of defects than the other model.

Hybrid approaches combining SPC and machine learning have been found to be particularly ef-
ficient at predicting defects in software development processes. In a study by G.M. Kapfhammer
et al. [42] a combination of SPC and artificial neural networks was used for defect prediction in
a software development process, and the authors discovered that the combined approach was
effective at predicting defects with high accuracy.
Chapter 3

Pre-Processing Of The Dataset

The initial point of our analysis is the Technical Debt dataset [76] discussed in section 2.1.7. To analyze this dataset for the purpose of our research, it was necessary to pre-process and filter the data to make it suitable for analysis [1]. This involved cleaning the data, formatting it in a specific way, and adding additional information to it. We used existing tables of the dataset and pre-processed them using information from the tables - GIT_COMMITS_CHANGES, SONAR_ANALYSIS, SONAR_MEASURES, SZZ_FAULT_INDUCING_COMMITS.

Preprocessing is an important step in any data analysis project. It helps to ensure that the data is in a format that is easy to work with and that it contains all the necessary information for the analysis. In the case of the Technical Debt dataset, pre-processing was necessary to identify commit periods containing pre-bug-inducing commits, low correlated metrics, and the rate of change of SonarQube metrics as well to exclude false positive bug-inducing commits identified by the SZZ algorithm which is used to compile the SonarQube dataset. ¹

The pre-processing and the consequent analysis are composed of six major steps.

The first step involves the analysis of SQL tables of the dataset and the extraction of bug-inducing / bug-fixing commit pairs (see Section 3.1 below). This allowed us to identify the commits that introduced bugs into the software and their respective commits that fixed those bugs along with the files involved in each such commit pair.

¹The processed dataset for this paper can be found in https://figshare.com/s/be8198c4ab320a1af7c0
The second step of the process is the filtering stage, which aims to refine the list of Bug inducing commits by eliminating the false positives identified by the SZZ algorithm. False positives can arise due to the algorithm’s reliance on heuristics, which can sometimes misidentify commits as Bug inducing. To ensure the accuracy of the final list, the filtering step is essential in removing any such falsely identified commits.

The third step involves the identification of a set of relevant commits for each such bug-inducing and bug-fixing commit pair. These pre-bug-inducing commits contain the files that lie in the intersection of files appearing in the corresponding bug-inducing / bug-fixing commit pair. In other words, the relevant commits are those that modify the same files as the bug-inducing commit and bug-fixing commit. By identifying these commits and further analyzing the pre-bug-inducing commits, it is possible to examine the changes that occurred in the lead-up to the bug-inducing commit, which may provide insights into the causes of the bug.

The fourth step of the process involves the identification of SonarQube metrics which exhibit a pair-wise low correlation across all projects. In this respect, we have identified seven pairs.

The fifth step involves using the metrics determined in the fourth step for our experiments and testing our hypothesis that these correlations change before an imminent bug-inducing commit.

The final step involves the analysis of the rate of change of the SonarQube metrics in each of two periods, 15 to 6 relevant commits and 5 to 1 relevant commits prior to a bug-inducing commit.

The rate of change was calculated by considering the slope of the least squares regression line that interpolates the metric values in two pre-bug-inducing commit periods as will be discussed in the following sections. By examining the rate of change of the metrics, it is possible to identify any patterns or trends that may be related to the introduction of a bug.

Once these pre-processing steps have commenced, then our experiments involved examining:

1. The low correlated metrics behave before an imminent bug-inducing commit.

2. The rate of change of metric values.

3. The rate of change of correlation and metrics together can serve to assess the imminent
3.1 Bug-Inducing and Bug-Fixing Commit Pairs

The first part of pre-processing the data set is to identify bug-inducing and bug-fixing commit pairs, as well as the files involved in these two commits forming this commit pair.

This process step consists of five phases:

In the first phase, the raw data set is analyzed and a new table ZALLPROJECTS_FAULT_INDUCING_COMMITS is created and populated with the information about bug-inducing and bug-fixing pairs, including Project ID, FAULT_INDUCING_COMMIT HASH, FAULT FIXING COMMIT HASH, DATE OF INDUCING COMMIT and DATE OF FIXING COMMIT.

This table is used to store information about all of the bug-inducing and bug-fixing commit pairs in the dataset. This information is extracted from the GIT_COMMITS_CHANGES and SZZ_FAULT_INDUCING_COMMITS tables of the technical debt dataset. Refer to table 3.1 for the sample data.

The second phase involved creating a new table ZALLPROJECTS_GIT_COMMITS_CHANGES and populating the information about individual commits in the dataset. This table includes the project ID, the commit hash, the files involved in the commit, and the date of the commit. The details are obtained from the GIT_COMMIT_CHANGES table of the existing dataset. This table contains information about all the commits in the dataset.

In the third phase, files associated with each of the bug-inducing commit is identified. This is achieved by cross-joining the records for the bug-inducing commits and dates and files information. This allows the process to identify the specific files involved in bug-inducing commits. We created a table ZALLPROJECTS_FILES_INDUCING_COMMITS cross join of ZALLPRO-
Chapter 3. Pre-Processing Of The Dataset

Table 3.1: Sample data of table ZALLPROJECTS_FAULT_INDUCING_COMMITS

<table>
<thead>
<tr>
<th>Project_ID</th>
<th>Fault_Inducing_COMMIT_HASH</th>
<th>Fault_Fixing_COMMIT_HASH</th>
<th>Date_of_Inducing_COMMIT</th>
<th>Date_of_Fixing_COMMIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.apache:archiva</td>
<td>b7c003acb6b2c2a930443c511cb087d2776b4891</td>
<td>fa3688a693bc10bf53e72965027a82df0a8ad267</td>
<td>2011-12-09 00:29:35+00:00</td>
<td>2012-03-01 13:15:45+00:00</td>
</tr>
<tr>
<td>org.apache:archiva</td>
<td>5f9755b5ae526a83eccd21b901e97498b223b776</td>
<td>d4c85abfac8d0920549f20735caf0e4028149c5c</td>
<td>2012-09-26 12:56:58+00:00</td>
<td>2014-09-01 14:04:41+10:00</td>
</tr>
<tr>
<td>org.apache:archiva</td>
<td>9e37277c4a5d4c353f71ec5060951c554f20bab5</td>
<td>d4c85abfac8d0920549f20735caf0e4028149c5c</td>
<td>2012-09-21 22:13:53+00:00</td>
<td>2014-09-01 14:04:41+10:00</td>
</tr>
<tr>
<td>org.apache:archiva</td>
<td>28898793f1c03b12593a1b1f1612f450c634c</td>
<td>d4c85abfac8d0920549f20735caf0e4028149c5c</td>
<td>2013-12-16 06:53:27+00:00</td>
<td>2014-09-01 14:04:41+10:00</td>
</tr>
<tr>
<td>org.apache:archiva</td>
<td>4dd185dade9f58e4c24014d484db7f870c71f4ecb</td>
<td>d4c85abfac8d0920549f20735caf0e4028149c5c</td>
<td>2013-12-04 12:09:22+00:00</td>
<td>2014-09-01 14:04:41+10:00</td>
</tr>
<tr>
<td>org.apache:archiva</td>
<td>b4f1ebe21cfaf75e6b2e860f4c5e68a94b54292d12</td>
<td>d4c85abfac8d0920549f20735caf0e4028149c5c</td>
<td>2014-02-15 11:59:04+00:00</td>
<td>2014-09-01 14:04:41+10:00</td>
</tr>
<tr>
<td>org.apache:archiva</td>
<td>1cbe6c1c3bef14da171be276b34dac4ff9af07</td>
<td>d4c85abfac8d0920549f20735caf0e4028149c5c</td>
<td>2014-02-15 11:59:13+00:00</td>
<td>2014-09-01 14:04:41+10:00</td>
</tr>
</tbody>
</table>

In the fourth phase, we identified the files associated with each of the bug-fixing commits. The same procedure is followed as the third phase but for bug-fixing commits. In this step, we created a table ZALLPROJECTS_FILES_FIXING_COMMITS which is also the cross-join of ZALLPROJECTS_FAULT_INDUCING_COMMITS and ZALLPROJECTS_FILES_INDUCING_COMMITS on FAULT_INDUCING_COMMITS. See table 3.4 for the sample data of ZALLPROJECTS_FILES_FIXING_COMMITS.

In the fifth and final phase, the files in the bug-fixing commit and the bug-inducing commit pair are intersected so that only the files which appear both in the bug-inducing and the bug-fixing commits in a pair, are kept. It is extracted from the intersection of tables of phases.
Table 3.2: Sample data of table ZALLPROJECTS_GIT_COMMITS_CHANGES

<table>
<thead>
<tr>
<th>PROJECT_ID</th>
<th>COMMIT_HASH</th>
<th>FILE</th>
<th>DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2b6be811c1d4f5e81339616208530d486608e42b</td>
<td>pom.xml</td>
<td>2005-11-28 03:15:35+00:00</td>
</tr>
<tr>
<td></td>
<td>af0ddbcfc1e8c2528decdb458f42e16cbe5051</td>
<td>pom.xml</td>
<td>2005-11-28 04:50:31+00:00</td>
</tr>
<tr>
<td></td>
<td>ea640e1803a8535ebfba2ade4ea9272b240067e8</td>
<td>pom.xml</td>
<td>2005-11-29 01:34:22+00:00</td>
</tr>
<tr>
<td></td>
<td>ea640e1803a8535ebfba2ade4ea9272b240067e8</td>
<td>pom.xml</td>
<td>2005-11-29 01:34:22+00:00</td>
</tr>
<tr>
<td></td>
<td>ea640e1803a8535ebfba2ade4ea9272b240067e8</td>
<td>pom.xml</td>
<td>2005-11-29 01:34:22+00:00</td>
</tr>
<tr>
<td></td>
<td>ea640e1803a8535ebfba2ade4ea9272b240067e8</td>
<td>pom.xml</td>
<td>2005-11-29 01:34:22+00:00</td>
</tr>
<tr>
<td></td>
<td>ea640e1803a8535ebfba2ade4ea9272b240067e8</td>
<td>pom.xml</td>
<td>2005-11-29 01:34:22+00:00</td>
</tr>
<tr>
<td></td>
<td>ea640e1803a8535ebfba2ade4ea9272b240067e8</td>
<td>pom.xml</td>
<td>2005-11-29 01:34:22+00:00</td>
</tr>
</tbody>
</table>

This information about these intersecting files is stored in a new table ZALLPROJECTS_FILES_INDUCING_INTERSECTION_FIXING_COMMITS (Table 3.5) with fields PROJECT ID, FAULT_INDUCING_COMMIT_HASH, FAULT_FIXING_COMMIT_HASH, DATE_OF_INDUCING_COMMIT, DATE_OF_FIXING_COMMIT and FILES.

### 3.2 Filtering SZZ Algorithm False Positives

Even though a substantial body of work has been conducted using the Technical Debt dataset, it has been argued that the SZZ algorithm, utilized to compile this dataset, introduces many false positive results [24], [25], [70]. More specifically, the SZZ algorithm has been proposed as a way to identify changes that are likely to introduce bugs. However, it has been shown that the algorithm in many occasions incorrectly identifies refactoring operations as bug-introducing changes [24], [25], [70]. Furthermore, the authors in [71] and in [26] discussed the impact of such false positives in bug prediction accuracy, while the authors in [22] present a framework for evaluating the quality of the results obtained by applying the SZZ algorithm on a system. These limitations and concerns on the SZZ algorithm manifested the need to apply a filtering step before we obtain the final bug-inducing, bug-fixing (BIC-BFC) pairs used for our analysis. More specifically, we sanitized the data set by filtering the BICs identified by the SZZ algorithm in the SonarQube data set. The filtering process is based on reconciling possible BICs with corresponding Jira bug reporting and tracking records. The reconciliation process is depicted in...
Fig. 3.1. Let us assume that the SonarQube Technical Debt dataset contains the bug-inducing bug-fixing pair \((BIC_i, BFC_j)\) (see Fig. 3.1). Then, for each such pair we check whether there is a Jira record \(JBR_i\) occurring within a 10-day period after \(BIC_i\) reporting a bug, and also there is a Jira record \(JFR_i\) occurring within a 10 period after \(BFC_j\) reporting the resolution of a bug. This heuristic provides a very strong constraint as it enforces the occurrence of bug reporting and bug fixing Jira records for each bug-inducing bug-fixing commit pair and aims at discarding any refactoring commits erroneously identified by SZZ as bug-introducing commits while retaining only commits that temporally associate with Jira bug reporting and bug fixing records. Our analysis yielded 40,137 distinct bug-inducing and bug-fixing commit pairs and after filtering we have selected 2,219 distinct bug-inducing bug-fixing commit pairs.
3.3 Pre Bug-Inducing Commits

The purpose of this step of the process is to identify relevant commits and consequently pre-bug-inducing commits. The relevant commits are referred to the commits that contain one or more files that lie in the intersection of the corresponding bug-inducing / bug-fixing commit pair. The motivation behind this type of analysis is to exclude any commits which do not bear any relationship with the bug-inducing commit / bug-fixing commit pair under consideration. By focusing on relevant commits, it is possible to better understand the changes that occurred leading up to the bug-inducing commit and to identify potential causes of the bug.

In this research, we assume that there are no impact or side effects between unrelated commits, that is one commit containing some files does not trigger bugs in commits containing other files. In this pre-processing step, we start from a bug-inducing commit, and we move backward to the past collecting all commits that contain one or more files that lie in the intersection of the corresponding bug-inducing / bug-fixing commit pair. In our experiments (see Chapter 5) we

<table>
<thead>
<tr>
<th>PROJECT_ID</th>
<th>FAULT_INCUING_COMMIT_HASH</th>
<th>FAULT_FIXING_COMMIT_HASH</th>
<th>DATE_OF_INCUING_COMMIT</th>
<th>DATE_OF_FIXING_COMMIT</th>
<th>FIXING_FILES</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.apache:arc</td>
<td>b7c003cb6b2c2a9</td>
<td>3e6b3a59c511bc087</td>
<td>d2776b4891</td>
<td>2011-12-09</td>
<td>00:29:35+00:00</td>
</tr>
<tr>
<td>org.apache:arc</td>
<td>5f9755b5ae525a8</td>
<td>3e6b3a59c511bc087</td>
<td>d2776b4891</td>
<td>2012-09-26</td>
<td>12:56:58+00:00</td>
</tr>
<tr>
<td>org.apache:arc</td>
<td>9e37277c4a5d4c3</td>
<td>5f9755b5ae525a8</td>
<td>3e6b3a59c511bc087</td>
<td>d2776b4891</td>
<td>2012-09-26</td>
</tr>
<tr>
<td>org.apache:arc</td>
<td>28998793f10c3b</td>
<td>12593a1b1f1612</td>
<td>f450cfef34c</td>
<td>2013-12-16</td>
<td>06:53:27+00:00</td>
</tr>
<tr>
<td>org.apache:arc</td>
<td>44dd815dade958</td>
<td>e4c24014d84db71</td>
<td>f870c714fcb</td>
<td>2013-12-04</td>
<td>12:09:22+00:00</td>
</tr>
</tbody>
</table>

Table 3.4: Sample data of table ZALLPROJECTS_FILES_FIXING_COMMITS
have considered two possibilities:

a) Looking back, starting from the bug-inducing commit, collecting five related commits (see Fig. 3.2)

b) Looking back from the bug-inducing commit, collecting ten related commits. (see Fig.3.3)

This step has two phases:

In the first phase, a table ZSONAR_METRICS with all the metrics for each commit is created and populated. This table includes information about the various metrics (Table 3.6) that are collected for each analysis, such as lines of code, cyclomatic complexity, and so on. In the technical debt dataset, the analysis is performed on commits [76]. In order to get the metrics for each commit, we set the COMMIT_HASH values as per the analysis key from the SONAR_ANALYSIS table.
3.3. Pre Bug-Inducing Commits

In the second phase, the relevant commits are identified and classified as either pre-bug-inducing (occurring before the bug-inducing commit) or pre-bug-fixing (occurring between the bug-inducing commit and the bug-fixing commit). For example, in Fig. 3.4 we see that the bug-inducing commit occurs at commit 17, while the bug-fixing commit occurs at commit 21. The uniformly normalized metrics values in the related commits are depicted in the line graph, as shown in Fig. 3.4.

This step in the process is important because it allows for a more focused analysis of the commits that are directly related to the bug-inducing commit and may potentially contain clues about the cause of the bug. By including related commits, it is possible to identify patterns and trends more easily in the data that may help with bug diagnosis and resolution.

To achieve this, we created a new table ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_INDUCTING cross join of table ZALLPROJECTS_FILES_INDUCTING_INTERSECTION_FIXING_COMMITS and ZALLPROJECTS_GIT_COMMITS_CHANGES.

In this final table ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_INDUCING, we added the following columns:
• **IS_COMMIT_HASH_ON_FILES_BEFORE_INDUCING**: It checked if the date of the specific commit is before inducing or not, if yes, it is set zero, else one.

• **IS_COMMIT_HASH_ON_FILES_AFTER_INDUCING_BEFORE_FIXING**: To check if the specific commit occurred after inducing commit and before fixing commit.

• **IS_COMMIT_HASH_ON_FILES_INDUCING_COMMIT**: To check if the specific commit is inducing commit or not.

• **IS_COMMIT_HASH_ON_FILES_FIXING_COMMIT** – To check if the specific commit is fixing commit or not.

• **PAIR_NUMBER** – To assign a unique identifier to each group’s relevant commits.

• **NCLOC, NEW_SQALE_DEBT_RATIO, VIOLATIONS, CODE_SMELLS, FUNCTIONS, LINES, CLASSES, CLASS_COMPLEXITY** – All these measures are added.
3.4 Metrics Correlation Values

The purpose of this step is to identify those metrics which collectively exhibit low correlation across all projects. The raw data set consists of 29 metrics, also known as features, for 22 different projects. The commit statistics for all these 22 projects are depicted in Table 3.8.

To identify the metrics with low correlation, the process considers all possible combinations and runs the Pearson correlation test for all 22 projects. The Pearson correlation test is a statistical test that measures the strength and direction of a linear relationship between two variables. In

Finally, we sorted the commits, as per the DATE of occurrence and PAIR_NUMBER. Please see table 3.7 for the example of the final data extracted for one of the pairs.
Chapter 3. Pre-Processing Of The Dataset

Figure 3.4: Example of metrics values in commits prior to a bug-inducing commit (commit 17) and between the bug-inducing and the bug-fixing commit (commit 18-21)

In this case, the variables being tested are the metrics.

The threshold for low correlation is set to 0.6, meaning that correlations between 0 and 0.6 or -0.6 and 0 are considered low. After running the tests, the analysis yields seven metrics that have a low correlation with the New SQALE Debt Ratio metric, as shown in Table 3.9.

By excluding metrics with high correlation, we are avoiding redundancy and enhancing the effectiveness of identifying patterns and trends in the data.

3.5 Correlation and Rate of Change of Metrics

In addition to identifying pairs of metrics that exhibit overall low correlation, we examined:

a) Correlation of five and ten commits before bug-inducing commit.

b) The difference in the rate of change of each of the correlations of these seven metrics with
3.6 Combined Metric Value and Correlation Value Rate of Change

New SQALE Debt Ratio in the period of the 15th to the 6th relevant commit prior to a bug-inducing commit, and the period from the 5th to just one relevant commit occurring prior to the bug-inducing commit.

c) The difference in the rate of change of each of the seven (including New SQALE debt Ratio) metrics in the two aforementioned periods (15 - 6 commits and 5 – 1 commits prior to a bug-inducing commit).

3.6 Combined Metric Value and Correlation Value Rate of Change

In this type of analysis, we consider the impact of both the rate of change combined with the rate of change in the correlation values in any of the seven pairs discussed in Section 3.5. As it will be presented later in Chapter 5, we considered two types of experiments

a) 10% or more difference in the rate of change in correlation values AND 10% or more difference in the rate of change in raw metric values.

b) 25% or more difference in the rate of change in correlation values OR 25% or more difference in the rate of change in raw metric values.
<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS_COMPLEXITY</td>
<td>Class complexity</td>
</tr>
<tr>
<td>NEW_LINES_TO_COVER</td>
<td>New lines to cover</td>
</tr>
<tr>
<td>VIOLATIONS</td>
<td>Violations</td>
</tr>
<tr>
<td>NEW_VIOLATIONS</td>
<td>New violations</td>
</tr>
<tr>
<td>SQALE_RATING</td>
<td>SQALE rating</td>
</tr>
<tr>
<td>SQALE_DEBT_RATIO</td>
<td>SQALE debt ratio</td>
</tr>
<tr>
<td>NEW_SQALE_DEBT_RATIO</td>
<td>New SQALE debt ratio</td>
</tr>
<tr>
<td>CODE_SMELLS</td>
<td>Code smells</td>
</tr>
<tr>
<td>NEW_CODE_SMELLS</td>
<td>New code smells</td>
</tr>
<tr>
<td>EFFORT_TO_REACH_MAINTAINABILITY_RATING_A</td>
<td>Effort to reach maintainability rating</td>
</tr>
<tr>
<td>BUGS</td>
<td>Bugs</td>
</tr>
<tr>
<td>NEW_BUGS</td>
<td>New bugs</td>
</tr>
<tr>
<td>RELIABILITY_REMEDICATION_EFFORT</td>
<td>Reliability remediation effort</td>
</tr>
<tr>
<td>NEW_RELIABILITY_REMEDICATION_EFFORT</td>
<td>New reliability remediation effort</td>
</tr>
<tr>
<td>RELIABILITY_RATING</td>
<td>Reliability rating</td>
</tr>
<tr>
<td>NEW_RELIABILITY_RATING</td>
<td>New reliability rating</td>
</tr>
<tr>
<td>VULNERABILITIES</td>
<td>Vulnerabilities</td>
</tr>
<tr>
<td>NEW_VULNERABILITIES</td>
<td>New vulnerabilities</td>
</tr>
<tr>
<td>SECURITY_REMEDICATION_EFFORT</td>
<td>Security remediation effort</td>
</tr>
<tr>
<td>NEW_SECURITY_REMEDICATION_EFFORT</td>
<td>New security remediation effort</td>
</tr>
<tr>
<td>SECURITY_RATING</td>
<td>Security rating</td>
</tr>
<tr>
<td>NEW_SECURITY_RATING</td>
<td>New security rating</td>
</tr>
<tr>
<td>CLASSES</td>
<td>Classes</td>
</tr>
<tr>
<td>FILES</td>
<td>Files</td>
</tr>
<tr>
<td>FUNCTIONS</td>
<td>Functions</td>
</tr>
<tr>
<td>COMMENT_LINES_DENSITY</td>
<td>Comment lines density</td>
</tr>
<tr>
<td>LINES_TO_COVER</td>
<td>Lines to cover</td>
</tr>
<tr>
<td>LINES</td>
<td>Lines</td>
</tr>
<tr>
<td>NCLOC</td>
<td>NCLOC</td>
</tr>
</tbody>
</table>
3.6. Combined Metric Value and Correlation Value Rate of Change

Table 3.7: Sample data of ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_INDUCING for one pair

<table>
<thead>
<tr>
<th>PROJECT_ID</th>
<th>COMMITS_BEFORE_FAULT</th>
<th>COMMITS_AFTER_FAULT</th>
<th>LOC BEFORE_FAULT</th>
<th>LOC AFTER_FAULT</th>
<th>DIFF LOC</th>
<th>MR BEFORE_FAULT</th>
<th>MR AFTER_FAULT</th>
<th>DIFF MR</th>
<th>CLASS BEFORE_FAULT</th>
<th>CLASS AFTER_FAULT</th>
<th>DIFF CLASS</th>
<th>PATH BEFORE_FAULT</th>
<th>PATH AFTER_FAULT</th>
<th>DIFF PATH</th>
<th>Fault Id</th>
<th>Fault Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.8: Project Statistics

<table>
<thead>
<tr>
<th>Project Name</th>
<th>Average LOC per Commit</th>
<th>Total # of Commits</th>
<th># of Commits Analyzed</th>
<th># of Distinct Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>batik</td>
<td>213850.96</td>
<td>83725</td>
<td>2164</td>
<td>14434</td>
</tr>
<tr>
<td>commons-beanutils</td>
<td>55390.82</td>
<td>7549</td>
<td>1210</td>
<td>499</td>
</tr>
<tr>
<td>commons-collections</td>
<td>21615.37</td>
<td>5663</td>
<td>1730</td>
<td>358</td>
</tr>
<tr>
<td>commons-cli</td>
<td>13314.65</td>
<td>3465</td>
<td>847</td>
<td>473</td>
</tr>
<tr>
<td>commons-io</td>
<td>33935.05</td>
<td>9986</td>
<td>1910</td>
<td>702</td>
</tr>
<tr>
<td>commons-jelly</td>
<td>55999.83</td>
<td>12318</td>
<td>1937</td>
<td>816</td>
</tr>
<tr>
<td>commons-jxl</td>
<td>18655.81</td>
<td>663</td>
<td>1532</td>
<td>599</td>
</tr>
<tr>
<td>commons-configuration</td>
<td>83412.97</td>
<td>12962</td>
<td>2929</td>
<td>1533</td>
</tr>
<tr>
<td>commons-daemon</td>
<td>2347.61</td>
<td>2940</td>
<td>981</td>
<td>249</td>
</tr>
<tr>
<td>commons-dbcp</td>
<td>24424.94</td>
<td>6878</td>
<td>1555</td>
<td>352</td>
</tr>
<tr>
<td>commons-dhtus</td>
<td>8185.04</td>
<td>1861</td>
<td>662</td>
<td>153</td>
</tr>
<tr>
<td>commons-digester</td>
<td>28863.58</td>
<td>9325</td>
<td>2142</td>
<td>1330</td>
</tr>
<tr>
<td>felix</td>
<td>26157.01</td>
<td>16648</td>
<td>3490</td>
<td>29600</td>
</tr>
<tr>
<td>httpcomponents-client</td>
<td>79191.72</td>
<td>24900</td>
<td>2714</td>
<td>1400</td>
</tr>
<tr>
<td>httpcomponents-core</td>
<td>54806.66</td>
<td>2633</td>
<td>1901</td>
<td>1666</td>
</tr>
<tr>
<td>commons-jxpath</td>
<td>35688.13</td>
<td>3972</td>
<td>596</td>
<td>335</td>
</tr>
<tr>
<td>commons-net</td>
<td>47690.75</td>
<td>9551</td>
<td>2699</td>
<td>599</td>
</tr>
<tr>
<td>santuario</td>
<td>124643.12</td>
<td>50338</td>
<td>2718</td>
<td>9162</td>
</tr>
<tr>
<td>commons-vfs</td>
<td>46029.78</td>
<td>15343</td>
<td>2079</td>
<td>714</td>
</tr>
<tr>
<td>zookeeper</td>
<td>97243.80</td>
<td>17789</td>
<td>222</td>
<td>1956</td>
</tr>
<tr>
<td>thrift</td>
<td>27217.14</td>
<td>27566</td>
<td>1944</td>
<td>2844</td>
</tr>
</tbody>
</table>

Table 3.9: Correlation of six metrics with New SQALE Debt Ratio

<table>
<thead>
<tr>
<th>Metric</th>
<th>Correlation Coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCLOC</td>
<td>-0.22</td>
</tr>
<tr>
<td># of Style Violations</td>
<td>-0.19</td>
</tr>
<tr>
<td># of Code Smells</td>
<td>-0.19</td>
</tr>
<tr>
<td>LOC</td>
<td>-0.22</td>
</tr>
<tr>
<td># of Classes</td>
<td>-0.18</td>
</tr>
<tr>
<td>Avg. Class Complexity</td>
<td>-0.07</td>
</tr>
<tr>
<td># of Functions</td>
<td>-0.18</td>
</tr>
</tbody>
</table>
Chapter 4

Evaluation of the likelihood of Bug Inducing Commit

Our analysis towards developing a framework for assessing imminent bug-inducing commits is focusing on two main directions.

The first direction deals with the analysis of how the pair-wise correlation values of each of the low correlated pairs of process, quality, and source code metrics behave before a bug-inducing commit as compared to any other period of the system’s lifetime.

The second direction of analysis involves examining the difference in the rate of change of the raw values of these eight (including the New SQALE Debt Ratio) as well as the difference in the rate of change in the correlation values of any of the seven pairs of low correlated metrics (the seven metrics correlated with New SQALE debt Ratio) before a bug inducing commit, compared to any other period of the system’s lifetime.

The goal of this analysis is to see if there are any patterns or changes in the correlation or the rate of change of these metrics that may indicate the presence of a bug-inducing commit. By analyzing both the raw values and the correlation values of these metrics, the framework aims to provide a more comprehensive understanding of the changes occurring in the system before a bug-inducing commit.

The sections below discuss in more detail these two different types of analyses.
4.1 Pair-wise metrics correlation analysis

The pair-wise metrics correlation analysis is composed of two phases:

a) Overall correlation analysis, where we identified the pairs of metrics that exhibit low correlation with New SQALE Debt Ratio across a system’s lifetime (see Table 3.9)

b) Commit-pairs correlation analysis, where we analyzed the correlation values of metrics of all commits in individual commits-groups.

c) Pre-bug-inducing commit correlation analysis, where we evaluated the change of behavior of correlation values of metrics of these pairs just before a bug-inducing commit.

We will look into the details of each of the phases in further sub-sections.

4.1.1 Overall correlation analysis

Based on data preprocessing discussed in Chapter 3 first we run the Pearson correlation test to identify pairs of metrics that exhibit low correlation values. The Pearson correlation test is a statistical measure that can be used to determine the strength and direction of the relationship between two continuous variables. In the context of this analysis, the Pearson correlation test is being used to identify pairs of metrics in a dataset that exhibit low correlation values.

Considering the nature of the problem (that is not a high-risk mission-critical problem) and the statistical guidelines, we consider that any absolute correlation value between two metrics that is less or equal to 0.6 is classified as a low correlation, while any absolute value greater than 0.6 as high correlation.

Our analysis started by considering 29 metrics (see Chapter 3) and we evaluated the correlation values for each pair of these 29 metrics for a total of 406 pairs. We calculated the correction matrix for each of the projects. Refer to table 4.1 for one of the project metrics “httpcomponents-client”.

Out of these 406 pairs we have identified seven pairs of metrics that exhibit low correlation between them and across all twenty-two projects. These pairs include the New SQALE Debt
4.1.2 Commit-groups correlation analysis

This analysis is based on the fact that if a bug-inducing commit is followed by one or more bug-fixing commits or vice versa, we refer them as separate commit groups. For example, if for \( BIC_i \) bug inducing commit has \( BFC_i \) and \( BFC_i \) as Bug fixing commits, in that case, there would be two different commit-pairs \(<BIC_i, BFC_i>\) and \(<BIC_i, BFC_i>\), or \(<BIC_j, BFC_j>\) and \(<BIC_k, BFC_k>\), if \( BIC_j \) and \( BIC_k \) have single bug-fixing commit \( BFC_{j,k} \).

As depicted in Fig.3.2 and Fig. 3.3, we consider two periods, one before a bug-inducing commit \( BIC_i \), and another period defined as the commits between a bug-inducing commit \( BIC_i \) and its corresponding bug-fixing commit \( BFC_i \) for the specific bug ‘i’ introduced in the bug-inducing commit \( BIC_i \).

Let us assume that the bug-fixing commit \( BFC_i \) contains the files \( S_1 = F_{i,1} \ldots F_{i,m} \) and the bug-inducing commit contains the files \( S_2 = F_{i,m} \ldots F_{i,w} \) where \( m, n \in [1, 2, \ldots n] \). Let us also assume that the intersection \( S = S_1 \cap S_2 = \{F_{ik}, \ldots F_{ip}\} \). In the twenty-two systems, we have
analyzed the the cardinality of set S was mostly 1 and rarely 2 or more.

We determined the commits that have actioned upon the same files and assigned them to the same commit groups. An example of the values of NCLOC of one commit group is depicted in Fig. 3.4.

As per the date and time of execution, the commits are divided into four categories within a commit group.

a) Pre-bug-inducing commits (1-16 commit in Fig. 3.4)

b) Bug inducing commit (17th Commit in Fig. 3.4)

c) Post-bug inducing commit and pre-bug fixing commit (18th to 20th commit in Fig. 3.4)

d) Bug-fixing commit (21st Commit in Fig. 3.4)

Finally, we determined the Pearson correlation of the seven metrics pairs values of all the commits for individual commits-groups.

4.1.3 Pre Bug-Inducing commit correlation value analysis

We define a period P of commits before the bug-inducing commit $BIC_i$ as the period that contains commits that occur before $BIC_i$, and these commits contain occurrences of any (or all) of the common files of inducing and fixing commits $F_{i,k}, \ldots F_{i,p}$ (discussed in section 4.1.2)

For our analysis, we considered two options for calculating the duration of the period P before a bug-inducing commit $BIC_i$. The first option is to consider the first five commits prior to $BIC_i$ that satisfy this file intersection property, and analyzed the Pearson Correlation for those five commits before bug-inducing commit of all the commit-groups. For example, in Fig. 4.1 we can see the correlation of NCLOC and New SQALE Debt Ratio is high five commits before bug-inducing commit.

The second option was to consider the first ten commits prior to $BIC_i$ that satisfy this property, and determined the Pearson Correlation for these ten commits before bug-inducing commit of all the commit-groups.
Chapter 4. Evaluation of the likelihood of Bug Inducing Commit

4.2 Pre Bug-Inducing Commit Rate of Change Analysis

As before, here we also consider two types of analyses.

In the first type we evaluate how the rate of change of metric values in a period of five commits prior to a bug-inducing commit (i.e. commits $PC_{i,5} \ldots PC_{i,1}$ compares to the rate of change of metric values in a period of ten commits prior to $PC_{i,5}$, that is commits $PC_{i,15} \ldots PC_{i,6}$ (See Fig. 4.2 and Fig. 4.3).

In the second type of analysis, we consider both the absolute change of the correlation values (see Section 4.1.3) either OR-combined or AND-combined with the difference in the rate of change of any of the six raw metric values.

These two types of analysis are discussed in more detail below.
4.2. PRE BUG-INDUCING COMMIT RATE OF CHANGE ANALYSIS

4.2.1 Metric Values-Only Rate of Change Analysis

In this type of analysis, we don’t focus on whether we have a change in the correlation values before a bug-inducing commit, but instead we look at the the difference in the rate of change of the individual raw values of the seven metrics in two-period segments before a bug-inducing commit.

More specifically, if we assume a bug-inducing commit $BIC_i$, we break the pre-bug-inducing commit period in two segments.

- The first segment consists of five relevant to the commit-pair commits, that is five commits prior to the bug-inducing commit $BIC_i$ ($PC_{i,5} \ldots PC_{i,1}$).
- The second segment is a ten-commit period ($PC_{i,15} \ldots PC_{i,6}$) prior to the first segment ($PC_{i,5} \ldots PC_{i,1}$).

We then compare the difference in the rate of change of the raw metric values in the first segment (i.e. immediately prior to the bug inducing commit $BIC_i$ - commits $PC_{i,5} \ldots PC_{i,1}$) versus the rate of change of the same metric in the second segment (i.e. commits $PC_{i,15} \ldots PC_{i,6}$). These two segments are depicted in Fig. 4.2. The rate of change is evaluated by computing the slope.

Figure 4.2: Schematic of relevant commits in segment $PC_{i,15} \ldots PC_{i,6}$ and relevant commits in segment $PC_{i,5} \ldots PC_{i,1}$ occurring prior to a bug-inducing commit $BIC_i$. 
Figure 4.3: Example of the rate of change (slope) of metric New SQALE Debt Ratio in the segment of 15 - 6 commits prior to bug-inducing commit #16 and the segment of 5 commits prior to the bug-inducing commit #16

of the least squares regression line in the first segment and in the second segment as depicted in Fig. 4.3.

The following steps are implemented in order to calculate the slope of least square regression line -

**Step 1: Normalizing the metric values of commits.**

We normalized the metric values with respect to the number of commits to avoid the inaccuracy of the slope calculation due to the wide range of metrics values. Metrics often vary significantly in magnitude, with some having values in a much larger range compared to others. When these raw, unnormalized metric values are plotted against the number of commits on a graph, the
resulting plot can appear distorted. The x-axis (representing the number of commits) may have relatively small values, while the y-axis (representing metric values) may have much larger values. This disparity in scales can lead to the slope being calculated as an excessively high value, even when the actual relationship between the metrics and commits doesn’t warrant such steepness. To address this issue, we adopted a normalization approach that scales the metric values down to a standardized range based on the number of commits involved in the calculation. By doing so, the y-axis values are brought into alignment with the x-axis values, ensuring that they are comparable and compatible for accurate slope calculations.

The metric values are normalized in the range of 0 to (n-1) where n is the number of commits for which the slope should be calculated. That is, 0 to 4 for calculating the slope five commits prior to bug inducing commit and 0-9 for calculating the slope of ten commits before the bug-inducing commit.

\[
\text{Normalized Value} = \frac{\text{Metric Value} - \min(\text{Metric Values})}{\max(\text{Metric Values}) - \min(\text{Metric Values})} \times (\text{No. of Commits involved} - 1)
\]

**Step 2: Calculating the Slope**

The slope is calculated of normalized metric values

\[
slope = \frac{N \sum xy - \sum x \sum y}{N \sum x^2 - (\sum x)^2}
\]

Where N is the number of commits for which slope is to calculated; x is the serial number of the commit and y is the metric value of the commit.

**Step 3: Calculating the degree of slope**

Determined the degree of slope using atan and toDegrees function of Java

\[
\text{Degree of slope} = \text{toDegrees}(\text{atan}(\text{slope}))
\]
We then examine whether we observe a significant change in the slope between the first and second segments. For example, in Fig. 4.3 we can see the slope of the least square regression line for the New SQALE Debt Ratio metric in the two segments prior to a bug-inducing commit (commit #16).

### 4.2.2 Combined Metric Value and Correlation Value Rate of Change Analysis

In this type of analysis, we consider indicators of an upcoming bug-inducing commit, either an OR or AND combination of both the difference in the rate of change in the correlation values of any of the seven pairs of metrics discussed in section 4.1 and the difference in the rate of change of the values of any of the raw values of the seven metrics which are low correlated with New SQALE Debt Ratio and, as these differences are observed in the two pre-bug-inducing commit segments \(PC_{i,15}, \ldots PC_{i,6} \) and \(PC_{i,5} \ldots PC_{i,1}\). More specifically, we examined as indicators:

a. 10% or more difference in the rate of change of raw metrics values AND-combined with 10% or more difference of rate of change in any of the seven correlation values.

b. 25% or more difference of rate of change of raw metrics values of the aforementioned seven metrics OR-combined with 25% or more difference of the rate of change in any of the seven correlation values. The results of this analysis are discussed in Chapter 5 and depicted in Table 5.4.
Chapter 5

Experimental Results

There were two sets of experiments to identify patterns or changes in the behavior of certain metrics and correlations in the period leading up to a bug-inducing commit, the results of which will be discussed in this chapter.

1. The first set of experiments aimed to identify what is the behavior of otherwise, low correlated metrics five and ten commits prior to a bug-inducing commit, compared to other periods of the system.

   Our hypothesis here is that metrics that otherwise exhibit a low correlation, they start exhibiting higher correlation in a period of five relevant commits prior to a bug-inducing commit. For completeness, we have also tested what is the change of behavior in a period of ten relevant commits prior to a bug-inducing commit.

   As it was discussed above, a relevant commit to a bug-inducing commit is a commit that occurs prior to a bug-inducing commit and contains one or more files which lie in the intersection of files contained on the bug-inducing and corresponding bug-fixing commit.

   To test this hypothesis, we examined the behavior of the metrics of five and ten commits leading up to a bug-inducing commit, and compared it to entire periods in the system. We then analyzed the data to see if there was any change in the correlations between the metrics in the period leading up to a bug-inducing commit.

2. The second set of experiments aimed to identify what is the difference of the rate of change
of a set of selected metrics and the behavior of the rate of change of correlation values in each of the seven pairs of metrics in the two-period segments prior to a bug-inducing commit. For the experiments we have considered a segment between five to one relevant commits prior to a bug-inducing commit and a segment between fifteen to six relevant commits prior to a bug-inducing commit.

Our hypothesis is that differences in the rate of change in the second period (i.e. fifteen to six commits prior to a bug-inducing commit) and the first period (i.e. five and one relevant commits prior to a bug-inducing commit) is an indicator of the likelihood of an impending bug-inducing commit.

Here we have looked at four different cases.

(a) The first case examined was how a 10% of more difference in the rate of change (i.e. slope) of selected metric values differs between the two-period segments and whether this change of behavior can be considered as an indicator of an imminent bug-inducing commit.

(b) The second case examined was how a 25% of more difference in the rate of change of any of the six correlation values differs in these two period segments.

(c) The third case examined whether a 10% or more difference in the rate of change of any of these seven metrics values AND combined with a 10% or more difference in the rate of change in any of the correlation values can be used as a indicator of an impending bug-inducing commit.

(d) Finally, the fourth case examined whether a 25% or more difference in rate of change of any of these seven metrics values OR combined with a 25% or more difference in the rate of change in any of the correlation values can be used as an indicator of an impending bug-inducing commit.

In the next sections, we will delve into the specifics of the two experimental sets previously mentioned. We will examine the details and results of each set of experiments, and discuss the implications and potential applications of the findings. By examining the results of these
5.1 Correlation Behavior as an Indicator of a Bug-Inducing Commit

This experiment aimed to investigate whether the correlation behavior of certain metrics, or features, in software development projects, could serve as indicators of bug-inducing commits. This experiment was conducted in five phases.

In the first phase, we computed the pair-wise Pearson correlation values among all combinations of pairs of all the 29 different metrics (i.e., features) in the Technical Debt dataset[76], comprised of data extracted from 31 projects. Three projects did not have information on bug-inducing and bug-fixing pairs and were omitted from our analysis. Six more projects were omitted because we were able to identify only three pairs of features that exhibited low correlation (i.e., 0.6 or -0.6), when these projects were included in our analysis. By excluding these 6 projects, we were able to identify seven pairs of features that exhibited low correlation with the New SQALE Debt Ratio feature, and therefore our analysis considered 22 open-source projects from the aforementioned dataset.

These seven pairs comprised of the New SQALE Debt Ratio feature with –

a) NCLOC

b) Number of Style Violations

c) Number of Code Smells

d) LOC

e) Number of Classes
f) Average Class Complexity

g) Number of Functions

The average, Pearson correlation values for these pairs are listed in Table 3.9.

In the second phase, we have identified bug-inducing and bug-fixing commit pairs across the lifetime of each of the 22 projects we have examined. For each such commit pair, we have identified the files that lie in the intersection of files appearing in the bug-inducing and the bug-fixing commit, as discussed in Chapter 3. In all 22 projects, we have identified 40195 distinct bug-inducing, bug-fixing commit pairs. After filtering due to reported limitations of the SZZ algorithm (see Section 3.2) we have selected 2,219 distinct bug-inducing bug-fixing commit pairs.

In the third phase, we determined the Pearson correlation of all the commits involved in each commit pair.

In the fourth phase, for each bug-inducing commit (of a commit-pair), we have identified five relevant commits which occur prior to this bug-inducing commit.

In the fifth phase, we examined whether there was any change of behavior of correlation scores in each of these seven pairs of features, five or ten relevant commits occurring prior to a bug-inducing commit.

The results of our analysis are depicted in Table 5.1 for five commits prior to bug-inducing commit and in Table 5.2 for ten commits prior to a bug-inducing commit. More specifically, in the top left most cells in Table 5.1, we observe that the probability (measured as a frequency of cases) for NCLOC have a correlation greater or equal of 0.6 with New SQALE Debt Ratio is 44.25%, while 5 related commits before a bug-inducing commit the probability of the correlation of NCLOC have a correlation greater or equal of 0.6 with \textit{New SQALE Debt Ratio} jumps to 77.86%. Note also that as depicted in Table 3.9, the average correlation across all 22 projects of NCLOC with \textit{New SQALE Debt Ratio} is -0.22. The numbers in parentheses in Table 5.1 depict the probability of a metric having a correlation greater than 0.6 (or 0.7, or 0.75) five commits prior to a non-BIC. For example, in cell (3,2) of Table 5.1 the probability of NCLOC having
5.1. **Correlation Behavior as an Indicator of a Bug-Inducing Commit**

Table 5.1: Behavior of correlations of the six metrics with *New SQALE Debt Ratio* in five relevant commits before a bug-inducing commit

<table>
<thead>
<tr>
<th>Metric</th>
<th>Probability of correlation ≥ 0.6 5 commits before BIC, and before a non BIC (in parentheses)</th>
<th>Probability of correlation ≥ 0.6 across all project commits</th>
<th>Probability of correlation ≥ 0.7 5 commits before BIC, and before a non BIC (in parentheses)</th>
<th>Probability of correlation ≥ 0.7 across all project commits</th>
<th>Probability of correlation ≥ 0.75 5 commits before BIC, and before a non BIC (in parentheses)</th>
<th>Probability of correlation ≥ 0.75 across all project commits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NCLOC</strong></td>
<td>77.86% (76.11%)</td>
<td>44.25%</td>
<td>75.21% (71.35%)</td>
<td>37.58%</td>
<td>71.67% (67.54%)</td>
<td>36.01%</td>
</tr>
<tr>
<td><strong>Violations</strong></td>
<td>82.58% (73.17%)</td>
<td>46.53%</td>
<td>76.28% (67.77%)</td>
<td>40.35%</td>
<td>73.21% (64.32%)</td>
<td>37.60%</td>
</tr>
<tr>
<td><strong>Code Smells</strong></td>
<td>82.44% (72.58%)</td>
<td>46.75%</td>
<td>75.99% (67.18%)</td>
<td>40.13%</td>
<td>73.33% (64.39%)</td>
<td>37.29%</td>
</tr>
<tr>
<td><strong>LOC</strong></td>
<td>78.85% (75.35%)</td>
<td>44.48%</td>
<td>75.21% (70.85%)</td>
<td>39.57%</td>
<td>72.08% (68.44%)</td>
<td>36.23%</td>
</tr>
<tr>
<td><strong>No. of Classes</strong></td>
<td>81.30% (75.69%)</td>
<td>55.16%</td>
<td>78.15% (71.19%)</td>
<td>44.62%</td>
<td>75.49% (68.69%)</td>
<td>40.37%</td>
</tr>
<tr>
<td><strong>Avg. Class Complexity</strong></td>
<td>64.79% (60.72%)</td>
<td>44.94%</td>
<td>57.59% (52.25%)</td>
<td>32.45%</td>
<td>55.43% (50.8%)</td>
<td>26.13%</td>
</tr>
</tbody>
</table>

A correlation with *New SQALE Debt Ratio* greater or equal to 0.6 five commits before a BIC is 82.58%, while BIC that the probability of NCLOC having a correlation with *New SQALE Debt Ratio* greater or equal to 0.6 five commits before a non BIC is 73.17%. Even though these results may appear not promising, we observe the same behavior in the correlation of all metrics with the *New SQALE Debt Ratio*, that is the correlation increases on average by 5 to 6 percent before a BIC in all cases and in all projects. The cases that exhibit the highest differences in values as compared to non-BIC are depicted in bold numbers in Table 5.4.

We obtained similar results when we performed the same experiment, but by considering the behavior of correlations in a period of 10 relevant commits. The results indicate that there is a change in the correlation but not as strong as when a period of 5 related commits is considered. More specifically, Table 5.2 (see last two columns) depicts that even though the probability of having a correlation of NCLOC with *New SQALE Debt Ratio* be greater or equal to 0.75 across the lifetime of all systems is approximately 38%, this probability jumps to approximately 60.2% in the period of ten related commits before a bug-inducing commit.

However, here correlations increased in the period of five commits before a BIC as compared to five commits prior to a random non-BIC on average by 9 percent in all cases.

Overall, these results suggest that certain metrics exhibit a high correlations in the period leading
up to a bug-inducing commit, and could potentially be used as indicators of bug-inducing
commits.

Table 5.2: Behavior of correlations of the six metrics with *New SQALE Debt Ratio* in ten
relevant commits before a bug-inducing commit

<table>
<thead>
<tr>
<th>Metric</th>
<th>Correlation ≥ 0.6 10 commits before BIC, and before a non BIC (in parentheses)</th>
<th>Correlation ≥ 0.6 across all project commits</th>
<th>Correlation ≥ 0.7 10 commits before BIC, and before a non BIC (in parentheses)</th>
<th>Correlation ≥ 0.7 across all project commits</th>
<th>Correlation ≥ 0.75 10 commits before BIC, and before a non BIC (in parentheses)</th>
<th>Correlation ≥ 0.75 across all project commits</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCLOC</td>
<td>68.82% (60.42%)</td>
<td>44.25%</td>
<td>63.34% (55.22%)</td>
<td>37.58%</td>
<td>59.26% (51.87%)</td>
<td>36.01%</td>
</tr>
<tr>
<td>Violations</td>
<td>68.51% (56.84%)</td>
<td>46.53%</td>
<td>63.51% (51.32%)</td>
<td>40.35%</td>
<td>59.82% (46.65%)</td>
<td>37.60%</td>
</tr>
<tr>
<td>Code Smells</td>
<td>68.64% (57.98%)</td>
<td>46.75%</td>
<td>62.81% (50.18%)</td>
<td>40.13%</td>
<td>59.75% (47.36%)</td>
<td>37.29%</td>
</tr>
<tr>
<td>LOC</td>
<td>68.54% (59.55%)</td>
<td>44.48%</td>
<td>63.55% (54.15%)</td>
<td>39.57%</td>
<td>59.28% (51.25%)</td>
<td>36.23%</td>
</tr>
<tr>
<td>No. of Classes</td>
<td>75.18% (60.73%)</td>
<td>55.16%</td>
<td>64.03% (51.45%)</td>
<td>44.62%</td>
<td>61.31% (48.66%)</td>
<td>40.37%</td>
</tr>
</tbody>
</table>
| Avg. Class
Complexity| 52.88% (40.22%)                                                              | 44.94%                                      | 46.95% (37.07%)                                                                | 32.45%                                      | 41.57% (31.72%)                                                                | 26.13%                                      |

5.2 Rate of Change as a Bug-Inducing Commit Indicator

In this set of experiments, we took a different approach and we have examined not how the
correlation of these seven features with the New SQALE Debt Ratio differs before an imminent
bug-inducing commit, but what is the difference in the rate of change of the values of any of the
correlations and of the raw values of these six features, in two periods before a bug-inducing
commit (i.e. the period of 15 to 6 relevant commits prior to a bug-inducing commit and the
period of 5 to 1) relevant commits before a bug-inducing commit).

Here, we have conducted four types of experiments.

The first type of experiment examined whether a difference of 10% or more in the rate of
change (i.e., the slope of the least squares regression line) in any of these seven metrics from
one pre-bug-inducing commit period (15 to 6 commits) to another (5 to 1 commit) can serve
as an indicator of a bug-inducing commit. Table 5.3 depicts the difference in the change in
slope between the two periods for each project and for each of the six features considered. For
example, for project batik we observe on average a 14.98% of change in the slope of NCLOC between the two pre-bug-inducing commit periods. Table 5.4 depicts the results of all four rate of change types of analyses. More specifically, the second column of Table 5.4 shows the probability of a bug-inducing commit occurring in the next five relevant commits when the difference in the rate of change of the different metric values is equal of more than 10% in commit 15-6, and 5-1, prior to a probable bug-inducing commit. The numbers in parentheses indicate the probabilities of the same event for the randomly selected non-bug-inducing commit. For example, a result (cell (2,2)) shown in Table 5.4 can be interpreted as “if we have a bug-inducing commit then in 62.38% of the cases the rate of change of NCLOC is on average more than 10% between the two pre-bug-inducing commit periods (15 - 6 and 5 - 1 commits prior to the bug inducing commits)”, and the number next to it as “if we do not have a bug inducing commit then in 64.07% of the cases the rate of change of NCLOC is on average more than 10% between the two non-pre-bug-inducing commit periods (15 - 6 and 5 - 1 commits prior to the non-bug inducing commits)”. These results, with the exception of the number of Violations (see cell (3,2)) indicate that the difference in the rate of change in metrics in the two periods is not a good indicator of an imminent BIC.
The second type of analysis examined whether a difference of 25% or more in the rate of change in the correlations of any of the seven metrics with New SQALE Debt Ratio from one pre-bug-inducing commit period (15-6 commits) to another (5-1 commits) can serve as an indicator of an imminent bug-inducing commit. For example, in 51.35% of the cases, we observe a change of more than 25% of the correlation of Avg. Class Complexity with New SQALE Debt Ratio between the two periods for a bug-inducing commit. Similarly, the number in parenthesis indicates that in 58.54% of the cases we observe the same change (i.e. more than 25%) but for randomly selected non-bug-inducing commits. We can conclude that the correlation change is not a good indicator of an imminent BIC.

The third type of analysis examined the frequency of observing an absolute difference of 10% or more in the correlations of any of the six metrics with the New SQALE Debt Ratio AND combined with a difference of 10% or more in the rate of change of the raw values of any of these seven metrics from one pre-bug-inducing commit period to another. For example, results in Table 5.4 (cell (2,4)) indicate that in 41.08% of the cases, the difference of correlations of NCLOC with New SQALE Debt Ratio in the two periods is more than 10%, AND the difference in the rate of change of values of NCLOC in the two periods is more than 10%,

<table>
<thead>
<tr>
<th>Metric</th>
<th>Diff. in Raw Metrics Rate of Change ≥ 10%</th>
<th>Correlation Diff. ≥ 25%</th>
<th>Correlation Abs. Diff. ≥ 10% AND Diff. in Raw Metrics Rate of Change ≥ 10%</th>
<th>Correlation Abs. Diff. ≥ 25% OR Diff. in Raw Metrics Rate of Change ≥ 25%</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCLOC</td>
<td>62.38% (64.07%)</td>
<td>51.35% (58.54%)</td>
<td>41.08% (48.08%)</td>
<td>61.25% (69.38%)</td>
</tr>
<tr>
<td>Violations</td>
<td>84.29% (78.29%)</td>
<td>54.04% (62.41%)</td>
<td>44.73% (53.60%)</td>
<td>64.84% (74.84%)</td>
</tr>
<tr>
<td>Code Smells</td>
<td>62.06% (67.96%)</td>
<td>53.43% (62.54%)</td>
<td>43.46% (52.67%)</td>
<td>63.99% (74.70%)</td>
</tr>
<tr>
<td>LOC</td>
<td>60.01% (64.39%)</td>
<td>48.55% (57.0%)</td>
<td>42.42% (48.86%)</td>
<td>57.95% (67.21%)</td>
</tr>
<tr>
<td>No. of Classes</td>
<td>64.58% (65.83%)</td>
<td>52.03% (56.77%)</td>
<td>38.93% (46.41%)</td>
<td>61.36% (66.44%)</td>
</tr>
<tr>
<td>Avg. Class Complexity</td>
<td>62.66% (66.11%)</td>
<td>63.60% (64.27%)</td>
<td>69.70% (69.75%)</td>
<td>84.36% (83.07%)</td>
</tr>
<tr>
<td>New SQALE Debt Ratio</td>
<td>73.86% (84.66%)</td>
<td>Not applicable</td>
<td>Not applicable</td>
<td>Not Applicable</td>
</tr>
</tbody>
</table>
before bug-inducing commit. The number in parenthesis indicates that in 48.08% of the cases we have the same event but for non-bug-inducing commits. Here, we can also conclude that the correlation change AND-combined with the rate of change of the raw metrics is not a good indicator for an imminent BIC.

Finally, the fourth type of analysis examined the frequency of observing a difference of 25% or more of the correlation values of any of the six metrics with New SQALE Debt Ratio OR combined with a change of 25% or more in the rate of change of any of these six metrics from one pre-bug-inducing commit period to another. For example, results in Table 5.4 (cell (2,5)) indicate that in 61.2% of the cases the difference of correlations of NCLOC with New SQALE Debt Ratio in the two periods is more than 25%, OR the difference in the rate of change of values of NCLOC in the two periods is more than 25%, for a bug-inducing commit. The number in parenthesis indicates that in 69.38% of the cases we have the same event but for a non-bug-inducing commit. We can also conclude that the correlation change OR combined with the rate of change of the raw metrics is not a good indicator for an imminent BIC.

However, the initial statistical analysis we have conducted indicates the behaviors discussed in all for cases above, and depicted in Table 5.4, may serve as an indicator that we will not have an imminent BIC.

5.3 Statistical Analysis of Experiments

In this section, we present a statistical analysis of bug-inducing commits (BIC) and non-bug-inducing commits (non-BIC) in software projects, aiming to investigate the potential differences between these two types of commits in terms of their correlation and rate of change of metrics.

To achieve this, we collected and categorized 24,476 random commits into BIC and non-BIC groups, with 12,843 belonging to the former and 11,633 to the latter. We calculated the correlation and rate of change of metrics for both categories in a consistent manner with Chapter 4.

The statistical analysis was conducted using a chi-square test discussed in section 2.1.5 with a
significance level of 0.05 and a sample size of 24,476 commits. The chi-square test was used to determine if there were any significant differences between the two types of commits in terms of their correlation and rate of change of metrics. We calculated the chi-square value for

<table>
<thead>
<tr>
<th>Table 5.5: Chi-square test on the correlation of 5 relevant commits</th>
</tr>
</thead>
<tbody>
<tr>
<td>chi-square ($\chi^2$) Statics</td>
</tr>
<tr>
<td>-------------------------------</td>
</tr>
<tr>
<td>$\chi^2$ Statics for correlation $&gt;=0.6$ and $&lt;0.8$</td>
</tr>
<tr>
<td>$\chi^2$ Statics for correlation $&gt;0.7$ and $&lt;0.7$</td>
</tr>
<tr>
<td>$\chi^2$ Statics for correlation $&gt;0.75$ and $&lt;0.75$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5.6: Chi-square test on the correlation of 10 relevant commits</th>
</tr>
</thead>
<tbody>
<tr>
<td>chi-square ($\chi^2$) Statics</td>
</tr>
<tr>
<td>-------------------------------</td>
</tr>
<tr>
<td>$\chi^2$ Statics for correlation $&gt;=0.6$ and $&lt;0.6$</td>
</tr>
<tr>
<td>$\chi^2$ Statics for correlation $&gt;0.7$ and $&lt;0.7$</td>
</tr>
<tr>
<td>$\chi^2$ Statics for correlation $&gt;0.75$ and $&lt;0.75$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5.7: Chi-square test on the rate of change of metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>chi-square ($\chi^2$) Statics</td>
</tr>
<tr>
<td>-------------------------------</td>
</tr>
<tr>
<td>$\chi^2$ Statics for Metric DIFF in Rate of Change $&gt;10$ and $&lt;10$</td>
</tr>
<tr>
<td>$\chi^2$ Statics for Correlation Diff in Rate of Change $&gt;25$ and $&lt;25$</td>
</tr>
<tr>
<td>$\chi^2$ Statics for (DIFF. in Correlation Change $&gt;10$ and DIFF. in Metrics change $&gt;10$) AND (DIFF. in Correlation Change $&lt;10$ and DIFF. in Metrics change $&lt;10$)</td>
</tr>
</tbody>
</table>

three scenarios: correlation of seven metrics with the New SQALE Debt Ratio in five relevant commits, correlation of seven metrics with the New SQALE Debt Ratio in ten relevant commits, and rate of change of metrics. The results are reported in Table 5.5, table 5.6, and table 5.7 respectively.

Our analysis showed that the calculated chi-square values for most of the scenarios were much greater than the critical value of approximately 3.84 for the chi-square distribution with 1 degree of freedom. Therefore, we rejected the null hypothesis and concluded that there is a significant difference between the observed and expected frequencies. This suggests that BICs
and non-BICs have a noteworthy contrast in terms of their correlation and the rate at which their metrics change.

The statistical analysis presented in this section provides evidence that there are significant differences between BICs and non-BICs in terms of their correlation and rate of change of metrics.

5.4 Limitations

It is important to recognize that the research presented in this thesis may be limited in its generalizability beyond the specific dataset and software development context analyzed. While the method proposed in the paper is based on analyzing correlations and rates of change of selected metrics, the behavior of these metrics and their correlation with bug-inducing commits may vary across different software development contexts. Additionally, the study did not explore the impact of other factors, such as project size or team composition, on the effectiveness of the proposed method. Therefore, the findings should be cautiously applied when extrapolated to other software development environments beyond those specifically analyzed in this study.
Chapter 6

Conclusion

In this research, we investigated and proposed two quantitative analysis techniques in order to assess the likelihood of an imminent bug-inducing commit.

The first type focuses on identifying a collection of process, quality, and source code metrics that exhibit low pair-wise correlation throughout the system’s lifetime, but their correlation profile changes prior to a bug-inducing commit. In this respect, this change of correlation behavior has shown that it can be used as an indicator that the system has entered a phase where a bug-inducing commit is highly probable to appear in the next 5 relevant commits. The second type of analysis focused on examining the rate of change trends of these metrics as well as the rate of change of the correlations in two periods prior to a bug-inducing commit. In this respect, we have considered a period of 15 to 6 and a period of 5 to 1 relevant commits prior to a bug-inducing commit.

Experimental results obtained from data on 22 open source systems provided by the SonarQube Technical Debt dataset indicate that elevated correlations ($\geq 0.6$ between Style Violations, Number of Code Smells and New SQALE Debt Ratio have a probability of approx. 82.5% being associated with a bug-inducing commit appearing in the next 5 related commits. This probability reaches 74% when the correlation level becomes $\geq 0.75$ (see Table 5.1). On the other hand, the probability of having the same observation but without having a bug-inducing commit is 73.17% and 64.32%. Even though these results do not seem very significant when we compare BIC and non-BIC cases, we consistently observe an increase in correlations of the level
of close to 5% five commits prior to a BIC and an increase of approximately 9% 10 commits prior to BIC. We can conclude that changes in correlations of the metrics with the New SQALE Debt Ratio provide significant evidence for an imminent BIC. Similarly, experimental results depicted in Table 5.4 indicate that a difference of ≥ 10% in the rate of change of Violations in the two segment periods is associated with a probability of 84.29% of an imminent bug-inducing commit. The results indicate that the change in the number of Violations in the two pre-BIC periods can possibly be used as a weak indicator of an imminent BIC, but the rest types of the changes in the two periods cannot serve as good indicators.

6.1 Future Work

This work can be extended in two main directions. In the first direction, new metrics can be considered and evaluated as predictors of imminent bug-inducing commits. Examples of such metrics can be the commit frequency of a file, the number of authors involved, and the number of incoming and outgoing flows as these can be estimated by the number of incoming and outgoing calls, definitions, and uses of data, as well as data flow metrics such as Function Points. The second direction focuses on a method for defining a file’s “state” as a function of a vector of metrics and consequently investigates how the state changes from one commit to the next. Such a model will define a state machine with transition probabilities. The hypothesis here is that transition patterns and analysis of state transition probabilities can then be used as predictors of an impending bug-inducing commit. Additionally, further research could be conducted to explore the impact of other factors, such as project size, team composition, and software development methodologies, on the effectiveness of the proposed method. Finally, it may be valuable to investigate how the proposed method could be integrated with existing software development tools and practices to streamline the bug detection and fixing process.
Bibliography

[1] https://figshare.com/s/be8198c4ab320a1af7c0.


[71] et al. Rodríguez-Pérez, G. Watch out for extrinsic bugs! a case study of their impact in just-in-time bug prediction models on the openstack project. TSE 2021.


[83] Biljana Stanić and Wasif Afzal. Static code metrics vs. process metrics for software fault prediction using bayesian network learners. 2015.


[95] Ruzhi Xu, Yunjiao Xue, Peiyao Nie, Yuan Zhang, and Desheng Li. doion cmmi-based software process metrics. *International Multi-Symposiums on Computer and Computational Sciences*, 1, 2008.


7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, 2009.

Appendix A

SQL Queries for pre-processing the Technical Debt Dataset

A.1 Determining the files that are common in fixing and inducing commits

A.1.1 Created a new table for inducing commit and fixing commit, along with their respective dates.

**(PROJECT_ID, FAULT_INDUCTING_COMMIT_HASH, FAULT_FIXING_COMMIT_HASH)** from **SZZ_FAULT_INDUCTING_COMMITS** and **(DATE_OF_INDUCTING_COMMIT, DATE_OF_FIXING_COMMIT)** from **GIT_COMMITS_CHANGES**

Query:

- Created a table “ZALLPROJECTS_FAULT_INDUCTING_COMMITS” with fields – **Project_ID**, **FAULT_INDUCTING_COMMIT_HASH**, **FAULT_FIXING_COMMIT_HASH**, **DATE_OF_INDUCTING_COMMIT**, **DATE_OF_FIXING_COMMIT**

- create table ZALLPROJECTS_FAULT_INDUCTING_COMMITS (Project_ID, FAULT_INDUCTING_COMMIT_HASH, FAULT_FIXING_COMMIT_HASH, DATE_OF_INDUCTING_COMMIT, DATE_OF_FIXING_COMMIT)
• Inserted values into the new table for fields “Project_id”, “FAULT_INDUCING_COMMIT_HASH”, “FAULT_FIXING_COMMIT_HASH” from existing table SZZ_FAULT_INDUCING_COMMITS

− INSERT INTO ZALLPROJECTS_FAULT_INDUCING_COMMITS (Project_ID, FAULT_INDUCING_COMMIT_HASH, FAULT_FIXING_COMMIT_HASH) select Project_ID, FAULT_INDUCING_COMMIT_HASH, FAULT_FIXING_COMMIT_HASH from SZZ_FAULT_INDUCING_COMMITS

• Inserted “DATE_OF_INDUCING_COMMIT” for the new field by equating the DATE from GIT_COMMITS_CHANGES

− UPDATE ZALLPROJECTS_FAULT_INDUCING_COMMITS SET DATE_OF_INDUCING_COMMIT = (SELECT DATE FROM GIT_COMMITS_CHANGES WHERE ZALLPROJECTS_FAULT_INDUCING_COMMITS.FAULT_INDUCING_COMMIT_HASH = GIT_COMMITS_CHANGES.COMMIT_HASH AND ZALLPROJECTS_FAULT_INDUCING_COMMITS.Project_ID = GIT_COMMITS_CHANGES.PROJECT_ID)

**Table Name:** ZALLPROJECTS_FAULT_INDUCING_COMMITS

### A.1.2 Created a table having commit hash, file and date information

distinct (Project_id, COMMIT_HASH, FILE, DATE) information from GIT_COMMITS_CHANGES

**Query:**

• Created a new table “ZALLPROJECTS_GIT_COMMITS_CHANGES” with fields “PROJECT_ID”, “COMMIT_HASH”, “FILE”, “DATE”

− CREATE TABLE ZALLPROJECTS_GIT_COMMITS_CHANGES (PROJECT_ID, COMMIT_HASH, FILE, DATE)
• Inserted distinct values into new table from exiting “GIT_COMMITS CHANGES” table

• INSERT INTO ZALLPROJECTS_GIT_COMMITS_CHANGES (PROJECT_ID, COMMIT_HASH, FILE, DATE) select PROJECT_ID, COMMIT_HASH, FILE, DATE from GIT_COMMITS_CHANGES

Table Name: ZALLPROJECTS_GIT_COMMITS_CHANGES

A.1.3 Table to get the files of inducing commits

cross join of ZALLPROJECTSFAULT_INDUCING_COMMITS and ZALLPROJECTS_GIT_COMMITS_CHANGES for inducing commits (Cross join of Table A.1.1 and A.1.2 for inducing commits)

Query:

• Created a new table “ZALLPROJECTS_FILES_INDUCING_COMMITS” to display the files on which inducing commits has worked on.

– CREATE TABLE ZALLPROJECTS_FILES_INDUCING_COMMITS
  (PROJECT_ID, FAULT_INDUCING_COMMIT_HASH, FAULT_FIXING_COMMIT_HASH, DATE_OF_INDUCING_COMMIT, DATE_OF_FIXING_COMMIT, PROJECT_ID2, COMMIT_HASH, INDUCING_FILES, DATE)

• Inserted the values into the new table by joining the table having inducing commits (table A.1.1) and table having commit with files information (table A.1.2)

• Insert into ZALLPROJECTS_FILES_INDUCING_COMMITS (PROJECT_ID, FAULT_INDUCING_COMMIT_HASH, FAULT_FIXING_COMMIT_HASH, DATE_OF_INDUCING_COMMIT, DATE_OF_FIXING_COMMIT, PROJECT_ID2, COMMIT_HASH, INDUCING_FILES, DATE)

SELECT * FROM ZALLPROJECTSFAULT_INDUCING_COMMITS CROSS JOIN ZALLPROJECTS_GIT_COMMITS_CHANGES WHERE ZALLPROJECTSFAULT_INDUCING_COMMITS.FAULT_INDUCING_COMMIT_HASH
A.1. Determining the files that are common in fixing and inducing commits

= ZALLPROJECTS_GIT_COMMITS_CHANGES.COMMIT_HASH AND ZALLPROJECTS_FAULT_INDUCING_COMMITS.Project_ID = ZALLPROJECTS_GIT_COMMITS_CHANGES.PROJECT_ID AND ZALLPROJECTS_FAULT_INDUCING_COMMITS.DATE_OF_INDUCING_COMMIT = ZALLPROJECTS_GIT_COMMITS_CHANGES.DATE

- Deleted duplicate columns generated during the joining process (PROJECT_ID2, DATE and COMMIT_HASH)
  - ALTER TABLE ZALLPROJECTS_FILES_INDUCING_COMMITS DROP COLUMN PROJECT_ID2;
  - ALTER TABLE ZALLPROJECTS_FILES_INDUCING_COMMITS DROP COLUMN DATE;
  - ALTER TABLE ZALLPROJECTS_FILES_INDUCING_COMMITS DROP COLUMN COMMIT_HASH;

Table Name: ZALLPROJECTS_FILES_INDUCING_COMMITS

A.1.4 Table to get the files of Fixing commits

cross join of ZALLPROJECTS_FAULT_INDUCING_COMMITS and ZALLPROJECTS_GIT_COMMITS_CHANGES for fixing commits (Cross join of Table output obtained from section A.1.2 and A.1.1 for fixing commits)

Query:

- Created a new table “ZALLPROJECTS_FILESFIXING_COMMITS” to display the files on which inducing commits has worked on.
  - CREATE TABLE ZALLPROJECTS_FILESFIXING_COMMITS (PROJECT_ID, FAULT_INDUCING_COMMIT_HASH, FAULT_FIXING_COMMIT_HASH, DATE_OF_INDUCING_COMMIT, DATE_OF_FIXING_COMMIT, PROJECT_ID2, COMMIT_HASH, FIXING_FILES, DATE)
- Inserted the values into the new table by joining the table having fixing commits (table A.1.1) and table having commit with files information (table A.1.2)

- Insert into ZALLPROJECTS_FILES_FIXING_COMMITS (PROJECT_ID, FAULT_INDUCING_COMMIT_HASH, FAULT_FIXING_COMMIT_HASH, DATE_OF_INDUCING_COMMIT, DATE_OF_FIXING_COMMIT, PROJECT_ID2, COMMIT_HASH, FIXING_FILES, DATE)

  SELECT * FROM ZALLPROJECTS_FAULT_INDUCING_COMMITS CROSS JOIN ZALLPROJECTS_GIT_COMMITS_CHANGES WHERE ZALLPROJECTS_FAULT_INDUCING_COMMITS.FAULT_FIXING_COMMIT_HASH = ZALLPROJECTS_GIT_COMMITS_CHANGES.COMMIT_HASH AND ZALLPROJECTS_FAULT_INDUCING_COMMITS.Project_ID = ZALLPROJECTS_GIT_COMMITS_CHANGES.PROJECT_ID AND ZALLPROJECTS_FAULT_INDUCING_COMMITS.DATE_OF_FIXING_COMMIT = ZALLPROJECTS_GIT_COMMITS_CHANGES.DATE

- Deleted duplicate columns generated during the joining process (PROJECT_ID2, DATE and COMMIT_HASH)

  - ALTER TABLE ZALLPROJECTS_FILES_FIXING_COMMITS DROP COLUMN PROJECT_ID2;

  - ALTER TABLE ZALLPROJECTS_FILES_FIXING_COMMITS DROP COLUMN DATE;

  - ALTER TABLE ZALLPROJECTS_FILES_FIXING_COMMITS DROP COLUMN COMMIT_HASH;

Table Name: ZALLPROJECTS_FILES_FIXING_COMMITS
A.1.5 Table to get the intersection of files of fixing and inducing commits to determine which files had issue

Intersection of table ZALLPROJECTS_FILES_INDUCING_COMMITS and ZALLPROJECTS_FILES_FIXING_COMMITS (Intersection of Table A.1.3 and A.1.4 for determining which files inducing and fixing commits used in common)

Query:

- Created a new table “ZALLPROJECTS_FILES_INDUCING_INTERSECTION_FIXING_COMMITS” for having the details of intersection of files of inducing and fixing commits

  CREATE TABLE ZALLPROJECTS_FILES_INDUCING_INTERSECTION_FIXING_COMMITS (PROJECT_ID, FAULT_INDUCING_COMMIT_HASH, FAULT_FIXING_COMMIT_HASH, DATE_OF_INDUCING_COMMIT, DATE_OF_FIXING_COMMIT, FILES)

- Inserted values into the new table by intersecting the table having all files of inducing commits (table 1.3) and table having all files of fixing commits (table 1.4)

  INSERT INTO ZALLPROJECTS_FILES_INDUCING_INTERSECTION_FIXING_COMMITS (PROJECT_ID, FAULT_INDUCING_COMMIT_HASH, FAULT_FIXING_COMMIT_HASH, DATE_OF_INDUCING_COMMIT, DATE_OF_FIXING_COMMIT, FILES)
  SELECT DISTINCT PROJECT_ID, FAULT_INDUCING_COMMIT_HASH, FAULT_FIXING_COMMIT_HASH, DATE_OF_INDUCING_COMMIT, DATE_OF_FIXING_COMMIT, FIXING_FILES FROM ZALLPROJECTS_FILES_FIXING_COMMITS
  INTERSECT
  SELECT DISTINCT PROJECT_ID, FAULT_INDUCING_COMMIT_HASH, FAULT_FIXING_COMMIT_HASH, DATE_OF_INDUCING_COMMIT, DATE_OF_FIXING_COMMIT, INDUCING_FILES FROM ZALLPROJECTS_FILES_INDUCING_COMMITS
Table Name: ZALLPROJECTS_FILES_INDUCING_INTERSECTION_FIXING_COMMITS

A.2 Determining the relevant Commits for Fault Inducing and Fixing Commits

All the commits that has worked on the same files as inducing and fixing commits, and occurred before fault fixing commit or at the same time as Fault fixing commit as referred to as ”Relevant commits”. “Commits at the same time as fault fixing commit” is considered to include the fault-fixing commit as well, in the same group.

A.2.1 Created a table having required SONAR_MEASURES with respect to COMMIT_HASH

Required measures are taken from Sonar_measures table and respective commit_hash (Revision) is taken from SONAR_ANALYSIS by equating Analysis ID in both tables

Query:

• Created a table “ZSONAR_METRICS” to store the values of sonar measures with respect to the commit

- CREATE TABLE ZSONAR_METRICS (  
  PROJECT_ID TEXT,  
  ANALYSIS_KEY TEXT,  
  COMMIT_HASH,  
  DEBT,  
  EFFORTS,  
  CLASS_COMPLEXITY REAL,  
  NEW_LINES_TO_COVER TEXT,  
  VIOLATIONS INTEGER,  
)
A.2. Determining the relevant Commits for Fault Inducing and Fixing Commits

NEW_VIOLATIONS TEXT,
SQALE_RATING INTEGER,
SQALE_DEBT_RATIO REAL,
NEW_SQALE_DEBT_RATIO REAL,
CODE_SMELLS INTEGER,
NEW_CODE_SMELLS TEXT,
EFFORT_TO_REACH_MAINTAINABILITY_RATING_A INTEGER,
BUGS INTEGER,
NEW_BUGS TEXT,
RELIABILITY_REMEDIATION_EFFECT INTEGER,
NEW_RELIABILITY_REMEDIATION_EFFECT TEXT,
RELIABILITY_RATING INTEGER,
NEW_RELIABILITY_RATING TEXT,
VULNERABILITIES INTEGER,
NEW_VULNERABILITIES TEXT,
SECURITY_REMEDIATION_EFFECT INTEGER,
NEW_SECURITY_REMEDIATION_EFFECT TEXT,
SECURITY_RATING INTEGER,
NEW_SECURITY_RATING TEXT,
CLASSES INTEGER,
FILES INTEGER,
FUNCTIONS INTEGER,
COMMENT_LINES_DENSITY REAL);

- Inserted the values into the table from existing table SONAR_MEASURES (existing table does not contain the commit_hash)

- INSERT INTO ZSONAR_METRICS (PROJECT_ID,
ANALYSIS_KEY,
CLASS_COMPLEXITY,
NEW_LINES_TO_COVER,
VIOLATIONS,
NEW_VIOLATIONS,
SQALE_RATING,
SQALE_DEBT_RATIO,
NEW_SQALE_DEBT_RATION,
CODE_SMELLS,
NEW_CODE_SMELLS,
EFFORT_TO_REACH_MAINTAINABILITY_RATING_A,
BUGS,
NEW_BUGS,
RELIABILITY.Remediation_Effort,
NEW.RELIABILITY.Remediation_Effort,
RELIABILITY.Rating,
NEW.RELIABILITY.Rating,
VULNERABILITIES,
NEW.VULNERABILITIES,
SECURITY.Remediation_Effort,
NEWSECURITY.Remediation_Effort,
SECURITY.Rating,
NEWSECURITY.Rating,
CLASSES,
FILES,
FUNCTIONS,
COMMENT_LINES_DENSITY
A.2. Determining the relevant commits for fault inducing and fixing commits

) SELECT PROJECT_ID, ANALYSIS_KEY,
CLASS_COMPLEXITY,
NEW_LINES_TO_COVER,
VIOLATIONS,
NEW_VIOLATIONS,
SQALE_RATING,
SQALE_DEBT_RATIO,
NEW_SQALE_DEBT_RATION,
CODE_SMELLS,
NEW_CODE_SMELLS,
EFFORT_TO_REACH_MAINTAINABILITY_RATING_A,
BUGS,
NEW_BUGS,
RELIABILITY_REMEDICATION_EFFORT,
NEW_RELIABILITY_REMEDICATION_EFFORT,
RELIABILITY_RATING,
NEW_RELIABILITY_RATING,
VULNERABILITIES,
NEW_VULNERABILITIES,
SECURITY_REMEDICATION_EFFORT,
NEW_SECURITY_REMEDICATION_EFFORT,
SECURITY_RATING,
NEW_SECURITY_RATING,
CLASSES,
FILES,
FUNCTIONS,
COMMENT_LINES_DENSITY
FROM SONAR_MEASURES

- Added the commit_hash value from SONAR_ANALYSIS table by equating the analysis_key of ZSONAR_METRICS (new table) and SONAR_ANALYSIS

  - UPDATE ZSONAR_METRICS SET COMMIT_HASH = (SELECT REVISION FROM SONAR_ANALYSIS WHERE ZSONAR_METRICS.ANALYSIS_KEY = SONAR_ANALYSIS.ANALYSIS_KEY)

**Table Name:** ZSONAR_METRICS

### A.2.2 Created a table to store all the data groupwise (Groups are based on the fault inducing and fault fixing commits, and all the commits that occurred before fault fixing)

A general group seems as shown in fig. A.1.

**Create a table to get all the commits for the group**

Cross join of output table obtained from ZALLPROJECTS_FILES_INDUCING_INTERSECTION_FIXING_COMMITS (section A.1.5) and ZALLPROJECTS_GIT_COMMITS_CHANGES (section A.1.2)

**Query:**

- Created a table “ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFOREFAULT_FIXING” to get all the groups (list of commits working on same files as inducing and fixing commits)

  - CREATE TABLE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_FIXING
    (PROJECT_ID,
    FAULT_INDUCING_COMMIT_HASH,
A.2. Determining the relevant Commits for Fault Inducing and Fixing Commits

Fault Fixing Commits

Figure A.1: Sample group details

FAULT_FIXING_COMMIT_HASH,
DATE_OF_INDUCING_COMMIT,
DATE_OF_FIXING_COMMIT,
FILES,
PROJECT_IDV2,
COMMIT_HASH_ON_FILE_BEFORE_FIX,
FILES2,
DATE_OF_COMMIT_HASH
)

- Insert values into the table using cross join of table having intersection of files de-
tails (ZALLPROJECTS_FILES INDUCING_INTERSECTION_FIXING_COMMITS) and GIT_COMMITS_CHANGES, where former table’s files are equal to the latter table’s files and limit the values to less than equal to the date of fixing commit

- INSERT INTO ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES BEFORE_FAULT_FIXING ( PROJECT_ID, FAULT_INDUCING_COMMIT_HASH, FAULT_FIXING_COMMIT_HASH, DATE_OF_INDUCING_COMMIT, DATE_OF_FIXING_COMMIT, FILES, PROJECT_IDV2, COMMIT_HASH_ON_FILE_BEFORE_FIX, FILES2, DATE_OF_COMMIT_HASH )

SELECT * FROM ZALLPROJECTS_FILES INDUCING INTERSECTION_FIXING_COMMITS CROSS JOIN ZALLPROJECTS_GIT_COMMITS_CHANGES WHERE ZALLPROJECTS_GIT_COMMITS_CHANGES.DATE <= ZALLPROJECTS_FILES INDUCING_INTERSECTION_FIXING_COMMITS.DATE OF_FIXING_COMMIT AND ZALLPROJECTS_FILES INDUCING_INTERSECTION_FIXING_COMMITS.FILES = ZALLPROJECTS_GIT_COMMITS_CHANGES.FILE

- Deleted duplicate columns generated during the joining process (PROJECT_IDV2 and FILES2)
A.2. Determining the relevant Commits for Fault Inducing and Fixing Commits

- ALTER TABLE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES _BEFORE_FAULT_FIXING DROP COLUMN PROJECT_IDV2;
- ALTER TABLE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES _BEFORE_FAULT_FIXING DROP COLUMN FILES2;

Specify if the commit is fault inducing or fixing or before inducing or after inducing and before fixing

Query:

- Add a new column IS_COMMIT_HASH_ON_FILES_BEFORE_INDUCING with default value = 0

- ALTER TABLE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES _BEFORE_FAULT_FIXING
  ADD
  IS_COMMIT_HASH_ON_FILES_BEFORE_INDUCING

- UPDATE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE _FAULT_FIXING set IS_COMMIT_HASH_ON_FILES_BEFORE_INDUCING = 0

- Set the value to 1 if the date of commit < date of inducing commit

- Update ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE _FAULT_FIXING
  Set
  IS_COMMIT_HASH_ON_FILES_BEFORE_INDUCING = 1
  WHERE
  DATE_OF_COMMIT_HASH < DATE_OF_INDUCING_COMMIT;

- Add new column IS_COMMIT_HASH_ON_FILES_AFTER_INDUCING_AND _BEFORE_FIXING with default = 0
Chapter A. SQL Queries for pre-processing the Technical Debt Dataset

- ALTER TABLE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES _BEFORE_FAULT_FIXING ADD IS_COMMIT_HASH_ON_FILES_AFTER _INDUCING_AND_BEFORE_FIXING
update ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES BEFORE _FAULT_FIXING set IS_COMMIT_HASH_ON_FILES_AFTER_INDUCING_AND _BEFORE_FIXING = 0

• Set the value to 1 if the date of commit > date of inducing commit && date of commit < date of fixing

- update ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES BEFORE _FAULT_FIXING
  set IS_COMMIT_HASH_ON_FILES_AFTER_INDUCING_AND_BEFORE_FIXING = 1 WHERE DATE_OF_COMMIT_HASH > DATE_OF_INDUCING_COMMIT AND DATE_OF_COMMIT_HASH < DATE_OF_FIXING_COMMIT

• Add a new column IS_COMMIT_HASH_ON_FILES_INDUCING_COMMIT with default value = 0

- ALTER TABLE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES _BEFORE_FAULT_FIXING ADD IS_COMMIT_HASH_ON_FILES_INDUCING _COMMIT;
- update ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES _BEFORE_FAULT_FIXING set IS_COMMIT_HASH_ON_FILES_INDUCING _COMMIT = 0

• Set the value to 1 if the date of commit = date of inducing commit

- update ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES BEFORE _FAULT_FIXING set IS_COMMIT_HASH_ON_FILES_INDUCING_COMMIT = 1 WHERE DATE_OF_COMMIT_HASH = DATE_OF_INDUCING_COMMIT;
A.2. Determining the relevant Commits for Fault Inducing and Fixing Commits

- Add a new column IS_COMMIT_HASH_ON_FILES_FIXING_COMMIT with default = 0
  
  ```
  ALTER TABLE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFOR
  E_FAULT_FIXING ADD IS_COMMIT_HASH_ON_FILES_FIXING_COMMIT;
  ```

- Update ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFOR
  E_FAULT_FIXING set IS_COMMIT_HASH_ON_FILES_FIXING_COMMIT = 0

- Set the value to 1 if the date of commit = date of fixing commit
  
  ```
  update ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFOR
  E_FAULT_FIXING set IS_COMMIT_HASH_ON_FILES_FIXING_COMMIT = 1 WHERE
  DATE_OF_COMMIT_HASH = DATE_OF_FIXING_COMMIT;
  ```

Add the Pair_Number (group_Number) for each inducing and fixing commit

Query:

- Added concat column to create groups: this concat column contains the concatenation of Inducing_commit_hash and Fixing_commit_hash
  
  ```
  ALTER TABLE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFOR
  E_FAULT_FIXING add COLUMN CONCAT;
  ```

- update ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFOR
  E_FAULT_FIXING set CONCAT = FAULT_INDUCING_COMMIT_HASH ||
  FAULT_FIXING_COMMIT_HASH;

- Added pair number based on concatenated values (inducing and fixing commit), each group is assigned with a pair Number
  
  ```
  CREATE TABLE YTEMP3 (CONCAT_COMMIT_HASH, ID INTEGER PRIMARY
  KEY AUTOINCREMENT );
  ```
– insert into YTEMP3 (CONCAT_commit_hash)

– select distinct concat from ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_FIXING

– Update ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_FIXING set PAIR_NUMBER = (select PAIR_NUMBER from YTEMP3)

Add commit measures to the table

Selected Measures:
- NCLOC
- NEW_SQALE_DEBT_RATION
- VIOLATIONS
- CODE_SMELLS
- FUNCTIONS
- LINES
- CLASSES
- CLASS_COMPLEXITY

Query:

• Added the measure values for commits from ZSONAR_METRICS table

- ALTER TABLE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_FIXING ADD COLUMN NCLOC;
- ALTER TABLE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_FIXING ADD COLUMN NEW_SQALE_DEBT_RATION;
- ALTER TABLE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_FIXING ADD COLUMN VIOLATIONS;
- ALTER TABLE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_FIXING ADD COLUMN CODE_SMELLS;
- ALTER TABLE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_FIXING ADD COLUMN FUNCTIONS;
A.2. Determining the relevant commits for fault inducing and fixing commits

- ALTER TABLE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_FIXING ADD COLUMN LINES;
- ALTER TABLE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_FIXING ADD COLUMN CLASSES;
- ALTER TABLE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_FIXING ADD COLUMN CLASS_COMPLEXITY;

- UPDATE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_FIXING SET NCLOC = (SELECT NCLOC FROM ZSONAR_METRICS WHERE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_FIXING.COMMIT_HASH_ON_FILE_BEFORE_FIX = ZSONAR_METRICS.COMMIT_HASH) ;
- UPDATE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_FIXING SET NEW_SQALE_DEBT_RATION = (SELECT NEW_SQALE_DEBT_RATION FROM ZSONAR_METRICS WHERE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_FIXING.COMMIT_HASH_ON_FILE_BEFORE_FIX = ZSONAR_METRICS.COMMIT_HASH) ;
- UPDATE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_FIXING SET VIOLATIONS = (SELECT VIOLATIONS FROM ZSONAR_METRICS WHERE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_FIXING.COMMIT_HASH_ON_FILE_BEFORE_FIX = ZSONAR_METRICS.COMMIT_HASH) ;
- UPDATE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_FIXING SET CODE_SMELLS = (SELECT CODE_SMELLS FROM ZSONAR_METRICS WHERE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_BEFORE_FAULT_FIXING.COMMIT_HASH_ON_FILE_BEFORE_FIX =
ZSONAR_METRICS.COMMIT_HASH) ;

- UPDATE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_Before_FAULT_FIXING SET FUNCTIONS = (SELECT FUNCTIONS FROM ZSONAR_METRICS
WHERE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_Before_FAULT_FIXING.COMMIT_HASH_ON_FILE_Before_FIX = ZSONAR_METRICS.COMMIT_HASH) ;

- UPDATE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_Before_FAULT_FIXING SET LINES = (SELECT LINES FROM ZSONAR_METRICS
WHERE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_Before_FAULT_FIXING.COMMIT_HASH_ON_FILE_Before_FIX = ZSONAR_METRICS.COMMIT_HASH) ;

- UPDATE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_Before_FAULT_FIXING SET CLASSES = (SELECT CLASSES FROM ZSONAR_METRICS
WHERE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_Before_FAULT_FIXING.COMMIT_HASH_ON_FILE_Before_FIX = ZSONAR_METRICS.COMMIT_HASH) ;

- UPDATE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_Before_FAULT_FIXING SET CLASS_COMPLEXITY = (SELECT CLASS_COMPLEXITY FROM ZSONAR_METRICS
WHERE ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_Before_FAULT_FIXING.COMMIT_HASH_ON_FILE_Before_FIX = ZSONAR_METRICS.COMMIT_HASH) ;

**TABLE OUTPUT:** ZALLPROJECTS_COMMITS_ON_INTERSECTION_FILES_Before_FAULT_FIXING
A.3 Final Dataset Preparation

- To avoid the duplicate commits in same group (commits working on multiple files) we deleted the column “FILES” and removed the duplicates.

- Rearranged the group in ascending order of date of commit.

FINAL OUTPUT:

ZALLPROJECTS_FINAL
Appendix B

Data extraction and processing Algorithms

B.1 Algorithm to determine the low correlation metrics

```java
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.*;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

public class DetermineNonCorrelatedSetsFinal2 {

    public static ArrayList<ArrayList> determineRelatedMetrics() throws IOException {
        FileInputStream fis = new FileInputStream(new File("PATH"));
        //creating workbook instance that refers to .xls file
        HSSFWorkbook wb = new HSSFWorkbook(fis);
        //creating a Sheet object to retrieve the object
        HSSFSheet sheet = wb.getSheetAt(0);
        int iRow = 1;
        int iColumn = 1;
        ArrayList listOfRelatedMetrics = new ArrayList<ArrayList>();
        while (sheet.getRow(iRow) != null) {
            //...
        }
    }
}
```
Algorithm to determine the low correlation metrics

```java
ArrayList<String> relatedMetrics = new ArrayList<String>();
relatedMetrics.add(sheet.getRow(iRow).getCell(0).getStringCellValue()); // added first cell value as the first element in the ArrayList
iColumn = 1;

while(sheet.getRow(iRow).getCell(iColumn)! = null)
{
    if((sheet.getRow(iRow).getCell(iColumn).getNumericCellValue()! = 1 && (sheet.getRow(iRow).getCell(iColumn).getNumericCellValue() < (-0.5)) || sheet.getRow(iRow).getCell(iColumn).getNumericCellValue() > 0.5))
    {
        relatedMetrics.add(sheet.getRow(0).getCell(iColumn).getStringCellValue());
    }
    iColumn++;
}
listOfRelatedMetrics.add(relatedMetrics);
iRow++;
}
return listOfRelatedMetrics; // return correlated metrics
// first element of each relatedMetrics is the primary element which is correlating with rest other elements
}

public static void printRelatedMetrics(ArrayList a)
{
    for(int i=0; i<a.size(); i++)
    {
        System.out.print(";" + a.get(i));
    }
}

private static HashSet<HashSet> calculateSetOfUnrelatedMetrics(ArrayList listOfRelatedMetrics) throws IOException {
```
FileInputStream fis = new FileInputStream(new File("PATH"));
//creating workbook instance that refers to .xls file
HSSFWorkbook wb = new HSSFWorkbook(fis);
//creating a Sheet object to retrieve the object
HSSFSheet sheet = wb.getSheetAt(0);
HashSet<HashSet> setOfUnrelatedMetrics = new HashSet<HashSet>();
int iRow = 1;
int iColumn = 1;
int totalCountOfColumns = 19;
int countForListOfRelatedMetrics = 0;
while (sheet.getRow(iRow) != null) {
    //added first cell value as the first element in the arrayList
    iColumn = 1;
    while (sheet.getRow(iRow).getCell(iColumn) != null) {
        int count = 0;
        int internalColumn = iColumn;
        ArrayList<String> listUnrelatedMetrics = new ArrayList<String>();
        listUnrelatedMetrics.add(sheet.getRow(iRow).getCell(0).getStringCellValue());
        while (count < totalCountOfColumns) {
            if (internalColumn > totalCountOfColumns) {
                internalColumn = internalColumn - totalCountOfColumns + 2;
            }
            if (sheet.getRow(iRow).getCell(internalColumn).getNumericCellValue() >= -0.5 &&
                sheet.getRow(iRow).getCell(internalColumn).getNumericCellValue() <= 0.5) {
                String eligibleElement = sheet.getRow(0).getCell(internalColumn).getStringCellValue();
                int flag = 0;
                for (int iPreviousElementsOfUnrelatedMetrics = 1; iPreviousElementsOfUnrelatedMetrics <
                    listUnrelatedMetrics.size(); iPreviousElementsOfUnrelatedMetrics++) {
                    
                }
String previousElement = listUnrelatedMetrics.get(iPreviousElementsOfUnrelatedMetrics);
for (int i = 0; i < listOfRelatedMetrics.size(); i++) {
    ArrayList<String> a = (ArrayList<String>) listOfRelatedMetrics.get(i);
    if (a.get(0).equals(previousElement)) {
        for (int j = 0; j < a.size(); j++) {
            if (a.get(j).equals(eligibleElement)) {
                flag = 1;
                break;
            }
        }
    }
}
if (flag == 0) {
    listUnrelatedMetrics.add(eligibleElement);
}
count++;
internalColumn++;
}
HashSet<String> unrelatedMetrics = new HashSet<>(listUnrelatedMetrics);
setOfUnrelatedMetrics.add(unrelatedMetrics);
iColumn++;
}
iRow++;
}
printHashSet(setOfUnrelatedMetrics);
return setOfUnrelatedMetrics;
}

public static void main (String args[]) throws IOException {

ArrayList listOfRelatedMetrics = determineRelatedMetrics();
HashSet<HashSet> setOfUnrelatedMetrics = calculateSetOfUnrelatedMetrics(listOfRelatedMetrics);

private static void printHashSet(HashSet<HashSet> setOfUnrelatedMetrics) {
    Iterator i = setOfUnrelatedMetrics.iterator();
    while(i.hasNext()) {
        System.out.println(i.next());
    }
}

B.2 Algorithm to calculate the correlation of 5 and 10 commits before bug inducing commit

import org.apache.commons.math3.stat.correlation.PearsonsCorrelation;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.xssf.usermodel.XSSFSheet;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.ArrayList;
public class CorrelationCalculation_givenInt {
    public static String filename_v2 = "path";
    public static XSSFWorkbook workbook = new XSSFWorkbook();
B.2. Algorithm to calculate the correlation of 5 and 10 commits before bug inducing commit 101

```java
public static int cellNo;
public static XSSFSheet newSheet = workbook.createSheet("sheet1");
public static void calculateCorrelation(int measure1, int measure2, int givenInt, String path, int column) throws IOException {
    FileInputStream fis = new FileInputStream(new File(path));
    //creating workbook instance that refers to .xls file
    XSSFWorkbook wb = new XSSFWorkbook(fis);
    //creating a Sheet object to retrieve the object
    XSSFSheet sheet = wb.getSheetAt(0);
    int i = 2; //starting from 3rd row
    double pairNumber = 0;
    int totalBic;
    while (sheet.getRow(i) != null && sheet.getRow(i).getCell(0) != null) {
        double corr = 0;
        totalBic = 0;
        int startingCommitRowNo = 1;
        int inducingCommitRowNo = 1;
        if (pairNumber != sheet.getRow(i).getCell(21).getNumericCellValue()) //pair ! = next pair no. that means starting of new pair
        {
            pairNumber = sheet.getRow(i).getCell(21).getNumericCellValue();
            startingCommitRowNo = sheet.getRow(i).getRowNum();
        }
        while (sheet.getRow(i).getCell(10).getNumericCellValue() != 1 &&
            sheet.getRow(i).getCell(9).getNumericCellValue() != 1 &&
            sheet.getRow(i).getCell(11).getNumericCellValue() != 1 &&
            sheet.getRow(i + 1) != null &&
            sheet.getRow(i + 1).getCell(21).equals(null) &&
            pairNumber == sheet.getRow(i + 1).getCell(21).getNumericCellValue()) //if not inducing commit and not AFTER INDUCING and not FIXING and same pair is continuing
        {
            
        }
    }
}
```
totalBic++;  
i++;  
}  
if(sheet.getRow(i).getCell(10).getNumericCellValue()==1 ||  
sheet.getRow(i).getCell(9).getNumericCellValue()==1 ||  
sheet.getRow(i).getCell(11).getNumericCellValue()==1 ||  
sheet.getRow(i+1)==null || pairNumber!=sheet.getRow(i+1).getCell(21).getNumericCellValue())  
{
    if(sheet.getRow(i).getCell(10).getNumericCellValue()!=1 &&  
sheet.getRow(i).getCell(11).getNumericCellValue()!=1 &&  
sheet.getRow(i).getCell(9).getNumericCellValue()!=1 &&  
    && sheet.getRow(i+1)==null  
    && sheet.getRow(i+1).getCell(21).getNumericCellValue()!=pairNumber))  
    
    inducingCommitRowNo=sheet.getRow(i+1).getRowNum();  
}  
else  
    inducingCommitRowNo = sheet.getRow(i).getRowNum();  
}  
int istartRow = 0;  
if(inducingCommitRowNo - startingCommitRowNo <= givenInt)  
{
    istartRow = startingCommitRowNo;  
}  
else  
{
    istartRow = inducingCommitRowNo - givenInt;  
}  
ArrayList measure1Values = new ArrayList<Double>();  
ArrayList measure2Values = new ArrayList<Double>();
for(int iRow = istartRow ; iRow<inducingCommitRowNo; iRow ++)
{
if(sheet.getRow(iRow).getCell(21).getNumericCellValue()!=pairNumber) // this will imply the current pair
{
iRow++; 
}
else {
if (sheet.getRow(iRow).getCell(measure1) != null && sheet.getRow(iRow).getCell(measure2) != null) {
measure1Values.add(sheet.getRow(iRow).getCell(measure1).getNumericCellValue());
measure2Values.add(sheet.getRow(iRow).getCell(measure2).getNumericCellValue());
}
}
}

double[] measure1ValuesArray = new double[measure1Values.size()];
double[] measure2ValuesArray = new double[measure2Values.size()] ;
for(int imeasures=0; imeasures<measure1ValuesArray.length; imeasures++)
{
measure1ValuesArray[imeasures] = (double) measure1Values.get(imeasures);
measure2ValuesArray[imeasures] = (double) measure2Values.get(imeasures);
}
if(measure1ValuesArray.length>=2) {
corr = new PearsonsCorrelation().correlation(measure1ValuesArray, measure2ValuesArray);
String output = sheet.getRow(i).getCell(21).getNumericCellValue() + ";" + sheet.getRow(i).getCell(1).getStringCellValue() + ";" + corr + ";" + measure1ValuesArray.length + ";" + measure1;
if(column==0 || newSheet.getRow(cellNo)==null) newSheet.createCell(cellNo).setCellValue(output);
else
    newSheet.getRow(cellNo).createCell(column).setCellValue(output);
    cellNo++;

    System.out.println(output);
}
    i++;}
else {
    i++;
}
}
else {
{
    i++;
    }
}
public static void main (String args[]) throws IOException {
    int ncloc=12, new_sqale_debt_ratio=13, violations=14, code_smells=15, functions=16, lines=17, classes = 18, class_complexity=22;
    int givenInt = 5; // 10 for calculating for 10 commits

    String path = "path";
    int column = 0;
    cellNo = 0;
    calculateCorrelation(ncloc, new_sqale_debt_ratio, givenInt,path, column);
    column++;
    cellNo = 0;
B.2. Algorithm to calculate the correlation of 5 and 10 commits before bug inducing commit 105

calculateCorrelation(violations, new_sqale_debt_ratio, givenInt, path, column);
column++;
    cellNo = 0;
calculateCorrelation(code_smells, new_sqale_debt_ratio, givenInt, path, column);
column++;
    cellNo = 0;
calculateCorrelation(functions, new_sqale_debt_ratio, givenInt, path, column);
column++;
    cellNo = 0;
calculateCorrelation(lines, new_sqale_debt_ratio, givenInt, path, column);
column++;
    cellNo = 0;
calculateCorrelation(classes, new_sqale_debt_ratio, givenInt, path, column);
column++;
    cellNo = 0;
calculateCorrelation(class_complexity, new_sqale_debt_ratio, givenInt, path, column);
FileOutputStream fileOut_v1 = new FileOutputStream(filename_v1);
workbook.write(fileOut_v1);
//closing the Stream
fileOut_v1.close();
//closing the workbook
//workbook.close();
//prints the message on the console
System.out.println("v1 Excel file has been generated successfully.");
column++;
    cellNo = 0;
calculateCorrelation(class_complexity, new_sqale_debt_ratio, givenInt, path1, column);
FileOutputStream fileOut_v2 = new FileOutputStream(filename_v2);
workbook.write(fileOut_v2);
//closing the Stream
fileOut_v2.close();
//closing the workbook
workbook.close();
//prints the message on the console
System.out.println("v2 Excel file has been generated successfully.");
}
}

B.3 Algorithm to calculate the Slope of 5 and 10 commits before bug inducing commit

import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.ss.usermodel.Row;
import org.apache.poi.xssf.usermodel.XSSFRow;
import org.apache.poi.xssf.usermodel.XSSFSheet;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.ArrayList;
import static java.lang.Math.atan;
import static java.lang.Math.toDegrees;

public class SlopeCalculation_normalisedPairwise {
    public static String filename = "outPath";
B.3. Algorithm to calculate the Slope of 5 and 10 commits before bug inducing commit 107

public static XSSFWorkbook workbook = new XSSFWorkbook();
public static int cellNo;
public static XSSFSheet newSheet = workbook.createSheet("sheet1");
private static void calculateSlope(int measure, String path, int column) throws IOException {
    FileInputStream fis = new FileInputStream(new File(path));
    XSSFFormat wb = new XSSFFormat(fis);
    XSSFSheet sheet = wb.getSheetAt(0);
    int activePairNo = 0;
    int i = 2;
    while (sheet.getRow(i) != null) // loop to iterate pair to pair
    {
        Row rowPairWise = sheet.getRow(i);
        int inducingCommitRowNumber = 2;
        int inducingCommit = 1;
        if (rowPairWise.getCell(0) != null && rowPairWise.getCell(0).getNumericCellValue() == 1) {
            ArrayList measureValues = new ArrayList<Double> ();
            ArrayList commitsValues = new ArrayList<Double> ();
            activePairNo = (int) sheet.getRow(i).getCell(21).getNumericCellValue();
            int startRowNumber = rowPairWise.getRowNum(); // get the startRowNumber of the pair
            Row rowInducingCommit = sheet.getRow(2);

            for (int j = startRowNumber; j < sheet.getPhysicalNumberOfRows(); j++) // loop to get the inducing commit
            {
                rowInducingCommit = sheet.getRow(j);
                if (rowInducingCommit != null && rowInducingCommit.getCell(10) != null && rowInducing-
                    Commit.getCell(10).getNumericCellValue() == 1 || (rowInducingCommit != null && rowInduc-
                    ingCommit.getCell(11) != null && rowInducingCommit.getCell(11).getNumericCellValue() == 1) ||
                    (rowInducingCommit != null && rowInducingCommit.getCell(9) != null && rowInducingCommit.
                    getCell(9).getNumericCellValue() == 1) || (rowInducingCommit != null && rowInducingCommit.get.
                    Cell(21) != null)) { // some conditions
mit.getCell(21).getNumericCellValue() != activePairNo))
{
    inducingCommitRowNumber = rowInducingCommit.getRowNum();
    break;
}

int givenInt = 5; // 10 for calculating for 10 commits
int imeasure = 0;
Row rowCalculateSlope = sheet.getRow(2);
if (inducingCommitRowNumber - startRowNumber <= givenInt) // if number of commits before inducing is less than or equal givenInt
{
    for (int j = startRowNumber; j < inducingCommitRowNumber; j++) {
        rowCalculateSlope = sheet.getRow(j);
        if (imeasure < givenInt) {
            if (rowCalculateSlope.getCell(measure) != null)
                measureValues.add(rowCalculateSlope.getCell(measure).getNumericCellValue());
            commitsValues.add(rowCalculateSlope.getCell(0).getNumericCellValue());
            imeasure++;
        }
    }
}
else // if number of commits before inducing is more than givenInt
{
    for (int j = (inducingCommitRowNumber - givenInt); j < inducingCommitRowNumber; j++) {
        // System.out.println("inside loop" + j);
        rowCalculateSlope = sheet.getRow(j);
        if (imeasure < givenInt) {
            if (rowCalculateSlope.getCell(measure) != null)
                measureValues.add(rowCalculateSlope.getCell(measure).getNumericCellValue());
            commitsValues.add(rowCalculateSlope.getCell(0).getNumericCellValue());
        }
    }
}
double x = 0, y = 0, x2 = 0, xy = 0;
int limit = measureValues.size() - 1;
double max = findMax(measureValues); // limit for normalization example, range from 0-9 for 10 commits and range 0-4 for 4 commits
double min = findMin(measureValues);
for(int j=0; j< measureValues.size(); j++)
{
    double value = (double) measureValues.get(j);
    double new_value = ((value-min)/(max-min))*limit;
    measureValues.set(j, new_value);
}
for (int j = 0; j < measureValues.size(); j++) {
    x = x + (double) commitsValues.get(j);
    y = y + (double) measureValues.get(j);
    xy = xy + ((double) commitsValues.get(j) * (double) measureValues.get(j));
    x2 = x2 + ((double) commitsValues.get(j) * (double) commitsValues.get(j));
}
int n = measureValues.size();
double slope = ((n * xy) - (x * y)) / ((n * x2) - (x * x));
slope = slope;
slope = atan(slope);
slope = toDegrees(slope);
"," + measure;
if(column==0 || newSheet.getRow(cellNo)==null) new-
Sheet.createCell(column).setCellValue(output);

else
newSheet.getRow(cellNo).setCellValue(output);
cellNo++;
System.out.println(output);
i++;

} else {
i++; // just to lower the number of iteration in the sheet (increases the performance of code)
}

private static double findMin(ArrayList measureValues) {
    double min = Integer.MAX_VALUE;
    for(int j=0; j< measureValues.size(); j++)
    {
        if(min > (double) measureValues.get(j))
        {
            min = (double) measureValues.get(j);
        }
    }
    return min;
}

private static double findMax(ArrayList measureValues) {
    double max = Integer.MIN_VALUE;
    for(int j=0; j< measureValues.size(); j++)
    {
        if(max< (double) measureValues.get(j))
        {

```
B.3. Algorithm to calculate the Slope of 5 and 10 commits before bug inducing commit 111

max = (double) measureValues.get(j);

}

return max;

}

public void main () throws IOException {

// in this code normalization is done pairwise in code
int ncloc = 12, new_sqaleDebtRatio = 13, violations = 14, code_smells = 15, functions = 16,
lines = 17, classes = 18, classComplexity = 22;
String path = ”path”;

int column=0;
cellNo = 0;
calculateSlope(ncloc, path, column);
cellNo = 0;
column++;
calculateSlope(new_sqaleDebtRatio, path, column);

   cellNo = 0;
column++;
calculateSlope(violations, path, column);

   cellNo = 0;
column++;
calculateSlope(code_smells, path, column);

   cellNo = 0;
column++;
calculateSlope(functions, path, column);
cellNo = 0;
column++;
calculateSlope(lines, path, column);
    cellNo = 0;
column++;
calculateSlope(classes, path, column);
    cellNo = 0;
column++;
calculateSlope(classComplexity, path, column);
FileOutputStream fileOut = new FileOutputStream(filename);
workbook.write(fileOut);
    //closing the Stream
fileOut.close();
    //closing the workbook
workbook.close();
}
Curriculum Vitae

Name: Parul

Post-Secondary Education and Degrees:
Chandigarh University, Mohali, Punjab, India
2015 - 2019 B.E.

University of Western Ontario, London, ON
2022 - 2023 M.Sc.

Honours and Awards:
Western Graduate Research Scholarship 2022-2023

Related Work Experience:
Teaching Assistant
The University of Western Ontario
2022 - 2023