6-18-2024 11:00 AM

Framework for Bug Inducing Commit Prediction Using Quality Metrics

Alireza Tavakkoli Barzoki, *Western University*

Supervisor: Kontogiannis, Kostas, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science

Abstract

This thesis relates to the topic of software defect prediction within the broader area of continuous software engineering. The approach presented in this thesis is employing source code and process metrics obtained for each commit, and is examining as to whether specific patterns, as the system moves from one commit to another, can predict an impending bug inducing commit. The thesis utilizes the SonarQube *Technical Debt* open source data which provides source code metrics and process metrics for each commit in 22 medium to large scale open source Apache projects.

Central to this research is the novel utilization of commits to trace transitions to bug-inducing commits, facilitating the construction of a predictive model. In this approach, each commit is denoted by a vector of metrics values which have undergone pre-processing so can be efficiently used. Each such a vector defines the "state" of a commit. A significant portion of the methodology is devoted to meticulous data preparation and analysis, including the delineation of commit transitions, feature selection, and rigorous data cleansing. This rigorous process is aimed at enhancing the precision and accuracy of pattern recognition, particularly in identifying transitions leading to bug-inducing commits.

Through the integration of advanced methodologies encompassing correlation analysis, clustering techniques (including K-Means and Hierarchical clustering), and a suite of classification strategies such as K-Means, Decision Trees, and innovative percentile-based classification, the study aims to identify emerging vector metrics state transition patterns which may be indicative of potential software bugs.

The results indicate that the proposed technique is promising on recognizing patterns indicative of potential impending bug inducing commits and sheds light on the practical implications of utilizing commit transitions in defect prediction strategies, offering insights into enhancing software development processes.

Keywords: Bug-inducing commit prediction, Quality metrics, Software engineering, Code commits, Predictive model, Clustering, Classification, Data preparation, Pattern recognition, Software defects

Summary for Lay Audience

This thesis explores an innovative approach in the field of software defect prediction, which is a critical area of continuous software engineering. The study focuses on using metrics from source code and processes, collected at each update or ”commit” in software projects, to predict future errors, or ”bugs,” that might be introduced into the software.

The research utilizes data from SonarQube Technical Debt, which includes detailed metrics from 22 substantial open-source Apache software projects. By examining the patterns in these metrics as the software changes over time, the thesis aims to identify early warnings of commits that could lead to software defects.

Key to this study is the idea of treating each software update as a ”state” defined by specific metric values. The thesis involves intensive data processing, including selecting important features and cleaning data thoroughly, which helps in accurately identifying patterns that precede software bugs.

Using sophisticated statistical and machine learning methods, including various clustering and classification techniques, the thesis strives to detect these critical patterns. The findings suggest that this method could be quite effective in foreseeing risky commits, providing valuable insights that could help improve software development practices by allowing teams to preemptively address potential issues.

Contents

List of Tables

List of Figures

List of Appendices

Chapter 1

Introduction

1.1 Preface

In the realm of software development, the presence of defects can result in significant setbacks, both in terms of time and resources. As the complexity of software projects continues to grow, the importance of effectively predicting and preventing defects becomes increasingly evident. In this thesis, we delve into the world of defect prediction, examining the various approaches, tools, and strategies that can empower software professionals to proactively identify and rectify defects, ensuring the delivery of high-quality software while adhering to tight project schedules. We focus on two main objectives. First, we aim to prepare our data by pre-processing it and creating the commit state model. Second, we intend to conduct a rigorous data analysis to derive meaningful conclusions regarding the prediction of imminent bug-inducing commits.

Contemporary research in the field of software defect prediction commonly generates datasets, methodologies, and frameworks that enable software engineers to concentrate their efforts on defect-prone code during development. This, in turn, enhances software quality and optimizes resource utilization, ultimately leading to more efficient development activities [100].

1.2 Motivation

In the realm of software development, the paradigm of continuous software engineering has emerged as a transformative approach to delivering high-quality software products rapidly and efficiently. At the heart of continuous software engineering lies the concept of the shift-left approach, which advocates for the early integration of quality assurance practices into the software development lifecycle.

However, despite the advancements in continuous software engineering, software defects remain a persistent challenge that can undermine the reliability and quality of software products. Traditional defect detection methods often rely on post-development testing, which can be time-consuming and costly to rectify. As software systems grow in complexity and scale, the need for proactive defect prediction becomes increasingly crucial.

The shift-left approach emphasizes the importance of detecting and addressing defects as early as possible in the development process. By integrating defect prediction techniques into

the early stages of development, software teams can identify potential issues before they manifest into critical bugs, thus minimizing the impact on project timelines and budgets.

Therefore, there is a compelling motivation to explore defect prediction within the context of continuous software engineering and the shift-left approach. By leveraging predictive analytics and machine learning algorithms, software development teams can gain insights into potential defect patterns, enabling them to take proactive measures to mitigate risks and improve software quality.

This thesis aims to contribute to the body of knowledge in defect prediction by investigating how predictive analytics can be integrated into the continuous software engineering paradigm. By examining the benefits and challenges of applying defect prediction techniques early in the development lifecycle, this research seeks to empower software teams to build more resilient and reliable software systems.

1.3 Rationale

In today's software development landscape, ensuring reliability, efficiency, and robustness in software systems is paramount. Software defects, commonly known as "bugs," not only compromise software quality but also incur substantial costs and can have far-reaching consequences, including compromised functionality and security vulnerabilities. Mitigating these defects is crucial in modern software engineering, where software underpins various aspects of daily life, from communication and commerce to healthcare and transportation [80].

This study holds significant potential to revolutionize defect prediction strategies, ultimately reducing defects in software systems and enhancing software quality. As software continues to evolve and integrate further into our lives, the importance of this research becomes increasingly evident. The rationale for this study lies in the need to enhance defect prediction methodologies by leveraging commit transitions to bug-inducing states within software engineering.

Commit transition modeling involves analyzing commit sequences within the version control system to identify patterns leading to bug-inducing commits. This enables early detection of potential defects, facilitating proactive intervention to prevent issues. By examining commit transitions, we gain a detailed understanding of how changes interact over time, identifying subtle relationships contributing to bug introduction.

This approach captures contextual information such as modified files, involved developers, and timing, providing a comprehensive view of the development process. Leveraging historical data and machine learning, predictive models can recognize patterns indicative of bug-inducing commits, aligning with the shift-left philosophy of early defect prevention.

Our methodology entails analyzing historical commit data to identify patterns associated with bug-inducing commits. We anticipate that our approach will enable:

- Early defect detection, - Improved development process understanding, - More accurate bug prediction, and - Reduced software defects and associated costs.

1.4 Thesis Contribution

Traditional defect prediction methods often rely on historical process and source code metrics, but may not fully capture the intricate metrics patterns from one commit to another, leading ultimately to a bug-inducing commit. This thesis explores defect prediction by considering that a collection of metrics values that correspond to process and source code features defines a state for each commit. The central hypothesis is that the analysis of how commit states evolve and the identification of state transition patterns could serve as predictors of impending bug-inducing commits. The thesis contributions can be summarized as follows:

- **Exploration of defect prediction in software development:** This thesis delves into the area of defect prediction, aiming to understand its nuances and complexities within the software development lifecycle. In its core, aims to model each commit as a collection of normalized metrics for each given commit, and which collectively define a *state* for this commit.

- **Analysis and selection of useful features for defect prediction:** Another contribution lies in analyzing a large pool of metrics and identify a set that can be used for defect prediction. These metrics provide valuable insights, enabling informed decision-making and continuous improvement in software development practices.

- **Enhancement of software quality and reliability:** By investigating defect prediction, the overarching aim is to contribute to the improvement of software quality and reliability, crucial for user satisfaction and system effectiveness.

- **Identification of bug indicators:** A key contribution is the identification of commit transition commit state patterns serving as potential indicators of software bugs. Understanding these patterns enables proactive measures to address and prevent issues in future development cycles.

- **Analysis of development process patterns:** Through empirical analysis, this thesis seeks to identify patterns within the software development process correlated with the occurrence of defects. This insight informs development practices and decision-making, mitigating risks effectively.

- **Proactive issue mitigation and prevention:** By uncovering patterns indicative of potential defects, the goal is to equip software development teams with knowledge and tools for proactive mitigation and prevention, enhancing overall software reliability.

- **Enhancement of development practices:** Ultimately, the culmination of these contributions aims to enhance the quality and reliability of software development practices. Leveraging insights from defect prediction, development teams can adopt proactive strategies to deliver higher-quality software products.

1.5 Scope and Limitations

In the context of this thesis, our twofold objective is to firstly curate an extensive and well-structured dataset encompassing various aspects of software development and its associated defect history. Subsequently, we will conduct rigorous data analysis to discern critical transitions and patterns within this dataset. These findings will enable us to draw meaningful conclusions regarding the causes and correlations of software bugs and defects, ultimately equipping us to formulate predictive models for anticipating and mitigating such issues in the dynamic landscape of software development.

One of the principal limitations of this research stems from data constraints. Despite working with a sizable database encompassing 21 distinct software projects and an initial commit database measuring 18.8 gigabytes, the data cleaning and curation process incurs substantial data loss. To put it differently, the proportion of usable data within our dataset is notably limited, but still usable. Extracting valuable insights from this abundance of data amidst a deluge of extraneous information proves to be a formidable challenge, necessitating extensive hours of analytical efforts.

Another unavoidable challenge encountered throughout this project is the constraint imposed by processing power. Dealing with a vast database and executing a multitude of resource-intensive scripts demands either a robust processing system or a substantial amount of time. Regrettably, we found ourselves lacking in both regards, which presented a significant impediment to the smooth progression of our work. The sheer size of the database was such that attempting to open a single table and visualize the data would result in system crashes, making it practically impossible to run queries directly. Consequently, the extraction and representation of data necessitated the use of specialized scripts to navigate and interact with the database.

1.6 Thesis Outline

In this introductory chapter, we provide an overview of the research to be presented in this thesis. We outline the key chapters that structure our investigation and provide a preview of the contents within. The introduction serves as the gateway to our study, laying the foundation for the subsequent chapters. Building upon the introduction, ”Chapter 2: Related Work & Background” delves into the historical and theoretical context of our research. We review the existing body of knowledge in the field, highlighting the key concepts, gaps, and relevant literature that motivate our study. This chapter provides the necessary background and context for a thorough understanding of the research presented.

In the subsequent two chapters, our methodology will be divided and detailed. To facilitate a clearer understanding of the processes employed in this thesis, an activity diagram has been provided at the beginning of each of these two chapters.

”Chapter 3: Data Pre-processing” details the research methods and approaches we employ to collect, analyze, and interpret data. We delve into the specifics of our research design, data sources, and the techniques used in our study. This chapter offers insight into the methods employed to prepare our data for our research and bring them into the proper format. In ”Chapter 4: Commit State Modeling,” we organize our pre-processed data into the desired format to develop a Commit State model, which is the core of our research. This model enables us to

conduct experiments that test and refine its accuracy, ultimately providing insights into patterns and trends linked to potential defects in future commits. By learning from historical data, the model helps us build a predictive tool that anticipates and mitigates software defects, improving coding practices and enhancing software quality. In ”Chapter 5: Experimental Results,” we present the empirical findings derived from our analysis and data exploration. This section serves as the core of our research, providing a comprehensive account of the results obtained. We interpret the data and highlight key patterns and insights that emerge from our study. The final chapter, ”Chapter 6: Conclusion,” synthesizes the key findings, contributions, and implications of our research. We reflect on the research objectives outlined in the introduction, discuss the significance of our results, and propose potential avenues for future research.

Chapter 2

Related Work & Background

2.1 Preface

In this chapter, we will review relevant literature and studies that are closely related to our research topic. This overview will provide context and insight into existing methodologies, approaches, and findings in the field. We will examine how these related works have influenced the current state of research and where there may be gaps or opportunities for further exploration.

Following the literature review, we will introduce and explain the key technical terms and concepts used throughout this document in the Background section. This discussion will provide clarity and ensure a common understanding of specialized terminology, theoretical frameworks, and technical methodologies that form the foundation of our research. By establishing this groundwork, we aim to offer a comprehensive understanding of the research landscape and set the stage for the subsequent analysis and discussion.

2.2 Related Work

2.2.1 Defect Prediction

The increased complexity of software systems makes the assurance of their quality very difficult. Therefore, a significant amount of recent research focuses on the prioritization of software quality assurance efforts. One line of work that has been receiving an increasing amount of attention for over 40 years is software defect prediction, where predictions are made to determine where future defects might appear. Since then, there have been many studies and many accomplishments in the area of software defect prediction. At the same time, there remain many challenges that face that field of software defect prediction [45]. We can build a prediction model with defect data collected from a software project and predict defects in the same project, i.e. within-project defect prediction. Researchers also proposed cross-project defect prediction to predict defects for new projects lacking in defect data by using prediction models built by other projects [59].

Defect Prediction Using Software Metrics

Determining whether a component is likely to be defective is strongly linked to several software metrics. Identifying and measuring these metrics is crucial for multiple purposes, such as estimating program performance, assessing the effectiveness of software processes, gauging the effort required for different processes, predicting the number of defects during development, and overseeing software project processes [73] [98]. Various software metrics commonly used for defect prediction include lines of code (LOC), McCabe metrics, Halstead metrics, and object-oriented software metrics. Consequently, automating the prediction of defective components using these metrics has become a prominent area of research [36].

Defect Prediction Using Machine Learning

Numerous machine learning techniques have been suggested and evaluated to tackle the software bug prediction challenge. These approaches include decision trees [46], neural networks [110] [97], Naive Bayes [54] [39], support vector machines [29], Bayesian networks [61], and Random Forests [7].

Dejaeger [13] conducted a study using data from an open-source dataset provided by the Eclipse Foundation and the NASA IV&V facility to assess the performance of fifteen Bayesian Network (BN) classifiers. The study utilized Halstead, Lines of Code, and McCabe complexity metrics for the evaluation. The findings suggest that the Augmented Naïve Bayes Classifier can achieve performance levels that are equal to or better than the Naïve Bayes Classifier.

In [24], Guisheng Fan introduced a model called defect prediction via an attention-based recurrent neural network (DP-ARNN). The process consists of several steps: First, DP-ARNN analyzes the abstract syntax tree of programs and converts them into vectors. Then, it encrypts these vectors as inputs to DP-ARNN, allowing the model to automatically learn syntactic and semantic features. This approach also includes a mechanism for producing features for precise defect prediction. To assess the model, the author used F1-score and area under the curve as evaluation criteria for seven open-source Java projects. The F1-score values for DP-ARNN, RNN, and CNN are 0.515, 0.506, and 0.473, respectively, while RF + RBM and RF achieve 0.310 and 0.396. In performance comparisons, DP-ARNN, RNN, and CNN outperformed traditional methods.

Cagatay Catal's work [7] examined how the size of a dataset, the selection of metric sets, and the choice of feature selection technique influence the outcomes of software fault prediction. The study found that random forest performs the best for large datasets, while the Naïve Bayesian Network algorithm is most effective for small datasets. In this study, 13 metrics were chosen to assess and compare the performance of different algorithms.

In [109], C. W. Yohannese proposed a framework for software default prediction and evaluated the model's performance across four scenarios: learning from standard datasets, feature-selected datasets, balanced feature selection data subsets, and noise-filtered and balanced feature selection subsets. The author concludes that combining feature selection, data balancing, and noise filtering methods for data preprocessing leads to better software default prediction performance compared to not using these techniques.

Ezgi Erturk's study [22] employs McCabe metrics to assess the performance of predictive models. The study utilizes both Artificial Neural Network and Support Vector Machine to

gauge the performance of an Artificial Neural Network Inference System. The results using McCabe metrics are 0.7795 for Support Vector Machine, 0.8685 for Artificial Neural Network, and 0.8573 for the Artificial Neural Network Inference System [22].

In a study conducted by Xin Xia [107], deep-learning techniques are proposed to predict fault proneness in code files. The framework consists of two phases: the model-building phase and the prediction phase. In the model-building phase, the goal is to create a statistical model based on historical changes, while in the prediction phase, the aim is to determine whether new changes are buggy or clean. The proposed framework utilizes 14 features and incorporates data preprocessing, which includes data normalization and resampling. The framework's performance is validated against six large open-source projects using F1-score and cost-effectiveness metrics. The results indicate a recall of 0.69 and an F1-Score of 0.45. In terms of cost-effectiveness, 20% of the lines of code help the framework predict 50% of defect-related changes.

File-level Defect Prediction

Moeyersoms et al. address the common challenge in rule-based classification systems: balancing comprehensibility and performance[57]. Typically, models that are easier to understand gain greater acceptance but often at the cost of reduced predictive accuracy. They use ALPA to discover a ruleset that mirrors the performance of complex black-box models, such as those using Random Forest and Support Vector Regression. The extracted rules lead to results similar to those of complex models while being easier to interpret.

Ostrand et al. [63] investigate whether including information about individual developers can help predict future faults in software releases by the same developer. They add this metric to an existing model based on Negative Binomial Regression, which already incorporates metrics such as file size, the number of releases a file has been part of, changes a file has undergone, previous faults found, and the programming language used. By predicting the number of faults each file may contain and focusing on the worst-performing 20% of files, they find these files account for 75% of the total faults. However, their findings suggest that including the developer metric only slightly improves results by about 1% and isn't a particularly effective measure for predicting a file's fault proneness.

Package-level Defect Prediction

Nagappan et al. [58] integrate historical failure data from bug databases with code complexity metrics to predict faulty Windows components, such as individual binaries. By mining five major commercial Microsoft projects, they gathered data on past faulty entities and applied Principal Component Analysis to find the best combination of metrics. These were then used in multiple regression models alongside past defect data. Their experimentation with applying failure metrics from one project to predict defects in another, even without previous failure history, showed that the effectiveness of such an approach depends on the interdependency of the projects. The R^2 values of the predictor across the projects ranged from 0.416 to 0.882.

Schroter et al. [77] explore whether certain problem domains are more fault-prone than others. They define a project's problem domain as the set of components it uses and assert that decisions made during the design phase impact future defect trends. By collecting data

on post-defect failures, files related to those failures, and import statements in those files, they provide a framework for managers to allocate resources efficiently based on potential fault-prone domains.

Class-level Defect Prediction

In their study, Huang et al. [40] apply Multi Instance Learning (MIL) to address the challenge of differing logical levels between training and validation sets, which can be a barrier for traditional supervised techniques. By treating classes as ”bags” and their attributes and methods as ”instances,” MIL preserves relational information. When applied to a NASA mission control project using four different MIL algorithms, the approach outperforms existing supervised learning methods in terms of accuracy and error rates.

Singh et al. [83] introduce two object-oriented metrics for identifying classes with poor structure and potential code smells. Public Factor evaluates the visibility of a class's methods and attributes, while Encapsulation Factor examines the relationship between a class's cohesion and data visibility. Using a binary regression model on Mozilla data, they found these metrics to be effective in predicting faulty classes and improving the AUC of a categorical model by 1% to 30% for detecting code smells.

Pandey et al. use Fuzzy Inferencing to predict buggy modules in [64]. They use a decision tree to derive rules for a Fuzzy Inference System, classifying software metrics into five categories: very low, low, medium, high, and very high. Modules are categorized as faulty or non-faulty and ranked based on their fault proneness. When tested on NASA's KC2 project data, their model achieved a classifier accuracy of 87.37%, surpassing other referenced models.

Method-level Defect Prediction

Kim et al. [47] introduce the concept of cache in software bug prediction, proposing that a file causing a fault may continue to produce faults and impact related entities. They create a cache, FixCache, which automatically updates each time a fix is committed and monitors the affected entities. Their approach results in a 73-95% hit rate for file-level entities and a 46-72% hit rate for method-level entities using a 10% cache.

Mizuno et al. [56] suggest using spam-based classification for defect prediction, leveraging advanced spam filtering techniques to classify source code as faulty or non-faulty. Utilizing CRM114 spam filtering software, they experiment with three classification techniques: Sparse Binary Polynomial Hash Markov model (SBPH), Orthogonal Sparse Bigrams Markov model (OSB), and Simple Bayesian model (BAYES). Testing the model on two open-source projects, they find SBPH achieves the highest accuracy with an average of 77.5% and acceptable recall and precision. OSB has better recall but lower accuracy than SBPH, while BAYES offers the best recall but poor accuracy and precision, making it less optimal.

Elish et al. [20] employ Support Vector Machines (SVMs) to identify buggy functions or methods, using four mission-critical NASA projects to evaluate the model's performance. They compare SVM results with eight other statistical and machine learning models, finding that SVMs have a better F1 score than five of them. Although SVMs show lower precision than most models, they achieve higher recall than all, which is significant in software testing as it reduces the risk of missing a fault.

2.2.2 Mining Software Repositories

Numerous research teams have explored the advantages of mining software repositories, including those that maintain change logs of software systems like GitHub, as well as those that hold supplementary information such as Bugzilla and Jira. This approach enables analysts to directly extract data from GitHub repositories and examine it to identify patterns that may signal defects in the software. In reference [44], Kalliamvakou et al. examine the quality of data extracted from GitHub repositories and offer researchers a set of recommendations on how to work with such datasets. They also highlight potential risks of using GitHub, which they discovered during their study. In reference [6], H. Inayat et al. utilized repository mining to gather data from GitHub projects, including details on commit history, code changes, and developer activity. They leveraged this data to forecast defects in the software development process using machine learning algorithms.

Another approach toward Mining software repositories taken by researchers is Bug-to-bug fixing commit link recovery, which involves examining version control systems to understand the connections between bugs and the commits that resolve them. In reference [3], Bachmann et al. address the challenge of recovering links for bug-fixing commits by mitigating the threat to validity posed by using inaccurate link recovery methods. Their approach includes extracting ground truth data and using it to assess the performance of a link recovery tool. In reference [96], Tantithamthavorn et al. explore how data mislabeling affects the overall performance of automated defect prediction models.

2.2.3 Software Metrics

In the current age of continuous engineering, frequent additions of new features in small increments under tight deadlines can sometimes lead to defects being overlooked. Nevertheless, maintaining high software quality remains crucial for organizations. This has been a key focus in software engineering research, where metrics are used to evaluate and improve software development and maintenance, resulting in higher-quality systems [66].

Software metrics provide quantitative measures of software attributes. By tracking and analyzing these metrics, we can assess code quality and predict bugs or other issues [65]. Metrics can be divided into process, project, and product categories. Process metrics assess software development and maintenance processes, helping to estimate system size and verify project timelines [82]. Project metrics examine various facets of a project, such as size, complexity, performance, and design features, aiding project managers in optimizing workflows [62].

Product metrics evaluate a product's properties at different stages of the Software Development Lifecycle (SDLC), including source code and design documents [88]. Metrics such as software quality metrics overlap multiple categories, such as process and product metrics. Code metrics, a subset of product metrics, are crucial for comparing system components and enhancing their performance. They can be further classified into dynamic and static code metrics based on whether the measurements occur during or before code execution [52].

Dynamic code metrics are gathered while the program runs to assess system behavior by comparing actual and expected outcomes, while static code metrics are collected prior to code execution [52]. These metrics aid in evaluating code complexity and lines of code. Selecting the right metrics can improve defect prediction performance. Researchers typically focus on

four main classes of metrics for predicting defects: quality metrics, process metrics, static code metrics, and technical debt metrics [70].

Size Metrics

Size metrics are among the earliest and most widely used static metrics for measuring software product size, providing an intuitive, well-defined, and easy-to-compute indication of programmer productivity [55] [28]. These metrics, such as lines of code (LOC), are chosen as primary metrics in most studies of software quality assessment or prediction models [106] [108]. Variants like source lines of code (SLOC) and Kilo-SLOC (KSLOC) offer refinements by excluding comments, empty lines, and non-functional symbols. However, LOC-based metrics have limitations [4] [49] [21].

Function Points (FP) is another significant size metric developed after LOC by Albrecht [2]. FP evaluates the volume of functionality delivered by a software module by considering inputs, outputs, inquiries, internal files, and external interfaces. Quality metrics such as defects per FP and defects per SLOC derive from FP and SLOC and serve as important measures of defect density.

Complexity Metrics

Complexity metrics are another type of static metric used in software measurement. One well-known example is McCabe's cyclomatic complexity (CC), introduced in [53]. CC quantifies the number of linearly independent paths in a program's control flow, offering insight into the complexity of the code's logic.

Object-Oriented Metrics

Object-Oriented Metrics (OO metrics) are used to measure the properties of code developed using object-oriented languages. The CK metrics suite, proposed by Chidamber and Kemerer in 1994, is a well-known example of OO metrics [11]. It includes measures such as Number of Children (NOC), Coupling Between Object Classes (CBO), Depth of Inheritance Tree (DIT), Weighted Method Count (WMC), and Lack of Cohesion in Methods (LCOM). Unlike McCabe's cyclomatic complexity, which focuses on procedural complexity, OO metrics aim to quantify higher-level inter-class or inter-method relationships. These metrics have become a robust choice for assessing OO systems, particularly as OO methodologies gain popularity [69] [42].

OO metrics have been validated in the context of software quality assurance through various studies. Singh et al. [84] conducted an empirical validation of the CK metrics suite using regression and machine learning models with a public NASA dataset. They found that CBO and WMC performed well in predicting fault proneness, while DIT, LCOM, and NOC were less reliable. Gyimothy et al. [35] examined the fault predictability of OO metrics, along with LOC, in the open-source Mozilla software system. Their experiment generally corroborated Singh's findings: CBO is a strong predictor of defect proneness, while DIT and NOC are unreliable. Additionally, they found LOC to perform well, concluding that it is an excellent candidate for a quick defect prediction model.

2.3 Background

2.3.1 Continuous Delivery

Continuous delivery (CD) is a software engineering methodology where teams work in short cycles to produce software that can be consistently released at any time. The process follows a pipeline through an environment similar to production, allowing for automation and minimizing manual intervention [10]. Continuous delivery focuses on building, testing, and releasing software more quickly and frequently. This approach helps minimize the cost, time, and risk associated with deploying changes by enabling smaller, incremental updates to applications in production. A simple, consistent deployment process is crucial for effective continuous delivery [79].

Continuous delivery views a deployment pipeline [41] as a lean Poka-Yoke [26], meaning it serves as a series of checks and validations that software must undergo before being released. Whenever code changes are committed to a source control repository, the build server compiles and packages the code if needed. Afterward, the code undergoes various testing methods, potentially including manual testing, to ensure it meets release standards.

2.3.2 Shift-Left Testing

Shift-left testing is a strategy in software and system testing that emphasizes performing testing earlier in the development lifecycle. This approach aims to move testing tasks to the left side of the project timeline, closer to the initial stages of development [25].

The concept is based on the idea of ”testing early and often,” which helps catch defects and issues before they become more challenging and costly to address later in the process. By identifying problems early on, teams can take corrective actions sooner, leading to higher quality software and more efficient development cycles [76].

The term ”shift-left testing” was coined by Larry Smith in 2001. Since then, the approach has become a key practice in modern software development, especially within agile and DevOps environments. It aligns with the goal of continuous integration and continuous delivery (CI/CD), where testing is integrated into the workflow from the outset [71].

Shift-left testing can involve various practices, such as unit testing, static code analysis, and integration testing, at earlier stages of the development process. This allows for faster feedback and helps ensure that potential issues are addressed promptly, ultimately improving the overall quality and reliability of the software being developed.

2.3.3 Correlation

Correlation in statistics pertains to the statistical association between two random variables or bivariate data, often used to gauge the extent of their linear relationship. While correlations, such as the relationship between parent height and offspring height, can offer predictive insights, they do not inherently signify causation. These associations are mathematically quantified by correlation coefficients, such as the widely used Pearson correlation coefficient, which measures linear relationships between variables. Other coefficients, like Spearman’s rank correlation, are designed to capture non-linear associations. Correlation serves as a valuable tool

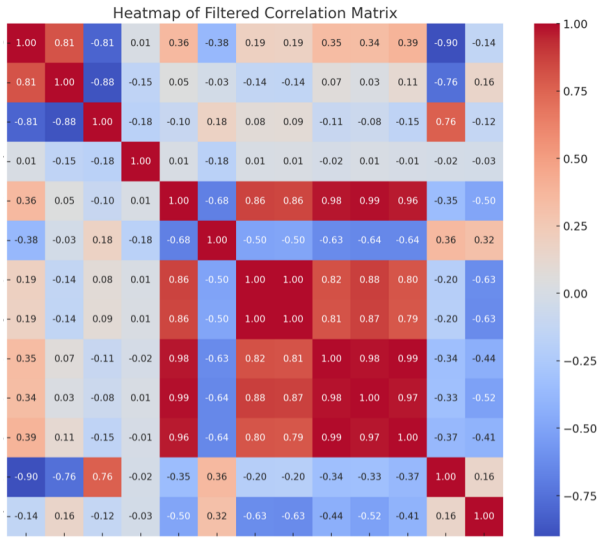


Figure 2.1: SonarQube Features Heatmap

for understanding relationships among variables, although it should be used judiciously, as it doesn't imply causation [12] [15].

Correlation Matrix

The ability to perform statistical comparisons between correlation coefficients measured on the same set of subjects in research is often crucial. This enables researchers to evaluate various aspects of their data. For instance, researchers might want to determine whether two different predictors are equally correlated with a specific outcome variable. In other cases, researchers may be interested in assessing whether an entire matrix of correlation values remains consistent over time. To facilitate these comparisons and tests, the article explores relevant literature, identifies statistics that should be approached with caution, and introduces a range of techniques suitable for use with medium to large datasets. The article also includes practical numerical examples to illustrate these concepts [89].

A correlation matrix is a tabular representation that displays correlation coefficients between different variables. Within this matrix, each cell provides the correlation measure between a pair of variables. This matrix serves multiple purposes, including data summarization, input for more sophisticated analyses, and as a diagnostic tool for advanced analytical procedures.

Usually a correlation matrix is represented by a heat-map. A heat map is a 2D data visualization method used to depict the values within a dataset by assigning them specific colors. The color variations in the heatmap can be represented by shifts in hue or intensity. While the term "heatmap" is relatively recent, the practice of shading matrices in this manner has been employed for more than a century [104]. You can see an example of what a heatmap is in Fig2.1.

2.3.4 Clustering

Cluster analysis, or clustering, is a fundamental task in data analysis aimed at grouping objects in a dataset based on their similarity to one another within clusters, with the objective of making objects in the same cluster more similar to each other than to those in different clusters. It is extensively used in various fields such as pattern recognition, image analysis, bioinformatics, and machine learning. Cluster analysis is not a specific algorithm but a general problem to be solved, with multiple algorithms available, each with its own perspective on defining clusters. The choice of the most suitable algorithm and parameters depends on the dataset and intended outcomes. Cluster analysis is an iterative, knowledge discovery process often involving adjustments to data preprocessing and model parameters to achieve desired results. Similar terms include automatic classification and community detection, with the focus varying between the groups formed and their discriminative power. Cluster analysis has historical roots in anthropology [18] and psychology [111] [99], with early applications in personality psychology classification [8].

K-Means Clustering

K-means clustering is a vector quantization method originating from signal processing, used to divide n observations into k clusters where each observation belongs to the cluster with the closest mean, forming a prototype of the cluster. This partitioning results in Voronoi cells (In mathematics, a Voronoi diagram is a partition of a plane into regions close to each of a given set of objects) within the data space. K-means minimizes within-cluster variances, specifically squared Euclidean distances, which is distinct from minimizing regular Euclidean distances. While the problem is computationally challenging (NP-hard (nondeterministic polynomial; at least as hard as any NP-problem)), efficient heuristic algorithms exist. These heuristics are akin to the expectation-maximization algorithm used in Gaussian mixture modeling and focus on iterative refinement, aiming for a local optimum. K-means tends to produce clusters of comparable spatial extent, whereas Gaussian mixture modeling allows for clusters with varying shapes. Additionally, it's important to note that K-means is an unsupervised algorithm, distinct from the supervised k-nearest neighbor classifier, despite the similar name. However, the nearest centroid classifier or Rocchio algorithm can be applied to classify new data into existing clusters formed by K-means.

Given a set of observations $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$, where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into $k(\leq n)$ sets $\mathbf{S} = \{S_1, S_2, \ldots, S_k\}$ so as to minimize the within-cluster sum of squares (i.e. variance). Formally, the objective is to find:

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \left\| \mathbf{x} - \boldsymbol{\mu}_i \right\|^2 = \arg\min_{\mathbf{S}} \sum_{i=1}^{k} |S_i| \operatorname{Var} S_i \tag{2.1}$$

where μ_i is the mean (also called centroid) of points in S_i, i.e.

$$\mu_i = \frac{1}{|S_i|} \sum_{\mathbf{x} \in S_i} \mathbf{x} \tag{2.2}$$

$|S_i|$ is the size of S_i, and $\|\cdot\|$ is the usual L^2 norm. This is equivalent to minimizing the pairwise squared deviations of points in the same cluster:

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \frac{1}{|S_i|} \sum_{\mathbf{x},\mathbf{y}\in S_i} \|\mathbf{x}-\mathbf{y}\|^2 \tag{2.3}$$

The equivalence can be deduced from identity $|S_i| \sum_{\mathbf{x}\in S_i} \left\|\mathbf{x}-\boldsymbol{\mu}_i\right\|^2 = \frac{1}{2} \sum_{\mathbf{x},\mathbf{y}\in S_i} \|\mathbf{x}-\mathbf{y}\|^2$. Since the total variance is constant, this is equivalent to maximizing the sum of squared deviations between points in different clusters (between-cluster sum of squares). This deterministic relationship is also related to the law of total variance in probability theory [48].

Hierarchical Clustering

Hierarchical clustering, a technique utilized in data mining and statistics, is a method for creating a cluster hierarchy. It employs two primary approaches:

- Agglomerative: a ”bottom-up” process where each observation initially forms its own cluster
- Divisive: a ”top-down” approach where all observations start in a single cluster and are recursively split

The process is driven by greedy decisions to merge or split clusters as needed. The results of hierarchical clustering are typically displayed in the form of a dendrogram, offering a visual representation of the cluster hierarchy [60]. Hierarchical clustering offers a notable benefit in that it accommodates the use of any valid distance metric. In fact, it doesn't necessitate the original observations themselves; it operates solely with a distance matrix.

2.3.5 Classification

Classification in statistics involves determining the category to which an observation belongs, such as classifying an email as ”spam” or ”non-spam” or diagnosing a patient based on their characteristics. Observations are often represented by quantifiable properties known as explanatory variables or features, which can be categorical, ordinal, integer-valued, or real-valued. Classifiers are algorithms that implement classification, mapping input data to categories. The terminology varies across fields, with statistics using explanatory variables and outcomes, while machine learning uses features and classes. In other domains, like community ecology, ”classification” typically refers to cluster analysis.

Decision Tree Classifier

To understand Decision Tree classifiers, it's essential to grasp the concept of Decision Tree Learning. Decision tree learning is an approach in statistics and machine learning that employs a tree-like structure to predict outcomes for observations. Tree models that handle discrete target variables are known as classification trees. In these tree structures, class labels are represented by the leaves, and branches depict combinations of features leading to those class

labels. Conversely, decision trees designed for target variables with continuous values, usually real numbers, are referred to as regression trees [92]. Due to their simplicity and interpretability, decision trees rank as one of the most widely used machine learning algorithms [105]. A classification chart or classification tree serves as a condensed representation of a classification scheme, aimed at visually conveying the organization and structure of a specific field or domain. This graphical tool offers a clear and concise overview of how elements within that domain are categorized and related to one another. It provides a valuable framework for understanding and navigating complex systems by highlighting the hierarchy, relationships, and distinctions within the classification scheme, making it a valuable resource for comprehending the underlying structure of diverse subject areas or industries [90]. In classification tree analysis, the focus lies on predicting the class, which represents the discrete category to which the data belongs.

Percentile Classification

A percentile is a statistical concept that provides information about the relative position of a particular data point within a given dataset. It is typically expressed as a percentage and is used to assess how a specific value compares to the rest of the data. For instance, the 50th percentile, known as the median, signifies the point at which half of the data values are lower, and half are higher. Percentiles are a valuable tool for analyzing data distributions and identifying values that fall within certain proportions of the dataset. Percentile classification is a methodology employed to discretize data by categorizing it into distinct classes or groups, contingent upon its placement within a statistical distribution. This process entails segmenting a dataset into predefined percentiles or percentile intervals. Each percentile interval corresponds to a specific class, and data points falling within the bounds of a particular interval are allocated to the corresponding class. This method proves valuable when the objective is to classify data into categories determined by their relative positioning within the overall distribution, rather than by fixed or predetermined criteria or thresholds. For instance, in the realm of education, percentile classification may be applied to stratify student performance on standardized tests into percentile categories (e.g., below average, average, above average) based on the comparison of their scores to the broader student cohort.

Binning in Classification

In classification, binning (or discretization) is a technique used to transform continuous or numerical features into categorical or discrete bins or intervals. This process involves dividing the range of values of a continuous feature into a set of predefined bins and assigning each data point to the appropriate bin based on its value. Binning is typically applied to simplify the modeling process, handle outliers, and capture non-linear relationships in the data.

- **Steps in Binning:**

1. **Dividing into Bins:** The first step is to determine the number of bins or intervals you want to create. The range of the feature values is then divided into these bins. The bin boundaries are defined based on criteria such as equal width, equal frequency (each bin contains approximately the same number of data points), or custom ranges.

2. **Assigning to Bins:** Each data point is assigned to one of the bins based on the value of the feature. For example, if you're binning ages into categories like "young," "middle-aged," and "elderly," you would assign each person's age to the corresponding bin.

3. **Feature Transformation:** After binning, the continuous feature is transformed into a categorical one. It's represented using integers or labels for each bin. This transformation simplifies the feature and reduces the impact of outliers.

4. **Modeling:** Binned features can be used in classification models just like any other categorical features. They may help capture non-linear relationships or patterns that might be missed when using the original continuous feature.

- **Advantages of Binning:**

- **Simplification:** Binning can make complex data more understandable and interpretable by dividing it into meaningful categories.

- **Handling Outliers:** Outliers can significantly affect the performance of classification models. Binning can help mitigate their impact by placing extreme values in the same bin.

- **Non-linearity:** Binning allows models to capture non-linear relationships between the feature and the target variable.

- **Reducing Overfitting:** Binning can reduce the risk of overfitting in some cases, especially when dealing with limited data.

- **Categorical Encoding:** Categorical features can be easier to work with, and some machine learning algorithms require categorical encoding.

However, it's essential to be mindful of the number of bins and the criteria used for binning. If not done correctly, it can lead to information loss or over-simplification. The choice of binning method and the number of bins should be made carefully, considering the specific characteristics of your data and the goals of your classification task.

2.4 Cosine Similarity

Cosine similarity is a mathematical measure used to quantify the similarity between two vectors, typically represented as $A = (A_i)i \in 1, 2, \ldots, n$ and $B = (B_i)i \in 1, 2, \ldots, n$. This similarity metric is determined by calculating the cosine of the angle θ between these vectors. It is formally defined as:

$$\cos(\theta) = \frac{A \cdot B}{|A||B|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}} \tag{2.4}$$

The cosine similarity measure yields values within the range of -1 to 1, which can be expressed as $-1 \leq \cos(\alpha) \leq 1$. Notably, this metric is solely contingent upon the angle between the two vectors, making it a valuable tool for assessing their similarity.

In scenarios like information retrieval and text mining, a common approach involves assigning unique coordinates to each word, and a document is consequently represented as a vector comprising the word occurrence counts within that document. The cosine similarity metric is employed to provide a valuable assessment of the likely similarity between two documents. This measure is particularly helpful for gauging the thematic resemblance between documents, while effectively mitigating the influence of document length on the assessment [85]. Cosine similarity is a versatile method utilized not only in information retrieval and text mining but also in the realm of data mining, where it helps gauge cluster cohesion. Its computational efficiency, particularly when handling sparse vectors, is a notable advantage, as it only considers non-zero coordinates. This similarity metric also goes by alternative names like Orchini similarity and Tucker coefficient of congruence. Furthermore, the Otsuka–Ochiai similarity, which applies cosine similarity to binary data, is an intriguing adaptation within this context [95].

2.5 Normalization

Normalization involves the transformation of statistical values by shifting and scaling them, aiming to create comparable normalized values across different datasets. This process helps mitigate the impact of significant influences, such as anomalies in time series data. Various normalization techniques exist, some involving rescaling to relate values to a reference size variable. It's important to note that such rescaled ratios are relevant for measurements with a ratio scale, where the ratios of measurements hold meaning, and not for interval measurements, where only the relative distances between values matter, but not their ratios. In theoretical statistics, parametric normalization often leads to the identification of pivotal quantities, which are functions with sampling distributions independent of parameters, and ancillary statistics, which are pivotal quantities computable from observations without knowledge of the underlying parameters [17]. Various statistical normalizations exist, encompassing nondimensional ratios involving errors, residuals, means, and standard deviations. These normalizations are designed to be scale-invariant. It's worth emphasizing that the meaningfulness of these ratios is contingent upon the levels of measurement, being applicable to ratio measurements where measurement ratios hold significance, but not to interval measurements, where only the relative distances between values have meaning.

2.5.1 Min-Max Normalization

Termed as either min-max scaling or min-max normalization, rescaling represents the most straightforward approach, involving the adjustment of feature ranges to fit within the specified range, typically $[0, 1]$ or occasionally $[-1, 1]$. The choice of the target range is determined by the data's characteristics. The standard formula for min-max scaling to $[0, 1]$ can be expressed as follows:

$$X_{\text{new}} = \frac{X - X_{\min}}{X_{\max} - X_{\min}} \tag{2.5}$$

In this context, where x represents the original value and x' is the normalized value. Imagine we are working with temperature data in Celsius, and the temperature range spans from 10°C

to 30°C. To rescale this data, we start by subtracting 10 from each temperature reading and then divide the result by 20 (the difference between the maximum and minimum temperatures).

For the more general case of rescaling a range to arbitrary values $[a, b]$, the formula can be expressed as follows where a, b are the min-max values:

$$x' = a + \frac{(x - \min(x))(b - a)}{\max(x) - \min(x)} \tag{2.6}$$

2.5.2 Mean Normalization

$$\frac{x - \bar{x}}{\max(x) - \min(x)} \tag{2.7}$$

In this context, x represents the original value, x' stands for the normalized value, and $\bar{x}$ denotes the average of the feature vector, calculated as the mean of x. It's worth noting that there's an alternative approach to mean normalization, which involves dividing by the standard deviation, often referred to as standardization.

2.6 Outlier Detection

In the realm of data analysis, anomaly detection, also known as outlier detection or occasionally novelty detection, typically involves identifying unusual items, events, or observations that significantly deviate from the majority of the data and do not adhere to a well-established concept of typical behavior [9]. Instances like these might raise concerns about their origin being potentially attributed to an alternative process [38] or seem incongruous when compared to the rest of the dataset [5].

2.6.1 Z-Score Method

The Z-score (or standard score) outlier detection method is a statistical technique used to identify outliers in a dataset. It's based on the idea of standardizing the data and then measuring how many standard deviations a particular data point is away from the mean of the dataset. Here's how the method works [1]:

1. **Standardization**: For each data point in the dataset, you subtract the mean of the dataset from that data point and then divide by the standard deviation. This process transforms the data into a new distribution with a mean of 0 and a standard deviation of 1. The formula for standardizing a data point x is:

$$Z = \frac{x - \mu}{\sigma}$$

Where: - Z is the Z-score of x. - x is the data point. - μ is the mean of the dataset. - σ is the standard deviation of the dataset.

2. **Identifying Outliers**: Once the data is standardized, you can then set a threshold, typically a Z-score value, beyond which data points are considered outliers. Common threshold values include Z-scores greater than 2 or 3, which correspond to data points that are significantly far from the mean. These data points are considered outliers.

3. **Detection**: Data points with Z-scores above the chosen threshold are flagged as potential outliers.

The Z-score method is effective in identifying outliers that are extreme in terms of their deviation from the mean, and it's particularly useful when the data follows a roughly normal distribution. However, it's sensitive to the distribution of the data, and outliers that follow a different distribution might not be accurately identified using this method. Therefore, it's important to consider the nature of the data and the chosen threshold when applying the Z-score method for outlier detection.

2.6.2 IQR

The Interquartile Range (IQR) outlier detection method is a statistical technique used to identify outliers in a dataset based on the spread of the data. It focuses on the distribution of the data's central 50% (interquartile range) and is less sensitive to extreme values compared to the Z-score method. Here's how the IQR method works:

1. **Calculation of the IQR**: - Calculate the first quartile (Q1), which is the 25th percentile, and the third quartile (Q3), which is the 75th percentile, of the dataset. - Compute the IQR by subtracting Q1 from Q3: IQR = $Q3 - Q1$.

2. **Identifying Outliers**: - Define the lower bound as $Q1 - 1.5 \times \text{IQR}$ and the upper bound as $Q3 + 1.5 \times \text{IQR}$. These bounds represent a range within which most of the data is expected to lie. - Any data point below the lower bound or above the upper bound is considered an outlier.

3. **Outlier Detection**: - Data points that fall below the lower bound or above the upper bound are flagged as potential outliers.

The IQR method is effective in identifying outliers that fall outside the typical range of the central 50% of the data. It is particularly useful when the data may not follow a normal distribution or when there are concerns about extreme values affecting the results. By focusing on the interquartile range, the IQR method is more robust against extreme values compared to methods like the Z-score [101].

It's important to note that the choice of the 1.5 multiplier in the upper and lower bounds is somewhat arbitrary and can be adjusted based on the specific requirements of the analysis. Common multipliers other than 1.5 include 2 and 3, which result in wider or stricter outlier detection, respectively.

2.6.3 SZZ Algorithm

The SZZ algorithm is a software maintenance and evolution technique used to identify the source code changes that introduced defects or bugs in a software project. It stands for "Simplified, Zero-One, and Zero." The SZZ algorithm was developed to help software developers and quality assurance teams understand the history of defects in a codebase and pinpoint the specific code revisions responsible for introducing those defects.

Here's a simplified explanation of how the SZZ algorithm works:

1. **Data Collection:**
    - The SZZ algorithm starts by collecting historical data from a version control system (e.g., Git, SVN) and the associated issue tracking system (e.g., Jira, Bugzilla).

This data includes information about code commits, such as the commit message, commit date, and the files modified in each commit.

2. **Defect Identification:**

   - The algorithm focuses on identifying defects or issues reported in the issue tracking system. These issues are typically labeled as bug reports, feature requests, or other types of software problems.

3. **Defect-Inducing Commits:**

   - For each defect, the algorithm works backward in time to identify the commit that introduced the defect. It does this by tracing the changes made to the files associated with the defect from the moment the issue was reported back to the first commit where the code related to the issue was modified.
   - This is done by analyzing the version history of the files and determining the ”blame” information, which associates lines of code with the commit that last modified them before the issue was reported.

4. **Marking Defect-Inducing Commits:**

   - The algorithm marks the commit(s) that introduced the defect as ”defect-inducing.” If there are multiple commits involved, they are typically marked as a set.
   - The defect-inducing commits are identified in a binary manner, hence the ”Zero-One” in the name. A commit is either marked as defect-inducing (1) or not (0).

5. **Analysis and Reporting:**

   - Once the defect-inducing commits are identified, the SZZ algorithm can be used to generate reports and statistics about the history of defects in the codebase. This information can be valuable for understanding which parts of the code are more error-prone, assessing the impact of specific changes, and improving software quality.

The SZZ algorithm is particularly useful for understanding the origins of defects in large and complex software projects. By identifying the commits that introduced defects, developers can analyze the code changes made during those commits, which can help with debugging and improving software quality.

It is inevitable that the SZZ algorithm is not perfect and may have false positives and false negatives, as it relies on historical data and heuristics to identify defect-inducing commits. Nevertheless, it is a valuable tool in software maintenance and quality assurance.

2.7 Shannon Entropy

In the realm of information theory, Shannon entropy serves as a fundamental concept. It quantifies the average amount of ”information,” ”surprise,” or ”uncertainty” associated with the

possible outcomes of a random variable. Let's delve into its mathematical formulation and implications.

Consider a discrete random variable X with its potential values residing in the set $\mathcal{X}$. This variable is distributed according to a probability mass function $p : \mathcal{X} \to [0, 1]$, where $p(x)$ represents the probability of X taking on a specific value x from $\mathcal{X}$. The Shannon entropy, denoted by $\mathrm{H}(X)$, captures the essence of this variability and is calculated as:

$$\mathrm{H}(X) = -\sum_{x \in \mathcal{X}} p(x) \log p(x)$$

Here, the symbol Σ signifies the summation operation over all possible values x of the variable. The logarithm function log is the base for which various applications may adopt different bases. For instance, using base 2 results in units of bits (or "shannons"), base e yields "natural units" known as nats, and base 10 produces units such as "dits," "bans," or "hartleys."

An intriguing perspective on entropy is viewing it as the expected value of the self-information of a variable. This concept intertwines with the essence of uncertainty and surprise encapsulated within the random variable.

Shannon's groundbreaking work on mathematical theory of communication [81] laid the foundation for this notion of entropy, which has since become a cornerstone in various fields, including communication theory, cryptography, and machine learning. Claude Shannon introduced the concept of information entropy in his seminal paper "A Mathematical Theory of Communication" in 1948, often referred to as Shannon entropy. Within Shannon's theory lies a conceptual framework comprising three key components: a data source, a communication channel, and a receiver. Shannon articulated the "fundamental problem of communication," which entails the receiver's task of accurately discerning the data originated by the source based on the signals it receives through the channel. Exploring diverse methods of encoding, compressing, and transmitting messages from a data source, Shannon demonstrated in his renowned source coding theorem that entropy serves as an absolute mathematical boundary dictating the optimal compression of data onto a perfectly noiseless channel without loss. Additionally, Shannon significantly bolstered this theorem for noisy channels through his development of the noisy-channel coding theorem.

2.8 Bayes' Theorem

The Bayes theorem, often known as Bayes' law or Bayes' rule, is a concept in probability theory and statistics that, depending on prior knowledge of potential contributing factors, estimates the likelihood of an event. It bears Thomas Bayes' name [43]. For instance, the Bayes theorem makes it possible to more accurately assess an individual's risk of developing health problems if it is known that such a risk increases with age. This is achieved by conditioning the risk assessment relative to the individual's age, instead of assuming that the individual is typical of the population as a whole. The following equation represents the mathematical formulation of Bayes' theorem [91]:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.8}$$

where A and B are events and $P(B) \neq 0$.

2.9 Conditional Probability

Conditional probability, in probability theory, is a measure of the likelihood that an event will transpire in the event that another event is already known to have occurred (by assumption, presumption, assertion, or evidence) [34]. This specific approach depends on event A having some kind of connection to event B. In this case, a conditional probability with respect to B can be used to analyze the event A. "The conditional probability of A given B", or "the probability of A under the condition B", is typically expressed as P(A—B) or, less frequently, $P_B(A)$. This is the case when the event of interest is A and the event B is known or presumed to have occurred. This can also be thought of as the ratio of the probabilities of both events occurring to the "given" one occurring (i.e., the number of times A occurs rather than not assuming B has occurred) or as the fraction of probability B that intersects with A [14]:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \tag{2.9}$$

For instance, there's only a 5% chance that a certain person will cough on any given day. However, the likelihood of the person coughing increases significantly if we know or believe that they are ill. For instance, if someone is coughing and the conditional probability that they are sick is 75%, then P(Cough) = 5% and P(Cough—Sick) = 75% would apply. In this example, A and B have a relationship, but it's not required for them to be dependent on one another or for it to happen at the same time.

2.10 Tools

In this section, we aim to introduce the tools employed in this project and briefly explain the functionality of each tool.

2.10.1 SQL

The data was initially housed in a SQL-based format, utilizing the SQLite Database Engine. SQLite is a database engine written in the C programming language. It is not a standalone app; rather, it is a library that software developers embed in their apps. As such, it belongs to the family of embedded databases.

2.10.2 Python

To extract and conduct all analyses on this data, we leveraged the Python programming language in combination with the pandas library, which provided robust tools for data manipulation and analysis. pandas is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series. It is free software released under the three-clause Berkeley Standard Distribution license.

2.10.3 CSV & JSON

Subsequently, the information extracted from the database was stored in multiple formats for ease of use and reference, with JSON and CSV formats being selected to preserve and make the data accessible for further research and analysis. A CSV file is a text file that has a specific format which allows data to be saved in a table structured format, and JSON is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays. It is a common data format with diverse uses in electronic data interchange, including that of web applications with servers.

2.10.4 Scikit-learn library

We relied on the Scikit-learn library for numerous techniques and methodologies employed throughout this project, including but not limited to clustering. Scikit-learn is an open-source machine learning library for Python. It provides a user-friendly and efficient environment for building, training, and evaluating machine learning models. The library includes a wide range of machine learning algorithms, tools for data preprocessing, model evaluation, and model selection, making it a fundamental component of the Python machine learning ecosystem. It is widely used for various data analysis and modeling tasks, including classification, regression, clustering, and more.

2.10.5 SonarQube

SonarQube is an open-source platform for continuous inspection of code quality. It is a widely used tool in the field of software development and helps development teams manage and improve the quality of their code. SonarQube provides a range of features and capabilities, including:

1. **Static Code Analysis**: SonarQube performs static code analysis to identify various code quality issues, including code smells, bugs, and vulnerabilities. It scans the source code and identifies potential problems by applying a set of predefined coding rules and quality standards.

2. **Code Quality Metrics**: It generates various code quality metrics and provides reports and dashboards to help teams understand the health of their codebase. Metrics may include code duplication, code coverage, cyclomatic complexity, and more.

3. **Continuous Integration and Automation**: SonarQube can be integrated into the continuous integration and continuous delivery (CI/CD) pipeline, allowing developers to get immediate feedback on the quality of their code with each code commit.

4. **Support for Multiple Languages**: It supports a wide range of programming languages, making it suitable for projects with diverse technology stacks.

5. **Security Scanning**: In addition to code quality, SonarQube has features for identifying and addressing security vulnerabilities in the code.

6. **Customization and Extension**: You can configure and customize the quality profiles and rules to match your specific coding standards and project requirements. Additionally, SonarQube supports the use of plugins to extend its functionality.

7. **Project and Portfolio Management**: It provides tools for managing and monitoring code quality across multiple projects, making it useful for larger organizations with numerous codebases.

8. **Historical Analysis**: SonarQube keeps a historical record of code quality metrics, enabling developers and teams to track improvements or deteriorations in the codebase over time.

SonarQube helps development teams proactively manage technical debt, improve code maintainability, and reduce the likelihood of introducing bugs and security vulnerabilities. It is a valuable tool for maintaining and enhancing the overall quality of software projects.

Chapter 3

Data Pre-processing and Modelling

3.1 Preface

In this Chapter, we detail the techniques used to pre-process and model the data for the purpose of defect prediction. More specifically, this Chapter provides a thorough explanation of the various steps involved in preparing our data for the development of our final model. We outline the methods and techniques used to clean, transform, and organize the raw data, ensuring it is suitable for modeling and analysis. This includes data selection, cleansing, normalization, and any feature engineering or selection processes that were applied. We will present an activity diagram to visually represent the sequence of pre-processing steps and illustrate the flow of data from its raw form to its readiness for model creation. This diagram serves as a road-map, offering a clear and concise overview of the entire pre-processing workflow and helping readers understand the systematic approach we took in this crucial phase of the project.

3.2 Dataset Description

The data employed in this project originates from a open source data set known as SonarQube ”TechnicalDebt” data set [51]. This database was curated by also linking JIRA issues to commits in various software projects, extracting relevant features, and pinpointing commits responsible for introducing and resolving bugs, all accomplished through the utilization of the SZZ algorithm, and a novel approach introduced by P. Parul and K. Kontogiannis [67].

The code quality tool SonarQube was utilized to extract metrics from code commits for the purpose of creating this database.

The database was constructed using project repositories from the Apache project, as detailed in Table 3.1, providing concise information about each project.

3.2.1 SonarQube Data Set Schema Outline

The SonarQube ”Technical Debt” data set includes a wide selection of defect, process, and source code metrics from 22 open source projects. The metrics are modelled in a relational database and organized in various tables. The data span the life time of each project and each commit.

Table 3.1: Project Statistics

Project Name	Average LOC per Commit	Total # of Commits	# of Commits Analyzed	# of Distinct Files
batik	213850.96	83725	2164	14434
commons-beanutils	55390.82	7549	1210	499
commons-codec	21615.37	5663	1730	358
commons-collections	95468.13	23854	2765	1724
commons-cli	13314.65	3465	847	473
commons-io	33935.05	9986	1910	702
commons-jelly	55989.83	12318	1937	816
commons-jexl	18635.81	6683	1532	599
commons-configuration	83412.57	12962	2929	1533
commons-daemon	2847.61	2940	981	249
commons-dbcp	24424.94	6878	1555	352
commons-dbutils	8185.04	1861	602	153
commons-digester	29863.58	9325	2142	1330
felix	286157.01	160648	3490	29600
httpcomponents-client	79191.72	23008	2714	1460
httpcomponents-core	54806.66	26033	1901	1666
commons-jxpath	36588.13	3972	596	335
commons-net	47690.75	9551	2089	599
santuario	124643.12	50338	2718	9162
commons-vfs	46029.78	15343	2079	714
zookeeper	97243.80	17789	222	1956
thrift	27217.14	27566	1944	2844

Table 3.2 provides an overview of the columns in the main database, each representing a metric in SonarQube. It includes a concise explanation of each metric to aid in understanding their respective meanings and purposes. For a more detailed explanation of SonarQube metrics, please refer to SonarQube Documentation website.

3.3 Outline of the Data Pre-Processing Phase

The data pre-processing phase consists of 12 discrete steps as depicted in the activity diagram in Fig 3.1.

The steps of the data pre-processing phase as depicted in Fig. 3.1 are summarized as follows:

1. **Database Cleaning**: The initial step involves cleaning the database. by excluding projects with very few commits, and projects with very few source code files. Also configuration or compilation directive files were excluded from the analysis, resulting to a database containing only source code files.

2. **Feature Selection**: The next step involves selecting the important features from the database for analysis. First, features with uniform or null values are removed. Features are then filtered, so that only features with low correlation to each other are retained. Finally, frequency analysis is conducted to identify features prevalent in most of nit all representing commits. The result of this step is a set of features that contains usable and rich data.

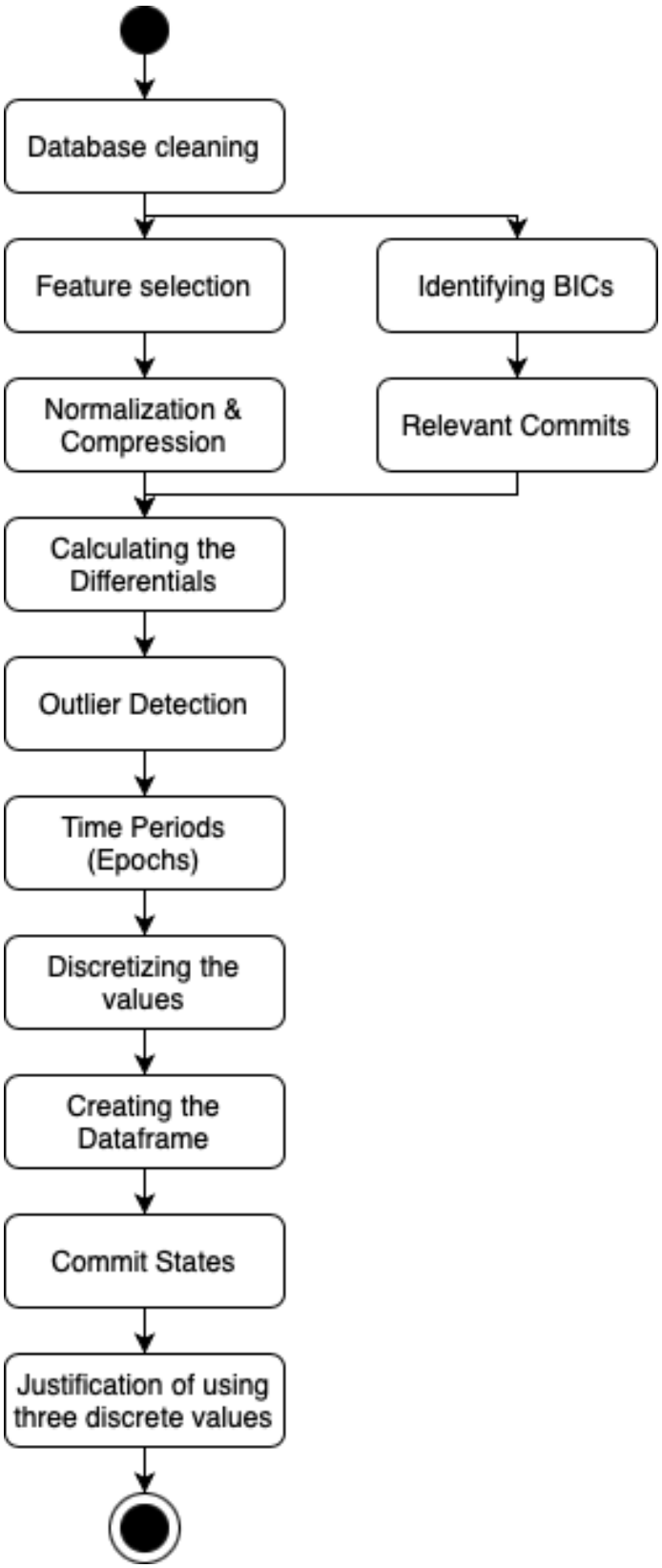


Figure 3.1: Activity Diagram for Pre-processing

Table 3.2: Explanation of SonarQube Metrics

Metric	Description
PROJECT_ID	Unique identifier for the project.
ANALYSIS_KEY	Unique identifier for the analysis.
COMMIT_HASH	Hash of the commit associated with the analysis.
CLASS_COMPLEXITY	Complexity of the classes in the commit.
NEW_LINES_TO_COVER	New lines of code to cover in the commit.
VIOLATIONS	Total number of code violations found in the commit.
NEW_VIOLATIONS	Number of new code violations found since the last analysis.
SQALE_RATING	Rating of technical debt based on SQALE methodology.
SQALE_DEBT_RATIO	Ratio of technical debt to size.
NEW_SQALE_DEBT_RATIO	Ratio of new technical debt to new code added.
CODE_SMELLS	Total number of code smells found in the commit.
NEW_CODE_SMELLS	Number of new code smells found since the last analysis.
BUGS	Total number of bugs found in the commit.
NEW_BUGS	Number of new bugs found since the last analysis.
RELIABILITY_REMEDIATION_EFFORT	Effort required to remediate reliability issues in the commit.
RELIABILITY_RATING	Rating of reliability based on the number of bugs found in the commit.
NEW_RELIABILITY_RATING	Rating of reliability based on the new bugs found since the last analysis.
VULNERABILITIES	Total number of vulnerabilities found in the commit.
NEW_VULNERABILITIES	Number of new vulnerabilities found since the last analysis.
SECURITY_REMEDIATION_EFFORT	Effort required to remediate security issues in the commit.
SECURITY_RATING	Rating of security based on the number of vulnerabilities found in the commit.
NEW_SECURITY_RATING	Rating of security based on the new vulnerabilities found since the last analysis.
CLASSES	Total number of classes in the commit.
FILES	Total number of files in the commit.
FUNCTIONS	Total number of functions in the commit.
COMMENT_LINES_DENSITY	Density of comment lines in the commit.
NEW_TECHNICAL_DEBT	New technical debt added since the last analysis.
JIRA_KEY	JIRA key associated with the commit.
EFFORTS_PER_FILE	Effort required to remediate issues per file in the commit.
EFFORTS_PER_COMPONENT	Effort required to remediate issues per component in the commit.
LINES_TO_COVER	Total lines of code to cover in the commit.
LINES	Total lines of code in the commit.
NCLOC	Non-commented lines of code in the commit.
NEW_SECURITY_REMEDIATION_EFFORT	Effort required to remediate new security issues since the last analysis.
EFFORT_TO_REACH_MAINTAINABILITY_RATING_A	represents the estimated effort required to bring the maintainability of a commit up to the highest rating.

3. **Identifying BICs**: In this step, bug-inducing commits (BICs) are identified using the SZZ algorithm and are then reconciled with JIRA records to increase accuracy of what constitutes a BIC as the SZZ algorithm is known to plagued with false positives.

4. **Relevant Commits**: Commits are considered relevant if they contain one or more common files. In this step, we identify these relevant commits and sort them by date. Relevant commits are grouped based on their impact on similar files and sorted by date. This process creates a flow of commits that establishes transitions later on.

5. **Normalization & Compression**: In this step, the values of the selected features are normalized to facilitate uniform type of processing. In this respect, two different approaches are employed to achieve normalized values as discussed in Section 3.6.

6. **Calculating the Differentials**: Next, the differentials (changes in each metric value) are calculated from one related commit to the next to understand the alterations in each commit for each metric.

7. **Outlier Detection**: Outliers are identified for each chosen metric using Z-score and IQR methods, with results compared for accuracy. The NCLOC feature, which highlights project refactoring, is crucial in this phase.

8. **Time Periods (Epochs)**: In this step, the commits are grouped onto epochs. The epochs are calculated based on each file and a new epoch occurs for a set of files when their

lines of code change significantly (i.e. deviate significantly form the mean value). These epochs allow the analysis to be performed in-context as during the life span of a project files may change significantly, and the analysis performed in one period may not be applicable anymore in another period.

9. **Discretizing the Values**: Values for each metric are discretized using three classification methods: decision tree, k-means, and percentile classification. The dominant answer for each value is then represented with a discrete value.

10. **Creating the Dataframe**: The dataframe is created based on the discrete values and previously selected features.

11. **Commit State**: This section explains the representation of commit states, their significance, and the insights they provide.

12. **Justification of Using Three Discrete Values**: The rationale behind using three discrete values in some experiments is discussed, including the objectives and benefits of this approach.

3.4 Database Cleaning

In order to facilitate the identification of pertinent commits and enhance the efficiency of subsequent analysis, it is imperative to initiate a data cleaning process aimed at refining the database and reducing its size.

3.4.1 Removing configuration files

To optimize classification accuracy and align the study's objectives, it is necessary to perform a filtering process that removes non-Java file records from the dataset. This step aims to enhance the quality of results by focusing solely on Java source code files, as configuration files are deemed irrelevant for the current study.

Filtering out non-Java file records serves several critical purposes, including:

- Improved Classification Accuracy:

  By exclusively considering Java files, the classification algorithm can effectively discern patterns and extract features specific to Java source code. The elimination of non-Java files reduces potential noise and interference, thereby enhancing the accuracy of the classification model.

- Relevance to Study Objectives:

  The study's focus is centered on Java source code analysis, which necessitates a concentrated examination of Java files. Configuration files, while essential for software development, do not directly contribute to the study's research questions and findings. Consequently, excluding such files streamlines the analysis process, allowing researchers to concentrate their efforts on Java-specific aspects.

- Keeping Relevant Commits and Database Arrangement:

  To facilitate future analysis and streamline the examination of relevant commits within the database, it is crucial to undertake a process of commit identification and aggregation. This procedure involves several steps, including the separation of files for each project, the isolation of bug inducing commits, and the organization of commits with shared fixing commits based on chronological order. By implementing these steps, all pertinent commits can be consolidated and conveniently placed adjacent to each other, thereby enhancing the efficiency and comprehensibility of subsequent analyses.

  The process of commit identification, aggregation, database sorting, and arrangement can be summarized as follows:

- File Separation:

  Initially, the commit records are divided based on the associated project files. This segregation enables a focused analysis on a per-project basis, allowing for a comprehensive understanding of the specific changes and developments within each project. By grouping commits by file, the subsequent steps can be performed with file-specific considerations in mind.

- Isolation:

  Within each project, the bug-inducing commits are identified and isolated. Bug-inducing commits are those that introduce changes or issues that subsequently require fixing. These commits often provide critical context and serve as a reference point for further analysis. By isolating bug-inducing commits, researchers can gain insight into the factors leading to subsequent fixing commits.

- Organization:

  Among the bug-inducing commits, those with common fixing commits are identified and sorted based on their chronological order. This sorting ensures that commits with similar fixing commits are grouped together, facilitating cohesive and sequential analysis. Examining commits in chronological order allows researchers to trace the progression of changes, understand the relationships between commits, and identify patterns or trends over time.

3.4.2 Eliminated projects

In the data pre-processing phase, a thorough evaluation was carried out to identify projects that primarily comprised configuration files and lacked Java files. This assessment aimed to maintain the study's relevance and alignment with the research objectives. Consequently, two projects, "batik" and "daemon," were excluded from the analysis due to their exclusive reliance on configuration files. This decision was made in recognition of the fact that configuration files, while essential for software development, do not directly contribute to the research questions and objectives of the study. By eliminating such projects, the subsequent analysis can focus exclusively on Java source code, allowing for a more targeted and accurate exploration of the desired aspects.

Moreover, further investigation revealed that the ”digester” project had only two remaining commit records, with all other commits related to configuration files. This limited number of relevant commit records rendered the ”digester” project unsuitable for inclusion in the study, as it hindered meaningful analysis and interpretation. Therefore, the decision was made to exclude the ”digester” project as well as two other projects that didn't have enough commits left leaving us with 17 projects in total. This rigorous assessment and exclusion of projects lacking Java files or predominantly composed of configuration files ensure the study's integrity and focus. By filtering out projects that do not align with the research objectives, the analysis can be conducted on a more relevant and representative subset of projects, providing valuable insights into the desired Java-specific aspects.

This refined database enables subsequent analyses to concentrate on the primary research questions, resulting in more precise and meaningful research outcomes.

3.4.3 Initial database

Through these steps, the relevant commits within the database are systematically organized and grouped for future analysis. The aggregation of commits adjacent to each other provides a holistic view of the development process, enabling comprehensive examinations of the relationships between bug-inducing and bug-fixing commits.

3.5 Feature Selection & Elimination

One of the initial steps towards creating a comprehensive and relevant database is the selection of the appropriate metrics that may be required for future analysis. We have two different sets of features: quality metrics and size metrics. Size metrics, by their nature, are highly correlated with each other. Although we include them in the calculation of the correlation matrix, this is not the proper way to choose them. Correlation is more significant for quality metrics. Therefore, our process involves selecting a specific set of important features and then using correlation to choose among the other features. In addition to correlation, we use other criteria to eliminate features based on their values or distribution, which we will explain in more detail in the following sections. We select a set of source code quality-related features and choose among them based on low correlation.

3.5.1 Data values Overview

To identify the optimal set of features for our study, we conducted a thorough evaluation of the 36 features initially obtained from the TechnicalDebtDataset. Initially, features containing a significant number of null values were excluded from further analysis as they proved ineffective. To further enhance our feature selection, we generated frequency plots to visually assess the distribution of feature values and pinpoint additional irrelevant features.

After this comprehensive evaluation, we arrived at a selection of 19 features for inclusion in the study, which are listed below.

- CLASS_COMPLEXITY
- VIOLATIONS
- SQALE_RATING
- SQALE_DEBT_RATIO
- NEW_SQALE_DEBT_RATIO
- CODE_SMELLS
- BUGS
- RELIABILITY_REMEDIATION_EFFORT
- RELIABILITY_RATING
- VULNERABILITIES
- SECURITY_REMEDIATION_EFFORT
- SECURITY_RATING
- CLASSES
- FILES
- FUNCTIONS
- COMMENT_LINES_DENSITY
- LINES_TO_COVER
- NCLOC
- EFFORT_TO_REACH_MAINTAINABILITY_RATING_A

3.5.2 Features Correlation

In order to gain deeper insights into the relationships between selected features, it is essential to carry out a correlation analysis and generate a correlation matrix. This process will enable us to refine the dataset further by identifying the features that exhibit stronger correlations and are more valuable for our analysis. The generated correlation matrix for the selected features is presented in Fig 3.2.

While examining the correlation matrix and heatmap, it becomes clear that correlation alone does not determine which metrics should represent a commit's state. Correlation measures the relationship between two metrics but does not account for their individual significance or the unique insights they provide. Additionally, it is important to remember that correlation does not imply causation. Just because two metrics are correlated does not mean one causes the other.

For instance, let's consider the metrics VIOLATIONS and CODE_SMELLS. These two metrics might show a high correlation, suggesting they often occur together. However, this does not mean they are redundant or that only one should be used. VIOLATIONS refer to specific instances where the code fails to comply with predefined rules or standards, such as not following naming conventions or exceeding complexity thresholds. This metric is crucial for identifying immediate issues that need correction to maintain code quality.

On the other hand, CODE_SMELLS represent deeper, more structural problems in the code, such as duplicated code, long methods, or large classes. These are not necessarily rule violations but indicate poor design choices that could lead to maintainability issues or bugs in the future. The correlation between VIOLATIONS and CODE_SMELLS does not mean that one causes the other, but rather that they often appear together due to underlying issues in the codebase.

By including both VIOLATIONS and CODE_SMELLS, we gain a comprehensive understanding of the code's health. VIOLATIONS help us catch and correct immediate issues, while

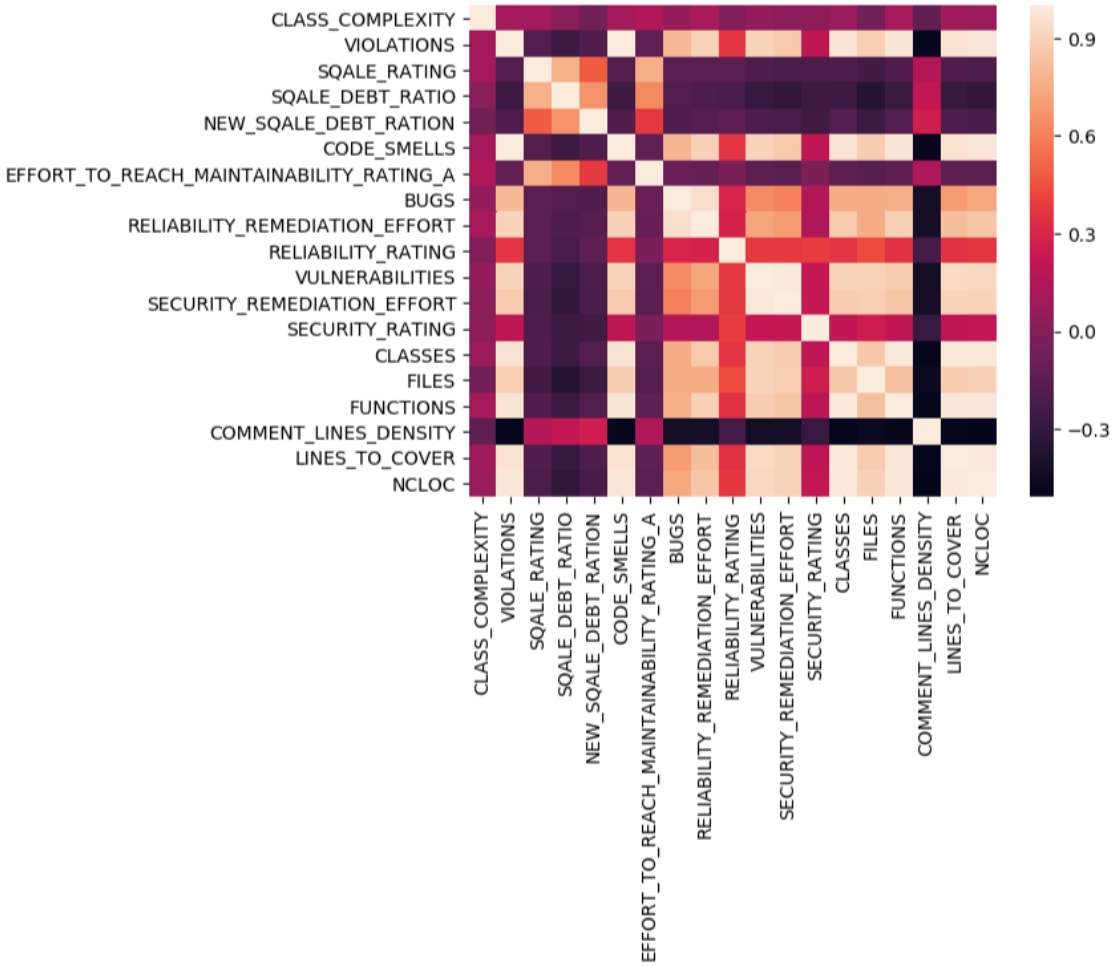


Figure 3.2: Correlation matrix plot

Quality Metrics	**Size Metrics**
CLASS_COMPLEXITY	CLASSES
VIOLATIONS	FILES
SQALE_RATING	FUNCTIONS
NEW_SQALE_DEBT_RATIO	LINES_TO_COVER
CODE_SMELLS	NCLOC
EFFORT_TO_REACH_MAINTAINABILITY	COMMENT_LINE_DENSITY
BUGS	-
RELIABILITY_REMEDIATION_EFFORT	-
RELIABILITY_RATING	-
VULNERABILITIES	-
SECURITY_REMEDIATION_EFFORT	-
SECURITY_RATING	-

Table 3.3: Metrics categorized by quality and size

CODE_SMELLS alert us to potential long-term problems. Ignoring one in favor of the other would result in a less thorough analysis.

Furthermore, using multiple metrics, even those that are highly correlated, allows us to capture different dimensions of code quality. For example, another pair of correlated metrics could be BUGS and VULNERABILITIES. While both might indicate problematic areas in the code, BUGS typically refer to functional issues found during testing, whereas VULNERABILITIES are security-related issues that could be exploited. Each metric provides distinct information necessary for a holistic view of the code's state.

Therefore, despite their correlation, the distinct nature and valuable insights provided by each metric justify their inclusion in our analysis. This approach ensures we do not overlook critical aspects of code quality, leading to more effective bug detection and overall software improvement. Understanding the difference between correlation and causation helps us to select the most relevant metrics and avoid misinterpretations that could compromise our study.

As shown in Table 3.3, we have 19 metrics categorized by whether they are quality metrics or size metrics.

3.5.3 Frequency Analysis

Subsequently, a frequency analysis was performed on the remaining features to identify those that displayed redundancy. In particular, features with values uniform across all records were removed as they provided little information for our study.

At the conclusion of the process, a total of 17 files will have been generated, with each file corresponding to a distinct project. These files adhere to a standardized structure, described as before. Each entry of these files is shown in Fig 3.3.

To enhance the relevance and comprehensiveness of the database, a thorough evaluation of the available columns was conducted, resulting in the removal of non-essential and redundant columns. The eliminated columns included:

- **PROJECT_ID**: This represents a unique identifier for a project within a system or

```
PROJECT_ID                                                                    org.apache:beanutils
FAULT_INDUCING_COMMIT_HASH                                  fa9f99a48f141abdb995884500e1add2dc83c19b
FAULT_FIXING_COMMIT_HASH                                    144cf664a4eb47cdc5e71ff79945f8a43c46ab90
DATE_OF_INDUCING_COMMIT                                                2003-03-03 22:33:46+00:00
DATE_OF_FIXING_COMMIT                                                  2006-11-06 04:05:27+00:00
FILES                                                                        PropertyUtils.java
COMMIT_HASH_ON_FILE_BEFORE_FIX                              0b1bf0fac7e14178e9aa923f89d3bf70375acacf
DATE_OF_COMMIT_HASH                                                    2002-04-28 01:16:48+00:00
IS_COMMIT_HASH_ON_FILES_BEFORE_INDUCING                                                        1
IS_COMMIT_HASH_ON_FILES_AFTER_INDUCING_AND_BEFORE_FIXING                                       0
IS_COMMIT_HASH_ON_FILES_INDUCING_COMMIT                                                        0
IS_COMMIT_HASH_ON_FILES_FIXING_COMMIT                                                          0
NCLOC                                                                                       8462
NEW_SQALE_DEBT_RATION                                                                    8.66855
VIOLATIONS                                                                                  1585
CODE_SMELLS                                                                                 1585
FUNCTIONS                                                                                    499
LINES                                                                                      18486
CLASSES                                                                                       50
EFFORTS_PER_COMPONENT                                                                        NaN
CONCAT                                      fa9f99a48f141abdb995884500e1add2dc83c19b144cf6...
PAIR_NUMBER                                                                                 4018
CLASS_COMPLEXITY                                                                            18.2
Name: 0, dtype: object
```

Figure 3.3: One entry in the concluding database

database. It helps in distinguishing one project from another, ensuring that data associated with each project is correctly attributed.

- **IS_COMMIT_HASH_ON_FILES_BEFORE_INDUCING**: This indicates whether the commit hash is present on files before any bug-inducing changes were made. It is a boolean value (true/false) that helps track the state of the files before the introduction of a bug.

- **IS_COMMIT_HASH_ON_FILES_AFTER_INDUCING_AND_BEFORE_FIXING**: This represents whether the commit hash is present on files after the bug-inducing changes have been made but before any bug fixes are applied. This helps in understanding the state of the files during the period when the bug exists but has not yet been fixed.

- **IS_COMMIT_HASH_ON_FILES_INDUCING_COMMIT**: This indicates whether the commit hash corresponds to the commit that induced (introduced) the bug. It helps in identifying the specific commit that caused the bug.

- **IS_COMMIT_HASH_ON_FILES_FIXING_COMMIT**: This shows whether the commit hash is present on the files associated with the commit that fixes the bug. It helps in pinpointing the exact commit where the bug was addressed and resolved.

- **EFFORTS_PER_COMPONENT**: This metric represents the amount of effort (usually in terms of time or complexity) required for each component of the project. It helps in understanding how much work is needed for different parts of the project, which can aid in project management and resource allocation.

- **CONCAT**: This is the concatenation of the bug-inducing commit hash and the bug-fixing commit hash. By combining these two hashes, it creates a unique identifier that links a

specific bug introduction to its resolution, facilitating better tracking and analysis of bug fixes.

- **PAIR_NUMBER**: This refers to a sequential number or identifier assigned to a pair of bug-inducing and bug-fixing commits. It helps in organizing and referencing these pairs systematically for analysis or reporting purposes.

These columns, while potentially informative in certain contexts, did not directly contribute to the specific research objectives of the study. Their removal streamlined the database, enhancing its clarity and relevance by focusing on the core aspects of interest.

3.5.4 Remaining Features

After a meticulous feature selection process, we have identified eight specific features to serve as the final metrics for this project. As we discussed, the features we are going to use in this thesis are quality features (specified in bold) and size features are disregarded in representing commit's state. To clearly understand the range of these metrics, we will examine the "beanutils" project as a case study, providing the maximum and minimum values for each metric within this context. These selected features are listed below:

- **NEW_SQALE_DEBT_RATIO** (Metric ID: 1): This metric relates to the Technical Debt in the codebase. It's a measure of the ratio between the new technical debt incurred in the recent changes and the effort required to fix it. A high value may indicate that a significant amount of new technical debt has been introduced. The maximum and minimum value for this feature in "beanutils" project is 8.914686333, and 0.02053004853.

- **VIOLATIONS** (Metric ID: 2): This metric refers to the total number of code violations or rule violations found by SonarQube. These violations might be related to coding standards, best practices, or other rules defined in project's quality profile. The maximum and minimum value for this feature in "beanutils" project is 5140, and 128.

- **CODE_SMELLS** (Metric ID: 3): Indicators of code that might be less than optimal in terms of maintainability or readability. This metric counts the total number of code smells detected in codebase. The maximum and minimum value for this feature in "beanutils" project is 5015, and 137.

- **CLASS_COMPLEXITY** (Metric ID: 4): A measure of how complex the classes are in terms of their structure and interactions. Higher values may suggest more complex and potentially harder-to-maintain classes. The maximum and minimum value for this feature in "beanutils" project is 67.7, and 12.7.

- NCLOC: This metric measures the number of lines in the codebase that are not comments. It's a way to gauge the size and complexity of the codebase without considering comments. The maximum and minimum value for this feature in "beanutils" project is 63562, and 812.

- FUNCTIONS: Represents the total number of functions or methods in the codebase. The maximum and minimum value for this feature in "beanutils" project is 7760, and 57.

Table 3.4: Example of Database Entry

Attribute	**Value**
FAULT_INDUCING_COMMIT_HASH	fa9f99a48f141abdb995884500e1add2dc83c19b
DATE_OF_INDUCING_COMMIT	2003-03-03 22:33:46+00:00
FILES	PropertyUtils.java
COMMIT_HASH_ON_FILE_BEFORE_FIX	ab939bf9fa5d41d093f93d5e95e14dad130b0d12
DATE_OF_COMMIT_HASH	2002-11-23 23:47:07+00:00
NEW_SQALE_DEBT_RATIO	7.478469368
VIOLATIONS	2063
CODE_SMELLS	2061
CLASS_COMPLEXITY	18.4
NCLOC	13531
FUNCTIONS	1072
LINES	32009
CLASSES	96

- LINES: This metric represents the total number of lines in the codebase, including both code and comments. The maximum and minimum value for this feature in ”beanutils” project is 149406, and 1988.

- CLASSES: Counts the total number of classes in the codebase. The maximum and minimum value for this feature in ”beanutils” project is 906, and 3.

These 8 features were selected by analyzing the commits. The first 4 features represent the quality of the code, while the others represent code size. Since we are looking for bugs, we use quality metrics to represent a commit's state. Quality metrics provide insight into the health and maintainability of the code, which are crucial for identifying potential issues that could lead to bugs. By focusing on quality metrics, we can better understand the underlying issues that may not be apparent through size metrics alone. Quality metrics such as code complexity and adherence to coding standards offer a deeper view of the potential risk areas within the code. On the other hand, size metrics, while useful for understanding the overall scope and scale of the project, do not directly correlate with the likelihood of bugs. Additionally, using more metrics increases the complexity of the states being analyzed, making it harder to comprehend and manage the data effectively. By prioritizing quality metrics, we can streamline our analysis and focus on the most relevant factors for bug prediction.

For instance, one can observe a database entry showcasing correlated selected features in Table 3.4.

3.6 Normalization & Compression

After first stages of data analysis, we found out the distribution of values among each feature is non-uniform, and therefore we cannot establish borders for the discrete representation of the features by simply dividing the scale of values by five. In order to better illustrate this

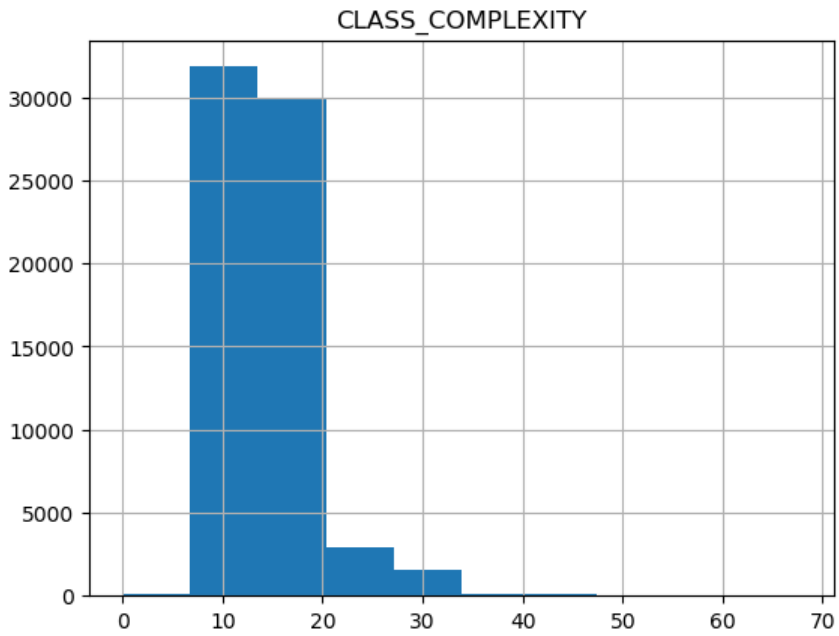


Figure 3.4: CLASS_COMPLEXITY values distribution

matter, consider the distribution of values for the CLASS_COMPLEXITY feature in Fig 3.4 as an example.

Upon comprehensive data examination, it's evident that the frequency of values across data points varies significantly, making a uniform scale division less effective due to varying prevalence. To address this, we prioritize values with higher frequencies. Achieving this involves normalization and classification. Normalization ensures data is brought to a common scale, while classification groups data into categories, facilitating structured and insightful analysis that considers varying frequencies and patterns. These techniques collectively provide a more robust understanding of the data, leading to more accurate and meaningful insights.

3.6.1 Normalization Techniques

For the purpose of Normalization, two techniques were employed: min-max normalization and mean normalization. Min-max normalization involved scaling metric values to a specific range. This technique preserved the relative proportions of metrics while ensuring they all fell within the same standardized range, allowing for direct comparisons. Mean normalization, on the other hand, centered the data around the mean value by subtracting the mean from each metric value. This approach helped eliminate biases caused by varying means and allowed for a more consistent assessment of metric values based on their deviations from the mean. These multiple normalization techniques collectively provided a comprehensive approach to ensure that metrics were prepared for fair and meaningful comparisons, enabling subsequent data analysis to be more accurate and insightful.

In the subsequent stage of the process, metric values are further refined by compressing them into a defined range, in this case between zero and 100, using a non-linear normalization technique. This compression preserves the inherent characteristics and relative magnitudes of

Table 3.5: Example of Database Entry with Normalization

Attribute	**Normalized Value**
FAULT_INDUCING_COMMIT_HASH	fa9f99a48f141abdb995884500e1add2dc83c19b
DATE_OF_INDUCING_COMMIT	2003-03-03 22:33:46+00:00
FILES	PropertyUtils.java
COMMIT_HASH_ON_FILE_BEFORE_FIX	ab939bf9fa5d41d093f93d5e95e14dad130b0d12
DATE_OF_COMMIT_HASH	2002-11-23 23:47:07+00:00
NCLOC	20.26932271
NEW_SQALE_DEBT_RATIO	83.85212808
VIOLATIONS	38.60734238
CODE_SMELLS	39.55391856
FUNCTIONS	13.17668441
LINES	20.36454164
CLASSES	10.29900332
CLASS_COMPLEXITY	10.36363636

the metrics, enabling more meaningful comparisons and analyses. This compression, provides insights into dataset dynamics and fluctuations. This approach helps assess disparities between normalized metric values, leading to a nuanced understanding of relative variances and relationships between metrics. The metric value compression proves instrumental in data transformation and analysis, enabling us to uncover intricate relationships and valuable findings within the dataset.

After applying normalization methods, the identical entry in the database is depicted in Table 3.5.

3.7 Identifying BICs

The process starts by extracting commit hashes related to bug-inducing commits from the primary database using a pairs file (The term ”Pairs file” derives from its content comprising pairs of bug-inducing and fault-fixing commit hashes). This method, described by P. Parul et al. [67], aims to retrieve commit hashes mentioned in these pairs from the main database. In this methodology, the SZZ algorithm is utilized to identify bug-inducing commits. To enhance accuracy and minimize false positives and noise, JIRA issues are linked to corresponding commits in the Version Control System (VCS). By examining all commits prior to an issue labeled as bugged, the outcomes are cross-referenced and verified. This process allows for a more reliable and accurate pinpointing of bug-inducing commits, ensuring a robust and dependable approach to tracking software defects. Through a search operation, commit hashes from the pairs file are cross-referenced with the main database to extract corresponding bug-inducing commit hashes. This efficient search process isolates and retrieves the relevant commit hashes, establishing a link between the pairs file and corresponding entries in the primary dataset.

The pairs file acts as a reference, specifying specific combinations of commit hashes of interest. By utilizing this information, the code systematically identifies and captures relevant

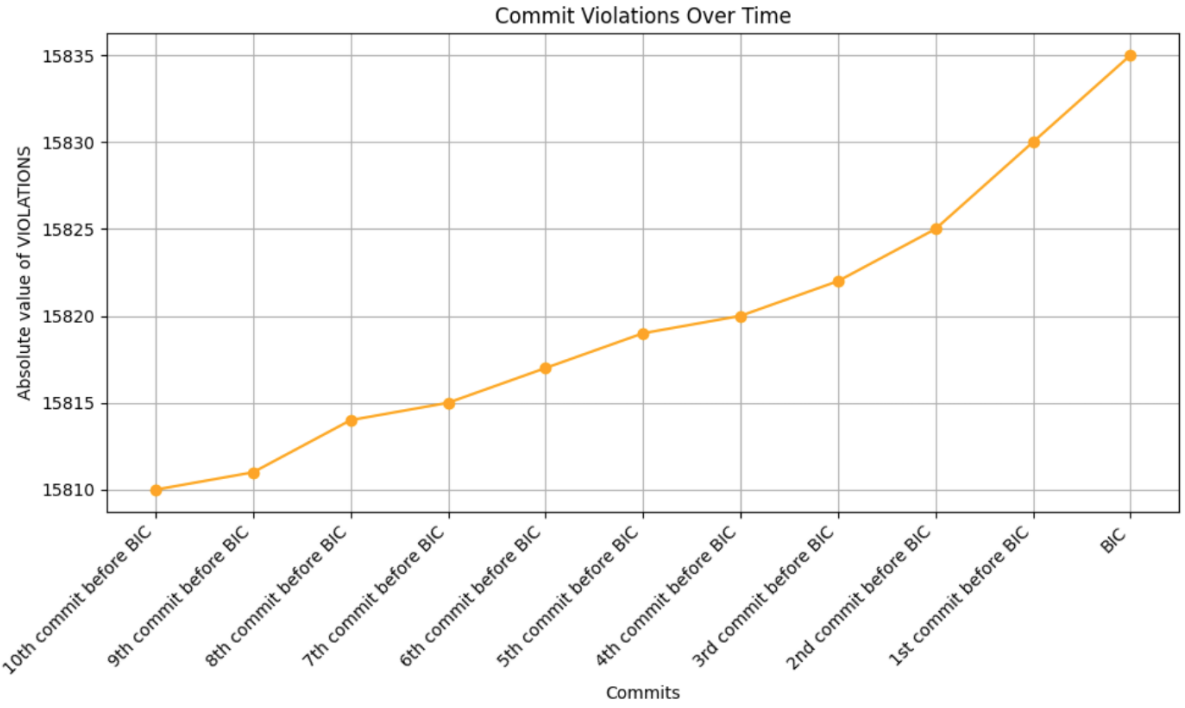


Figure 3.5: Changes in VIOLATIONS metric leading to a bug-inducing commit

commit hashes from the main database, contributing to the gradual population of a list with multiple DataFrames.

3.8 Relevant Commits

A crucial aspect of our research involves identifying relevant commits to construct the commit transition model. In the original database, there is no inherent relationship between each entry and the subsequent commits, which may pertain to different files and thus lack relevance to each other. To address this, we need to separate commits based on the files they affect and sort them chronologically. This approach ensures that commits affecting the same file are grouped together, making alterations in different features evident.

We utilized graphical representations to visualize changes observed in specific features. For a illustrative graph depicting these modifications, please refer to Figure 3.5.

3.9 Calculating the Differentials

The process of constructing commit states involves capturing nuanced characteristics beyond absolute metric values. It adopts a contextual and relative perspective on metrics, considering changes over time or in comparison to other observations, facilitating the identification of trends, patterns, and anomalies within commit data.

To implement this approach, computing differentials for each metric is crucial. Differentials quantify changes in metric values relative to preceding observations, enabling a nuanced

Table 3.6: Example of Database Entry with Differentials

Attribute	**Normalized Value**	**Diff Value of Related Commits**
NCLOC	20.26932271	0.005346332
NEW SQALE DEBT RATION	83.85212808	0.006060606
VIOLATIONS	38.60734238	0.007235142
CODE SMELLS	39.55391856	0.007242628
FUNCTIONS	13.17668441	0.006896552
LINES	20.36454164	0.005063122
CLASSES	10.29900332	0.010752688
CLASS COMPLEXITY	10.36363636	0.01754386

analysis that captures relative shifts and reveals patterns and fluctuations. They offer a comprehensive perspective for understanding metric dynamics, aiding trend identification, anomaly detection, and comparative evaluations across different observations.

After normalizing metric values through various techniques, the next step is calculating differentials between these normalized metrics. These differentials capture relative changes or variations in metrics over time or across different data points. By subtracting the normalized metric value at a particular point from its corresponding value at a previous point or from a defined reference point, it becomes possible to analyze how metric values evolved or fluctuated over time. These differential values serve as a crucial dataset for exploring patterns, trends, and deviations within the metrics, laying the foundation for in-depth data analysis and the extraction of valuable insights from the dataset.

Furthermore, differentials are also computed on absolute and non-normalized values as an additional approach to explore whether they could yield more meaningful results.

Table 3.6 displays both the differentials and normalized values for each metric. As elucidated in this section, the differential quantifies the extent of changes in each feature and metric. To streamline, the example refrains from providing detailed commit identities in the subsequent context to avoid redundancy and keep the table concise.

3.10 Outlier Detection

As part of our data exploration and analysis, our next undertaking involves the identification and characterization of outliers within each metric. Detecting outliers provides valuable insights into the distribution and peculiarities of the data, enabling a deeper understanding of its underlying patterns, variations, and potential anomalies. To accomplish this task, we have selected the widely recognized and statistically grounded Z-score method.

3.10.1 Z-Score method

The process of detecting outliers using the Z-score method can be summarized as follows:

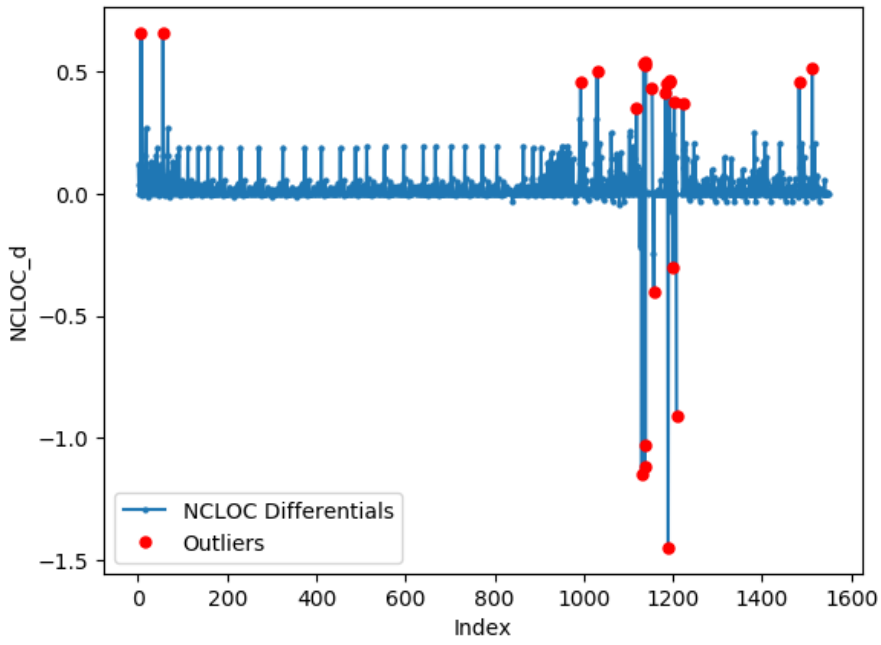


Figure 3.6: Z-Score Outliers in NCLOC differentials with threshold 3

Metric Selection: To comprehensively assess the data, we consider each metric individually. This approach allows us to examine the uniqueness and potential anomalies within each specific measurement, providing a more granular understanding of the dataset.

Z-Score Calculation: For each metric, the Z-score is computed for every data point. The Z-score is determined by subtracting the mean of the metric from the data point value and then dividing the result by the standard deviation. This transformation normalizes the data, expressing it in terms of standard deviations from the mean.

Threshold Determination: To classify a data point as an outlier, a threshold value is established. The choice of the threshold is crucial, as it determines the degree of deviation necessary to identify an observation as an outlier. In our study, we have set the default threshold to 3.0, considering data points with Z-scores greater than this threshold as outliers. The threshold value was carefully selected through an extensive analysis, which included testing multiple thresholds. We aimed to find a threshold that aligns with our expectations and requirements, providing a reasonable number of outliers considering our data size and value distribution.

Outlier Identification: Based on the calculated Z-scores and the established threshold, data points exceeding the threshold are flagged as outliers. These observations possess values that deviate significantly from the mean, signifying their potential as influential or anomalous elements within the dataset.

By employing the Z-score method for outlier detection, we gain a quantitative measure of the deviation of each data point from the mean, allowing for a systematic and objective identification of potential outliers.

As an example, you can observe the outlier detection results for the NCLOC differentials metric in Figure 3.6.

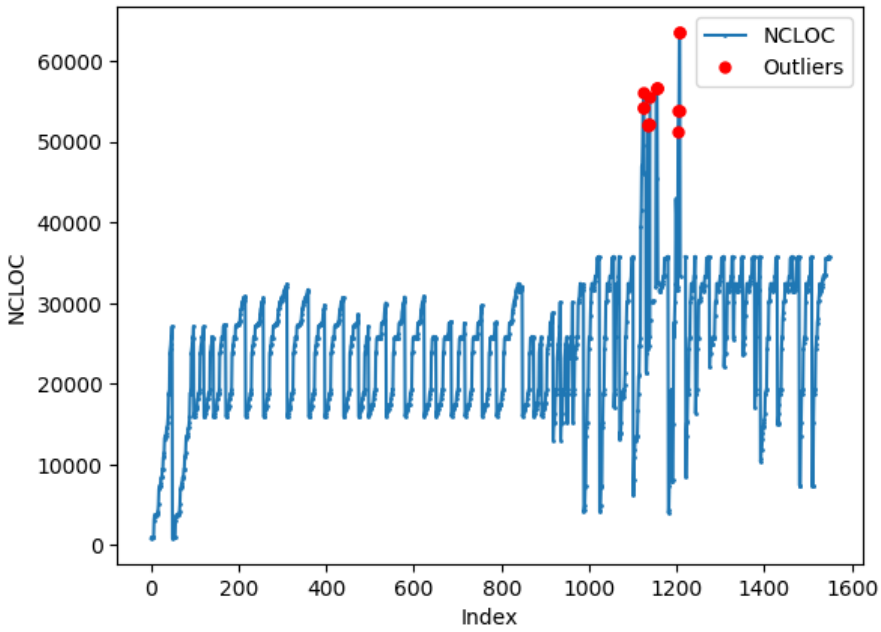


Figure 3.7: IQR Outliers in NCLOC

3.10.2 Interquartile Range (IQR)

To calculate the IQR, we first arrange the data points in ascending order. Then, we locate the median, which divides the dataset into two halves: the lower half (Q1) and the upper half (Q3). The IQR is determined by subtracting Q1 from Q3, representing the range within which the middle 50% of the data lies.

By focusing on this central portion of the data distribution, the IQR is less influenced by extreme values or outliers that may exist in the dataset. It provides a robust measure of variability that is less affected by these unusual observations, making it particularly useful for outlier detection.

We aim to compare the performance of the Z-Score method, previously employed for outlier detection, with the newly introduced IQR method. By implementing the IQR method, we expect to gain insights into how it identifies and handles outliers in our data. This comparative analysis will contribute to a more comprehensive understanding of the strengths and limitations of each technique, thus facilitating improved comprehension of our dataset.

As an example, you can observe the outlier detection results for the NCLOC metric in Figure 3.7.

3.10.3 Comparison and result

Upon applying the outlined outlier detection techniques to our dataset, we observed that both the Z-Score and IQR methods consistently identify and flag almost identical sets of outliers across various metrics. This consistency is because the Interquartile Range and Z-Score outlier detection methods usually produce similar results when the data distribution closely approximates a normal or Gaussian distribution, as is the case with our dataset. While minor dissimi-

larities may arise due to implementation nuances, the convergence in the number of identified outliers substantiates the reliability and robustness of these methods in detecting anomalous data points. This finding underscores the credibility of the Z-Score and IQR methods in accurately capturing instances deviating significantly from expected data patterns. Therefore, we used the union of the results from both these outlier detection methods for outlier detection and subsequent evaluation of their impact on our data analysis. It's worth noting that the outliers were calculated for absolute values, normalized values, and the differentials of both.

3.10.4 Standard Deviation

An additional approach was employed to identify distinct outliers in the dataset, involving the iterative evaluation of each metric value and the detection of borders whenever a value exceeded the average of preceding values by more than one standard deviation. This threshold was chosen based on the number of outliers detected and verification against our data to ensure it was meaningful. This method allowed us to detect significant shifts or deviations in the dataset, particularly in the magnitude of the "NCLOC" metric, which signified periods of considerable change or distinct data segments.

By utilizing the standard deviation as a measure of dispersion, deviations beyond the average value were identified as substantial and used as demarcation points. These points defined distinct temporal intervals within the data, facilitating further targeted analysis. This strategy offered a nuanced approach to demarcating the data, capturing key transitions and anomalies to improve the classification process and enhance the understanding of the dataset.

3.11 Classification bounds

To discretize our values and perform classifications effectively, we need to establish a specific scope for classification. This is essential because our database includes data from multiple projects, and varying factors within many metrics can complicate classification. For instance, in one project, if the normalized NCLOC value changes from 12 to 20, that change might be considered significant within the context of that project. However, in another project where NCLOC values reach up to 80, a change to 20 may not be significant. Additionally, factors such as refactoring in one project and other variables can influence our analysis. Therefore, we require epochs to carry out classification within these defined contexts.

The outlier analysis conducted revealed a significant concept regarding the classification of metrics into discrete values, primarily focusing on the "NCLOC" metric. The outliers observed within the "NCLOC" metric signify potential instances of substantial refactoring or significant changes in the project's structure. In response, the idea emerges to consider each detected outlier as a threshold for classification, effectively defining boundaries within which metric values can be categorized.

This approach facilitates the organization of metric values into distinct ranges or categories, aligning with the boundaries established by the detected outliers. Essentially, the outliers act as key reference points, demarcating the limits for classifying metric values and offering valuable insights into potential transformations or notable shifts within the project's structure.

The proposed classification methodology based on ”NCLOC” outliers not only aids in structuring and categorizing metric values but also enables the identification of patterns and trends within the data. It supports the exploration of how other metrics' values align or deviate concerning the defined outlier boundaries. Consequently, this classification framework enhances the understanding of interdependencies and associations among metrics, shedding light on the potential impact of outliers on the overall project dynamics.

3.11.1 Time Periods (Epochs)

To refine the classification range, it was vital to establish clear boundaries, accurately represent outliers, and confirm an adequate number of commits and bug-inducing commits within specific time priods known as ”periods.” These conditions are essential for preserving the accuracy and robustness of the classification process while minimizing the risk of misclassification or bias. To achieve this, the study set a distance threshold of 0.3 to identify outliers as classification boundaries, as detailed in Section 3.10. This value was selected after testing various thresholds and observing that 0.3 resulted in a reasonable number of outliers, as demonstrated in Figs 3.6 and 3.7. This approach ensured a reliable and consistent classification methodology, enhancing the validity of the research and experimentation.

All these methods were applied to both the absolute values and normalized values, including the calculation of differentials for both. Ultimately, the pertinent commits were grouped, sorted chronologically, and outliers within those groups were identified. This allowed for the classification methods to be applied to each distinct period.

3.12 Discretizing the values

After obtaining the selected features along with their respective absolute numerical values, normalized values, and differentials, as well as obtaining the periods, the next step involves discretizing the metrics. To achieve this, a discrete form with five distinct values was employed. This approach enhances data representation and analysis, aiding in pattern and trend identification. Discretization, a crucial process in data analysis, improves result quality by mitigating the impact of outliers and simplifying data while preserving its core characteristics. The selection of five values makes the discretization process more manageable and facilitates result interpretation. The discrete values are presented below.

- 0: Very Low
- 1: Low
- 2: Medium
- 3: High
- 4: Very High

In order to make the analysis more tractable, we later in the thesis combined the last two discrete values and the first two and come up with three categories as follows:

- 1: Low
- 2: Medium
- 3: High

3.12.1 Classification

To streamline the analysis of extensive datasets, it is advisable to initiate the process with a categorization based on scale. This entails segmenting the data into distinct intervals or categories, such as 50, 100, 200, 400, and 10,000, and tallying the data points falling within each interval. By maintaining uniformity among the intervals, this method simplifies the analysis of extensive information, rendering it more manageable and aiding in the interpretation of data in discrete categories. The primary aim is to offer a more lucid view of the data and facilitate efficient decision-making.

This step involves the equitable distribution of data among the final discrete values. The goal is to distribute data points evenly across each discrete value while upholding data integrity. While achieving perfect equality may be unattainable, efforts are made to allocate the data as evenly as possible, striving for the closest approximation to equality within practical constraints. The process is outlined below.

Note that the assignment of each value to a group depends on the classification scope, and it is impossible to establish a fixed threshold for every value and definitively state that a metric value lower than X belongs in group Y. This is because the aim is to enhance the classification process by basing groups solely on changes in specific metrics within the commit stream, which adds depth and nuance to the categorization.

The data analysis process then progresses to the classification of these normalized values into five distinct categories. This classification is carried out using three techniques: Decision Tree, K-Means Clustering, and Binning using Percentile Classification.

The combination of these processes allows for a comprehensive analysis of the metric values, with data being grouped into meaningful clusters. The methodology helps identify similarities and patterns within the dataset, aiding in the exploration and interpretation of the underlying data structure.

Binning Using Percentile Classification

Percentile Classification is employed as a classification method. This technique involves dividing the metric values into predefined percentile ranges, such as quintiles or percentiles. Each range represents a category, and the metric values are assigned to the corresponding percentile category based on their relative position within the distribution. This approach ensures that the metric values are classified into five equal-sized groups, allowing for an equal representation of the data across the categories. As an example, you can see the result of this classification method on normalized differential values of CODE_SMELLS metric in Fig 3.8.

Table 3.7 displays the results of percentile classification on both the normalized values and differentials.

The column ”Percentile Classification on Diff” displays the classification results based on the differential value of the corresponding metric.

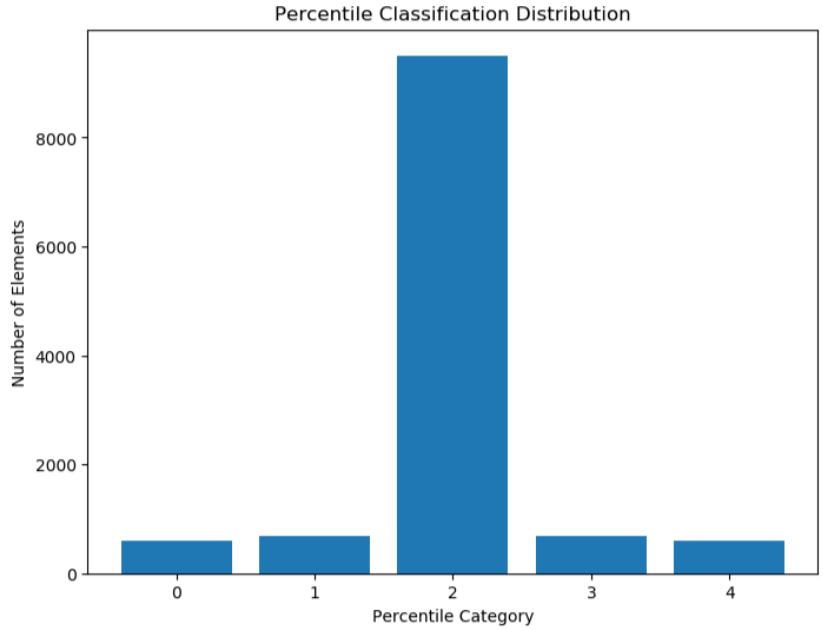
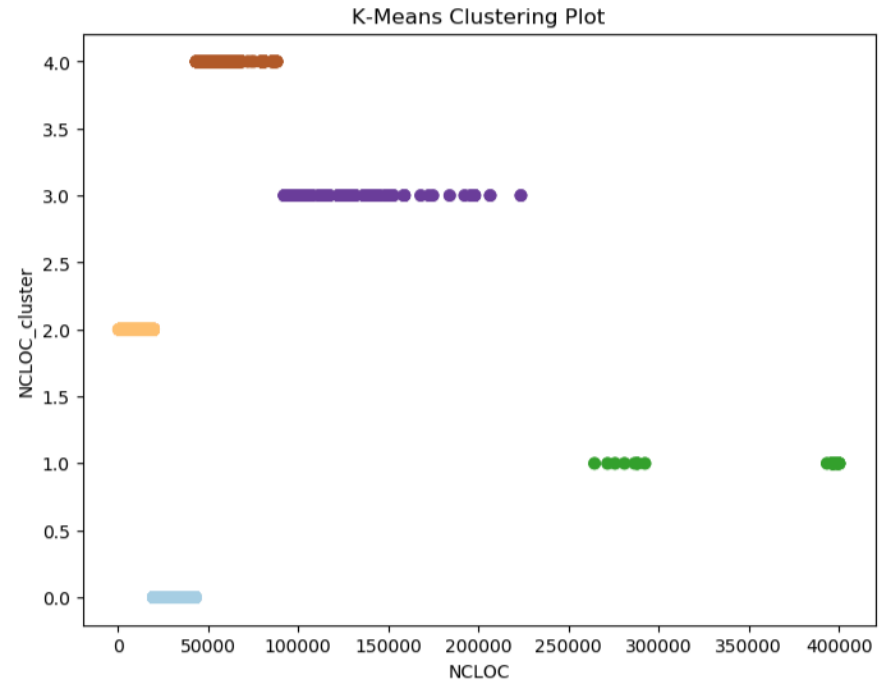


Figure 3.8: Pecentile Classification Result

Table 3.7: Example of Database Entry with Percentile Classification

Attribute	**Normalized Value**	**Diff Value of Related Commits**	**Percentile classification**	**Percentile Classification of Diff**
NCLOC	20.26932271	0.005346332	4	1
NEW SQALE DEBT RATIO	83.85212808	0.006060606	0	1
VIOLATIONS	38.60734238	0.007235142	4	0
CODE SMELLS	39.55391856	0.007242628	4	0
FUNCTIONS	13.17668441	0.006896552	4	0
LINES	20.36454164	0.005063122	4	1
CLASSES	10.29900332	0.010752688	4	0
CLASS COMPLEXITY	10.36363636	0.01754386	1	2




```
Number of elements in Cluster 0: 7180
Number of elements in Cluster 1: 55
Number of elements in Cluster 2: 5597
Number of elements in Cluster 3: 341
Number of elements in Cluster 4: 1155
```

Figure 3.9: K-Means Clustering Result

K-Means Clustering

The data analysis process consists of several crucial steps, including the selection of the desired number of clusters, which, in this specific context, focuses on the creation of five distinct clusters. Subsequently, data is prepared for clustering by transforming it into a 2D array, where each row corresponds to a data point, and a single column represents the selected feature under consideration.

After this preparation, a K-Means object is instantiated with the predefined number of clusters, and the data is fitted to this object, initiating the clustering process. As clustering concludes, each data point is assigned predicted labels, which are then integrated as a new column in the original dataframe. This step delivers valuable insights into the distribution of data points across different clusters, providing a clear count of instances within each category. The approach harnesses the K-Means clustering algorithm to effectively categorize data points within a single column into distinctive clusters, facilitating a comprehensive assessment of data distribution and the identification of cohesive data groups based on their proximity to cluster centroids. As an example, you can see the result of this classification method on compressed values of NCLOC metric in Fig 3.9.

Table 3.8 displays the results of K-Means Clustering on both the normalized values and differentials.

Table 3.8: Example of Database Entry with K-Means Clustering

Attribute	**Normalized Value**	**Diff Value of Related Commits**	**K-Means Clustering**	**K-Means Clustering on Diff**
NCLOC	20.26932271	0.005346332	3	0
NEW SQALE DEBT RATIO	83.85212808	0.006060606	3	1
VIOLATIONS	38.60734238	0.007235142	4	0
CODE SMELLS	39.55391856	0.007242628	4	0
FUNCTIONS	13.17668441	0.006896552	1	3
LINES	20.36454164	0.005063122	2	3
CLASSES	10.29900332	0.010752688	2	0
CLASS COMPLEXITY	10.36363636	0.01754386	0	2

Decision Tree

In this methodology, we establish five distinct categories by sourcing data from a designated column within a specified dataset. Following this, any missing data points are imputed using suitable techniques, such as mean or median values from the dataset, ensuring data completeness. The data is then divided into these predefined categories. Subsequently, a decision tree classifier is constructed, customized according to the desired number of categories, and trained on the dataset. Predicted labels for each data point are obtained and added as a new column to the original dataset. Finally, the algorithm provides a count of instances within each category, furnishing valuable insights for interpreting and analyzing the resulting classifications. This approach is essential for uncovering patterns and trends in numerical data that might not be readily discernible. The Decision Tree algorithm serves as a classification method, responsible for partitioning metric values based on specific decision criteria. It establishes a hierarchical structure of decision nodes that sequentially segment the data using predetermined features or thresholds, enabling the identification of distinct categories through iterative division based on the most informative attributes. As an example, you can see the result of this classification method on differential values of NEW_SQALE_DEBT_RATIO metric in Fig 3.10.

Upon completing these steps, we generate a vector comprising multiple discrete values for each commit, encapsulating the relevant categories applied to the data.

Table 3.9 displays the results of Decision Tree Classification on both the normalized values and differentials.

Finding the dominant answer

Employing three distinct classification techniques, metric values are categorized into five meaningful and interpretable groups, ensuring comprehensive and robust categorization of normalized metric values. This comparative approach assesses the consistency and reliability of categorization outcomes by identifying the category assigned by at least two out of the three

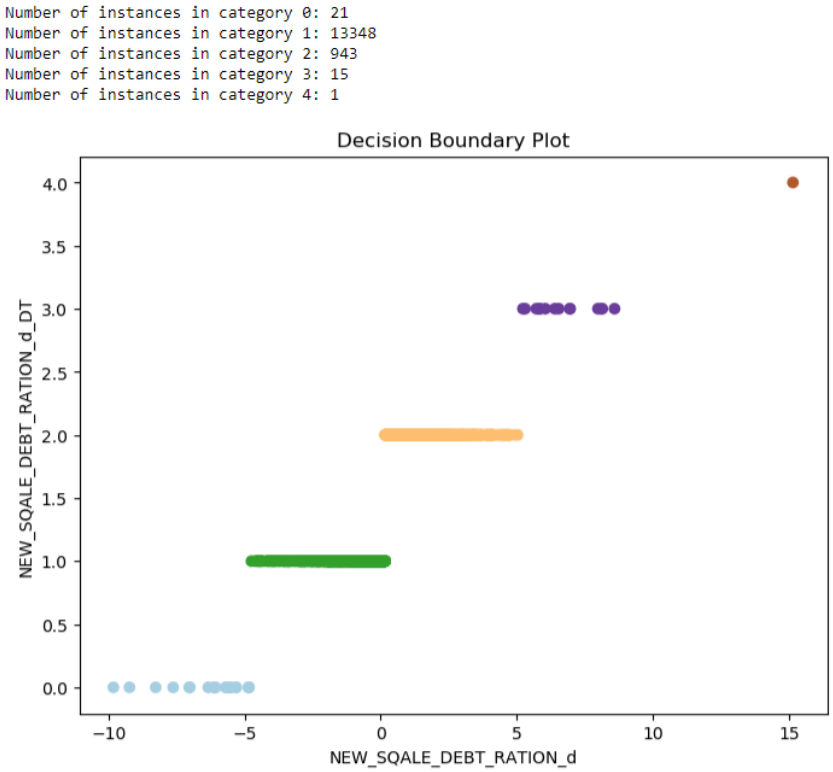


Figure 3.10: Decision Tree Classifier Result

Table 3.9: Example of Database Entry with Decision Tree Classification

Attribute	**Normalized Value**	**Diff Value of Related Commits**	**Decision Tree Classification**	**Decision Tree Classification on Diff**
NCLOC	20.26932271	0.005346332	4	0
NEW SQALE DEBT RATIO	83.85212808	0.006060606	0	2
VIOLATIONS	38.60734238	0.007235142	4	0
CODE SMELLS	39.55391856	0.007242628	4	0
FUNCTIONS	13.17668441	0.006896552	4	0
LINES	20.36454164	0.005063122	4	1
CLASSES	10.29900332	0.010752688	4	0
CLASS COMPLEXITY	10.36363636	0.01754386	0	3

Table 3.10: Example of Database Entry after Classification

Attribute	**Normalized Value**	**Diff Value of Related Commits**	**Discrete Value**	**Discrete Value of Diff**
NCLOC	20.26932271	0.005346332	4	0
NEW SQALE DEBT RATIO	83.85212808	0.006060606	0	**1**
VIOLATIONS	38.60734238	0.007235142	4	**0**
CODE SMELLS	39.55391856	0.007242628	4	**0**
FUNCTIONS	13.17668441	0.006896552	4	0
LINES	20.36454164	0.005063122	4	1
CLASSES	10.29900332	0.010752688	4	0
CLASS COMPLEXITY	10.36363636	0.01754386	0	**2**

methods, providing valuable guidance for subsequent analyses and informed interpretations based on the consistently assigned category.

To enhance our dataset's comprehensiveness, our objective is to identify the most prevalent classification outcome among various methods applied to each commit and feature category. We aim to extract the predominant answer resulting from all the classification techniques used, significantly enriching our database for more in-depth analysis. In cases where no clear consensus emerges and each classification method produces a distinct result, we prioritize the classification outcome generated by the Decision Tree classifier, even though such occurrences are infrequent.

In the end, we save both the dominant answers and our original classification results. We do not discard each classification result but retain all three in case one of the classification methods proves to be more effective than the others.

As an example, let's examine the results of classification for the CLASS COMPLEXITY metric across our three classification methods. The classification of the absolute value of this metric using percentile classification resulted in a value of 1 (Low), with K-means, it was 0 (Very Low), and with the decision tree, it was 0 (Very Low). Two of these methods categorize this value as 0 (Very Low), which constitutes the majority, so the final discrete value for this metric is 0 (Very Low).

Next, let's consider the results of classifying the amount of change or $ValueofCommit_n - ValueofCommit_{n-1}$ for this metric. The classification of the differential value using percentile classification resulted in 2 (Medium), with K-means, it was also 2 (Medium), and with the decision tree, it was 3 (High). Since two of these methods categorize this value as 2 (Medium), which is the majority, the final discrete value for this metric is 2 (Medium).

Table 3.10 displays the results of Classification on both the normalized values and differentials.

Now, as you can see, we have eight metrics available for assessing the state of a commit. As mentioned earlier in Section 3.5.4, four of these metrics relate to code size (NCLOC,

FUNCTIONS, LINES, CLASSES), while the other four pertain to code quality (NEW SQALE DEBT RATIO, VIOLATIONS, CODE SMELLS, CLASS COMPLEXITY). Moving forward, only the metrics related to code quality will be used to represent the state of the commit.

Another key point to consider is the availability of two distinct sets of discrete values to work with. One set is derived from classifying the absolute normalized values, while the other comes from classifying the amount of change correlated with the metric from the previous relevant commit (differential values). Moving forward, we will prioritize the use of discrete values associated with the differential to represent the commit state.

The rationale for this decision is that, within the context of a stream of commits, classification based on absolute values tends to increase from the first to the last commit. Using these values to represent the commit's state would lead to limited meaningful transitions, as most of our metrics would remain static. In contrast, classifying each metric's change from one commit to the next provides a more dynamic and informative representation of commit states, allowing us to capture meaningful transitions and insights into the evolution of the codebase over time.

The two states representing this specific commit are 0440 (Absolute Values) and 1002 (Differential). These four features prioritize code quality over code size in our analysis.

The conclusive decision for this research was to utilize classifications based on differentials to represent the commit's state. This metric accurately reflects alterations and changes in our values. Consequently, the final state for this commit will be denoted as 1002. Which means the following:

- NEW SQALE DEBT RATIO: Low
- VIOLATIONS: Very Low
- CODE SMELLS: Very Low
- CLASS COMPLEXITY: Medium

3.13 Creating the Dataframe

Upon generating the vectors for each commit, we must create a comprehensive database that encompasses these vectors and other pertinent information, including the commit hash and project name, which serve as unique identifiers for each database entry.

Furthermore, to streamline the data manipulation and classification techniques employed in this study, we will merge them into a unified dataframe, incorporating absolute values as needed. Subsequently, this resulting dataset will be stored in a suitable format for further analysis. This step enhances data comprehensiveness by consolidating all relevant information into a single entity, simplifying interpretation and analysis.

Each entry in our database conforms to a predefined structure, ensuring consistency and facilitating seamless querying, manipulation, and analysis of the data.

In Table 3.11, a portion of our final database containing absolute value metrics is displayed for reference.

Table 3.11: A Slice of One Database Entry

Attribute	Value	Discrete Value
Fault Inducing Commit Hash	fa9f99a48f141abdb995-884500e1add2dc83c19b	-
Files	PropertyUtils.java	-
Commit Hash on File Before Fix	ab939bf9fa5d41d093f9-3d5e95e14dad130b0d12	-
Date of Commit Hash	2002-11-23 23:47:07+00:00	-
NCLOC	20.2693	0
New Sqale Debt Ration	83.8521	1
Violations	38.6073	0
Code Smells	39.5539	0
Functions	13.1767	0
Lines	20.3645	1
Classes	10.299	0
Class Complexity	10.3636	2

3.14 Commit States

As mentioned in the previous section, we use differential values after classification to represent the commit's state. This approach is necessary because using absolute values within a scope of only six commits results in states that are nearly identical, with few exceptions. Consequently, this would lead to transitions and patterns consisting of six identical states, making the analysis both impossible and meaningless. After completing the outlined procedures, every commit will be depicted by a state comprising four features. For instance, if a commit's state is denoted as 2104, it signifies that the NEW_SQALE_DEBT_RATIO feature is medium, VIOLATIONS is low, CODE_SMELLS is very low, and CLASS_COMPLEXITY is very high. In certain experiments, this state might be equivalent to 2113.

3.15 Justification of using three discrete values

Incorporating three discrete values instead of five in some of our experiments holds significant importance for several compelling reasons. This section delineates some of these pivotal rationales.

1. **Simplicity**: A smaller number of discrete values simplifies the interpretation and analysis of data. With only three values, the distinctions between states are clearer and easier to understand, reducing the cognitive load on analysts and stakeholders.

2. **Reduced Complexity**: Working with three discrete values simplifies the computational and analytical processes involved. It streamlines algorithms, reduces computation time, and minimizes the complexity of statistical analyses.

3. **Efficiency**: With fewer discrete values, data storage requirements are reduced, leading to more efficient data management. Additionally, visualization and reporting tasks become simpler, as there are fewer categories to represent graphically or in tabular format.

4. **Robustness**: In some cases, using fewer discrete values can lead to more robust models. With fewer categories, the model is less sensitive to noise or small variations in the data, which can improve its generalizability and stability.

5. **Practicality**: Using three discrete values aligns better with practical considerations or requirements of some of these experiments. It is more intuitive to work with a smaller set of categories, facilitating communication and decision-making.

Nevertheless, it's imperative to weigh the trade-offs. While opting for three discrete values provides simplicity and efficiency, it may compromise the granularity and nuance in data representation. The decision between employing three or five discrete values hinges on the particular requirements, objectives, and limitations of the project or analysis.

As a consequence of this decision, the entry in our database for the running example will be illustrated in Table 3.12.

Table 3.12: A Slice of One Database Entry with three discrete values

Attribute	**Value**	**Discrete Value**
NCLOC	20.2693	1
New Sqale Debt Ration	83.8521	1
Violations	38.6073	1
Code Smells	39.5539	1
Functions	13.1767	1
Lines	20.3645	1
Classes	10.299	1
Class Complexity	10.3636	2

By opting for the classification based on differentials, the state for this commit will be denoted as 1112, signifying a low-low-low-medium classification for the features NEW SQALE DEBT RATION, VIOLATIONS, CODE SMELLS, and CLASS COMPLEXITY.

Chapter 4

Commit State Modeling

4.1 Preface

After completing the pre-processing phase, the next step is to construct a state model denoting the transitions form one commit state to the next along with the corresponding transition frequencies. This chapter will explain the methodology used to create the graph model, including the approach, techniques, and tools employed to represent the relationships and connections between different commits.

We will discuss how we establish nodes and edges within the state model, with nodes representing individual commits, and edges illustrate the frequency of transitions between these commits.

By providing a detailed explanation of our methodology, we aim to give the reader a comprehensive understanding of how we transform the pre-processed data into a structured graph model, setting the stage for subsequent analysis and insights into the relationships and dynamics present in our commit data.

4.2 Outline for the Commit State Modelling Phase

All the procedures outlined in this section adhere to the guidelines specified in the accompanying activity diagram in Fig 4.1

The steps depicted in the Activity diagram 4.1 are discussed in the following sections and are summarized as follows:

1. Commit Transition Model: We begin by creating transitions, placing the states of consecutive commits in sequence to identify patterns.

2. Discrete Dataset: We construct the database from the previous chapter using discrete values and the earlier established transitions, followed by a review of the dataset.

3. Transition Trimming: In this phase, we refine the transition database by selecting only valuable transitions based on predetermined criteria.

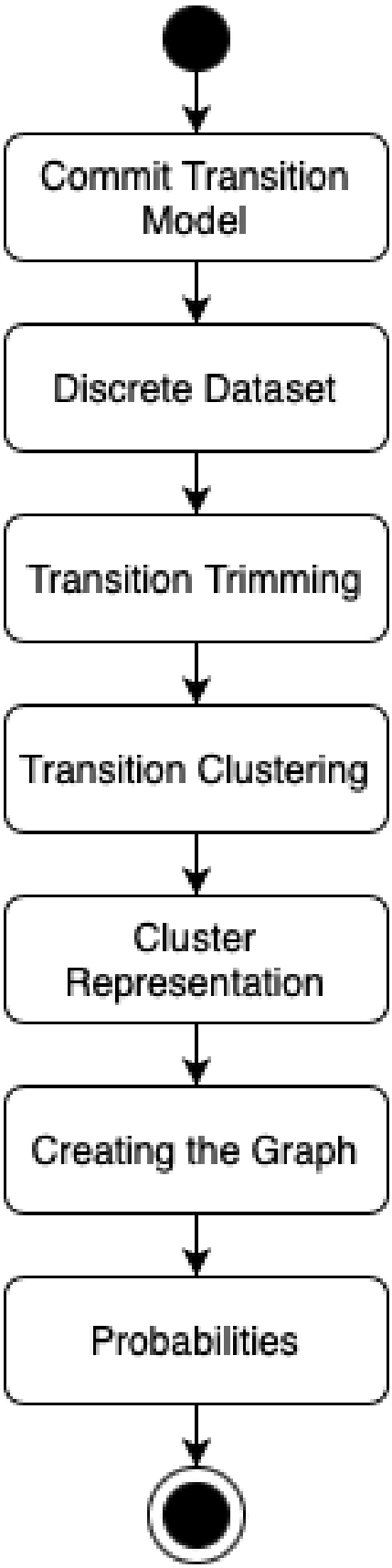


Figure 4.1: Activity Diagram for Commit State Model

4. Transition Clustering: We cluster the transitions, grouping similar transitions into categories to reduce dataset size and enhance the quality of the analysis. This is achieved using K-Means and Hierarchical Clustering metrics.

5. Cluster Representation: We represent clusters containing transitions with a unified depiction for consistency.

6. Graph Creation: We create a graph, treating each commit state as a node and transitions between commits as edges.

7. Probabilities: We calculate the probability of each transition, which serves as the value of each edge in the final graph.

We explain these steps in more detail in the following sections.

4.3 Commit Transition Model

After acquiring a comprehensive dataframe that consolidates all pertinent data, it becomes essential to establish a chronological sequence of commits leading to a bug-inducing commit. This entails recording the indices of all bug-inducing commits and their respective predecessors.

To accomplish this objective, we systematically traverse the databases established for each project, meticulously extracting fault-inducing commits and their antecedent five commits. Comprehensive information, encompassing pertinent metrics, is meticulously compiled and incorporated into our ultimate dataset.

In this research, we define each commit as belonging to a specific state. Transitions between different states, or in other words, transitions between successive commits or a series of commits, are referred to as ”patterns” in our study. For instance, when we discuss a pattern, we may be referring to a sequence of 2-6 commits that are connected and occur in a specific order. Each pattern will be as below;

$$State_1 \rightarrow State_2 \rightarrow State_3 \rightarrow State_4 \rightarrow State_5 \rightarrow State_6$$

Each state is a four dimensional vector with each of them initially consisted of five discrete values: 0, 1, 2, 3, and 4, with each value representing a specific quality metric relevant to our research. For simplicity and better performance of our experiments, we later combined categories 0 and 1 into one group and categories 3 and 4 into another, resulting in three discrete values. These quality metrics provide insights into the characteristics and quality of the code at each state, allowing us to assess the impact of different commits on the overall software quality and identify patterns that may lead to bug-inducing changes.

4.4 Discrete Dataset

In our resulting dataset, each line contains six states. Each state is represented by a four-digit number, which corresponds to the category of a specific metric. As previously mentioned, we

solely utilized metrics representing code quality to describe each commit's state. Code size metrics may not provide significant insights in this context. (Section 3.5.4) The metrics we are interested in are as follows:

- **NEW_SQALE_DEBT_RATIO:** This metric measures code quality debt.
- **VIOLATIONS:** This metric assesses code violations.
- **CODE_SMELLS:** It represents the presence of code smells.
- **CLASS_COMPLEXITY:** This metric gauges class complexity.

Each metric described earlier falls into one of five categories: very low (0), low (1), medium (2), high (3), and very high (4). Subsequently, these categories are converted into three final values.

These metrics' values change over transitions, starting from the fifth commit before a bug-inducing commit and continuing until the sixth state, which corresponds to the bug-inducing commit itself.

4.5 Transition Trimming

Upon acquiring the dataset, a critical step in streamlining the analysis process involves eliminating redundant paths leading to a bug-inducing commit. Redundancy reduction enhances the dataset's clarity and conciseness, focusing on essential information.

Additionally, we take measures to enhance data integrity and quality by excluding all paths that contain one or more null states as values. This data cleansing process ensures that the dataset remains consistent and free from potential inconsistencies that may arise from incomplete or missing information. These actions contribute to a more robust and accurate foundation for subsequent analysis and evaluation.

As a result, our final dataset comprises 4,204 records, where each record uniquely represents a path leading to a bug-inducing commit.

The final transitions file will have a structure as presented in Table 4.1

Table 4.1: State Transitions

Transition Number	**Fifth Commit Before BIC**	**Fourth Commit Before BIC**	**Third Commit Before BIC**	**Second Commit Before BIC**	**First Commit Before BIC**	**Bug Inducing Commit**
case 1	1333	2113	1221	2333	0002	4444
case 2	1333	1221	1223	0444	1221	4002
case 3	3002	1333	2112	1221	2113	3002
case 4	3112	3002	3112	3003	3112	3112
case 5	4002	1333	3112	1221	2223	3112

In the provided table, each cell represents a commit state across a series of transitions. For example, in case 1, the state of the fifth commit before the bug-inducing commit (BIC) is represented by the number 1333, which corresponds to low-high-high-high states across the four chosen metrics. The transition then progresses to 2113 until it culminates at the bug-inducing commit. Here's the sequence of transitions for each metric:

- **New Sqale Debt Ratio**: Low → Medium → Low → Medium → Very Low → Very High
- **Violations**: High → Low → Medium → High → Very Low → Very High
- **Code Smells**: High → Low → Medium → High → Very Low → Very High
- **Class Complexity**: High → High → Low → High → Medium → Very High

These transitions illustrate the evolution of the commit state for each metric over time.

4.6 Transition Clustering

Two clustering techniques are employed to identify natural groupings or patterns within the data. It aids in the classification of similar states or transitions, making it easier to discern clusters of related information and explore the relationships between them.

Distances

For these particular methods, we utilized the cosine similarity values that had been previously computed for each transition as the distance metric between elements. This measure of cosine similarity served as a crucial foundation, enabling us to assess the similarity between transitions and inform the clustering process based on the computed distances.

As an illustrative example, let's examine the first two distinct transitions from our transitions file, as shown in Table 4.2.

Table 4.2: First Two Distinct Transitions

Transition Number	**Fifth Commit Before BIC**	**Fourth Commit Before BIC**	**Third Commit Before BIC**	**Second Commit Before BIC**	**First Commit Before BIC**	**Bug Inducing Commit**
case 1	1333	2113	1221	2333	0002	4444
case 2	1333	1221	1223	0444	1221	4002

We treat each of these transitions as a 24-dimensional vector. To illustrate, consider the transitions '133321131221233300024444' and '133312211223044412214002'. The cosine similarity between these two vectors is calculated as 0.7778. Since this value serves as a measure of similarity and falls within the range of 0 to 1, we can obtain a distance metric by subtracting it from 1. In this instance, our defined distance value is 0.2222.

4.6.1 K-Means Clustering

Within our finalized dataset, we conducted K-Means clustering to explore potential groupings among the transitions. K-Means clustering is an unsupervised machine learning technique that helps identify clusters of similar elements based on specified features. In our analysis, we initiated this process by pre-defining a range of cluster numbers, ranging from 50 to 400.

Starting with 50 clusters, this configuration results in larger clusters that contain less similar elements. As the number of clusters increases, for instance, to 400, the granularity of the groupings becomes finer, with elements exhibiting more similarities grouped together. This stepwise exploration of different cluster numbers allows us to examine the data at varying levels of granularity and discern patterns and relationships that might not be apparent when the clusters are too broad or too specific.

The objective of this clustering analysis is to gain a comprehensive understanding of the transitions' inherent structure and to identify potential patterns and relationships between different states. By adjusting the number of clusters, we can fine-tune the level of detail in our analysis, uncovering insights that are sensitive to the intricacies of the data. This multi-faceted approach contributes to a more nuanced interpretation of the dataset, providing valuable insights into the transitions and their relationships.

Implementation

In the context of K-Means clustering, our objective is to identify an ideal number of clusters for our dataset, we define ideal as close-to-uniform distribution of elements among all the clusters. To achieve this, we perform the clustering process with different numbers of clusters and evaluate the results. The aim is to strike a balance where we have a reasonable number of clusters that effectively group data points, ensuring that each cluster contains a well-distributed set of elements. This process helps us uncover the underlying structure of the data and can be a critical step in gaining insights into patterns and relationships within the dataset. The selection of an appropriate number of clusters is a fundamental decision in K-Means clustering and can significantly impact the quality and interpretability of the final clustering results.

In the end, we have determined that using 100 clusters is the optimal choice. The distribution of data within each cluster is visualized in Table 4.3.

Table 4.3: Distribution of elements among 5 out of 100 clusters in K-Means

Cluster Number	Number of Elements
1	37
2	51
3	53
4	34
5	30

To select an optimal number of clusters, we aim for a relatively even distribution of elements across all clusters, as previously discussed. In this clustering method, we have determined that 100 clusters provide an optimal grouping of the data, as mentioned earlier. Within

these clusters, we observe variations in the number of elements. The largest cluster contains 97 elements, while the smallest encompasses 7.

However, a challenge with this approach lies in the fact that, as previously mentioned, we are employing cosine similarity as the similarity metric. Unfortunately, the elements within each cluster do not exhibit acceptable cosine similarities, resulting in substantial variations in distances. This issue is primarily attributed to the low number of clusters chosen for this clustering technique and dataset.

On the other hand, increasing the number of clusters to a range of even 400 does not resolve this problem. To achieve a more uniform distribution of similar elements within each cluster, it would require a substantial increase in the number of clusters, exceeding 1000, which is not a feasible solution for our purposes.

For a clearer understanding of the situation, you can examine the characteristics of eight out of the 100 clusters, including their sizes, maximum distances, and minimum distances within each cluster, as presented in Table 4.4.

Cluster Size	Max Distance	Min Distance
37	0.2507	0.0025
51	0.3900	0.0044
53	0.3492	0.0041
34	0.2981	0.0025
30	0.1900	0.0062
59	0.4133	0.0046
41	0.3097	0.0034
66	0.4412	0.0049

Table 4.4: Range of distances inside each cluster in K-Means

As shown in the table, the distribution of elements across clusters is uneven, which may not necessarily indicate that the clustering technique is inadequate. However, the significant maximum distance between elements within each cluster suggests that the K-Means clustering approach may not be the most suitable method for classifying our transitions. This large distance implies that the clusters may contain diverse or loosely related groups of data, undermining the efficacy of the technique for our purposes. As a result, it may be worth exploring alternative clustering methods that can better capture the nuances and relationships within our data.

4.6.2 Hierarchical Clustering

In addition to the K-Means clustering approach, we considered employing the Hierarchical Clustering method, which appears to be a more apt choice for our analysis. Hierarchical clustering allows us to uncover intricate structures within the data by forming a hierarchy of clusters. This method offers the advantage of being able to interpret the data at different levels of granularity. To perform hierarchical clustering, we pre-defined specific thresholds and linkage methods, including single, complete, average, and Ward, which dictate how clusters are

formed. These linkage methods have distinct ways of calculating the distances between clusters and are chosen based on the nature of the data and the goals of the analysis.

In our approach, the distances between transitions are measured using cosine similarity. Each transition is treated as a 24-character vector, and the cosine similarity metric quantifies the angle between these vectors, indicating the similarity between transitions. This allows us to assess how closely related different transitions are in a multi-dimensional space.

To systematically apply hierarchical clustering, we first establish a fixed threshold that applies to all linkage methods. We then assess how each linkage method reacts to this uniform threshold. Subsequently, we observe the resulting clusters and their composition for each linkage method. Based on this analysis, we can determine the most appropriate threshold for each linkage method. The thresholds are not fixed in advance; rather, they are determined by the method's behavior during the analysis.

By applying this approach to hierarchical clustering, we can tailor the clustering process to the unique characteristics of our data, ensuring that we obtain meaningful and interpretable clusters. This method provides a versatile way to explore the dataset, uncover relationships between transitions, and identify significant patterns that may vary based on the linkage method and threshold applied. The results are systematically saved for further in-depth analysis, providing a valuable resource for understanding the structure of the data and the relationships between different transitions.

Implementation

Before discussing the implementation, let's briefly outline the methodology involved in the clustering analysis:

Threshold Selection

In the initial phase, we conducted an exploratory analysis by testing a range of thresholds, ranging from 0.1 to 0.9, in order to determine the optimal one. It's important to note that the choice of threshold plays a crucial role in this context; a lower threshold produces a greater number of clusters, whereas a higher threshold leads to fewer clusters.

Following this preliminary examination, we gained insights into what range of thresholds would be most suitable for each linkage method. Based on this understanding, we made appropriate adjustments to the set of thresholds to ensure that they align with the characteristics of the data and the objectives of our analysis. This iterative process allowed us to fine-tune the clustering parameters for each linkage method, ultimately facilitating a more effective and tailored clustering outcome.

Linkage Methods

The code employs four distinct linkage methods, namely single, complete, average, and ward, to construct hierarchical clusters from the data. Each of these methods provides distinct insights into the underlying structure of the dataset. To ensure the most meaningful results, we employed tailored thresholds for each linkage method. Subsequently, we meticulously stored the most optimal results obtained for each method. This approach allowed us to capture the

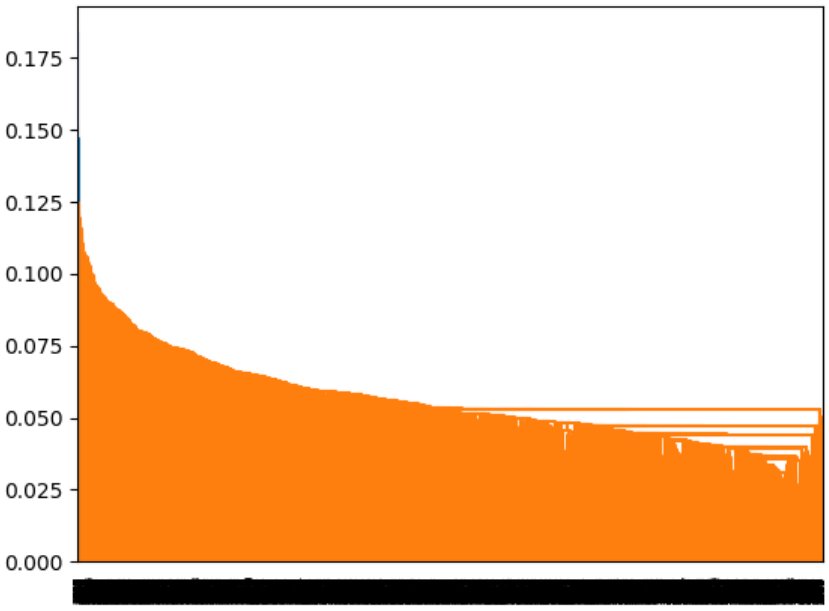


Figure 4.2: Single Dendrogram

nuances and variations within the data while preserving the best outcomes for further analysis and interpretation. Subsequently, a Dendrogram plot was generated for each of the employed linkage methods.

The dendrogram plot for each method is presented in Figs 4.2, 4.3, 4.4, and 4.5.

As an example, consider the preliminary threshold set for the single linkage method with a threshold of 0.1. In this configuration, the clustering process yields 50 clusters. Notably, one cluster stands out with a substantial 4,126 elements, while the remaining clusters are relatively smaller, containing only 1 to 5 elements. The results, as depicted in Fig. 4.6, were generated using a dedicated script to extract and visualize this data.

Given the significant imbalance in the distribution of elements among the clusters, it is evident that the current clustering result is sub-optimal. The non-uniform distribution raises concerns about the effectiveness of the clustering. To address this, we plan to explore adjustments in the threshold value and closely monitor how the linkage method responds to these changes.

Conversely, a promising combination is observed when utilizing the average linkage method with a threshold set to 0.2. This configuration, as illustrated in Fig. 4.7, produces a reasonable number of clusters, ensuring a more even distribution of elements across these clusters.

In this clustering method, we obtain a reasonable number of clusters (142) with each cluster having a minimum of 2 elements and a maximum of 274. This outcome is considered quite satisfactory, particularly for an initial threshold selection.

The acceptable Threshold for each linkage method and the results of clustering can be found in Tables 4.5, 4.6, 4.7, and 4.8.

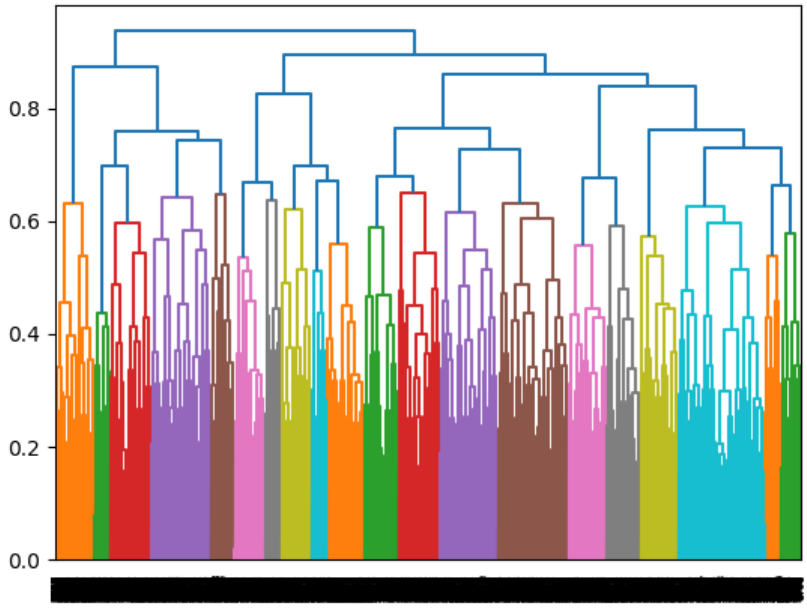


Figure 4.3: Complete Dendrogram

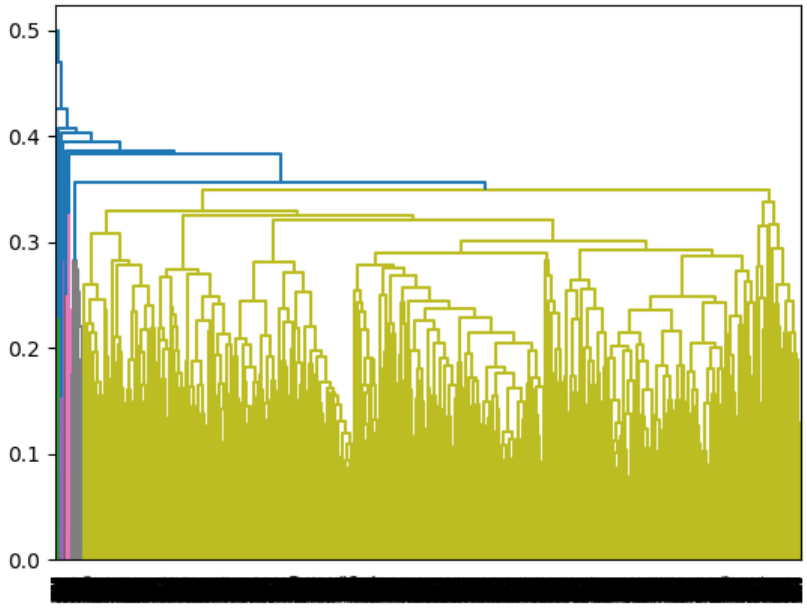


Figure 4.4: Average Dendrogram

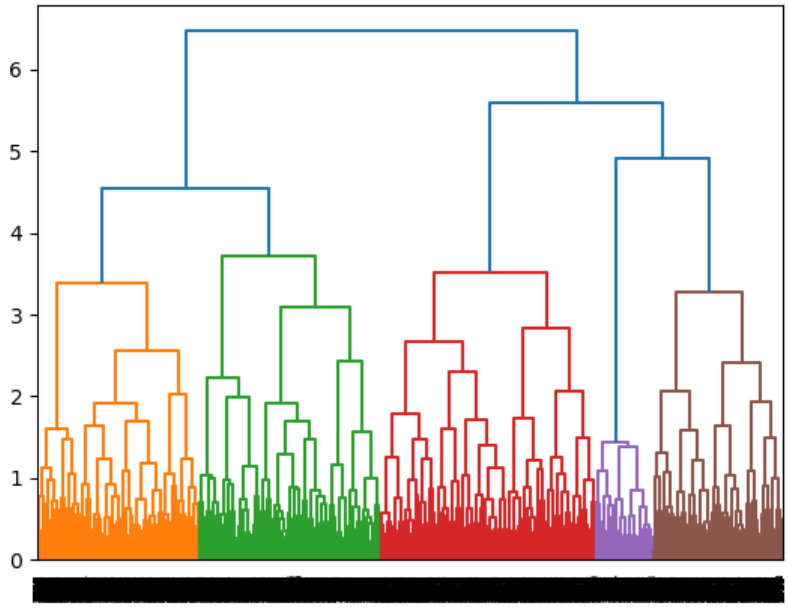


Figure 4.5: Ward Dendrogram

```
Method: single, Dendrogram saved as: h_cluster/single_dendrogram.pngMethod: single, Threshold: 0.1, Number of clusters: 50
Cluster Sizes:
Cluster 1 has 2 elements: [464, 2537]
Cluster 2 has 2 elements: [465, 2538]
Cluster 3 has 2 elements: [546, 2695]
Cluster 4 has 2 elements: [242, 360]
Cluster 5 has 2 elements: [493, 2585]
Cluster 6 has 1 elements: [232]
Cluster 7 has 1 elements: [1998]
Cluster 8 has 3 elements: [972, 3391, 3392]
Cluster 9 has 2 elements: [434, 2474]
Cluster 10 has 4 elements: [346, 2208, 2209, 2210]
Cluster 11 has 2 elements: [1839, 1840]
Cluster 12 has 2 elements: [1903, 1904]
Cluster 13 has 2 elements: [1052, 3484]
Cluster 14 has 3 elements: [148, 1896, 1897]
Cluster 15 has 2 elements: [793, 3182]
Cluster 16 has 5 elements: [1481, 3843, 3844, 3845, 3846]
Cluster 17 has 2 elements: [433, 2473]
Cluster 18 has 3 elements: [2021, 2241, 2242]
Cluster 19 has 2 elements: [1091, 3517]
Cluster 20 has 3 elements: [1007, 3432, 3433]
Cluster 21 has 3 elements: [879, 2682, 3288]
Cluster 22 has 4126 elements: [1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 21, 22, 24, 25, 27, 28, 29,
30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91,
92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117,
118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142,
143, 144, 145, 146, 147, 149, 150, 151, 152, 153, 155, 157, 158, 159, 161, 162, 163, 165, 166, 167, 168, 169, 170, 171, 172,
```

Figure 4.6: Single, Threshold: 0.1

```
Method: average, Threshold: 0.2, Number of clusters: 142
Cluster Sizes:
Cluster 1 has 2 elements: [242, 360]
Cluster 2 has 2 elements: [256, 3901]
Cluster 3 has 1 elements: [1968]
Cluster 4 has 10 elements: [1186, 1187, 1189, 1190, 1216, 3618, 3619, 3621, 3632, 4086]
Cluster 5 has 15 elements: [1526, 1527, 1529, 1532, 1534, 1559, 1663, 3898, 3899, 3955, 3956, 3957, 4151, 4152,
4153]
Cluster 6 has 3 elements: [258, 1052, 3484]
Cluster 7 has 2 elements: [232, 1998]
Cluster 8 has 12 elements: [20, 492, 509, 1510, 2416, 2417, 2584, 2627, 3069, 3070, 3551, 3885]
Cluster 9 has 2 elements: [433, 2473]
Cluster 10 has 6 elements: [491, 493, 1178, 2582, 2583, 2585]
Cluster 11 has 11 elements: [153, 1084, 1642, 3510, 4083, 4104, 4105, 4106, 4112, 4113, 4114]
Cluster 12 has 1 elements: [23]
Cluster 13 has 4 elements: [464, 465, 2537, 2538]
Cluster 14 has 5 elements: [1046, 1181, 1197, 3478, 3613]
Cluster 15 has 5 elements: [1193, 1194, 1220, 1221, 3623]
Cluster 16 has 10 elements: [1199, 1205, 1213, 1472, 1475, 3624, 3627, 3631, 3835, 3838]
Cluster 17 has 19 elements: [231, 241, 344, 347, 470, 472, 1994, 1995, 1996, 1997, 2018, 2019, 2020, 2146, 2204,
2211, 2212, 2543, 2545]
```

Figure 4.7: Average, Threshold: 0.2

Threshold	Number of Clusters	Cluster with Max Elements
0.01	3375	8
0.02	2639	14
0.03	2142	37
0.04	1584	544
0.05	1049	1718
0.06	596	2903
0.07	335	3488
0.08	173	3859
0.09	95	4029

Table 4.5: Cluster Statistics, Single

Threshold	Number of Clusters	Cluster with Max Elements
0.31	206	103
0.32	187	103
0.33	175	103
0.34	159	103
0.35	146	103
0.36	141	103
0.37	132	103
0.38	123	103
0.39	113	103

Table 4.6: Cluster Statistics, Complete

Threshold	Number of Clusters	Cluster with Max Elements
0.71	125	88
0.72	124	88
0.73	120	88
0.74	114	89
0.75	113	89
0.76	109	89
0.77	107	89
0.78	103	89
0.79	102	89

Table 4.7: Cluster Statistics, Ward

Threshold	Number of Clusters	Cluster with Max Elements
0.11	703	55
0.12	604	55
0.13	504	81
0.14	432	100
0.15	366	116
0.16	300	130
0.17	249	206
0.18	212	206
0.19	176	265

Table 4.8: Cluster Statistics, Average

Transition ID	Cosine Similarity
(2498, 2500)	0.9981
(2715, 2718)	0.9981
(2492, 2500)	0.998
(2714, 2716)	**0.998**
(2714, 2718)	0.998
(2715, 2719)	0.998
(2565, 2567)	0.9979
(2708, 2709)	0.9979
(2709, 2710)	0.9979
(3169, 3170)	0.9979
(46, 1742)	0.9978
(784, 3168)	0.9978
(1235, 3640)	0.9978
(2493, 2496)	0.9978
(2628, 2630)	0.9978

Table 4.9: Highest Cosine Similarities between Transitions

4.6.3 Transition Similarities

To enhance our comprehension of how transitions are grouped together, we provide a set of cosine similarity values for different transitions found within Table 4.9. This table consists of two columns: the left column enumerates specific transitions with numerical identifiers, and the right column displays the calculated cosine similarity scores between these transitions. These cosine similarity values serve as a quantitative measure of how closely related or similar these transitions are. After performing these calculations, we sort the list of transition pairs in descending order, allowing us to identify and focus on the transitions with the highest recorded cosine similarities. These highest recorded cosine similarities play a pivotal role in revealing the most significant relationships or connections between transitions within the figure. This analytical process is invaluable for gaining deeper insights into the underlying patterns and associations among the transitions, and it has practical applications in various domains, including data analysis, natural language processing, and network analysis.

For example, in 'Table 4.10,' we can observe that transitions 2714 and 2716 are highlighted with a remarkably high cosine similarity score of 0.998. As previously explained in Section 3.15, we sometimes require 5 discrete values for certain analyses and experiments. This necessity is why you see 5 discrete values in this table. This indicates that these two transitions share a particularly strong resemblance or connection, as the cosine similarity score approaches 1, signifying almost identical behavior between them. This specific case underscores the significance of using cosine similarity to uncover and emphasize the most closely related transitions within the dataset.

Table 4.10: Transitions 2714 and 2716

Transition Number	**Fifth Commit Before BIC**	**Fourth Commit Before BIC**	**Third Commit Before BIC**	**Second Commit Before BIC**	**First Commit Before BIC**	**Bug Inducing Commit**
2714	0443	4441	4440	0334	3112	0333
2716	0443	4441	4440	0443	4112	0332

4.7 Cluster Representation

As outlined earlier, each transition includes six states, each of which corresponds to a commit. Within each state, four metrics are evaluated with values ranging from 1 to 3. As a result, each transition is represented by 24 characters, all of which are grouped into a single cluster. For better understanding, let's consider two transitions within a hypothetical cluster, consisting of only four states each:

$$\text{t1:}\mathbf{1}\mathit{33} - \mathbf{1122} - \mathbf{1}\mathit{2}\mathbf{33} - \mathbf{111}\mathit{1}$$

$$\text{t2:}\mathbf{12}\mathit{3}\mathbf{1} - \mathbf{1122} - \mathbf{1}\mathit{3}\mathbf{33} - \mathbf{111}\mathit{2}$$

Should these transitions be clustered together, we're presented with a variety of metric choices that differ between the transitions, depicted as:

$$\text{T:1[23]3[13]} - 1122 - \text{1[23]33} - \text{111[12]}$$

Subsequently, we proceed with filtering. Each basket of metrics undergoes Shannon Entropy calculation, and identifying the metric value with the highest frequency. Subsequently, we retain only the 20th percentile of unique entropy values, discarding the rest due to their limited informational value. Among these selected baskets, we preserve those with a majority representation of at least 60 percent. For instance, a basket like [112233] exhibits a majority of only 33%, rendering it unsuitable for determining a representative metric value. Conversely, a basket such as [1111111123] holds a majority representation of 80%, making it eligible for retention if its entropy falls within the first 20th percentile among unique entropy values.

Please note that when calculating Shannon entropy and determining the majority for each set of options, we examined the frequency of each transition rather than just considering distinct transitions. For instance, if we consider three hypothetical transitions, each comprising only two commits as illustrated below:

$$t1 : 321\mathbf{2} - 33\mathbf{3}2$$

$$t2 : 321\mathbf{2} - 33\mathbf{2}2$$

$$t3 : 321\mathbf{1} - 33\mathbf{2}2$$

The representation would be as follows:

$$T : 321[122] - 33[223]2$$

And if we base the cluster representation solely on distinct values, it would be as follows:

$$T : 321\mathbf{2} - 33\mathbf{2}2$$

However, if $t1$ occurs 100 times and $t2$ and $t3$ occur only once each, the influence of $t1$ on determining the majority is significantly greater than the other two. Therefore, the correct representation should consider the frequency of occurrences, as shown:

$$T : 321\mathbf{2} - 33\mathbf{3}2$$

This approach accurately reflects the transitions' impact on the overall cluster and aligns with their true influence within the dataset.

A significant aspect to note is that we do not completely remove all transitions that do not meet our desired thresholds for majority and Shannon entropy. Instead, we carefully analyze each distinct cluster that falls below the target entropy threshold and scrutinize the transitions within these clusters. If there is a unique transition within the clusters that possesses a frequency equal to or greater than that of the smallest accepted cluster, we treat it as a separate group. This approach is intended to minimize data loss and preserve valuable information, ensuring a more nuanced and comprehensive analysis of the data.

4.8 Creating the Commit State Transition Model

By treating each state as a distinct node and every transition as a vector in a directed graph, our objective was to construct a network that represents the model we've developed. This network was intended to provide deeper insights into the relationships among states and potentially uncover underlying patterns. However, it's worth noting that our database is exceptionally extensive, and this ambitious network quickly became unwieldy due to its sheer size, rendering it nearly obsolete. Despite the challenges posed by the monumental scale of the database, we continued to explore alternative approaches and methods to extract valuable insights.

To enhance comprehension, the final state transition model, before applying the clustering techniques, is visually represented in Figure 4.8.

To understand the network better, you can see a fraction of it in Fig 4.9

Within each node, the state of the commit is denoted by a four-digit numerical identifier. Additionally, the numerical value on each vector signifies the frequency with which each specific transition has occurred.

Using the clustering techniques on the transitions is essential for comprehension, as is evident.

Fig. 4.10 shows the resultant network after clustering techniques have been applied and layers have been created based on the number of commits prior to the bug inducing commit.

The image is based on an interface we created, where you could enlarge to view the precise states along with their frequency.

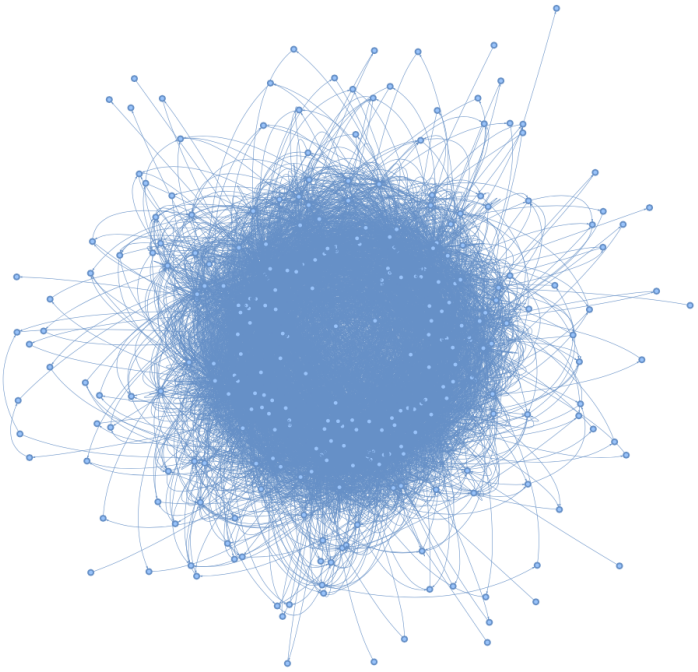


Figure 4.8: Transition Network

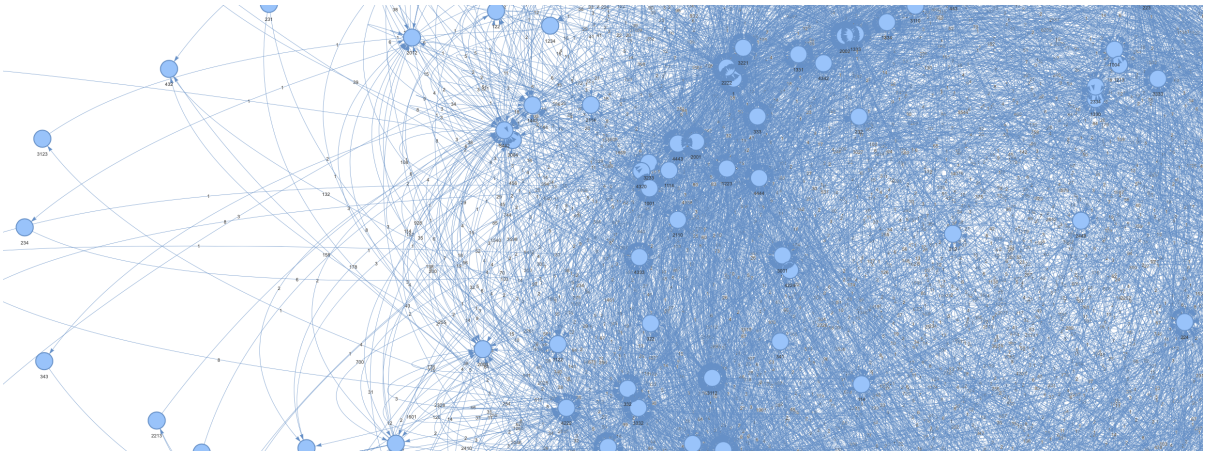


Figure 4.9: Transition Network Slice

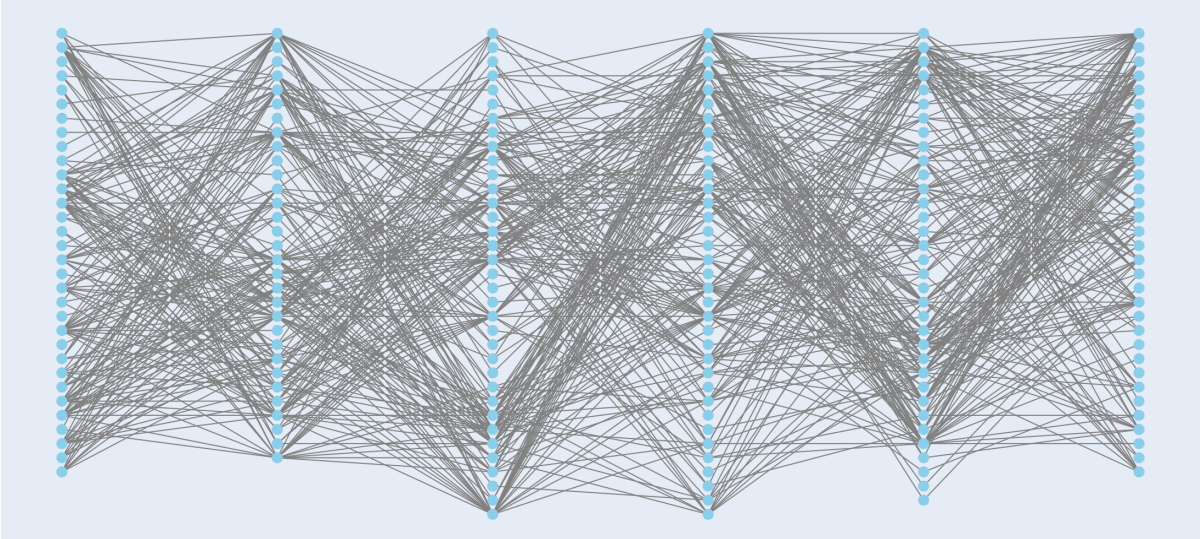


Figure 4.10: Transition Graph

4.9 Calculating State Transition Probabilities

In the upcoming phase of our research, our objective is to assess the likelihood of each state culminating in a commit that introduces a bug. To achieve this, we will systematically analyze our database for each transition, identifying comparable transitions that result in bug-inducing commits, as well as those that do not. This thorough examination will provide valuable insights into the factors contributing to bug occurrences within our system. To achieve this, we employed a methodology involving the calculation of probabilities associated with transitions to a bug-inducing commit. For instance, considering the following pattern:

$$C1 \rightarrow C2 \rightarrow C3 \rightarrow C4 \rightarrow C5 \rightarrow BIC$$

We computed the probability of transitioning from $C1 \rightarrow C2$ leading to a bug-inducing commit by analyzing occurrences in our database relative to all cases. Subsequently, we repeated this process for each subsequent commit in the sequence. This process will furnish us with the likelihood of a bug-inducing commit occurring, given a particular pattern. For our future endeavors, we'll also compute the probability of a pattern, given a bug-inducing commit, as it's essential for our analysis.

4.9.1 Conditional Probability

The conditional probability can be calculated using the formula:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

To gain valuable insights into the most probable paths that could lead to a bug-inducing commit (BIC), it is crucial to analyze and understand conditional probabilities in the context of our data. Using Bayes' Theorem, we can derive the following equation:

$$P(pattern_{C2\to C3}|pattern_{C1\to C2}) = \frac{P(pattern_{C1\to C2}|pattern_{C2\to C3})P(pattern_{C2\to C3})}{P(pattern_{C1\to C2})} \tag{4.1}$$

Refer to Section 4.3 for the definition of patterns.

In this context, $pattern_{C1\to C2}$ represents the vector that results from concatenating the state representations of commit C_1 and commit C_2 associated with the edge $C1 \to C2$, and $pattern_{C2\to C3}$ represents the value linked to the edge $C2 \to C3$.

This allows us to express that:

$$P(pattern_{C2\to C3} \cap pattern_{C1\to C2}) = P(pattern_{C2\to C3}|pattern_{C1\to C2}) \cdot P(pattern_{C1\to C2})$$

We aim to calculate the probability for each transition:

$$P(\text{BIC} \mid \text{pattern}_{Ci\to Cj})$$

This is the probability that a transition will lead to a bug-inducing commit (BIC) given a specific pattern pattern_x. By analyzing the probabilities for various combinations of patterns such as $P(\text{BIC} \mid \text{pattern}_1)$, $P(\text{BIC} \mid \text{pattern}_1 \cap \text{pattern}_2)$, and so on, up to $P(\text{BIC} \mid \text{pattern}_1 \cap \ldots \cap \text{pattern}_5)$, we can better understand the potential risks in the commit history.

By employing Bayes' Theorem and the conditional probability formula, in combination with known probabilities such as $P(\text{pattern}_x)$ and $P(\text{BIC})$, we can calculate the probability of each transition leading to a BIC. These calculations extend across all edges along the path $C1 \to C2 \to C3 \to C4 \to C5 \to \text{BIC}$.

To enhance understanding, the conditional probability $P(pattern_x|pattern_y)$ can be interpreted as the frequency of occurrence of pattern x given that the preceding pattern is y. This interpretation helps us understand the relationships between different patterns and their impacts on the probability of reaching a bug-inducing commit.

In these equations, C denotes a commit and BIC refers to a bug-inducing commit. Through these analyses, we can identify the most probable paths leading to a BIC and mitigate potential risks.

Chapter 5

Experimental Results

5.1 Preface

Following the successful creation of the final dataset and the Transition Graph, we conducted a round of analysis to gain insights into the data. In this section, we will present the results of this analysis. We conducted a series of distinct experiments to explore different aspects of patterns that lead to bug-inducing commits. In the initial set of experiments, we aimed to identify common patterns that result in a bug-inducing commit by splitting our data into different groups and comparing the common patterns between two groups. This approach is analogous to splitting data into a training set and a test set, allowing us to examine whether the test results align with the training data.

In the subsequent set of experiments, we focused on the states of commits just before the bug-inducing commit, rather than the entire transition. Our goal was to predict the upcoming state in the development process for the developer and assess the potential risk of leading to a bug-inducing commit.

Finally, we present the results of our transition clustering analysis, which involves examining our database for areas such as the frequency of feature changes during a specific number of transitions and how these metrics evolve over time. This analysis provides insights into the patterns and dynamics within the development process, enabling us to better understand how certain transitions might lead to bugs and offering developers guidance on how to avoid them.

5.2 Common Patterns Experiments

In this section, we will elaborate on three distinct methods employed to experimentally assess the accuracy of our model.

5.2.1 Top-Ten most probable transitions

The first experiment aimed to identify the top ten most likely patterns leading to a bug-inducing commit across the entire dataset. To achieve this, we employed a methodology involving the calculation of probabilities associated with transitions to a bug-inducing commit.

To ensure that our calculations are not merely capturing a limited subset of existing data in these experiments, we examine the least likely patterns to lead to a bug-inducing commit, in addition to calculating the most likely patterns. If the number of common patterns in the bottom list is equal to or greater than the number of common patterns in the top list, it suggests that our experiment may be just a sample and not indicative of a predictive model.

However, if this is not the case, and as the probability of leading to a bug-inducing commit increases, the transitions converge toward specific patterns, then our experiment is meaningful and potentially provides valuable insights into identifying bug-inducing commits.

Individual Project-Based

Following the identification of the top-ten likeliest paths leading to a bug-inducing commit across the whole data, our next objective was to evaluate the degree of overlap in these patterns across each project.

The findings, presented in the table 5.1, illustrate the frequency of occurrence of these patterns within each project. Additionally, we provide the total number of commits and the count of unique transitions observed within each project.

Project	Common Top-Patterns	Common Bottom-Patterns	Number of Commits	Unique Transitions
1	0	1	263	23
2	1	2	3971	31
3	0	1	3296	30
4	2	1	5347	49
5	3	3	22082	86
6	1	1	8379	61
7	5	0	51317	133
8	5	0	46203	134
9	4	1	63684	159
10	7	0	124231	204
11	5	0	155169	225
12	6	0	119831	197
13	7	0	117637	198
14	6	0	245168	278
15	8	0	354572	301
16	8	0	211221	275
17	6	0	169422	254

Table 5.1: Common Top-10 Most and Least Probable Patterns, Total Number of Commits, and Unique Transitions Across Projects.

The ”Common Top-Patterns” column indicates the frequency of the top ten most likely patterns for each project, while the ”Common Bottom-Patterns” column shows how often the top ten least likely patterns occurred in each project. The ”Total Number of Commits” column reflects the total number of commits in each project, and the ”Unique Transitions” column reports the number of distinct patterns observed in each project.

To better understand Table 5.1, let’s take a look at project 11, which has 225 distinct transitions (a transition pattern occurring multiple times is counted as one). This project includes 155,169 commits. We analyzed all the commit state transitions in project 11 and calculated

the likelihood of each transition leading to a bug-inducing commit as described in Section 4.9. We then identified the top 10 most probable patterns that lead to a bug-inducing commit for project 11 and compared them to the top 10 patterns across all other projects combined. This comparison revealed that there are 5 common patterns between the lists. The transitions in Project 11 that are least likely to result in a bug-inducing commit are extracted in the following step, and we repeat this process for all the other projects put together. To determine if there are any common patterns, then we compare the final ten items in each list. As Table 5.1 illustrates, there are none. This demonstrates how a particular set of patterns is reached by the top of our list.

This comprehensive table provides insights into the occurrence of bug-inducing commit patterns across different projects, alongside their respective commit and transition characteristics.

As is apparent, in projects with a greater number of commits and unique transitions, it's logical to observe an increase in the number of matches among the top-ten likeliest paths leading to a bug-inducing commit.

Below are the top-ten transitions most likely to lead to a bug-inducing commit:

- 3112-3313-1321-1123-3113
- 1213-1131-1323-3113-3311
- 1131-3112-1133-3211-1133
- 1133-1131-1311-3133-1121
- 1321-1131-1133-1121-1133
- 3112-3131-1121-1131-3113
- 1131-1133-3112-3113-1131
- 1133-3133-1133-3111-1133
- 3111-1133-1133-1131-3111
- 3113-1133-1131-1131-3131

The following shows the intersection of these top-ten transitions with the top-ten transitions most likely to lead to a bug-inducing commit for each project:

- **Project 1:** None
- **Project 2:** 3112-3131-1121-1131-3113
- **Project 3:** None
- **Project 4:** 1131-3112-1133-3211-1133
1133-1131-1311-3133-1121

- **Project 5:**
  1133-1131-1311-3133-1121
  3112-3313-1321-1123-3113
  1131-1133-3112-3113-1131

- **Project 6:** 1133-3133-1133-3111-1133

- **Project 7:**
  1133-1131-1311-3133-1121
  1213-1131-1323-3113-3311
  3112-3313-1321-1123-3113
  3113-1133-1131-1131-3131
  1131-3112-1133-3211-1133

- **Project 8:**
  3112-3313-1321-1123-3113
  1133-1131-1311-3133-1121
  1133-3133-1133-3111-1133
  1213-1131-1323-3113-3311
  1131-3112-1133-3211-1133

- **Project 9:**
  1133-1131-1311-3133-1121
  1133-3133-1133-3111-1133
  3112-3313-1321-1123-3113
  1213-1131-1323-3113-3311

- **Project 10:**
  1133-3133-1133-3111-1133
  3111-1133-1133-1131-3111
  1133-1131-1311-3133-1121
  3112-3131-1121-1131-3113
  3113-1133-1131-1131-3131
  1213-1131-1323-3113-3311
  1131-1133-3112-3113-1131

- **Project 11:**
  1131-1133-3112-3113-1131
  1133-1131-1311-3133-1121
  3113-1133-1131-1131-3131
  3112-3313-1321-1123-3113
  1133-3133-1133-3111-1133

- **Project 12:**
  3112-3313-1321-1123-3113
  1133-1131-1311-3133-1121
  3112-3131-1121-1131-3113
  1131-1133-3112-3113-1131
  1133-3133-1133-3111-1133
  3111-1133-1133-1131-3111

- **Project 13:**
  1131-3112-1133-3211-1133
  1133-1131-1311-3133-1121
  1133-3133-1133-3111-1133
  3112-3131-1121-1131-3113
  3111-1133-1133-1131-3111
  3113-1133-1131-1131-3131
  3112-3313-1321-1123-3113

- **Project 14:**
  3112-3313-1321-1123-3113
  1213-1131-1323-3113-3311
  1133-1131-1311-3133-1121
  1321-1131-1133-1121-1133
  3112-3131-1121-1131-3113
  1133-3133-1133-3111-1133

- **Project 15:**
  3112-3313-1321-1123-3113
  1213-1131-1323-3113-3311
  1131-3112-1133-3211-1133
  1133-1131-1311-3133-1121
  1321-1131-1133-1121-1133
  3112-3131-1121-1131-3113
  1131-1133-3112-3113-1131
  3111-1133-1133-1131-3111

- **Project 16:**
  3112-3313-1321-1123-3113
  1213-1131-1323-3113-3311
  1131-3112-1133-3211-1133
  3111-1133-1133-1131-3111
  1321-1131-1133-1121-1133
  3112-3131-1121-1131-3113
  1131-1133-3112-3113-1131
  3113-1133-1131-1131-3131

- **Project 17:**
  1133-1131-1311-3133-1121
  1321-1131-1133-1121-1133
  3112-3131-1121-1131-3113
  1131-1133-3112-3113-1131
  1133-3133-1133-3111-1133
  3111-1133-1133-1131-3111

The results clearly demonstrate that as the scope of our project expands, the patterns most likely to lead to bug-inducing commits converge into a distinct set of patterns. By excluding one project and then calculating the top ten patterns most likely to lead to bug-inducing commits, we can compare this list with another list that includes the said project. This comparison supports our hypothesis. However, this convergence does not hold true for the patterns least likely to lead to bug-inducing commits, indicating that our experiments are capturing more than just a subset of a particular dataset.

Project Magnitude-Based

We categorize projects into different groups based on their magnitude, determined by the number of commits within each project. This classification allows us to conduct the same experiment across various categories. The magnitude of each project is represented by the number of commits, as illustrated in the graph depicted in Fig 5.1.

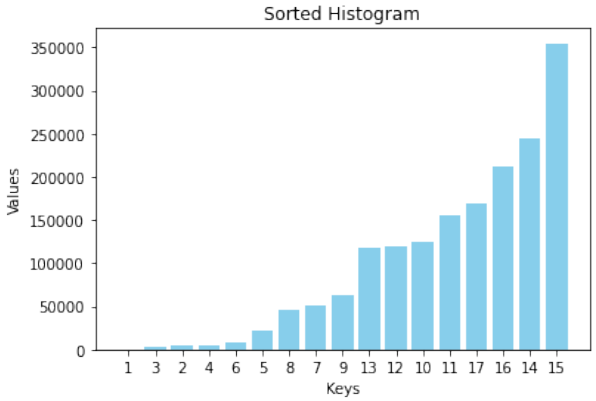


Figure 5.1: Number of commits in each project

Based on the first method of classification, where projects are grouped into 'small,' 'medium,' and 'large' categories based on the number of commits and percentile classification, the results for the occurrences of the top-ten patterns are as follows:

Project Size	Project Numbers	Common Top-Patterns	Common Bottom-Patterns	Unique Transitions
Small	1, 2, 3, 4, 5, 6	4	1	154
Medium	7, 8, 9, 10, 12, 13	7	1	271
Large	11, 14, 15, 16, 17	9	0	303

Table 5.2: Classification and Results of Top-Ten Occurrences

Please keep in mind that this Table, is calculated with considering all the projects while testing one group, if we want to exclude the group we testing while calculating the top and bottom ten, the results will be as below;

Project Size	Project Numbers	Common Top-Patterns	Common Bottom-Patterns	Unique Transitions
Small	1, 2, 3, 4, 5, 6	3	0	154
Medium	7, 8, 9, 10, 12, 13	7	0	271
Large	11, 14, 15, 16, 17	8	0	303

Table 5.3: Classification and Results of Top-Ten Occurrences

Another approach to classify projects into categories was based on their magnitude. Under this method, projects with fewer than 10,000 commits were assigned to group 1, those with between 10,000 and 100,000 commits to group 2, projects with between 100,000 and 200,000 commits to group 3, and projects with more than 200,000 commits to group 4.

Group	Project Numbers	Common Top-Patterns	Common Bottom-Patterns	Unique Transitions
1	1, 2, 3, 4, 6	3	2	137
2	5, 7, 8, 9	4	0	201
3	10, 11, 12, 13, 17	6	1	249
4	14, 15, 16	9	0	302

Table 5.4: Classification and Results of Top-Ten Occurrences based on Magnitude

Now with excluding the group in test from our bulk of data in other category, we will reach this table:

Group	Project Numbers	Common Top-Patterns	Common Bottom-Patterns	Unique Transitions
1	1, 2, 3, 4, 6	2	1	137
2	5, 7, 8, 9	4	0	201
3	10, 11, 12, 13, 17	6	0	249
4	14, 15, 16	8	0	302

Table 5.5: Classification and Results of Top-Ten Occurrences based on Magnitude

5.2.2 80-20 split

To further evaluate our model's performance, we employed a method of data partitioning. Our approach involved dividing the dataset into two distinct groups.

Initially, we randomly sampled 80% of the entire dataset and conducted the analysis to determine the top-ten likeliest transitions leading to bug-inducing commits, as per our previous methodology. Subsequently, we repeated the same process for the remaining 20% of the data.

Following this, we established a fixed number of random samples and calculated the number of matches between the two groups. These matches indicated the consistency of identified patterns across the dataset partitions. To quantify our model's accuracy, we computed the average of these match counts.

The results of our evaluation provide insights into the effectiveness and robustness of our model across different data partitions. By assessing the consistency of identified patterns between the two groups, we gain a better understanding of the reliability of our model in predicting bug-inducing commits.

After applying this technique to 100 random samples and tallying the overlaps in the top-ten of the 80 percent and 20 percent, then calculating the average number of overlaps, our model attained an accuracy of 86.1%. This implies that 8.61 out of 10 likeliest paths leading to a bug-inducing commit can be predicted by our model.

Similarly, we also aim to calculate the number of common patterns that are *least* likely to lead to a bug-inducing commit across the 20% and 80% splits of random samples. This approach helps us ensure that our pattern list converges to specific transitions, rather than just reflecting a subset of the same data.

The accuracy of these common patterns across 100 random samples was 1.6%, indicating that, on average, 0.16 patterns were common between the two lists after 100 tests with random samples.

5.2.3 Project-Based

In the previous experiment, we combined data from all projects. However, now we want to evaluate how our model performs within the scope of each individual project. We will follow the same process as before, but this time, the dataset will be restricted to the transitions of each project separately. Since the projects in the ”Small” category don't have enough commits to split into an 80-20 ratio, we will exclude them from this experiment. We will proceed with only the projects in the ”Medium” and ”Large” categories. The results of this experiment are presented in Table 5.6.

Project Number	Number of Commits	Average of the Top	Average of the Bottom
7	51,317	3.3	0.96
8	46,213	3.05	1.01
9	63,684	3.61	0.72
10	124,231	4.55	0.68
11	155,169	4.7	0.63
12	119,831	4.61	0.69
13	117,637	4.03	0.67
14	245,168	5.49	0.49
15	354,572	6.23	0.32
16	211,221	5.28	0.54
17	169,422	4.93	0.61

Table 5.6: Accuracy of projects 7 to 17

The ”Average of the Top” column represents the average occurrence of the top ten most probable patterns that lead to a bug-inducing commit, based on an analysis of 100 random samples. In contrast, the ”Average of the Bottom” column reflects the average occurrence of the top ten least probable patterns that lead to a bug-inducing commit, also based on 100 random samples.

To better understand the data in Table 5.6, let's use project 11 as an example. We randomly sample 20% of the transitions in this project and calculate the top ten most likely patterns to lead to a bug-inducing commit, as well as the top ten least likely patterns to lead to a bug-inducing commit, within both the 20% and 80% fractions of our transitions. We then compare the results from these two groups. This sampling process is repeated 100 times, and the average of the common patterns for both lists is calculated and reported. According to Table 5.6, in project 11, the average of common patterns for the least likely patterns to lead to a bug-inducing commit is 0.63, while the average of common patterns for the most likely patterns to lead to a bug-inducing commit is 4.7, based on 100 random samples.

5.2.4 Conclusion

After conducting these experiments, we conclude that our model is an effective predictive tool for identifying the most likely patterns that lead to a bug-inducing commit. By assessing the

probability that each transition will result in a bug-inducing commit and ranking them based on this probability, we find that the model consistently converges to a specific set of patterns.

These patterns can be recognized and analyzed by developers, allowing them to assess the impact of each metric and take preventative measures to avoid transitions that could lead to bugs. By dividing our dataset into different portions and testing this theory across multiple scopes—either within a single project or across several projects—we can validate the model's effectiveness.

This approach demonstrates that, in any new set of data, there is a high probability that these patterns will lead to a bug-inducing commit in most cases. By consistently identifying and focusing on these patterns, developers can take proactive measures to improve software quality and reduce the likelihood of introducing bugs.

5.3 Statistical Experiments

To enhance our understanding of the model, we conducted statistical analyses focusing on the states within transitions. We treated each state as a focal point and systematically counted the subsequent states in all transitions to identify transition patterns. Sorting the results allows us to obtain a more comprehensive view of our analysis scope and highlights the most frequently occurring subsequent states. Reporting the percentage of each state leading to a specific outcome, rather than absolute values, provides a more meaningful perspective. Additionally, analyzing the percentage of states linked to bug-inducing commits offers further insights.

Given the importance of analyzing feature changes, we categorized states into five discrete levels: Very Low, Low, Medium, High, and Very High.

This analysis clarifies the frequent sequences and transitions involving specific states, offering valuable insights into the model's behavior and dynamics.

For instance, we extracted the most frequent transition following each state in our transition database. To avoid excessive detail, we excluded transitions with a frequency below 100 and a percentage under 35%. The results are presented in Table 5.7.

In this table, the "source state" represents the state of the initial commit, while the "target state" refers to the state of the subsequent commit that follows the initial one. The "frequency" column indicates the number of times a transition occurs from the source state to the target state in our database.

The "percentage" column provides the proportion of transitions from the source state to the target state out of all transitions from the source state to other states. This value helps to understand the likelihood of moving from a given source state to a specific target state.

The "percentage of bug-inducing" column quantifies the proportion of transitions from the source state to the target state that result in a bug-inducing commit. This means that the target state is associated with a bug-inducing commit.

This model can serve as a predictive tool for anticipating future states that might culminate in a bug-inducing commit. In essence, it enables us to infer that if certain alterations occur in the selected features, there exists a probability of X that developers will encounter a bug in subsequent commits.

This table represents only a fraction of our complete database, but we have similar data for all states. It helps developers understand the current state of their commit based on specific

Source State	Target State	Frequency	Percentage (%)	Percentage of Bug-Inducing (%)
1221	3001	2351	42.46	2.51
0332	2001	3598	38.71	0.17
0442	3002	2063	56.06	0.05
4221	3001	1572	37.9	3.18
0333	1002	4974	46.77	0.06
3223	1112	3361	62.36	0.03
1443	2112	789	67.96	0.38
1003	2112	309	44.02	0.32
0222	2112	4204	36.84	0.26
3000	2112	595	53.18	14.62
4442	3112	3924	36.8	0.33
0331	2002	4989	40.52	16.0
0221	2112	4202	45.84	2.62
2012	2002	1129	53.74	8.06
4111	3111	375	60.68	22.4
4233	4441	286	60.6	1.75

Table 5.7: Data with source and target states, frequency, percentage, and percentage of bug-inducing

features and identify the most likely future state for each metric. Additionally, it indicates how likely a developer is to encounter a new bug during the development process. This insight is crucial for managing potential risks and making strategic decisions.

To improve comprehension of Table 5.7, let's assume the developer is currently in state 3000. This means their most recent commit has the following attributes:

- **NEW_SQALE_DEBT_RATION**: High
- **VIOLATIONS**: Very Low
- **CODE_SMELLS**: Very Low
- **CLASS_COMPLEXITY**: Very Low

Based on our findings, the next commit is most likely to move to state 2112, with a probability of 53.18% which entails:

- **NEW_SQALE_DEBT_RATION**: Medium
- **VIOLATIONS**: Low
- **CODE_SMELLS**: Low
- **CLASS_COMPLEXITY**: Medium

Statistically, this transition carries a 14.62% chance of resulting in a bug-inducing commit. To summarize the expected changes for each metric:

- **NEW_SQALE_DEBT_RATION**: Decrease one level ↓
- **VIOLATIONS**: Increase one level ↑
- **CODE_SMELLS**: Increase one level ↑
- **CLASS_COMPLEXITY**: Increase two levels ↑

As a simpler example, consider the transition from state 4111 to 3111 as depicted in Table 5.7. In this transition, the only change is in the first metric, NEW_SQALE_DEBT_RATION, which decreases from Very High to High (one level down ↓). If a developer encounters this state, they should be aware that there is a 60.68% probability of transitioning to state 3111. Additionally, this transition carries a 22.4% chance of resulting in a bug, making it a risky transition that should be avoided.

5.3.1 Similar States

Consider an alternative analytical approach: categorizing states based on a similarity threshold. This involves grouping states that share similarities above a certain level. Instead of closely tracking transitions between individual states, the method quantifies transitions from a specific state with a similarity level of alpha from the source state. This streamlines the analysis by simplifying comparisons.

This approach provides a broader view of the data, focusing on transitions from states with a given similarity level rather than individual state transitions. By condensing the analysis, it reveals trends and patterns more efficiently and conclusively. This method is particularly useful for large datasets, reducing computational demands and offering a more comprehensive understanding of the system's behavior and tendencies.

In Table 5.8, the transitions between a specific source state and target states with equal cosine similarity to the source state are displayed. This table presents the same information as Table 5.7, but instead of using fixed target states, targets are grouped based on their cosine similarity to the source state.

For example, if the current state is 1221, the next commit will have a cosine similarity of 0.4 with the source state with a likelihood of 43.44%. In this case, there is a 4.68% chance of encountering a bug in the next commit.

Like Table 5.7, we set a threshold to exclude frequencies below 100 and percentages lower than 30 in Table 5.8 to avoid excessive detail, but the data is available for all the source and target states.

5.4 Additional Experiments

5.4.1 Metrics Toggle

Another noteworthy and useful result is the nubmer of changes for each metric in a number of transitions, in Tab 5.9, the times each metric in the commit state changes in a specific number of transitions is shown.

Source State	Similarity with Target	Frequency	Percentage (%)	Percentage of Bug-Inducing (%)
1221	0.4	3986	43.44	4.68
0332	0.1907	5868	39.51	0.22
0442	0.1849	2548	57.9	0.1
4221	0.8222	2409	39.13	5.89
0003	0.417	964	35.53	3.18
0333	0.5164	6238	47.21	0.1
3223	0.9636	6690	63.66	0.05
1443	0.7807	1155	71.71	0.74
1003	0.8	415	50.76	0.52
0222	0.7303	7430	37.23	0.45
3000	0.6325	1029	55.01	25.35
4442	0.8593	4749	37.36	0.41
0331	0.1622	9928	40.96	21.3
4222	0.8386	12377	35.65	19.53
0221	0.6325	7663	46.13	3.65
2012	0.9428	2046	54.28	11.91
4111	0.9934	522	65.34	33.85
4233	0.9038	343	62.72	2.15

Table 5.8: Summary of transitions with source state, similarity with target, frequency, percentage, and percentage of bug-inducing.

Metric	Number of Changes
CLASS_COMPLEXITY	20374
NEW_SQALE_DEBT_RATION	24018
CODE_SMELLS	21619
VIOLATIONS	21576

Table 5.9: Metric Changes in a slice of transitions

In Table 5.9, we can observe that all the metrics associated with a specific number of transitions exhibit equal changes. There isn't one particular metric that stands out by changing significantly more than the others. This observation is indicative of the fact that our choice of metrics for defining states is well-suited to the task.

The uniformity in the changes across all metrics suggests that each metric effectively captures the nuances and characteristics of the transitions being studied. This balance in metric behavior underscores our sound selection process in designing the criteria for defining states. It implies that no single metric is disproportionately influencing the state definition, ensuring a comprehensive and well-rounded approach to the analysis.

This consistency in the behavior of metrics not only validates our choice but also enhances the reliability of our results. It indicates that the chosen metrics collectively provide a robust foundation for characterizing and understanding transitions without introducing bias or skew. Therefore, our metrics are working harmoniously and are appropriately attuned to the specific requirements of our analysis, contributing to the overall validity and effectiveness of our research.

Chapter 6

Conclusion and Future Works

6.1 Preface

In this chapter, we will summarize the key findings and conclusions of our research, offering a comprehensive overview of the insights and outcomes we have achieved throughout the study. We will discuss how our results contribute to the existing body of knowledge in the field and highlight the implications of our findings for practice and future research.

Furthermore, we will suggest potential areas for further investigation and development, building on the groundwork laid by our research. These ideas for future work may include exploring new methodologies, extending the scope of our study to different contexts or data sets, or refining our approach to improve accuracy and applicability.

By outlining possible directions for future research, we aim to inspire further inquiry and encourage other researchers to build upon our work, advancing the field and enhancing our understanding of the topic.

6.2 Conclusion

Based on the information provided, it's evident that our model, built upon an analysis of top-ten likeliest paths leading to bug-inducing commits across multiple projects, demonstrates promising predictive capabilities. By leveraging techniques such as conditional probability and overlap analysis, we've established a robust methodology for identifying patterns indicative of potential bugs.

Our findings reveal a correlation between project activity, measured by the number of commits and unique transitions, and the frequency of matches among the top-ten paths. Projects with higher activity levels tend to exhibit more matches, indicating a logical relationship between project dynamics and bug occurrence.

Moreover, through rigorous testing involving 100 random samples, our model achieved an impressive accuracy rate of 86.1%. This signifies that the model successfully predicts over 8 out of 10 likeliest paths leading to bug-inducing commits, underscoring its efficacy in bug prediction.

In conclusion, our approach offers valuable insights into bug prediction, enabling developers to proactively identify potential issues and mitigate risks during the software development

lifecycle. Continued refinement and validation of our model hold the potential to further enhance its predictive capabilities and contribute to more robust software development practices.

6.3 Comparison

This thesis introduces a novel approach to predicting bug-inducing commits by employing a blend of source code and process metrics to capture the state of each commit transition. Unlike traditional methods, which often rely on static analysis at specific points in the software development life cycle, this methodology dynamically tracks changes across commits to predict potential vulnerabilities.

The advantages of this method include:

- **Dynamic Analysis:** By analyzing commit transitions rather than static code snapshots, the model captures the evolutionary nature of code changes, providing a more contextual prediction.
- **Comprehensive Metrics Utilization:** It leverages a broad range of metrics, including those derived from SonarQube Technical Debt data, enabling a deeper insight into the quality of code over time.
- **Advanced Classification Techniques:** The use of sophisticated clustering and classification techniques, such as K-Means and Hierarchical clustering, enhances the model's ability to identify subtle patterns that may lead to defects.

These innovative aspects not only enhance the accuracy of bug prediction but also contribute to more informed decision-making during software development processes. By integrating these techniques, the research provides actionable insights into improving code quality and reducing the frequency of bug-inducing changes.

Most current studies, including Hammouri et al. [37], employ machine learning techniques for bug prediction. While these methods tend to achieve higher accuracy, they also present several challenges and issues that must be addressed. Bug prediction using software metrics offers several advantages including simplicity and interpretability, as it uses straightforward statistical techniques that are easily understandable by project managers and developers without deep machine learning knowledge. This approach can be quickly implemented since it requires fewer computational resources and uses direct metrics like lines of code and complexity measures from the codebase without needing complex model training. Software metrics also facilitate historical and comparative analysis, allowing teams to track how metric changes correlate with bug occurrences and identify potential risk factors. Additionally, the low resource requirement of this method makes it ideal for organizations with limited technical infrastructure, enabling efficient bug prediction with minimal overhead.

In the study by Gupta et al. [32], Object-Oriented metrics were employed to predict software bugs, achieving a prediction accuracy of 76.27%. In contrast, our framework significantly enhances this performance, surpassing their results by a notable margin of 10 percent. This improvement in accuracy is achieved through a sophisticated integration of both source code and process metrics, which provide a more dynamic and comprehensive analysis of the commit

states. Our approach not only refines the prediction accuracy but also extends the capability to anticipate bug-inducing commits more effectively. This advance is crucial for the development of more reliable software systems and supports proactive debugging practices that can save substantial resources and time in software development projects.

Our methodology presents significant improvements over conventional strategies discussed in [33], which predominantly utilizes static software metrics for bug prediction. The cited study highlights the utility of established metrics such as DIT, WMC, CBO, and LoC in open-source projects. In contrast, our document proposes a more dynamic and advanced model that monitors changes across commits by integrating both code and process metrics. This strategy not only enriches the dataset for a more thorough analysis but also incorporates sophisticated clustering and classification techniques, which could enhance predictive accuracy by up to 10%. Moreover, our approach supports real-time application in active projects, empowering software teams to quickly address potential bug-inducing modifications, which ultimately reduces development time and costs while boosting software quality and reliability.

6.4 Risks to Validity and Limitations

This section delineates the potential risks to validity and the limitations of our study, which may impact the reliability and applicability of the results.

6.4.1 Internal Validity

The internal validity of our study might be compromised by the potential for feature selection bias. While the selection of eight specific metrics was rigorously conducted, the exclusion of other variables might limit the model's ability to fully capture the predictors of bug-inducing commits. Moreover, the performance of the predictive model is evaluated on data from a set of projects that may not fully represent all types of software development contexts, possibly affecting the model's accuracy and leading to overfitting.

6.4.2 External Validity

The generalizability of our findings is a concern, as the study primarily utilizes data from open-source projects. These projects often differ significantly from commercial software in terms of code complexity, development practices, and team dynamics. Thus, the results may not directly extend to other contexts without further validation, particularly in environments that differ markedly from the open-source ecosystem.

6.4.3 Construct Validity

The construct validity could be at risk due to the operationalization of the metrics. The metrics used to predict bug-inducing commits, such as code complexity, might not comprehensively reflect all aspects of code quality and developer activity. Additionally, the interpretation and measurement of these metrics could vary, affecting the consistency and reliability of the data across different studies.

6.4.4 Reliability Validity

Reliability validity concerns the consistency of the measurement process. Our methodology relies on automated tools and scripts to extract and process metrics, which might introduce errors if these tools fail to correctly interpret the codebase's nuances. Furthermore, any changes in the tools or data extraction methods over time could lead to inconsistencies in the data collected for longitudinal studies.

6.4.5 Conclusion Validity

Our conclusions are subject to validity risks related to the statistical methods employed. The modeling techniques and assumptions made during the analysis may not hold across different datasets or configurations, which could lead to erroneous interpretations of the data. The limited diversity in project samples might also affect the robustness of statistical inferences, potentially leading to misleading conclusions about the effectiveness of the metrics.

Given these limitations and validity concerns, our study's findings should be viewed as preliminary. They underscore the need for further research involving a diverse range of software projects and more robust statistical methodologies to confirm the utility of the identified metrics in predicting bug-inducing commits. Future work should also explore the integration of additional variables and alternative analytical techniques to enhance the predictive power and applicability of the models.

6.5 Future Work

The current study has laid the foundation for an in-depth analysis of commit transitions and their implications within software development projects. While we have made significant strides in understanding patterns, relationships, and anomalies, there remain several avenues for future research and exploration.

1. **Temporal Analysis**: To enhance our understanding of the temporal aspects of software development, future work could focus on investigating how commit transitions evolve over time. Analyzing transitions on a timeline can reveal dynamic trends and shifts in development practices.

2. **Machine Learning Integration**: Incorporating machine learning techniques for predictive analysis could be a promising direction. By leveraging historical data, machine learning models can provide insights into the likelihood of a transition leading to a bug-inducing commit, thereby supporting proactive quality assurance efforts.

3. **Large-Scale Dataset Handling**: Expanding the scope of this research to encompass even larger datasets presents a challenge that merits further exploration. Developing efficient methods for handling and analyzing monumental datasets can enable more comprehensive insights.

4. **Diverse Software Contexts**: Applying the methodologies and findings from this study to a broader range of software contexts, such as open-source projects, proprietary

software, and various programming languages, can yield a more comprehensive understanding of software development dynamics.

5. **Visualization Tools**: The creation of interactive visualization tools can aid software developers and project managers in exploring commit transition patterns. Such tools could facilitate the identification of critical transitions and their implications.

6. **Commit Transition Prediction**: Investigating the feasibility of predicting commit transitions that are likely to lead to bug-inducing commits could be a valuable direction. Predictive models can provide proactive recommendations to developers, potentially preventing defects early in the development process.

7. **Human Factors and Code Review**: Exploring the role of code reviews and human factors in commit transitions and their impact on software quality is another avenue for future research. Understanding the influence of human actions can help improve development processes.

8. **Cross-Domain Insights**: Extending the research to examine the relationships between commit transitions and software metrics in different domains, such as web development, mobile applications, and embedded systems, can provide cross-domain insights into best practices.

In summary, this study serves as a foundational exploration of commit transitions within software development. Future research can build upon these findings to deepen our understanding of software quality, development practices, and the dynamic nature of commit transitions. By addressing these areas, we can contribute to more effective software development and quality assurance processes.

Bibliography

[1] Vaibhav Aggarwal, Vaibhav Gupta, Prayag Singh, Kiran Sharma, and Neetu Sharma. Detection of spatial outlier by using improved z-score test. In *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, pages 788–790. IEEE, 2019.

[2] Allan J Albrecht. Measuring application development productivity. In *Proc. joint share, guide, and ibm application development symposium*, pages 83–92, 1979.

[3] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links: bugs and bug-fix commits. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 97–106, 2010.

[4] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on software engineering*, 28(1):4–17, 2002.

[5] Vic Barnett, Toby Lewis, et al. *Outliers in statistical data*, volume 3. Wiley New York, 1994.

[6] Hassan Raza Bukhari, Rafia Mumtaz, Salman Inayat, Uferah Shafi, Ihsan Ul Haq, Syed Mohammad Hassan Zaidi, and Maryam Hafeez. Assessing the impact of segmentation on wheat stripe rust disease classification using computer vision and deep learning. *IEEE Access*, 9:164986–165004, 2021.

[7] Cagatay Catal and Banu Diri. Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. *Information Sciences*, 179(8):1040–1058, 2009.

[8] Raymond B Cattell. The description of personality: Basic traits resolved into clusters. *The journal of abnormal and social psychology*, 38(4):476, 1943.

[9] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):1–58, 2009.

[10] Lianping Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE software*, 32(2):50–54, 2015.

[11] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[12] Frederick Emory Croxton. Applied general statistics. In *Applied general statistics*, pages 754–754. 1967.

[13] Karel Dejaeger, Thomas Verbraken, and Bart Baesens. Toward comprehensible software fault prediction models using bayesian network classifiers. *IEEE Transactions on Software Engineering*, 39(2):237–257, 2012.

[14] Frederik Michel Dekking. *A Modern Introduction to Probability and Statistics: Understanding why and how*. Springer Science & Business Media, 2005.

[15] Cornelius Frank Dietrich. *Uncertainty, calibration and probability: the statistics of scientific and industrial measurement*. Routledge, 2017.

[16] Andrew P Dillon. *A study of the Toyota production system: From an Industrial Engineering Viewpoint*. Routledge, 2019.

[17] Yadolah Dodge. *The Oxford dictionary of statistical terms*. OUP Oxford, 2003.

[18] Harold Edson Driver and Alfred Louis Kroeber. *Quantitative expression of cultural relationships*, volume 31. Berkeley: University of California Press, 1932.

[19] JJ Dziak, Runze Li, and AT Wagner. Weighted tvem sas macro users' guide (version 2.6). *University Park, PA: The Methodology Center, Penn State*, 2017.

[20] Karim O Elish and Mahmoud O Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649–660, 2008.

[21] Mahmoud O Elish, Hamoud Aljamaan, and Irfan Ahmad. Three empirical studies on predicting software maintainability using ensemble methods. *Soft Computing*, 19:2511–2524, 2015.

[22] Ezgi Erturk and Ebru Akcapinar Sezer. A comparison of some soft computing methods for software fault prediction. *Expert systems with applications*, 42(4):1872–1879, 2015.

[23] Brian S Everitt, Sabine Landau, and Morven Leese. Cluster analysis arnold. *A member of the Hodder Headline Group, London*, pages 429–438, 2001.

[24] Guisheng Fan, Xuyang Diao, Huiqun Yu, Kang Yang, Liqiong Chen, et al. Software defect prediction via attention-based recurrent neural network. *Scientific Programming*, 2019, 2019.

[25] Donald Firesmith. Four types of shift left testing. *podcast, Software Engineering Institute website, September*, 2015.

[26] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering and beyond: trends and challenges. In *Proceedings of the 1st International Workshop on rapid continuous software engineering*, pages 1–9, 2014.

[27] Santi Garcia-Vallvé and PERE Puigbo. Dendroupgma: a dendrogram construction utility. *Universitat Rovira i Virgili*, pages 1–14, 2009.

[28] Yossi Gil and Gal Lalouche. On the correlation between size and metric validity. *Empirical Software Engineering*, 22(5):2585–2611, 2017.

[29] David Gray, David Bowes, Neil Davey, Yi Sun, and Bruce Christianson. Using the support vector machine as a classification method for software defect prediction with static code metrics. In *Engineering Applications of Neural Networks: 11th International Conference, EANN 2009, London, UK, August 27-29, 2009. Proceedings 11*, pages 223–234. Springer, 2009.

[30] Eilam Gross. Practical statistics for high energy physics. *CERN Yellow Reports: School Proceedings*, 3:199–199, 2018.

[31] John R Grout and Brian T Downs. A brief tutorial on mistake-proofing, poka-yoke, and zqc. *Lean Business Solutions UnitedStates*, 2009.

[32] Dharmendra Lal Gupta and Kavita Saxena. Software bug prediction using object-oriented metrics. *Sādhanā*, 42:655–669, 2017.

[33] Varuna Gupta, N Ganeshan, and Tarun K Singhal. Developing software bug prediction models using various software metrics as the bug indicators. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 6(2), 2015.

[34] Allan Gut and Allan Gut. *Probability: a graduate course*, volume 200. Springer, 2006.

[35] Tibor Gyimóthy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software engineering*, 31(10):897–910, 2005.

[36] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2011.

[37] Awni Hammouri, Mustafa Hammad, Mohammad Alnabhan, and Fatima Alsarayrah. Software bug prediction using machine learning approach. *International journal of advanced computer science and applications*, 9(2), 2018.

[38] Douglas M Hawkins. *Identification of outliers*, volume 11. Springer, 1980.

[39] Peng He, Bing Li, Xiao Liu, Jun Chen, and Yutao Ma. An empirical study on software defect prediction with a simplified metric set. *Information and Software Technology*, 59:170–190, 2015.

[40] Peng Huang and Jie Zhu. Predicting defect-prone software modules at different logical levels. In *2009 International Conference on Research Challenges in Computer Science*, pages 37–40. IEEE, 2009.

[41] Jez Humble, Chris Read, and Dan North. The deployment production line. In *AGILE 2006 (AGILE'06)*, pages 6–pp. IEEE, 2006.

[42] Ahmed Abd Elhalim Ibrahim, Amr Kamel, and Hesham Hassan. Object oriented metrics and quality attributes: A survey. In *Proceedings of the 10th International Conference on Informatics and Systems*, pages 312–319, 2016.

[43] James Joyce. Bayes' theorem. 2003.

[44] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101, 2014.

[45] Y. Kamei and E. Shihab. Defect prediction: Accomplishments and future challenges. 2016.

[46] Taghi M Khoshgoftaar and Naeem Seliya. Tree-based software quality estimation models for fault prediction. In *Proceedings Eighth IEEE Symposium on Software Metrics*, pages 203–214. IEEE, 2002.

[47] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. Predicting faults from cached history. In *29th International Conference on Software Engineering (ICSE'07)*, pages 489–498. IEEE, 2007.

[48] Hans-Peter Kriegel, Erich Schubert, and Arthur Zimek. The (black) art of runtime evaluation: Are we comparing algorithms or implementations? *Knowledge and Information Systems*, 52:341–378, 2017.

[49] Lov Kumar and Santanu Ku Rath. Software maintainability prediction using hybrid neural network and fuzzy logic approach with parallel computing concept. *International Journal of System Assurance Engineering and Management*, 8:1487–1502, 2017.

[50] Pierre Legendre, Louis Legendre, et al. Numerical ecology: developments in environmental modelling. *Developments in Environmental Modelling*, 20(1), 1998.

[51] Valentina Lenarduzzi, Nyyti Saarimaki, and Davide Taibi. The technical debt dataset. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE'19, page 2–11. Association for Computing Machinery, 2019.

[52] Panagiotis Louridas. Static code analysis. *Ieee Software*, 23(4):58–61, 2006.

[53] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[54] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33(1):2–13, 2006.

[55] Everald E Mills. Metrics in the software engineering curriculum. *Annals of Software Engineering*, 6(1):181–200, 1998.

[56] Osamu Mizuno, Shiro Ikami, Shuya Nakaichi, and Tohru Kikuno. Spam filter based approach for finding fault-prone software modules. In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, pages 4–4. IEEE, 2007.

[57] Julie Moeyersoms, Enric Junque de Fortuny, Karel Dejaeger, Bart Baesens, and David Martens. Comprehensible software fault and effort prediction: A data mining approach. *Journal of Systems and Software*, 100:80–90, 2015.

[58] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461, 2006.

[59] J. Nam and S. Kim. Heterogeneous defect prediction. page 508–519, 2015.

[60] Frank Nielsen and Frank Nielsen. Hierarchical clustering. *Introduction to HPC with MPI for Data Science*, pages 195–211, 2016.

[61] Ahmet Okutan and Olcay Taner Yıldız. Software defect prediction using bayesian networks. *Empirical Software Engineering*, 19:154–181, 2014.

[62] Mauricio J Ordonez and Hisham M Haddad. The state of metrics in software industry. In *Fifth International Conference on Information Technology: New Generations (Itng 2008)*, pages 453–458. IEEE, 2008.

[63] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. Programmer-based fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, pages 1–10, 2010.

[64] Ajeet Kumar Pandey and Neeraj Kumar Goyal. Test effort optimization by prediction and ranking of fault-prone software modules. In *2010 2nd International Conference on Reliability, Safety and Hazard-Risk-Based Technologies and Physics-of-Failure Methods (ICRESH)*, pages 136–142. IEEE, 2010.

[65] Pooja Paramshetti and DA Phalke. Survey on software defect prediction using machine learning techniques. *International Journal of Science and Research*, 3(12):1394–1397, 2014.

[66] Parul Parul. *Evaluating the Likelihood of Bug Inducing Commits Using Metrics Trend Analysis*. PhD thesis, The University of Western Ontario (Canada), 2023.

[67] Parul Parul, Kostas Kontogiannis, and Chris Brealey. Prediction of bug inducing commits using metrics trend analysis. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 830–839, 2023.

[68] Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., 1984.

[69] Sandeep Purao and Vijay Vaishnavi. Product metrics for object-oriented systems. *ACM Computing Surveys (CSUR)*, 35(2):191–221, 2003.

[70] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 432–441. IEEE, 2013.

[71] V Shobha Rani, A Ramesh Babu, K Deepthi, and Vallem Ranadheer Reddy. Shift-left testing in devops: A study of benefits, challenges, and best practices. In *2023 2nd International Conference on Automation, Computing and Renewable Systems (ICACRS)*, pages 1675–1680. IEEE, 2023.

[72] Harry Robinson. Using poka-yoke techniques for early defect detection. In *Sixth International Conference on Software Testing Analysis and Review*, pages 134–145, 1997.

[73] Daniele Romano and Martin Pinzger. Using source code metrics to predict change-prone java interfaces. In *2011 27th IEEE international conference on software maintenance (ICSM)*, pages 303–312. IEEE, 2011.

[74] Charles Romesburg. *Cluster analysis for researchers*. Lulu. com, 2004.

[75] SOKAL RR. A statiscal method for evaluating systematic relationships. *Univ Kans sci bull*, 38:1409–1438, 1958.

[76] Robert Sabourin and Mónica Wodzislawski. Teaching testing to programmers. what sticks, and what slides off? a journey from teflon to velcro. 2020.

[77] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 18–27, 2006.

[78] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press Cambridge, 2008.

[79] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE access*, 5:3909–3943, 2017.

[80] A. Shahrokhi and R. Feldt. A systematic review of software robustness. *Elsevier Information and Software Technology*, 55(1):1–17, 2013.

[81] Claude Elwood Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[82] Gurdev Singh, Dilbag Singh, and Vikram Singh. A study of software metrics. *IJCEM International Journal of Computational Engineering & Management*, 11(2011):22–27, 2011.

[83] Satwinder Singh and Karanjeet Singh Kahlon. Effectiveness of encapsulation and object-oriented metrics to refactor code and identify error prone classes using bad smells. *ACM SIGSOFT Software Engineering Notes*, 36(5):1–10, 2011.

[84] Yogesh Singh, Arvinder Kaur, and Ruchika Malhotra. Empirical validation of object-oriented metrics for predicting fault proneness models. *Software quality journal*, 18:3–35, 2010.

[85] Amit Singhal et al. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.

[86] Karl Smith. *Precalculus: A functional approach to graphing and problem solving*. Jones & Bartlett Publishers, 2013.

[87] Thorvald Sorensen. A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on danish commons. *Biologiske skrifter*, 5:1–34, 1948.

[88] Biljana Stanic. Static code metrics vs. process metrics for software fault prediction using bayesian network learners, 2015.

[89] J. H. Steiger. Tests for comparing elements of a correlation matrix. *Psychological Bulletin*, 87(2):245–251, 1980.

[90] Richard Winship Stewart. *American military history*, volume 2. Center of Military History, US Army, 2005.

[91] Alan Stuart and Keith Ord. Kendall's advanced theory of statistics, distribution theory (volume 1), 1994.

[92] Matthias Studer, Gilbert Ritschard, Alexis Gabadinho, and Nicolas S Müller. Discrepancy analysis of state sequences. *Sociological methods & research*, 40(3):471–510, 2011.

[93] Gabor J Szekely, Maria L Rizzo, et al. Hierarchical clustering via joint between-within distances: Extending ward's minimum variance method. *Journal of classification*, 22(2):151–184, 2005.

[94] Ameet Talwalkar. Mehryar mohri afshin rostamizadeh.

[95] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to data mining*. Pearson Education India, 2016.

[96] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, Akinori Ihara, and Kenichi Matsumoto. The impact of mislabelling on the performance and interpretation of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 812–823. IEEE, 2015.

[97] Mie Mie Thet Thwin and Tong-Seng Quah. Application of neural networks for software quality prediction using object-oriented metrics. *Journal of systems and software*, 76(2):147–156, 2005.

[98] Fadel Toure, Mourad Badri, and Luc Lamontagne. Predicting different levels of the unit testing effort of classes using source code metrics: a multiple case study on open-source software. *Innovations in Systems and Software Engineering*, 14:15–46, 2018.

[99] Robert Choate Tryon. Cluster analysis: correlation profile and orthometric (factor) analysis for the isolation of unities in mind and personality. edwards brother, incorporated. *Ann Arbor*, 1939.

[100] R. S. Wahono. A systematic literature review of software defect prediction: Research trends, datasets, methods and frameworks. *Journal of Software Engineering*, 1(1), 2015.

[101] Xiang Wan, Wenqian Wang, Jiming Liu, and Tiejun Tong. Estimating the sample mean and standard deviation from the sample size, median, range and/or interquartile range. *BMC medical research methodology*, 14:1–13, 2014.

[102] Joe H Ward Jr. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association*, 58(301):236–244, 1963.

[103] Michael J Way, Jeffrey D Scargle, Kamal M Ali, and Ashok N Srivastava. *Advances in machine learning and data mining for astronomy*. CRC Press, 2012.

[104] Leland Wilkinson and Michael Friendly. The history of the cluster heat map. *The American Statistician*, 63(2):179–184, 2009.

[105] Xindong Wu, Vipin Kumar, J Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J McLachlan, Angus Ng, Bing Liu, Philip S Yu, et al. Top 10 algorithms in data mining. *Knowledge and information systems*, 14:1–37, 2008.

[106] Meng Yan, Xin Xia, Xiaohong Zhang, Ling Xu, Dan Yang, and Shanping Li. Software quality assessment model: a systematic mapping study. *Science China Information Sciences*, 62:1–18, 2019.

[107] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26. IEEE, 2015.

[108] Gokul Yenduri and Thippa Reddy Gadekallu. A systematic literature review of soft computing techniques for software maintainability prediction: State-of-the-art, challenges and future directions. *arXiv preprint arXiv:2209.10131*, 2022.

[109] Chubato Wondaferaw Yohannese and Tianrui Li. A combined-learning based framework for improved software fault prediction. *International Journal of Computational Intelligence Systems*, 10(1):647–662, 2017.

[110] Jun Zheng. Cost-sensitive boosting neural networks for software defect prediction. *Expert Systems with Applications*, 37(6):4537–4543, 2010.

[111] Joseph Zubin. A technique for measuring like-mindedness. *The Journal of Abnormal and Social Psychology*, 33(4):508, 1938.

Appendices

Defect

In software development and quality assurance, several terms—bug, error, fault, and defect—are used to describe various aspects of software imperfections. Understanding these terms and their relationships is crucial for effective software testing and debugging.

- **Bug:** A bug refers to an unintended flaw or fault in a software program that causes it to behave unexpectedly or produce incorrect results. Bugs can range from minor issues, such as typos in code, to critical problems that impact the functionality and performance of the software.
- **Error:** An error occurs when the actual behavior of a software program deviates from its expected behavior or specification. It represents a mistake or discrepancy that may result from human error, hardware failures, or bugs in the code.
- **Fault:** A fault is the root cause underlying an error or defect in software. It is an imperfection or flaw in the code or system design that leads to erroneous behavior when the software is executed under certain conditions.
- **Defect:** A defect is any deviation or imperfection in a software product that does not meet its requirements or specifications. It is a broader term encompassing bugs, errors, and faults. Defects can manifest as issues such as crashes, incorrect calculations, or unexpected behavior during software operation.

These terms are interconnected: a bug is a specific instance of unintended behavior caused by a defect, an error results from a fault or defect, and a defect represents the general term for any deviation from expected software behavior.

In practice, software developers and testers identify and resolve defects through various testing techniques, including unit testing, integration testing, and system testing. By addressing bugs, errors, and faults early in the development process, software teams can improve the reliability, performance, and overall quality of their products.

Poka-Yoke

Poka-yoke is a Japanese term that translates to ”mistake-proofing” or ”error prevention.” It is also known as a forcing function or behavior-shaping constraint. In essence, poka-yoke

is a strategy for designing processes and systems that prevent mistakes and defects by either avoiding errors altogether, correcting them, or making them immediately noticeable as they happen.

The goal of poka-yoke is to help operators of equipment and systems minimize the potential for human error and its associated risks. This approach can involve implementing safeguards such as automatic stops, alarms, and visual cues that alert the user to potential issues [72].

Shigeo Shingo, an industrial engineer, formalized the concept and introduced the term as part of the Toyota Production System. Poka-yoke has since become a fundamental aspect of lean manufacturing and quality control practices across various industries, emphasizing the importance of designing processes that are robust and resilient against errors [16] [31].

Linkage Method

Hierarchical clustering methods rely on measures of dissimilarity between sets of observations to decide cluster combinations (agglomerative) or cluster splits (divisive). This is typically accomplished using a distance metric, such as the Euclidean distance, which quantifies the dissimilarity between individual data points, and a linkage criterion that characterizes dissimilarity between sets based on the pairwise distances of their observations. The choice of both distance metric and linkage criterion significantly influences the clustering outcome, with the distance metric determining the similarity of objects and the linkage criterion shaping the cluster structures. For instance, complete-linkage often yields more spherical clusters, in contrast to single-linkage. Some common linkage methods are explained below [19] [93];

- Complete: an agglomerative hierarchical method where individual elements begin in separate clusters and are successively merged into larger ones until all elements are within a single cluster. It's also called farthest neighbor clustering, and the clustering results are depicted using a dendrogram, showcasing the merging sequence and distances [87] [50] [23].

$$\max_{a \in A, b \in B} d(a, b) \tag{1}$$

- Single: a hierarchical method, combines the clusters containing the nearest pair of elements in a step-by-step, bottom-up approach. It often produces elongated clusters, where nearby elements are close, but elements at the cluster ends may be distant from each other [23]. Widely used in astronomy for analyzing galaxy clusters, especially those with extended matter strings, it's known as the friends-of-friends algorithm [103].

$$\min_{a \in A, b \in B} d(a, b) \tag{2}$$

- Average: a straightforward hierarchical clustering method, following a bottom-up approach. It has a weighted version called WPGMA, often credited to Sokal and Michener [75]. The term "unweighted" signifies that all distances equally contribute to each computed average, while the simple averaging in WPGMA produces weighted results and

the proportional averaging in UPGMA results in an unweighted outcome [27].

$$\frac{1}{|A| \cdot |B|} \sum_{a \in A} \sum_{b \in B} d(a, b) \tag{3}$$

- Ward: a versatile criterion for hierarchical cluster analysis, where clusters are merged based on an objective function, initially presented by Joe H. Ward, Jr [102]. One of its known applications is the "Ward's minimum variance method," often using the error sum of squares as the objective function. The nearest-neighbor chain algorithm can efficiently reproduce the clusters defined by Ward's method.

$$\frac{|A| \cdot |B|}{|A \cup B|} \|\mu_A - \mu_B\|^2 = \sum_{x \in A \cup B} \|x - \mu_{A \cup B}\|^2 - \sum_{x \in A} \|x - \mu_A\|^2 - \sum_{x \in B} \|x - \mu_B\|^2 \tag{4}$$

Euclidean distance

In mathematics, the Euclidean distance, also known as the Pythagorean distance, measures the length of a line segment between two points in Euclidean space. It's calculated using the Pythagorean theorem based on the Cartesian coordinates of the points. The names come from the ancient Greek mathematicians Euclid and Pythagoras, although they didn't represent distances as numbers, and the connection to distance calculation was made much later. When dealing with non-point objects, distance is typically defined as the shortest distance between any pair of points from the two objects. Various formulas exist for computing distances between different types of objects, like the distance from a point to a line. In advanced mathematics, the concept of distance has been extended to abstract metric spaces, and different distances beyond the Euclidean have been explored. In certain statistical and optimization applications, the square of the Euclidean distance is used instead of the distance itself.

The distance between any two points on the real line is the absolute value of the numerical difference of their coordinates, their absolute difference. Thus if p and q are two points on the real line, then the distance between them is given by [86]:

$$d(p, q) = |p - q| \tag{5}$$

Heuristic Algorithms

In mathematical optimization and computer science, a heuristic is a strategy devised to expedite problem-solving when conventional methods prove too sluggish in identifying an exact or even an approximate solution, or when these methods are unable to locate any precise solution within a search space. This acceleration is accomplished by prioritizing speed over qualities like optimality, completeness, accuracy, or precision, effectively trading them for a quicker route to a solution. Heuristics can be thought of as a form of shortcut.

Moreover, a heuristic function, often referred to simply as a heuristic, serves as a guide within search algorithms. It assigns rankings to alternatives at each branching juncture based on available information, helping determine the most promising path to pursue. For instance, it might offer an approximation of the exact solution, assisting in expediting the decision-making process [68].

Mixture Models

In statistics, a mixture model is a probabilistic framework used to describe the existence of subpopulations within a larger population. It doesn't necessitate that an observed dataset assigns each individual observation to a specific subpopulation; instead, it captures the idea that the data may be a combination of multiple underlying subpopulations.

K	=	number of mixture components
N	=	number of observations
$\theta_{i=1\dots K}$	=	parameter of distribution of observation associated with component i
$\phi_{i=1\dots K}$	=	mixture weight, i.e., prior probability of a particular component i
$\boldsymbol{\phi}$	=	K-dimensional vector composed of all the individual $\phi_{1\dots K}$; must sum to 1
$z_{i=1\dots N}$	=	component of observation i
$x_{i=1\dots N}$	=	observation i
$F(x\|\theta)$	=	probability distribution of an observation, parametrized on θ
$z_{i=1\dots N}$	$\sim$	Categorical($\boldsymbol{\phi}$)
$x_{i=1\dots N}\|z_{i=1\dots N}$	$\sim$	$F(\theta_{z_i})$

Figure .1: a basic parametric mixture model

Supervised Learning

Supervised learning (SL) is a machine learning approach in which a model is trained using input data, typically represented as a vector of predictor variables, and corresponding desired output values, often referred to as human-labeled supervisory signals. The training data is used to construct a function that can map new input data to the expected output values. In an ideal scenario, the algorithm should accurately predict output values for previously unseen instances. Achieving this relies on the learning algorithm's ability to generalize from the training data to new, unencountered situations in a manner consistent with its inductive bias. The statistical quality of this generalization process is quantified by the generalization error [94].

Rocchio Algorithm

The Rocchio algorithm is rooted in a relevance feedback approach commonly utilized in information retrieval systems, tracing its origins to the SMART Information Retrieval System developed between 1960 and 1964. Like numerous other retrieval systems, the Rocchio algorithm is constructed upon the vector space model. It operates under the assumption that users generally have a broad understanding of which documents are deemed relevant or irrelevant. Consequently, the user's initial search query is adjusted to incorporate a specific proportion of relevant and irrelevant documents, aiming to enhance the search engine's recall and, potentially, precision. The precise balance between relevant and irrelevant documents included in the query is determined by the weightings assigned to the variables a, b, and c, as detailed in the Algorithm section [78]. The formula and variable definitions for Rocchio relevance feedback are as follows:

$$\vec{Q}_m = a\vec{Q}_o + b\frac{1}{|D_r|}\sum_{\vec{D}_j \in D_r}\vec{D}_j - c\frac{1}{|D_{nr}|}\sum_{\vec{D}_k \in D_{nr}}\vec{D}_k \tag{6}$$

Otsuka–Ochiai coefficient

Within the field of biology, a related concept to cosine similarity exists called the Otsuka–Ochiai coefficient, named after Yanosuke Otsuka and Akira Ochiai, and is sometimes referred to as the Ochiai–Barkman or simply the Ochiai coefficient [74]. Its representation is as follows:

$$K = \frac{|A \cap B|}{\sqrt{|A| \times |B|}} \tag{7}$$

In this context, we have two sets, denoted as A and B, and $|A|$ represents the count of elements within set A. If these sets are represented as binary vectors, it becomes evident that the Otsuka–Ochiai coefficient aligns with the concept of cosine similarity.

Z-Score

In statistics, the standard score represents how many standard deviations a given raw score is either above or below the mean of the observed or measured data. Raw scores exceeding the mean are associated with positive standard scores, whereas those falling below the mean yield negative standard scores [30].

To calculate the standard score, one subtracts the population mean from an individual raw score and then divides the resulting difference by the population standard deviation. This process of converting a raw score into a standard score is known as standardization or normalization, although it's worth noting that "normalizing" can encompass various types of ratios.

Standard scores are often referred to as z-scores, and these terms are frequently used interchangeably, as demonstrated in this article. Additional equivalent terms include z-value, z-statistic, normal score, standardized variable, and "pull" in the context of high-energy physics.

Curriculum Vitae

Name:	Alireza Tavakkoli Barzoki
Post-Secondary Education and Degrees:	Sharif University of Technology Tehran, Iran 2017 - 2022 B.Sc. University of Western Ontario London, ON 2022 - 2024 M.Sc.
Related Work Experience:	Research and Teaching Assistant The University of Western Ontario 2022 - 2024
Portfolio:	AlirezaTavakkoli.io