2023-05-01

Analyzing Software Maintenance Through Machine Learning and Mining Software Repositories Approaches

Sayed Mohsin Reza
University of Texas at El Paso

ANALYZING SOFTWARE MAINTENANCE THROUGH MACHINE LEARNING AND MINING SOFTWARE REPOSITORIES APPROACHES

SAYED MOHSIN REZA

Doctoral Program in Computer Science

APPROVED:

Mahmud Shahriar Hossain, Ph.D., Chair

Yoonsik Cheon, Ph.D.

Hugo Gutierrez, Ph.D.

Stephen L. Crites, Jr., Ph.D.
Dean of the Graduate School

to my

MOTHER, FATHER and WIFE

with love

ANALYZING SOFTWARE MAINTENANCE THROUGH MACHINE LEARNING AND MINING SOFTWARE REPOSITORIES APPROACHES

by

SAYED MOHSIN REZA

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

May 2023

Acknowledgements

I express my deep gratitude to my advisor, Dr. Mahmud Shahriar Hossain, and my previous advisor, Dr. Omar Badreddin, for their unwavering support, motivation, and exceptional guidance throughout my Ph.D. journey. Their patience and wisdom have been invaluable in guiding me through the research and writing of this dissertation. I consider myself fortunate to have had such dedicated mentors who consistently pushed me to achieve my best.

I extend my sincere appreciation to my Ph.D. committee members, Dr. Yoonsik Cheon and Dr. Hugo Gutierrez, for their insightful comments and feedback. Their contributions have been instrumental in shaping this work and have helped me broaden my research perspectives. I am grateful for their dedication, time, and commitment to making this work a success.

I am also grateful to the professors and staff at the University of Texas at El Paso Computer Science Department for their hard work and dedication in providing me with the means to complete my degree and prepare for a career as a computer scientist. Additionally, I am grateful to my family, especially my parents, for their unwavering love, support, and encouragement throughout my academic journey. Their constant motivation has been a source of strength and inspiration for me.

Finally, I am incredibly grateful to my dear wife, Laila Noor, for her unwavering support and understanding during my Ph.D. Her continuing, loving support without complaint has been invaluable to me.

Abstract

The rapid growth of software systems demands meticulous planning and maintenance to accommodate the evolution of the code base over extended periods. Without maintenance, software systems will become more complex, low in quality, and hence unsustainable. Software engineers who perform maintenance often strive to optimize code quality or minimize code smells in a timely manner. Several techniques have been used to detect code quality or code smells as a part of software maintenance. Most of these techniques are based on heuristics, which create detection rules using a few metrics. These approaches have reasonable accuracy but do not work in cross-project evaluation. The recent efforts in devising automatic Machine Learning (ML) based quality or code smell detection techniques have achieved unsatisfactory results so far. Reasons include the use of a smaller dataset, fewer input features, within-project classification, or a lack of user-friendly tools for data collection.

This dissertation explores the use of modern techniques in Mining Software Repositories (MSR), identifying code smells, code quality, and issue labels using machine learning approaches. The mining process is optimized through the use of phase-by-phase caching and efficient data retrieval from open-source platforms. To identify code quality attributes, traditional machine-learning approaches were applied to a large set of metrics. For the identification of code smells, traditional ML, and neural network-based ML techniques were utilized. A deep learning-based ML technique is proposed to classify the issue labeling of reported issues.

The first contribution of this dissertation is the development of a novel mining tool for extracting software artifacts. The proposed tool, ModelMine, is capable of mining software repositories, issues, and files from open-source platforms. A synthesized dataset containing code quality, issue, and code smell data is created. The second contribution is the application of ML approaches to classify code quality attributes and the comparison of

their performance. The evaluation results indicate that Random Forest (RF) significantly improves accuracy without generating false negatives or false positives, which can result in false alarms in code quality classification. The third contribution is the investigation of unexpected side effects (code smells and technical debt) of software repositories. The analysis revealed that handwritten code quality is impacted by a higher level of technical debt and code smells. The results also show that the performance of neural network-based ML approaches is better than traditional ML approaches in classifying code smells. The fourth contribution of this research is the development of a deep learning approach for issue label classification. The result shows that the proposed approach outperforms classifying issue labels compared to existing research.

In conclusion, this dissertation presents a comprehensive study of the use of modern techniques in the MSR field and machine learning approaches to identify code quality, code smells, and issue labels. The results of this research have practical implications for software quality assurance and issue management. Also, it will provide a foundation for machine learning approaches in software maintenance activities. To further improve the performance of ML, I will incorporate a larger dataset into the ML models through the enhancement of the ModelMine tool. The future research plan includes developing a methodology using NLP techniques to extract insights from textual data associated with software code and investigating the use of VR-based data visualizations for software maintenance.

Table of Contents

Page

Signature Page i

Title Page iv

Acknowledgements v

Abstract vi

Table of Contents viii

List of Tables xiii

List of Figures xv

Chapter

1 Introduction 1

1.1 Background of the Research Problem 1

1.1.1 Mining Software Artifacts from Open Source Repositories 2

1.1.2 Machine Learning Approaches in Code Quality Classification 2

1.1.3 Machine Learning Approaches in Code Smell Detection 3

1.1.4 Machine Learning Approaches in Issue Label Identification 3

1.2 Motivation 4

1.3 Research Objective and Questions 4

1.4 Contributions 6

1.5 Significance of the Research 7

1.6 Dissertation Overview 8

1.6.1 Chapter 2: Literature Overview 9

1.6.2 Chapter 3: ModelMine: A Tool to Facilitate Mining Software Artifacts from Open Source Repositories 9

1.6.3 Chapter 4: Performance Analysis of Machine Learning Approaches in Software Code Quality Classification 10

1.6.4 Chapter 5: Evaluating the Accuracy of Machine Learning Algorithms for Code Smells Detection . . . 10
1.6.5 Chapter 6: Issue Label Identification: Towards A Machine Learning-based Approach . . . 11
2 Literature Review . . . 12
2.1 Overview of the Research Field . . . 12
2.2 Software Artifacts . . . 13
2.2.1 Tools in Mining Software Repositories . . . 13
2.2.2 Software Artifacts Mining . . . 14
2.2.3 Software Artifacts Analysis . . . 15
2.2.4 Software Artifacts Visualization . . . 16
2.2.5 Other Research on Software Artifacts . . . 17
2.3 Machine Learning in Code Quality Research . . . 17
2.3.1 Code Quality Features . . . 17
2.3.2 Machine Learning Approaches . . . 19
2.3.3 Code Quality Analysis Techniques . . . 20
2.3.4 Other Code Quality Research . . . 20
2.4 Machine Learning in Code Smell Research . . . 21
2.4.1 Traditional ML Approaches . . . 23
2.4.2 Neural Network-based ML Approaches . . . 24
2.5 Gaps in the Literature . . . 25
3 ModelMine: A Tool to Facilitate Mining Software Artifacts from Open Source Repositories . . . 27
3.1 Introduction . . . 28
3.2 ModelMine Architecture . . . 30
3.2.1 Indexing Phase . . . 30
3.2.2 Paging Phase . . . 30
3.2.3 Query Reduction Phase . . . 31

3.2.4 Querying Phase . . . 31
3.2.5 Data Representation . . . 31
3.2.6 Results Ranking . . . 31
3.3 ModelMine User Interface . . . 32
3.3.1 Model-based Repository Search . . . 32
3.3.2 Model-based Artifact Search . . . 33
3.3.3 Model-based Commit Search . . . 33
3.4 Evaluation . . . 34
3.5 Conclusion . . . 38
4 Performance Analysis of Machine Learning Approaches in Software Code Quality Classification . . . 40
4.1 Introduction . . . 41
4.2 Research Methodology . . . 42
4.2.1 Research Questions . . . 42
4.2.2 Proposed Research Framework . . . 43
4.2.3 Dataset Collection . . . 44
4.2.4 Dataset Cleaning & Analysis . . . 47
4.2.5 Machine Learning Classifiers & Evaluation Metrics . . . 48
4.3 Result and Discussion . . . 48
4.3.1 Correlation Results . . . 48
4.3.2 Performance Results . . . 49
4.4 Conclusion . . . 52
5 Evaluating the Accuracy of Machine Learning Algorithms for Code Smells Detection 53
5.1 Introduction . . . 54
5.2 Background . . . 55
5.3 Study Design . . . 57
5.3.1 Research Questions . . . 57
5.3.2 Code smell Characteristics of Handwritten Code in MDE Projects . 58

5.3.3 Code Smells in Recent Studies . . . 60
5.3.4 Data Collection for Code Smells in MDE Projects . . . 70
5.3.5 Machine Learning Approaches in Code Smell Detection . . . 75
5.4 Results . . . 75
5.4.1 Results Based on Code Smells (RQ1 & RQ2) . . . 75
5.4.2 Results Based on Technical Debt (RQ3) . . . 77
5.4.3 Code Smells Considered in Recent Studies (RQ4) . . . 78
5.4.4 ML Approaches Considered in Recent Studies (RQ5) . . . 80
5.4.5 Performance Comparison of ML Approaches (RQ6) . . . 84
5.5 Discussion & Analysis . . . 85
5.6 Threats to Validity . . . 88
5.6.1 Construct Validity . . . 89
5.6.2 External Validity . . . 90
5.7 Conclusion . . . 91
6 Issue Label Identification: Towards A Machine Learning-based Approach . . . 93
6.1 Introduction . . . 94
6.2 Study Design . . . 96
6.2.1 Research Questions . . . 96
6.2.2 G-Issue Architecture . . . 98
6.2.3 Proposed Deep Learning Approach . . . 99
6.2.4 Data Collection . . . 100
6.2.5 Terminology . . . 102
6.3 Results & Discussion . . . 102
6.3.1 Performance Evaluation . . . 102
6.3.2 Analysis of Issue Lifetime . . . 103
6.3.3 Evolution of Issues . . . 104
6.3.4 Performance of Proposed Deep Learning Approach . . . 107
6.4 Conclusion . . . 108

7 Conclusion 109
7.1 Contributions 109
7.2 Future Research Plan 110
References 112

Appendix

A CSIQ: A Synthesized Dataset of Software Artifacts 128
A.1 Dataset Overview 128
A.2 Significance of the Data 130
A.3 Data Description 130
A.4 Experimental Design, Materials, and Methods 137
A.5 Repository Links 139
A.6 Version links 140
B Machine Learning in Code Smell Detection 142
B.1 Data Sources Links 142
B.2 List of Questions for Data Collection 143
B.3 Dataset Used in the Primary Studies 144
B.4 Independent Variables Considered in the Primary Studies 145
B.5 List of Source Code Metrics and Definitions 148
Biosketch 149

List of Tables

3.1 Evaluation Metrics for Tasks 35
3.2 Performance Evaluation Results 36
3.3 Usability Study Questionnaires 37

4.1 Selected Repositories with Metadata Information 44
4.2 Source Code Metrics Used in this Study 46
4.3 Confusion Matrices of Classifiers for Predicting Software Complexity 50
4.4 Performance of Machine Learning Models 50

5.1 Selected Repositories Name, its Type, and URL 59
5.2 Basic Information of Subject Software Repositories 61
5.3 Query Strings Applied to Extract Articles 63
5.4 Data Sources and Search Results 64
5.5 Reviewed Articles in the Literature 65
5.6 Variables Description 72
5.7 File Search Criteria 73
5.8 Detected Types of Code Smells 73
5.9 Code Smells & Technical Debt Results 74
5.10 Selected Machine Learning Approaches 76
5.11 Code Smells and Their Frequencies Identified in the Primary Studies . . . 78
5.12 Machine Learning Algorithms Reported in Primary Studies 83
5.13 Accuracy Comparison of Traditional and Neural Network-based Machine Learning Approaches 85

6.1 Selected Repositories with Metadata Information 101
6.2 Performance Comparison of Different Tools 103

6.3 Statistics on Days it Takes to Solve an Issue 104
6.4 Performance of Proposed Approach Compared with Existing Research . . . 107

A.1 Dataset Specifications 129
A.2 Selected Repositories with Metadata Information 131
A.3 Version & Metadata Information of Selected Repositories 131
A.4 List of Source Code Metrics and Code Quality Attributes 133
A.5 List of Code Smells and Their Associated PMD Rule Names 138
A.6 PMD Command & Parameters to Extract Code Smells 139
A.7 Selected Repository Links 139
A.8 Version Link of Selected Repositories 140

B.1 Data Sources and Links 142
B.2 List of Questions & Metadata for Data Collection 143
B.3 Datasets Reported in the Primary Studies 144
B.4 Independent Variables Considered in the Primary Studies 145
B.5 Source code metric names along with their definitions 148

List of Figures

3.1 Architecture of ModelMine Tool 29
3.2 Model-based Repository Search 32
3.3 Model-based Artifact Search 33
3.4 Model-based Commit Search 34
3.5 Usability Study Results 38

4.1 Proposed Methodology 43
4.2 Complexity Distribution among Repositories 45
4.3 Relationship of Input Variables with Target Variable 45
4.4 Correlation among Source Code Metrics and Quality Attribute 49
4.5 Relative Performance of ML Classifiers 51
4.6 Relative FP and FN Rate of ML Classifiers 51

5.1 Repository Selection Process 58
5.2 Article Selection Process 62
5.3 Code Smells in MDE, DD & Non-DD Repositories 77
5.4 Technical Debt (TD) Result 78
5.5 Code smells Reported in the Primary Studies 81
5.6 Frequency of Machine Learning Models in Code Smells Detection 81
5.7 Frequency of Evaluation Metrics Used for ML Algorithms in Primary Studies 82
5.8 Average Code Smell Density Results 86
5.9 Average Technical Debt Density Results 87
5.10 Comparative Accuracy Analysis of Machine Learning Models 89

6.1 Example of Labels Attached to Issues for the Spring-framework Project in GitHub. 95

6.2 Architecture of G-Issue Tool 96
6.3 Search & Result Screenshot of G-Issue Tool 99
6.4 Classifier for Proposed Deep Learning Approach 100
6.5 Box Plot of Days it Takes to Solve Issues among Repositories 105
6.6 Evolution of Issue-related Artifacts Over Time among Repositories 106

Chapter 1

Introduction

This dissertation explores the intersection of mining software repositories and machine-learning approaches in software engineering. In recent years, there has been a significant increase in research using machine learning methods in software engineering. By leveraging software repositories, their versions, and commit histories, researchers can extract valuable information about software development activities to support studies in areas like cost estimation, testing, quality assurance, and more. This information has driven advancements in software engineering research, including code smell detection, code review automation, and software issue management. In this chapter, we introduce the research background, problems, motivation, contributions, and overview of the dissertation.

1.1 Background of the Research Problem

The rise of distributed software development has led to the need for effective management of technical artifacts such as code, commits, issues, and milestones. This has resulted in the development of source code management software like BitBucket, GitHub, and GitLab, which combine the development, security, and operation of the software in a single application. As software repositories contain critical information regarding software development processes, leveraging this information has become an essential area of research in advancing the field of software engineering. The implications of such research are far-reaching, spanning beyond code quality estimation, bug identification, code smell detection, and re-engineering activity prediction.

1.1.1 Mining Software Artifacts from Open Source Repositories

In recent years, the field of mining software repositories has experienced significant growth due to the valuable information that software repositories, their version and commit histories can provide regarding software development processes. However, existing mining tools have primarily focused on extracting data from code or commit history, leaving a gap in support for extracting non-textual artifacts, which limits the scope of extractable data.

Moreover, source code management systems are becoming popular and diversified developers contribute to community-based software development, the number of issues reported related to bugs, errors, and missing documentation continues to increase. Addressing these issues in a timely and effective manner is crucial to enhancing software quality, features, and documentation. Failure to do so can lead to software quality degradation and even render the software unusable. Therefore, there is a need to mine the software issue-related artifacts to understand the behavior of the software and improve the management of these issues. However, existing tools lack the ability to efficiently and accurately mine issues, identify issues' labeling, and assign them to the correct developer. It is therefore important to develop a more effective and efficient tool for mining software data, which is a key motivation for my research in this area.

1.1.2 Machine Learning Approaches in Code Quality Classification

Maintainability is an essential part of software development. Studies have shown that code quality attributes such as complexity, coupling, and cohesion are critical to maintainability [1, 2]. The use of source code metrics to improve software quality factors has been proven to lead to better software maintenance [3, 4, 5]. Ignoring software maintenance in the current version of a project can lead to the accumulation of technical debt, which can be costly and time-consuming to repay [6, 7]. Traditional tools used to analyze software quality can be time-consuming. Therefore, it is essential to apply modern techniques such as machine

learning to classify software quality and undertake preventive maintenance. This issue motivates me to research machine learning techniques to improve software quality and undertake preventive maintenance.

1.1.3 Machine Learning Approaches in Code Smell Detection

Code smell, as an indicator of poor software design, is a major issue that can lead to higher software maintenance costs and jeopardize the sustainability of software. While several detection rules, heuristics, and traditional machine learning-based approaches have been proposed to identify code smells, recent advances in machine learning have led to an increasing trend in using neural network-based approaches. However, there is still a need for a comprehensive study to compare the accuracy of neural network-based approaches with traditional machine learning approaches. This gap in the research is what motivates me to investigate the effectiveness of applying machine learning, especially neural network-based approaches, to code smell detection.

1.1.4 Machine Learning Approaches in Issue Label Identification

Machine learning approaches have revolutionized the way we approach various tasks, and Source Code Management (SCM) issue label identification is no exception. With the sheer volume of issues that repositories receive, it can be challenging to label them automatically and correctly, and this is where machine learning comes in. By leveraging algorithms and data analysis, machine learning can accurately classify and assign appropriate labels to SCM issues, thus saving developers time and effort.

Furthermore, SCM's issue label identification through machine learning allows for consistent and standardized labeling, reducing errors and improving overall efficiency. Very few studies have been done on machine learning in SCM issue label identification. And that is what makes me motivated towards such research.

1.2 Motivation

The motivation for this research stems from the increasing importance of software maintenance in the context of distributed software development. The rise of source code management software has created a need to extract valuable information from software repositories, which can be used to improve software quality, identify bugs and code smells, and predict re-engineering activities. However, current tools are limited in their ability to extract repositories, codes, and non-textual artifacts and efficiently mine issue-related artifacts.

Moreover, machine learning techniques are becoming popular in research related to software development, maintenance towards code quality classification, code smell detection, and issue label identification. However, due to a lack of a huge dataset of software artifacts, and the low performance of existing ML approaches, software researchers are unable to use developed ML approaches for practical purposes. And that's why I am motivated to develop a more efficient tool for mining software data and investigate whether more data can improve machine learning performance in software maintenance research.

Overall, the research is motivated by the need to fill gaps in the current understanding and implementation of mining software data and the process of performance improvement of ML approaches in software maintenance.

1.3 Research Objective and Questions

My research involves investigating tools and techniques in software engineering research to improve code quality and identify code smells for better software maintenance. I want to understand what makes a better tool to mine software artifacts and which machine learning approaches are providing better performance in classifying code quality, code smells, and issue labeling. The objective of this research is to explore and evaluate the machine learning approaches for identifying code quality, code smells, and issue labels and to develop a tool that can extract software artifacts for ML training purposes.

In this dissertation, the following research questions drive this investigation:

RQ1. How does the proposed mining tool compare with existing state-of-the-art tools in terms of performance & usability metrics?

The question reveals the importance of the proposed tool in terms of performance and usability. The performance dimension gives valuable information on the estimated time, maximum memory consumption, and cyclomatic complexity. The usability dimension is motivated to look at the user interface, learning curve, data visualization, and error reporting to the targeted audience. The finding reveals the comparison with existing state-of-the-art tools. I describe the research methodology and results in Chapter 3.

RQ2. Which code metrics are correlated most with code quality attributes and How accurately can machine learning approaches classify code quality?

This question reveals the relationships between code quality attributes and source code metrics. To answer this question, we apply statistical correlation on source code metrics and code quality attributes collected from open-source source code repositories to find out the relationship. Also, this question is targeted to find out the accuracy of machine learning approaches in class-level code quality detection. We apply several machine learning techniques and evaluate the performance. This question reveals the best technique for detecting code quality from source code metrics. I describe the research methodology and results for this RQ in Chapter 4.

RQ3: What are the predominant code smells in software repositories, and how accurately can machine learning techniques identify code smells?

The question is targeted at identifying the code smells in open-source repositories. It explores the current state of the art of code smell detection using machine learning approaches. I identified code smells that are being investigated in recent studies, and prevalent code smells are listed. The question is also targeted at finding the accuracy of several ML classifiers for detecting code smells. I describe the research methodology and results related to the performance comparison of all the machine learning approaches in Chapter 5.

RQ4: How accurately can machine learning techniques classify software issue labels?

This question finds out the automatic issue labeling by using modern machine-learning techniques. To answer this research question, I use the G-Issue mining tool to mine issues from open-source repositories and convert issue words to vectors for the creation of an issue dataset. Finally, I applied the deep learning technique to detect the issue's labels. I describe the research methodology and results in Chapter 6.

1.4 Contributions

The contributions of the dissertation lie in two main areas: (1) mining software repositories; and (2) machine learning approaches in software maintenance. Under ML approaches in software maintenance, there are three contributions to the development of ML approaches in code quality, code smells, and issue label classification.

The first contribution of the dissertation is the development of a groundbreaking mining tool, ModelMine, that can extract repositories, model-based artifacts, and designs from open-source repositories and compare the performance of the ModelMine tool with a state-of-the-art tool [8]. The details of the contributions can be found in Chapter 3. The second contribution is the investigation and comparison of the performance of traditional Machine Learning approaches for code quality attribute classification from source code metrics and report the best ML technique towards code quality classification [9]. The details of the second contribution can be found in Chapter 4.

In the third contribution, I investigate the generation of unexpected code smells in software repositories, and compare the performance of the traditional and neural network-based ML approaches in code smell detection [7]. The details of the contribution are discussed in Chapter 5. The fourth contribution is the development of a deep-learning approach to issue label identification. The details of this contribution are discussed in Chapter 6.

In preparation for this dissertation, I have authored a collection of scholarly works, consisting of four conference papers (three of which have been published [8, 9, 7] and one currently under review), as well as two journal papers (one of which has been published [5] and the other one is under review).

In addition, there are ongoing research projects, including a systematic literature review of machine learning algorithms for code smell detection, which is currently under review by the Wiley Journal of Software Practice and Experience. Another area of research focuses on evaluating the performance of "G-Issue," a proposed mining tool for issue-related artifacts, against other state-of-the-art tools while also examining the issue lifetime and evolution of well-known and well-maintained repositories over time.

1.5 Significance of the Research

The significance of this research can be summarized as follows:

- **Practical ML approaches in software maintenance:** The research has practical implications for software quality assurance, maintenance, and issue management. The proposed ML approaches can help software engineers to identify code smells, code quality, and issue labels in a timely and accurate manner, leading to improved software quality and sustainability.

- **Development of usable tools & ML techniques:** The research makes several contributions to the fields of mining software repositories and machine learning for software maintenance. The development of a novel mining tool, ModelMine, and the use of ML approaches to classify code quality attributes, code smells, and issue labels are significant contributions of this research.

- **Improvement of existing approaches:** The research demonstrates the limitations of existing approaches that are based on heuristics and proposes modern ML tech-

niques that are more accurate and efficient. For instance, the proposed deep learning-based ML technique for issue-label classification outperforms existing research.

- **Research extensibility:** The research opens up paths for future research, such as incorporating a larger dataset into the ML models by collecting more data using the ModelMine tool and developing a methodology using NLP techniques instead of traditional ML approaches to extract insights from textual data associated with software code for software maintenance.

In summary, this research has important implications for improving software quality and sustainability and makes significant contributions to the field of software maintenance. The proposed tools and techniques have great significance for future software engineering research.

1.6 Dissertation Overview

The dissertation showcases my contributions to the field of software engineering, specifically in the areas of mining software repositories and utilizing machine learning for code quality analysis, code smell detection, and issue label identification. Chapter 2 provides an in-depth literature review of the current state of research in these domains, highlighting the existing approaches and their limitations. Subsequently, Chapters 3-6 present my original contributions, which bring forth innovative solutions to address some of the challenges faced in the field. Finally, Chapter 7 concludes the dissertation by summarizing the key takeaways from my work and discussing its implications for future research. Overall, this dissertation aims to provide a comprehensive overview of my contributions to the field and offer a valuable resource for researchers and practitioners alike.

1.6.1 Chapter 2: Literature Overview

This chapter provides an overview of the existing literature on the topics of mining software repositories and machine learning in code quality and code smell research. The review is structured into four main sections, each focusing on a distinct area of research:

1. Studies that aim to mine, analyze, and visualize software artifacts,
2. Studies that apply machine learning techniques to code quality classification,
3. Studies that apply machine learning for code smell detection, and
4. Studies that apply machine learning for issue label identification.

Each section provides a comprehensive examination of the current state of research in the respective area, with a focus on identifying the major challenges and limitations, as well as the proposed solutions. The literature review highlights the mixed results reported in the field, with some studies demonstrating the capabilities of certain tools for software artifact mining, while others report on the performance of machine learning models. Ultimately, this chapter provides a solid foundation for the subsequent chapters of the dissertation.

1.6.2 Chapter 3: ModelMine: A Tool to Facilitate Mining Software Artifacts from Open Source Repositories

In this chapter, I introduce a proposed cutting-edge mining tool designed to extract model-based artifacts and designs from open-source repositories. By leveraging this tool, we can uncover valuable insights into software designs and practices within open-source communities. Additionally, I explore ways to simplify the process of mining software repositories and discuss research methods that can be employed to improve the performance of mining efforts. Also, I present a synthesized dataset of software data that includes code quality attributes, code smells, and issues extracted from various open-source repositories using

this tool. I provide insights into the research process that can be employed to obtain similar data. By leveraging this dataset, researchers can gain valuable insights into software quality, and code smell and identify potential software issues that need to be addressed.

1.6.3 Chapter 4: Performance Analysis of Machine Learning Approaches in Software Code Quality Classification

In this chapter, I present a study that explores the classification of software code quality components in software design using machine learning approaches. The study includes an examination of the relationship between source code metrics and code quality, with a particular focus on class complexity, coupling, and lack of cohesion. Additionally, we compare the performance of various ML techniques for code quality classification to identify the most effective approaches. Through this chapter, I hope to provide valuable insights into the use of ML for improving software quality.

1.6.4 Chapter 5: Evaluating the Accuracy of Machine Learning Algorithms for Code Smells Detection

In this chapter, I present a case study focused on evaluating the quality and identifying code smells in software repositories. I examine the quality of handwritten code developed within the context of Model-Driven Engineering (MDE). The study reveals key code smells that are more prevalent in handwritten code within MDE projects. By highlighting these issues, I hope to provide insights into the areas that require attention to improve software quality and maintainability. Also, I discuss the machine learning algorithms for code smell detection in recent studies on code smell detection, with a particular focus on research articles published between 2015 and 2021. This chapter reveals the predominant code smells and machine learning techniques in recent studies. Finally, I compare the performance of neural network-based ML approaches with traditional ML approaches in classifying code smells. Through this study, I aim to provide a comprehensive overview of the current state

of research in code smell detection and identify research areas for future investigation.

1.6.5 Chapter 6: Issue Label Identification: Towards A Machine Learning-based Approach

In this chapter, I present research focused on software issues reported during community-based software development. I examine software issue-related artifacts to gain a deeper understanding of software behavior through software issues. I investigate the performance of a proposed issue-related artifacts mining tool, "G-Issue", and compare it with other state-of-the-art tools. Through analysis, I also explore the issue lifetime and evolution of issues over time within well-known and maintained repositories. Also, I propose a deep learning approach to identify software issue labels automatically. By providing insights, I hope to contribute to the development of more effective machine learning models for software issue label classification.

Chapter 2

Literature Review

This chapter provides an overview of the existing literature supporting research areas in this dissertation. It begins by presenting the background and scope of the research field and then proceeds to examine key research areas, identify gaps in the current literature, and outline the proposed theoretical framework.

2.1 Overview of the Research Field

The Mining Software Repository (MSR) and Machine Learning research in the software engineering field has gained significant attention in recent years due to the abundance of rich software data that are readily available. The data stored in these repositories provide valuable insights into software development activities and have become increasingly important in software maintenance and re-engineering research. However, access to these repositories and data extraction has been challenging in the past, due to limited access and the complexity of understanding the APIs of open-source systems.

The MSR community was established with the first International Workshop on Mining Software Repositories held at the International Conference on Software Engineering (ICSE) in 2003. Since then, MSR has grown into a prominent conference in its own right, attracting a large amount of interest within the software engineering community. MSR-related publications in top research venues continue to grow in size and quality, and many MSR-related papers have won awards at prestigious venues.

Machine learning has been increasingly applied to software maintenance, particularly in the areas of code quality and code smell research. Code quality refers to the overall

characteristics of software code, including its readability, maintainability, and efficiency. Code smells, on the other hand, are specific patterns or structures within code that may indicate potential problems or areas for improvement.

Machine learning techniques have been applied to automatically detect and classify code smells, as well as to predict the impact of code changes on code quality. These approaches can help developers prioritize their efforts, focus on the most critical issues, and ultimately improve the overall quality of their software code. However, there are still challenges in applying machine learning to code quality and code smell research, including the need for high-quality training data and the difficulty of interpreting the results of complex machine learning models.

2.2 Software Artifacts

The general idea of retrieving software-related data on demand is not new and can be related to Data as a Service (DaaS). These data are collected from source code, metadata information, structured data, graphics files, etc. [10]. Some tools provide static and dynamic processes to collect metadata information as well as file structure. However, the process is not trivial to collect, analyze and visualize specific types of software artifacts from open-source repositories.

2.2.1 Tools in Mining Software Repositories

Tools in Mining Software Repositories (MSR) research field involve the application of data mining, machine learning, and statistical techniques to software repositories such as source code, bug reports, and version control systems. These tools are used to extract useful insights and knowledge about software development practices, including software evolution, bug fixing, and code quality. Several research has been done on the development of tools to mine software artifacts. In one research, GHTorrent was developed to provide repositories of GitHub in a static way that was archived each year since 2013 [11]. As a public reposi-

tory of software repositories, it provides cross-domain analysis on metadata information of repositories [10, 11]. A study by Goes [12] in 2014 showed that Big Data concepts in software repositories such as GHTorrent are characterized by 5-Vs (Volume, Variety, Velocity, Value, and Veracity). However, it limits research on current data files and commits history [13, 14]. This issue was also mentioned by Gousios et al [11]. The author mentioned that the advantage of GHtorrent in terms of repository information that are mostly static helps to analyze software structure and basic metadata information, not dynamic attributes such as recent commits, comments, or issues [11].

Another example, PyDriller, a Python framework for mining software repositories can extract recent information from open-source repositories such as commits, developer information, modifications, differences, and source codes [15]. MetricMiner is another cloud-based application suitable for mining software repositories for metrics calculation, data extraction, and statistical inference [16]. These tools focus on extracting data primarily from either code or commit history, with limited support for mining non-textual artifacts.

Software textual or non-textual artifacts mining from public repositories can be connected to software quality, and maintainability research to analyze when files are created, and how those files are maintained and sustained throughout the project life cycle. A study conducted by Gregorio et al. [17] reported that creating a model-based dataset helps to study the impact of specific extension models on the code structure and how software quality and productivity are changing throughout the project life cycle. A study by Noten et. al. [14] showed that such dynamic model files allow one to analyze recent data on repositories, artifacts, or commit information to maintain software quality. Overall, the field has contributed to software maintenance research and has led to the development of many useful software tools.

2.2.2 Software Artifacts Mining

Software Artifacts Mining is a research field that focuses on analyzing and understanding software artifacts, such as source code, documentation, and bug reports. The field involves

the use of data mining, machine learning, and other techniques to extract valuable insights from these artifacts, with the aim of improving software quality, maintenance, and development processes. Several studies have been conducted research on such artifacts from different perspectives such as sentiment analysis [18], label prediction[19], issue management [20] & mining [21].

Software artifact mining has improved software quality, bug identification, and network analysis. Several studies have uncovered interesting and actionable artifacts from software data. Several mining tools have emerged to enable such research, and discovery [15, 8, 22]. For example, PyDriller, a Python framework for mining software repositories, can extract recent information from open-source repositories such as commits, developer information, modifications, differences, and source codes [15]. However, the tool has no feature to extract issues. MetricMiner is another application suitable for mining software repositories for metrics calculation, data extraction, and statistical inference [16]. These tools focus on extracting data primarily from either code or commit history, with limited support for mining issue-related artifacts. The insights generated from this research field can help developers understand how software is built and maintained, leading to better software design and more effective software development practices.

2.2.3 Software Artifacts Analysis

Software Artifacts Analysis is a research field that involves the systematic analysis of software artifacts, such as source code, documentation, and requirements. This field aims to identify and understand the characteristics and properties of these artifacts, and how they relate to software quality, maintainability, and evolution. Issue-related artifact analysis has gained popularity last few years. Several studies have researched on issue lifetime [19, 23], how long it will take to close an issue, and empirical studies on the life expectancy of issues based on labels. Kikas et al. conducted research on 4000 repositories to find temporal dynamics of issues in GitHub [19]. The study found that projects with a shorter observation time tend to have higher volumes of open issues. In addition, Kikas proposed a prediction

model trained from static, dynamic, and contextual features to predict the lifetime of an issue. The results showed that the average issue lifetime for community-created issues is 39 days, but the team-created issues are 5.9 days. Techniques used in this field include static and dynamic analysis, and testing, with the goal of improving software quality and reducing the risk of failures.

2.2.4 Software Artifacts Visualization

Software Artifacts Visualization is a research field that focuses on developing visual representations of software artifacts, such as source code, dependencies, and software architectures. The goal of this field is to provide developers and stakeholders with a better understanding of complex software systems, making it easier to comprehend and maintain them. Visualization techniques have been popular in textual, camera control & network data [24]. However, this research is becoming popular in software engineering too. As an example, an issue-related artifact has textual(issue title, body) and network (issue assignee, label) data. Very few research has been conducted on issue-related artifact visualization [25, 20]. Liao et al. applied the visualization technique on issue-related behavior by analyzing seven projects with 98,074 issues in total [25] and proposed an SRF to measure the importance of user behaviors. The results found that issue-related user behaviors are critical, and not all issues that are assigned labels could be closed rapidly. Another empirical research by Bissyand et al. investigated the adoption of an issue tracker based on the projects, developers, and type of issues [20]. The results found that small-sized team projects are less likely to have issues and visualized the top 10 labels in GitHub, where bug and feature requests are at the top with 18.36% and 10.29%, respectively. Techniques used in this paper include 2D, interactive visualizations but are planned to show software artifacts visualization in 3D.

2.2.5 Other Research on Software Artifacts

There is other research on software artifacts conducted in aspects of sentiment [26, 18, 27], bug and issue tracking [28]. Jurado et al. conducted a study on 10,829 issues from 9 well-known & open-source repositories to do sentiment analysis [18]. The study proposed a new technique to identify the underlying sentiments in the text found in issues and their comments. Results showed that issues' titles and text leave underlying sentiments, which can be used to analyze the development process. Another set of research on issues-related artifacts is automatic labeling of Issues [29, 30, 31]. Kallis et al. introduced a tool called Ticket Tagger, which is developed using Node.js, de-facto server-side JavaScript, and uses machine learning approaches on issue artifacts to label an issue automatically [30].

2.3 Machine Learning in Code Quality Research

Machine Learning in Code Quality is a research field that aims to apply machine learning techniques to improve the quality of software code. This field involves developing models that can detect code quality, and recommend improvements to code. The goal of this field is to help developers write better code, reduce the risk of defects and failures, and improve the maintainability of software systems. The earlier techniques to measure the code quality depended on low-level metrics. Estimating quality characteristics has been a well-known problem for many years. The reason behind this, quality has a direct impact on software maintenance and software development [1, 2]. Many researchers and practitioners proposed different methodologies to improve code quality for software maintenance. In this section, we discuss the existing research on those areas with mentioning advantages and limitations.

2.3.1 Code Quality Features

Code Quality Features are the characteristics of software code that determine its quality, maintainability, and reliability. These features include readability, consistency, modular-

ity, testability, and efficiency. Several research on code quality from source code metrics includes fault-prone modules detection [32], early detection of vulnerabilities [2], improvement of network software security [33, 34, 35], software redesign [36], etc. All of these researches are targeted to reduce the maintenance effort and cost during software development. Chowdhury et al. investigate the efficacy of applying cohesion, complexity, and coupling to automatically predict vulnerability and complexity entities [2]. This study used machine learning and statistical approaches to predict vulnerability that learn from the cohesion, complexity, and coupling metrics. The results indicate that structural information from the non-security realm such as cohesion, complexity, and coupling is useful in vulnerability prediction which minimizes the maintenance effort.

Another study proposed by Briand et al. [37] analyzed the correspondence between object-oriented metrics and fault proneness. This research result is created based on a few classes analysis. Gegick et al. [38] developed a heuristic model to predict vulnerable components and complexity. The model was successful on a large commercial telecommunications software and predicted vulnerable components with an 8% false positive rate and 0% false negative rate.

In another research, Varela et al. conducted a systematic study to collect code quality features and perform an analysis of over five years of data [39]. They found almost 300 different source code features used by different organizations, practitioners, and researchers. Among all analyzed attributes, complexity, coupling, and lack of cohesion were the most prevalent quality features for software maintenance [1, 2]. They also reported the machine learning techniques' performance in the detection of code quality. Another study by Janes et al. demonstrated that source code metrics could predict the fault-prone modules with the help of the following quality features: Response Set for a Class (RFC) and Coupling Between Object class (CBO) [40]. Dallal et al. utilized similar object-oriented features to improve predictions using ML techniques [41]. Recently, Shin and William demonstrated an approach to predict vulnerabilities using object-oriented code quality features, such as complexity, coupling, and cohesion [42, 43]. All these efforts can reduce the risk of

generating bugs and failures, make code easier to maintain, and improve the overall quality of software systems.

2.3.2 Machine Learning Approaches

Machine learning approaches are becoming increasingly popular in improving code quality by automating code analysis and identifying code quality. These approaches can be used to classify the impact of code changes on quality. Machine learning approaches are particularly useful in dealing with large and complex codebases, where manual analysis can be time-consuming and error-prone. Several research on machine learning approaches has been done. In a few research, Code quality was identified through a threshold or fuzzy-based approaches. But machine learning techniques open a new arena for code quality detection. One key challenge in using ML to classify software quality features is identifying the most influential source code metrics to be fed into the ML algorithms. Chang et al. identified some of the influential code metrics that impact software usability using fuzzy logic [44]. Another study by Watanabe et al. was conducted to collect a software defect dataset with 8-object-oriented code metrics from two projects and proposed a method to do Cross Project Defect Prediction (CPDP) [45]. As an extension of this research, Rahman et al. collected nine datasets with the same process but got different influential source code metrics to improve the performance of CPDP in terms of costs [46]. However, collecting a small number of object-oriented metrics limit any prediction accuracy [47, 48]. Wang et al. discussed the importance of a significant number of software quality parameters in detecting software defects. In a study, the author showed the dynamic version of AdaBoost.NC with object-oriented code metrics improves the overall performance of software quality predictions [49]. Not only Wang but also Kehan et al. discussed what code metrics are needed to choose for defect predictions and how such metric selection impacts the predictions [50].

2.3.3 Code Quality Analysis Techniques

Correlation analysis among features in software engineering is a technique used to identify relationships between different code quality features. By examining the correlation between these features, developers can gain insights into how changes in one feature might affect others, and prioritize efforts to improve code quality accordingly. This analysis is useful in understanding the overall quality of software code and identifying areas for improvement. Many prior studies have used correlation in their software quality research [2, 51, 52]. Chowdhury et al. experimented with a hypothesis that complexity is positively correlated to vulnerabilities. He also experimented with his other correlation hypothesis related to coupling and cohesion [51]. The results showed empirical evidence that using correlation in software engineering research is providing a better outcome. Another study by Marco et al. found a correlation between object-oriented metrics and bugs [53]. The author experimented with correlation with major, minor, and high-priority bugs. The results showed a correlation between change coupling and defects higher than the one observed with complexity metrics. Another recent work on software quality research experimented with and evaluated the correlation between quality features and source code metrics [52]. The experimental results showed that unit complexity metrics correlate with vulnerability and make stronger vulnerability predictors than coupling metrics.

2.3.4 Other Code Quality Research

There is very limited literature on code quality aspects in a specific environment such as Model Driven Engineering (MDE), Unified Modeling Language (UML), etc. In one research, Hutchinson et al. [54] conducted an empirical study of MDE projects in the context of the industry by questionnaire and interviews. They reported on the understanding of social and organizational factors on MDE usages and investigated factors of failure and success aspects of MDE such as benefits of code generation.

In another research, Fernandez-Saez et al. [55] conducted interviews and questionnaires

on UML and software modeling with employees of a software company that works on software maintenance projects. The results of this survey suggest that UML modeling is beneficial, however, there are concerns about integrating modeling into the overall software engineering approach. Nurgoho and Chaudron [56] analyzed the impacts of UML modeling (class diagram and sequence diagram) in terms of defects density with software modules that are not modeled and found that UML modeling reduces the defect density in code than not modeled modules. However, there is a research gap in investigating the characteristics of handwritten code (HC) in MDE and Non-MDE software sub-systems from open-source platforms.

2.4 Machine Learning in Code Smell Research

Code smells indicate a violation of software design principles. Detecting code smells using machine learning is comparably better technology to minimize violations of design and improve software quality. Many prior studies used such concepts to find design flaws and enable software redesign.

Over the last two decades, several research has been conducted to understand the reason and the impact of code smells in code quality and technical debt management [57, 58]. In code smell research, researchers used current or historical source code metrics information or textual properties to exploit code smell [59, 60]. In terms of approaches, there are two ways of approaches used by researchers. In one way, researchers used heuristic approaches [60, 61, 62] and in other ways, researchers used machine learning approaches [60, 63, 64, 65].

In the machine learning approach, researchers apply traditional approaches and neural network-based approaches [66, 67, 68]. In one study, Fabiano et al. focus on the comparison of traditional heuristic approaches and supervised machine learning approaches for metric-based code smell detection [60]. Kreimer [69] proposed a decision tree-based approach to identify specific code smells (long method and large class) to visualize design flaws. The technique is applied based on a few source code metrics and quality features which minimize

the power of the adaptive learning approach. Vaucher et al. [70, 71, 72] applied Bayesian belief networks to detect God's class (Blob class). The author used the box plot to perform the discretization of God Class. The weak point of the research lies in the approach with few numbers of classes analyzed. A similar approach was proposed by Amorim et al. [73] to detect the Blob class, long parameter list, long method, and feature envy. The author applied the decision tree technique rather than trying different machine learning to prove that the decision tree works better than others. However, a detailed study was done by Fontana et al. using different machine learning techniques including J48, JRip, Random Forest, Naive Bayes, SMO, and LibSVM in predicting the severity of code smells [74, 75, 76]. The paper reported each technique's performance, but a weak point exists in balancing different severity levels that were missing and might create bias in the results.

Several other systematic literature reviews showed the machine learning techniques for identifying essential code smells that hurt the maintenance of software systems [77, 78, 60]. A systematic literature review on machine learning techniques for code smell detection by Azeem et al. [79] considered 2456 papers published between 2000 and 2017. The paper revealed a promising way to identify the search keywords and organize the research perspectives. The results uncovered the predominant code smells, machine learning approaches, and evaluation strategies. Another similar systematic literature review conducted on 319 papers by Zhang et al.[80] reported code smells. The study findings indicate that duplicated code is the most identified code smell. This literature review reveals that all the data used to detect code smells are objective rather than subjective. Similarly, Gupta et al. [81] identified that code clone receives the most attention among all code smells. They performed a literature review on sixty research papers. They found that most of the literature in the area of research is focused on code smell detection tools and techniques rather than the actual impact of code smells.

Some studies report on code smell detection at the class and method levels. Pritom et al. [65] reported on a study that detects class-level code smells using different machine-learning techniques. Firstly, 4120 classes were selected for this study after pre-processing.

Six machine learning algorithms (Naive Bayes Classifier, Multilayer Perceptron, Logit-Boost, Bagging, Random Forest, and Decision Tree) were used to predict change proneness in the classes. The study results show that code smell is indeed a powerful predictor of class change proneness. The models perform with 70% accuracy except for a few exceptions. Additionally, many researchers conducted similar studies to measure the attributes of software systems' good and bad aspects [82].

Lafi et al. [83] presented research that discussed the importance of detecting code smells and different code smell detection machine learning algorithms. They discuss the significance of maintenance issues related to code smell and finding optimal code smell detection techniques. Further, a group of researchers investigated which code smells should be considered while identifying code smells using machine learning approaches [84]. They reported one class level and two method level code smell for software systems. They found very high accuracy in the prediction model. Such as 98.57% accuracy for the data class, 97.86% accuracy for the God class, 99.67 % feature envy, and 99.76% accuracy for the long method.

Rasool et al.[85] conducted a study on code smell detection techniques and tools for mining code smells. They observed that most of the subject systems used for code smell detection are open-source or local. They also reported a lack of evidence of which code smell detection technique is appropriate. They conclude that code smell detection techniques should be flexible on the subject systems. Additionally, code smell detection techniques are classified into seven categories [86]. Many code smell detection techniques are based on source code metrics. Further, the code smell threshold is determined from the source code under analysis.

2.4.1 Traditional ML Approaches

In the past decade, traditional machine-learning approaches have been widely used for code smell detection. These approaches involve extracting features from source code and using classification algorithms to identify code smells. Techniques such as support vector

machines, decision trees, and random forests have been commonly applied. However, the effectiveness of these approaches heavily relies on the quality of extracted features and the choice of classification algorithms. The most common machine learning approaches used to detect code smells are supervised methods [60, 87]. In such an approach, independent variables (source code metrics as predictors) are used to detect dependent variables (code smell of classes). The setup of those machine learning approaches varied based on the dataset size, training-testing size, within-project testing, cross-project testing, etc. Besides that, there are variations in ML approaches in terms of probabilistic approach or tree-based approach.

Amorim et al. [73] evaluated the Decision Tree machine learning approach to detect code smells (Blob, Long Parameter List, Long Method, Feature Envy). The results showed that performance metrics F1 measure are above 67%. Similar results were reported by Fontana et al. [63]. In his proposal for four different code smells (God Class, Data Class, Long Method, and Feature Envy) and more than four different machine learning approaches, Naive Bayes, Random Forest, Support Vector Machine, and Decision Tree are used. According to the results, Decision Tree performs well in detecting code smells. A recent study by Dewangan et al. [88] showed a different set of results where Random Forest becomes the best algorithm to detect code smells. As there are differences in results, I will verify that with the collection of code smells from popular and well-maintained repositories in this dissertation.

2.4.2 Neural Network-based ML Approaches

Recent research has explored the use of neural network-based machine learning approaches for code smell detection. These approaches involve using deep learning techniques to automatically learn features from source code, such as through the use of convolutional or recurrent neural networks. These approaches have shown promise in achieving high detection accuracy and overcoming the limitations of traditional feature-based approaches. However, they require large amounts of labeled data and extensive computational resources for training. Several research used neural network-based ML approaches for code smell

detection[67, 66, 89]. In general, neural network models are being trained with a large amount of data from the repository analysis and are required high-end machines contrary to traditional Machine Learning techniques.

A recent work by Lin et al. showed why deep learning techniques are becoming popular in code smell detection [66]. Another novel approach by Liu et al. showed neural networks and advanced deep learning techniques could automatically select features of source code for code smell detection, and could automatically build the complex mapping between such features and target labels [67]. In both studies, the performance of detecting comparison is better in terms of accuracy, precision, recall, and F_1 score. Sharma et al. also apply deep learning models to see the feasibility of such a technique and also investigate applying transfer learning [89]. Experiments by this study showed that transfer learning is feasible for code smells with performance comparable to that of direct learning.

2.5 Gaps in the Literature

The literature gap in the field of Mining Software Repositories is the limited support for software engineering researchers in targeting mining software artifacts specially models and non-textual artifacts for system development from open-source platforms. The current textual-based mining tools, such as PyDriller and MetricMiner, have limitations in mining non-textual artifacts, as well as having a smaller size of extractable data compared to modeling artifacts. These deficiencies have limited the depth and breadth of the extractable data, thus limiting the scope of the possible research. Also, the literature gap of the existing research is the lack of efficient and effective issue mining, analyzing, and visualizing in open source communities. Existing efforts on issue mining are limited to specific tasks such as title prediction, and sentiment analysis. There is a missing effort in comparing the performance of my developed tools with state-of-the-art tools in terms of execution time and memory usage

Although neural network-based machine learning approaches have shown promise for

code smell detection, the literature gap lies in the comparison of their performance against traditional machine learning approaches. Existing studies have primarily focused on evaluating the performance of either traditional or neural network-based approaches independently, without directly comparing their performance. Moreover, it remains unclear which approach performs better under different scenarios, such as varying levels of code repositories. Therefore, a systematic comparison between these approaches is needed to understand the trade-offs and benefits of each and guide the selection of appropriate techniques for code smell detection in practice.

The field of automatic software issue labeling has seen an increase in the use of deep learning approaches in recent years. This is due to their ability to learn complex patterns and relationships in data, which can be difficult for traditional machine-learning approaches to capture. However, there is a literature gap in terms of comparing the performance of deep learning approaches with traditional machine learning approaches in this specific task. While some studies have reported promising results with traditional machine learning, there is a need for more comprehensive and rigorous comparative evaluations to establish the superiority of one approach over the other. This gap presents an opportunity for my dissertation to contribute to the development of more accurate and efficient automated software issue labeling techniques.

Chapter 3

ModelMine: A Tool to Facilitate Mining Software Artifacts from Open Source Repositories

This chapter presents ModelMine, a novel mining tool for extracting software repositories and other artifacts from open-source platforms to uncover information about software designs and practices in open-source communities. The proposed approach supports mining software repositories, files, commits, issues, etc. This chapter is related to a research paper that was published in the 2020 ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS 2020) conference [8].

Mining Software Repositories have opened up new pathways and rich sources of data for research and practical purposes. This research discipline facilitates mining software-related data from open-source repositories and analyzing software quality, defects, development activities, processes, patterns, and more. Contemporary mining tools are geared towards data extraction, and analysis primarily from textual artifacts and have limitations in representation, ranking, and availability. This chapter presents ModelMine, a novel mining tool that focuses on mining model-based artifacts and designs from open-source repositories. ModelMine is designed particularly to mine software repositories, artifacts, and commit histories to uncover information about software designs and practices in open-source communities. ModelMine supports features that include the identification and ranking of open-source repositories based on the extent of the presence of model-based artifacts and querying repositories to extract models and design artifacts based on customizable criteria.

It supports phase-by-phase caching of intermediate results to speed up the processing to enable efficient mining of data. We compare ModelMine against a state-of-the-art tool named PyDriller in terms of performance and usability. The results show that ModelMine has the potential to become instrumental in cross-disciplinary research that combines modeling and design with repository mining and artifacts extraction.

URL: `https://www.smreza.com/projects/modelmine/`

3.1 Introduction

Mining software repositories have witnessed tremendous growth in the past few years. Software repositories, their versions, and commit histories to contain significant information about software development activities and contribute to establishing research agendas in software development, cost estimation, testing, and quality assurance [90, 91, 16, 92]. With such research advancement, the impact of modeling, code smells, code reviews, and prediction of change-prone classes become more influential in software engineering research [59, 93, 94].

Unfortunately, there is limited support for software engineering researchers who target mining models and design artifacts for system development from open-source repositories [95, 96, 97]. Mining tools like PyDriller [15] and MetricMiner [16] can extract data primarily from either code or commit history, with limited support for mining non-textual artifacts. Current textual-based mining tools expose some key deficiencies when mining repositories and model artifacts. First, textual artifacts are relatively smaller in size compared to modeling artifacts. Second, models and design artifacts are much less prevalent than code in the majority of repositories. These deficiencies have limited the depth and breadth of the extractable data, and have consequently limited the scope of the possible research. As such, the ultimate goal of the paper is to propel research in software design and related practices by facilitating a tool that enhances data extraction for model-based artifacts.

This paper presents a novel model-mining tool called ModelMine which facilitates min-

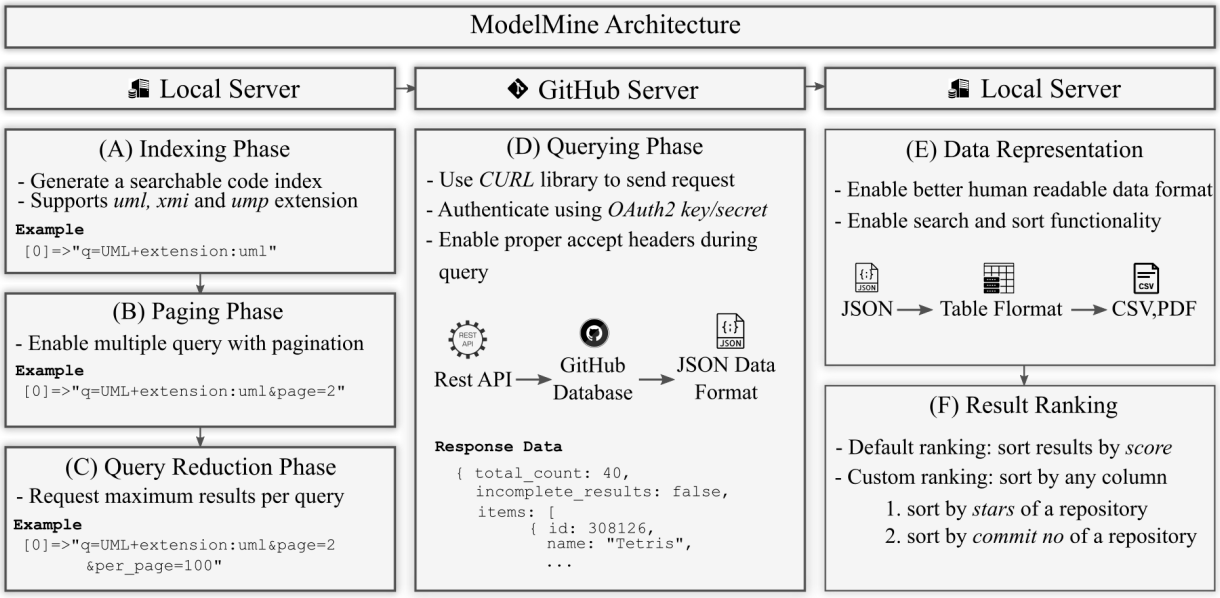


Figure 3.1: Architecture of ModelMine Tool

ing models by analyzing model artifacts and repository metadata information. The tool brings some key benefits to researchers in software design and modeling. First, it enables the ranking of repositories based on the prevalence of model-based artifacts. Second, it enables faster data extraction for non-textual artifacts using a phased data pre-fetching approach. Third, it supports different filtering mechanisms to extract models from open-source repositories without requiring extensive data mining knowledge or expertise.

To evaluate the usefulness of our tool, we compare our tool with the state-of-the-art PyDriller [15], a Python framework in terms of execution time, memory consumption, cyclomatic complexity, and usability. Results show that ModelMine requires less time, and memory, and can achieve similar results as PyDriller without having the coding knowledge. In addition to that, ModelMine can mine repositories and file artifacts which are unsupported in PyDriller.

The chapter is organized as follows. Section 3.2 presents ModelMine architecture, followed by a demonstration of ModelMine user interfaces. We evaluate the tool compared against a well-known tool named PyDriller in Section 3.4. In the last section, we conclude

the paper with future work ideas.

3.2 ModelMine Architecture

In this section, we discuss the architecture that we use to build ModeMine. The tool adopts a six-phased approach (indexing, paging, query reduction, querying, data representation, and results ranking) to mine model-based repositories, artifacts and commit history from open source repositories. The architecture of the tool is visualized in Figure 3.1 and the details are provided in the following subsections.

3.2.1 Indexing Phase

In this phase, ModelMine processes model extensions to generate a searchable code index. The tool supports several types of code extensions (UML, XMI, UMP, and SysML, etc.) which are required to index in a search query. To get proper search results for the model file extension, we create indices one by one if there are multiple file extensions. A sample query indices are shown in Figure 3.1(A).

3.2.2 Paging Phase

In this phase, the ModelMine tool works on paging the results. Since generating more results require more time and server load, we introduce the paging concept to limit the number of response results. If a researcher wants to generate more results, the ModelMine tool allows them to perform that at the cost of time. Without pagination, the researcher gets maximum set limit results. To request further results, ModelMine adds a new query parameter named *page*. A sample example is shown in Figure 3.1(B) with the paging concept.

3.2.3 Query Reduction Phase

In this phase, we implement a technique to overcome the issue limit per request. To process each request, the ModelMine tool applies a query reduction technique that enables a request to have a maximum number of results set by the administrator. Such a limit is very important during the development of any mining tools [98]. However, to get a maximum number of results per query, ModelMine introduces a new query parameter named *per_page* and it reduces approximately 70% of requests to get more results per request. A sample example is shown in Figure 3.1(C).

3.2.4 Querying Phase

In this phase, ModelMine prepares the queries created in the last three phases. Two important steps are required to secure the server. First, authentication, and second, blocking too many requests within a short period of time. In authentication, the ModelMine tool uses the $OAuth2 - key/secret$ technique [99] to protect the tool from unauthenticated users. For the second part, we introduce middleware to block multiple requests from the same IP within a short period of time [100]. A sample example is shown in Figure 3.1(D).

3.2.5 Data Representation

Data representation is the phase where ModelMine prepares the resulting data and its format of presentation. In the background, ModelMine represents the response in JavaScript Object Notation (JSON) format but for users, the responses are presented in a human-readable format (table, CSV, excel, pdf). Additionally, ModelMine provides search and sort operations to maximize user satisfaction with data presentation.

3.2.6 Results Ranking

The ranking of the results is used to position the responses on a scale. To ensure a better result ranking system in ModelMine, we use the score concept to serialize the results. This

default score system represents the relevance of a search item relative to the other items in the result set. However, this ranking system can be changed to repository popularity, watchers, etc.

3.3 ModelMine User Interface

In this section, we discuss user interface features and the mining process in ModelMine. The tool provides a simple extensible user interface to mine software repositories, artifacts, and commit history. The tool has three unique user interface features: (1) repository search, (2) artifacts search, and (3) commit history search to ensure the possibility of mining different types of datasets for MSR research.

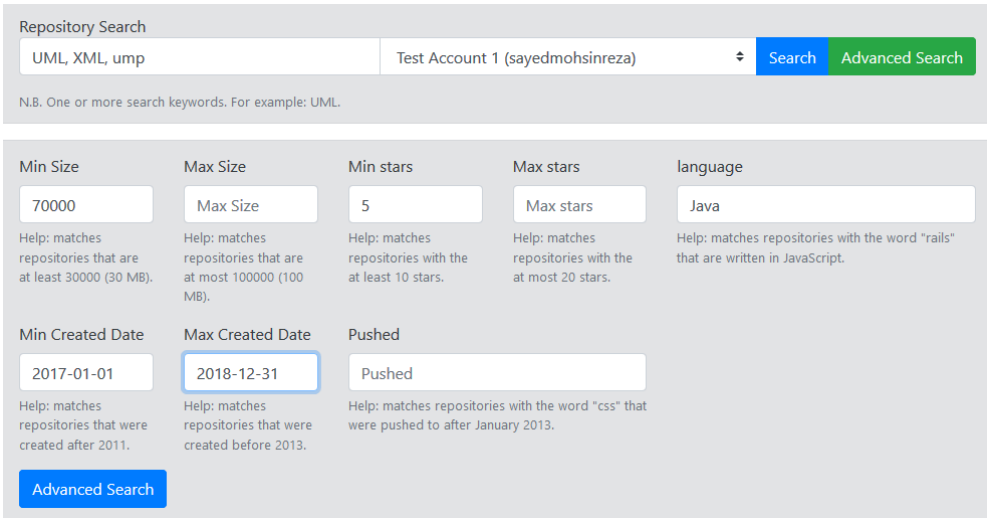


Figure 3.2: Model-based Repository Search

3.3.1 Model-based Repository Search

A typical first exploratory step involves searching for repositories with prevalent modeling artifacts. Researchers can query code repositories with any model file extensions (i.e. UML, XMI, UMP, etc.) as shown in Figure 3.2. In addition to that, ModelMine allows additional

criteria to query for repositories. Additional criteria include repository size, popularity, primary programming language, and repository timestamped information.

3.3.2 Model-based Artifact Search

ModelMine provides another user interface to search model-based files in open-source repositories. Figure 3.3 visualizes the user interface for this operation where researchers can search model files with extensions. In addition to the extension, the user can also provide a basic string that may exist inside the model file metadata.

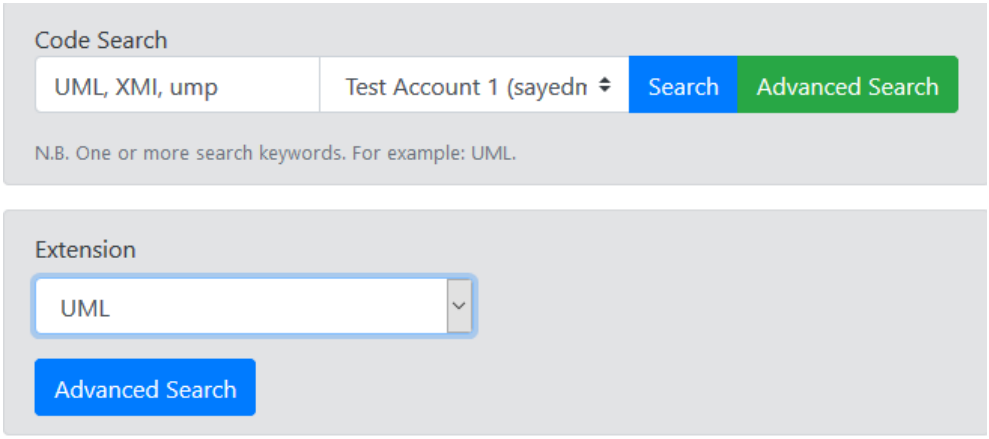


Figure 3.3: Model-based Artifact Search

3.3.3 Model-based Commit Search

Once a subset of repositories is identified using the repository search feature described in section 3.3.1, researchers are able to search for commit history. In a repository, there are multiple types of files and ModelMine allows researchers to investigate specific extension-based commit searches. This feature allows researchers to analyze the version of the files of repositories as well as the behavior of software code updates of specific file extensions for different projects [14]. The user interface for this operation is visualized in Figure 3.4.

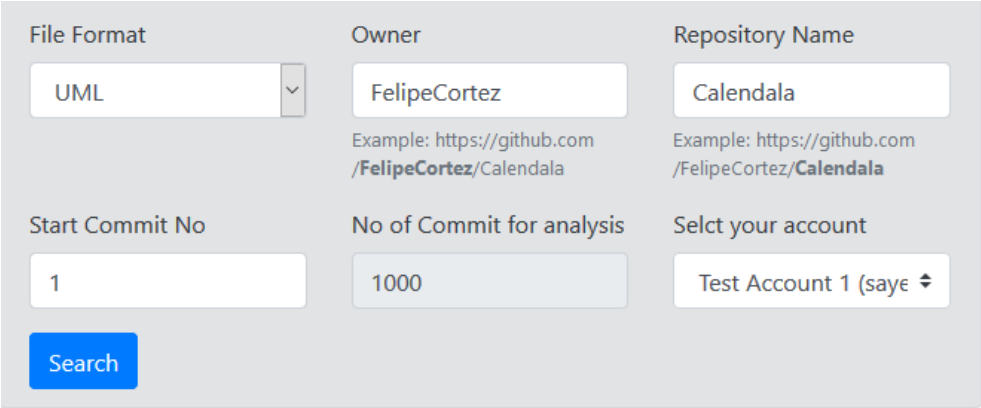


Figure 3.4: Model-based Commit Search

3.4 Evaluation

The evaluation focuses on two dimensions; performance and usability. The performance dimension is motivated by the fact that model-based artifacts are much rarer in repositories compared to code, and tend to be significantly larger in size. This often translates to computational complexity in identifying and extracting model-based artifacts. The usability dimension is motivated by the targeted audience; software design practitioners and researchers who are not necessarily competent in data mining and data extraction.

For reference, we compare ModelMine with PyDriller [15], a well-established Python framework for mining software repositories. We identify five unique tasks that are common for the majority of MSR research. The tasks are as follows:

1. **Task 1 (Size related):** Retrieve the list of repositories that include at least one model artifact developed in UML and the repository size is larger than 30 MB.

2. **Task 2 (Time related):** Retrieve the list of repositories that include at least one model artifact developed in UML and the repository was created between January 2019 and December 2019.

3. **Task 3 (File property related):** Retrieve the list of artifacts with a specific file extension: *.uml*.

4. **Task 4 (Commit related):** Retrieve the list of commits with a model artifact and the repository has at least one model artifact.

5. **Task 5 (File property + commit related):** Retrieve the list of commits with any model artifacts (any model-based file extension) and the repository has at least one model artifact.

These tasks are implemented using both frameworks: (1) ModelMine and (2) PyDriller. To compare the frameworks, we use two different types of metrics: (1) performance & (2) usability metrics. Such evaluation metrics are used in the evaluation of different software mining repositories tools [15, 101]. The reason behind evaluation is to check how easy the tool is to learn and whether the results are provided in a meaningful way or not. The metrics we used to evaluate our tool are provided in Table 3.1.

Table 3.1: Evaluation Metrics for Tasks

No	**Metric Type**	**Metric Name**	**Subcategory**	**Shorthand**	**Unit**
1	Performance Metrics	Execution Time		ET	Second
2		Max Memory		MM	Kilobytes
3		Cyclomatic Complexity		CCOM	N/A
4	Usability Metrics	Subjective Satisfaction	User Interface	UI	Rating (1-5)
5			Learning Curve	LC	
6			Data Visualization	DV	
7			Error Reporting	ER	

In the performance and usability study, we select ten participants who are actively working in software engineering research. Eight of them are doctoral students and two of them are master's students in computer science. We request participants to perform a usability study on ModelMine and PyDriller and fill up a questionnaire that collects all the usability metrics. The performance metrics are collected when the participants used the tools to do the evaluation. The details of the evaluation are available online at

https://www.smreza.com/projects/modelmine/eval/. The results of the performance analysis of each task are shown in Table 3.2.

The result clearly shows that PyDriller takes more time and memory than ModelMine. The reason behind this result is that PyDriller fetches the whole git file of a selected repository and then mines the commit information. But ModelMine fetches the information directly without downloading any file. As there is no intermediate process, ModelMine takes less time and memory. Note that, the current version of PyDriller [15] is unable to search repositories (Task 1 & 2) or artifacts (Task 3).

Table 3.2: Performance Evaluation Results

Tasks	**Metrics**	**ModelMine**	**PyDriller**
Task 1 (Size)	ET	0.793 s	Not supported
	MM	701 KB	
	CCOM	5	
Task 2 (Time)	ET	0.675 s	Not supported
	MM	698 KB	
	CCOM	5	
Task 3 (Property)	ET	0.633 s	Not supported
	MM	611 KB	
	CCOM	4	
Task 4 (Commit)	ET	1.422 s	3.8 s
	MM	630 KB	24220 KB
	CCOM	4	5
Task 5 (Composite)	ET	1.508 s	9.3 s
	MM	660 KB	24616 KB
	CCOM	4	4

In the usability study, we adopt a framework proposed by Altalhi et al. [102] where

usability analysis is designed based on a list of questionnaires presented to participants. Table 3.3 lists the questions and their related category. Similar usability studies were used to analyze overall tool ratings as well as task-based user satisfaction ratings [102, 103].

Table 3.3: Usability Study Questionnaires

Category	**Questions**
User Interface	(1) How easy are the tools to navigate?
	(2) How clearly do the tools provide results in a meaningful way?
Learning Curve	(3) How easy are the tools to learn?
Data Visualization	(4) How well does the tool about presenting the data and modeling results?
Error Reporting	(5) How relevant is the error reporting?

Participants are required to answer the given questions for each task with a scale from 1-5 to express their usability responses. The lower value represents less usability for each criterion. Figure 3.5 shows the usability results of ModelMine compared with PyDriller.

The results clearly show that ModelMine has better usability in all usability criteria than PyDriller. In the user interface & learning curve category, ModelMine has 50% more ratings than PyDriller. The evaluation study also asks participants to provide comments on their ratings. One participant mentions that ModelMine provides a platform to mine data without worrying about the data collection part. Most of the MSR tools need extensive tasks to mine and have limitations in data preparation, ranking, and availability. ModelMine has incorporated most of them to enrich the mining model experience from open-source repositories. Another participant comments that ModelMine provides a faster learning experience than PyDriller due to its easy UI design and better readability.

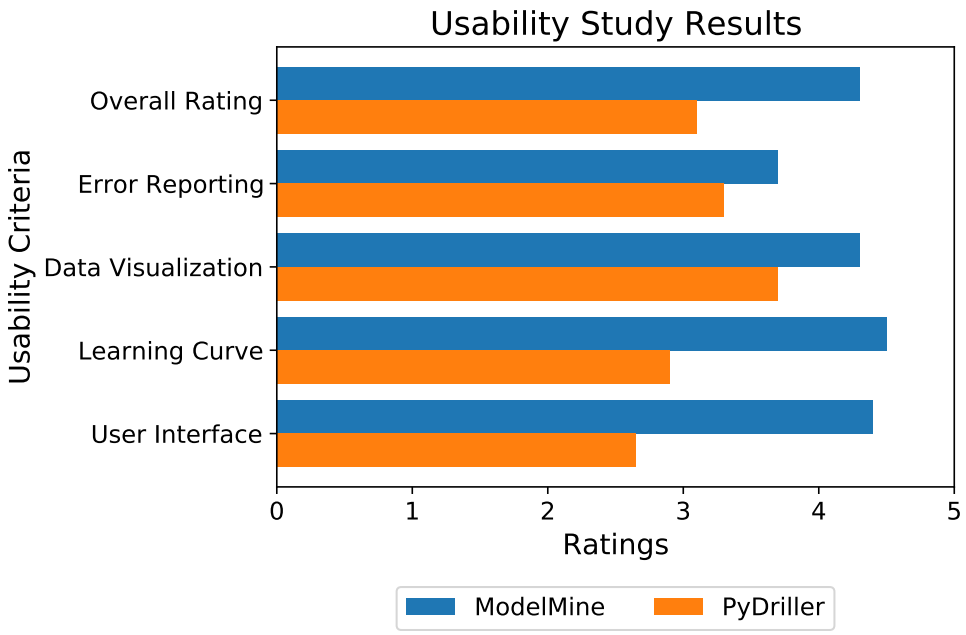


Figure 3.5: Usability Study Results

3.5 Conclusion

Prevalent mining tools are geared towards textual artifacts and tend to exhibit poor performance in mining models and software designs. Model-based artifacts are much less prevalent and tend to be much larger in size when compared to codes and other textual-based artifacts. This paper presents ModelMine, a tool crafted to facilitate mining models and designs from open-source repositories. It enables the ranking of repositories based on the presence of designs and models and uses phased data pre-fetching to enhance performance and broaden the scope of the mining processes. To evaluate our tool, we compare it with the state-of-the-art tool PyDriller and perform a usability and performance analysis with ten graduate students. Results show that ModelMine performs better than PyDriller in both analyses. The reported results demonstrate a significant potentiality of ModelMine in MSR research. ModelMine fills an important gap in the repository mining landscape and aims at realizing research that combines data mining and data extraction from open-source repositories.

ModelMine tool can be extended with image-based model mining features. We plan to mine image-based model files (i.e. png, jpg, jpeg, etc.) which will enrich MSR research. We also plan to ensure the best search results by analyzing text inside those image-based model files to understand the model semantically.

Chapter 4

Performance Analysis of Machine Learning Approaches in Software Code Quality Classification

This chapter presents a study on the classification of software code quality components in software design using machine learning (ML) approaches. The objective of the paper is to examine the relationship between source code metrics and code quality features such as class complexity, coupling, and cohesion and to compare the performance of various ML techniques for classification. This chapter is related to published research which was published at the 2020 International Conference on Trends in Computational and Cognitive Engineering (TCCE 2020) [9].

Software design is one of the core concepts in software engineering. This covers insights and intuitions of software evolution, reliability, and maintainability. Effective software design facilitates software reliability and better quality management during development which reduces software development costs. Therefore, it is required to detect and maintain these issues earlier. Class complexity, coupling, and cohesion are part of software quality features. The objective of this paper is to classify class code quality features from source code metrics using Machine Learning approaches and compare the performance of the approaches. We collect ten popular and quality-maintained open-source repositories and extract eighteen source code metrics that relate to code quality features for class-level analysis. First, we apply statistical correlation to find out the relation between the source code metrics and code quality features. Second, we apply five alternative ML techniques

to build complexity predictors and compare the performances. The results report that the following source code metrics: Depth Inheritance Tree (DIT), Response For Class (RFC), Weighted Method Count (WMC), Lines of Code (LOC), and Coupling Between Objects (CBO) have the most impact on class complexity. Also, we evaluate the performance of the techniques and results show that Random Forest (RF) significantly improves accuracy without providing additional false negatives or false positives that work as false alarms in complexity prediction.

4.1 Introduction

Software design is a process of creating software artifacts, primitive components, and constraints. Effective software design with object-oriented structures facilitates better software quality, re-usability, and maintainability [32]. One of the quality factors is complexity. This quality attribute is determined by many factors related to code structures, object-oriented properties, and source code metrics [104]. The less the complexity of software, the less the cost of software development will be [105, 91]. This motivates us to research software complexity prediction.

In the software life cycle, the more the complexity is, the maintenance becomes costly, unpredictable, and human-intensive activity [104]. Moreover, high maintenance efforts often affect software sustainability that many software systems become unsustainable over time [106, 96, 107]. Therefore, software redesign becomes an essential step where the complexity of the software needs to be reduced. Such action will enhance software maintainability and reduce the associated costs [108, 95]. Having set the importance of complexity detection for software redesign, we are motivated to predict class-level complexity from source code metrics.

Some studies introduced McCabe complexity, a widely accepted metric developed by Thomas McCabe to show the level of software complexity [36]. Another approach to the calculation of software complexity was based on counting the number of operators and

operands in software. But the calculation and counting process of total operators and operands is tedious [109].

In this paper, we use machine learning techniques to build a complexity predictor. The reason behind using machine learning is to get rid of manual processes or code rules to detect class complexity. Also, successful research on detecting software defect, and vulnerability using ML techniques motivate us [2, 110]. We use five ML classifiers, analyze the performance of the classifiers and report the best technique in complexity prediction.

The rest of the paper is organized as follows. We describe research methodology in Section 4.2. Results & Evaluation are discussed in Section 4.3 and finally, we conclude the paper in Section 4.4.

4.2 Research Methodology

This research has two main goals. First, analyze source code metrics to what extent it is possible to predict complexity. Second, report the best ML approaches evaluating relative effectiveness in the prediction of complexity from source code metrics. The details of our research questions, data sets, and machine-learning approaches are discussed in the following subsections.

4.2.1 Research Questions

This research is focused on answering two primary research questions.

Research Question 1: How source code metrics are correlated with quality attribute: class complexity?

This question reveals the relationships between complexity and source code metrics, such as the number of attributes, lines of code, etc. To answer this question, we apply statistical correlation on 18 source code metrics and complexity collected from 10 different source code repositories to find out the relationship.

Research Question 2: How accurately can machine learning approaches predict class complexity from source code metrics?

This question is targeted to find out the accuracy of machine learning approaches in class-level complexity detection. We apply 5 machine learning techniques and evaluate the performance. This question reveals the best technique for detecting class complexity from source code metrics.

4.2.2 Proposed Research Framework

The proposed research is built upon three steps. First, extracting source code metrics and complexity from classes of large code bases. Second, prepare the dataset for complexity prediction by applying the data cleaning process. Third, apply ML techniques and evaluate to find out the best one.

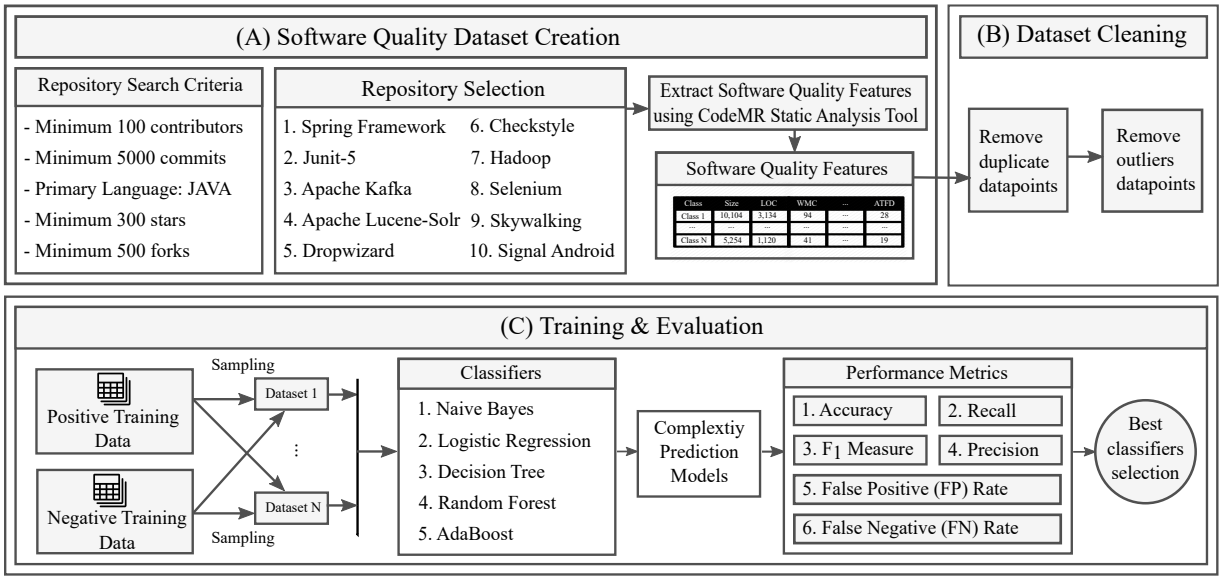


Figure 4.1: Proposed Methodology

For the first step, we extract source code metrics and quality feature: complexity from a large number of classes. The details of the dataset creation process are discussed in subsection 4.2.3. In the second step, we apply the data-cleaning process to get a better-

learned ML model. Uncleaned data fed into machine learning techniques may result in a bad model creation [111]. The details of the process are discussed in subsection 4.2.4. For the final step, we select several ML techniques and train the dataset to detect highly complex classes. We also assess ML prediction effectiveness using performance metrics. The detailed picture of the study is shown in Figure 4.1.

Table 4.1: Selected Repositories with Metadata Information

Serial	Repository Name	Commits	Contributors	Stars	Forks	Lines of Code	Classes
1	Spring Framework	21154	491	38200	25800	232447	5628
2	Junit5	6286	146	4000	899	16856	659
3	Apache Kafka	7787	691	16300	8700	119299	2463
4	Apache Lucene-Solr	33899	194	3600	2500	602185	8850
5	Dropwizard	5448	345	7700	3200	14268	508
6	checkstyle	9408	232	5400	7400	26030	454
7	Hadoop	24001	280	10600	6600	695992	10496
8	selenium	25354	518	18100	5800	36031	1175
9	skywalking	5753	245	14000	4100	61588	2531
10	Signal-Android	5777	206	13400	3400	116268	2861

4.2.3 Dataset Collection

The dataset for complexity prediction needs a diverse set of repositories. We search code-base repositories using ModelMine tool [8] with the following criteria; a repository with primary language Java, a minimum of 5000 commits, at least 100 active contributors, a minimum of 3000 stars, and 500 forks. The selected repositories are shown in Table 4.1 with repository metadata information.

To validate the diversity of repositories, we consider a high number of stars and forks as a proxy for the popularity of repositories and a high number of commits as a proxy for maintenance. Also, we consider repository size as follows: low (1-1000 classes), medium (1001-5000 classes), and high (more than 5000 classes) in size. This selection implies

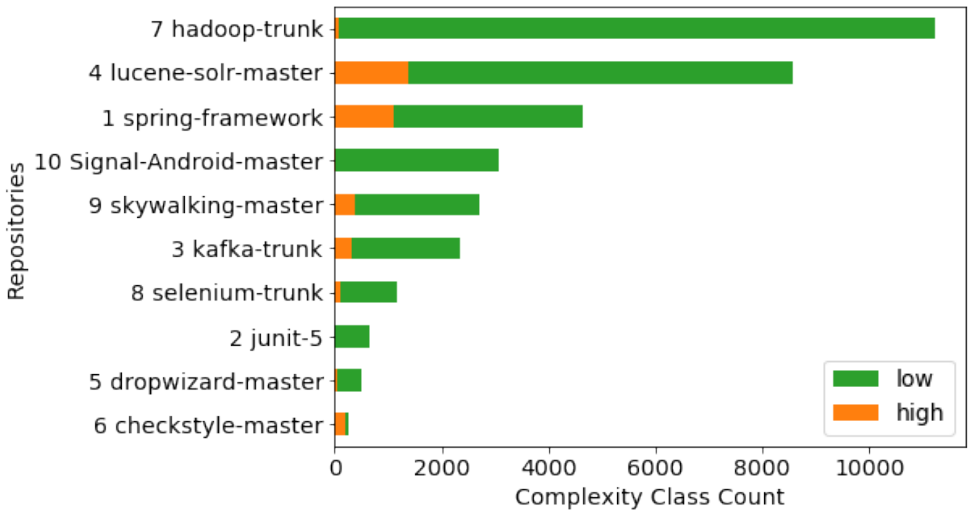
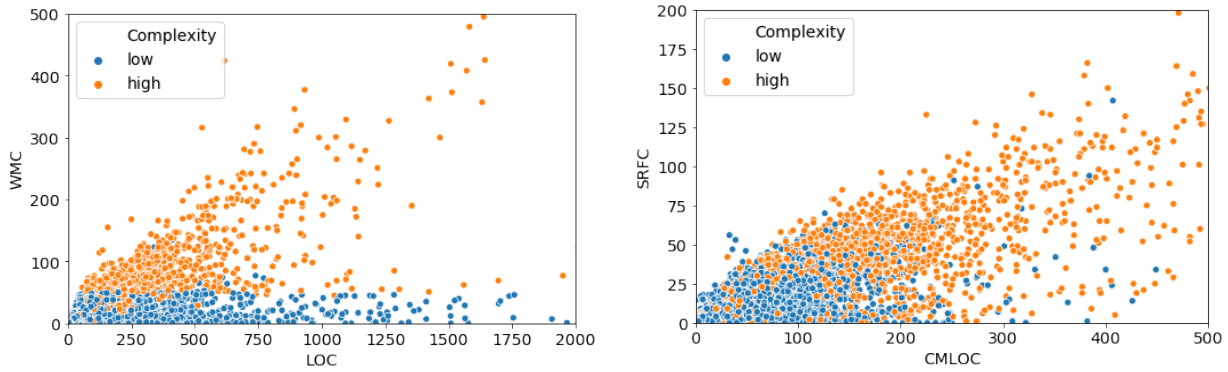


Figure 4.2: Complexity Distribution among Repositories

diversity in the complexity of classes. Figure 4.2 shows the number of complexity classes against each selected repository where 3 of them are selected from low, 4 of them are selected from medium, and the rest of them are selected from the high volume of category.



(a) WMC, LOC vs Complexity

(b) SRFC, CMLOC vs Complexity

Figure 4.3: Relationship of Input Variables with Target Variable

After extracting code repositories, we extract source code metrics for each class in the repository using the CODEMR tool [4]. The tool provides 18 unique source code metrics for each class. The details of the source code metrics are described in Table 4.2. The target

variable data is collected also for each class using the same tool with different processes. The data is then combined using the class file name for training and testing purposes.

Table 4.2: Source Code Metrics Used in this Study

No	**Source Code Metric Name**	**Description**
1	Class Lines of Code (CLOC)	The number of all non-commented and nonempty lines of a class.
2	Weighted Method Count (WMC)	The weighted sum of all class' methods.
3	Depth of Inheritance Tree (DIT)	The location of a class in the inheritance tree.
4	Number of Children (NOC)	The number of associated sub-classes of a class.
5	Coupling Between Object Classes (CBO)	The number of classes that another class is coupled to.
6	Response for a Class (RFC)	The number of the methods that can be potentially invoked in response by an object of a class.
7	Simple Response For a Class (SRFC)	The number of the methods that can be potentially invoked in response by an object of a particular class.
8	Lack of Cohesion of Methods (LCOM)	Measure how methods of a class are related to each other.
9	Lack of Cohesion Among Methods (LCAM)	Measure cohesion based on parameter types of methods.
10	Number of Fields (NOF)	The number of fields (attributes) in a class.

No	Source Code Metric Name	Description
11	Number of Methods (NOM)	The number of methods in a class.
12	Number of Static Fields (NOSF)	The number of static fields in a class.
13	Number of Static Methods (NOSM)	The number of static methods in a class.
14	Specialization Index (SI)	Measures the extent to which sub-classes override their ancestor's classes.
15	Class-Methods Lines of Code (CMLOC)	The total number of all nonempty, non-commented lines of methods inside a class.
16	Number of Overridden Methods (NORM)	The number of methods that are inherited from a superclass and have return type as the method that it overrides.
17	Lack of Tight Class Cohesion (LTCC)	Measures cohesion between the public methods of a class and subtract from 1.
18	Access to Foreign Data (ATFD)	The number of classes whose attributes are directly or indirectly reachable from the class.

4.2.4 Dataset Cleaning & Analysis

Data cleaning is a critically important step for complexity prediction. To get optimal performance results of ML approaches, we clean the data in two stages. First, by identifying column variables that have a single value or very few unique values. In this stage, we also remove the duplicate observations. In the second stage, we apply a box plot for each source code metric and find the outliers. This technique helps to remove the biased data points from the dataset.

After cleaning the dataset, we have come up with a much more differential and clear

dataset for complexity prediction. Figure 4.3a visualizes the relationship between weighted method count, lines of code, and complexity. Figure 4.3b visualizes the relationship between response for class, method lines of code, and complexity.

4.2.5 Machine Learning Classifiers & Evaluation Metrics

This subsection provides a brief overview of five alternative machine learning classifiers used to build class complexity predictors. The machine learning classifiers are as follows: (1) Naive Bayes (NB), (2) Logistic Regression (LR), (3) Decision Tree (DT), (4) Random Forest (RF), and (5) Ada Boost (AB). These classifiers are well-known classifiers in building vulnerability predictors and used in several similar research [2, 112, 113]. The statistical performance of selected ML classifiers is calculated by performing a 10-fold cross-validation technique. Cross-validation is a technique for assessing how accurately a predictive model will perform in practice after generating the model [114]. The objective of such an operation is to reduce the variability of the results.

4.3 Result and Discussion

This section describes the results of correlation analysis, and complexity prediction using ML models and compares the performance of ML classifiers.

4.3.1 Correlation Results

The results of Pearson correlation reveal the impact of source code metrics on quality attribute: complexity. Figure 4.4 visualizes the correlation between source code metrics and complexity. It is clear in the figure that not any single metric highly impact on complexity. This quality attribute is formed based on a combined behavior of source code metrics. Among the code metrics, DIT, SRFC, RFC, WMC, CMLOC, and CBO have a moderately high impact on complexity. Generally, classes with a higher number of WMC, LOC, or

DIT are associated with a high number of defects in the software and it becomes hard to maintain over time. [110]. This issue is also mentioned by Subramanyam et. al. that DIT and CBO have influenced class complexity [110]. In another research, Chowdhury et. al. experimentally showed that WMC, DIT, RFC, and CBO code-level metrics are strongly correlated to vulnerabilities that are directly generated from file complexity [51]. This answers research question 1.

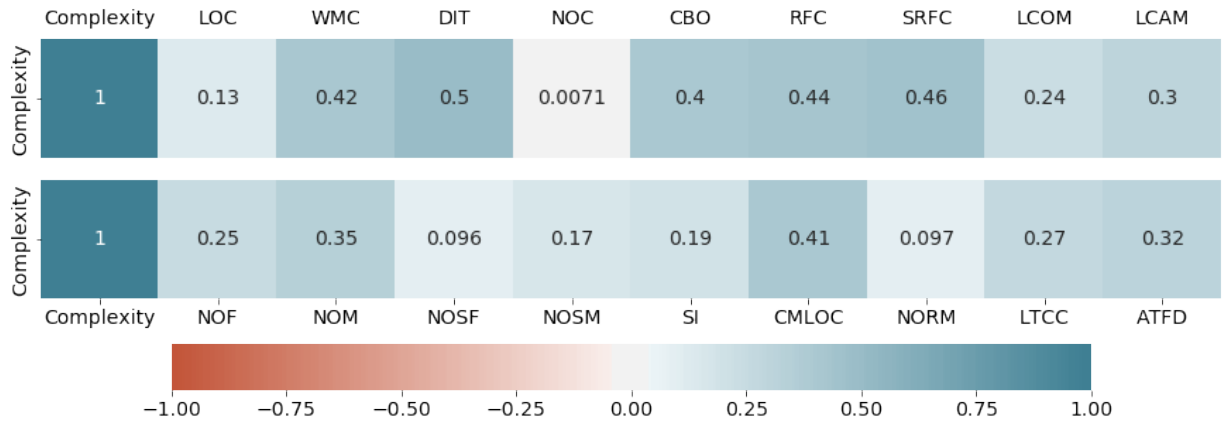


Figure 4.4: Correlation among Source Code Metrics and Quality Attribute

4.3.2 Performance Results

In this subsection, we discuss the performance of ML complexity predictors. We use the following evaluation metrics: accuracy, precision, recall, F_1 score, FP rate, and FN rate to compare the performances. At first, we generate confusion matrices from the validation set. Table 4.3 visualizes the confusion matrices of the classifiers for predicting software complexity.

We evaluate the techniques using the following metrics: accuracy, precision, recall, F_1 score, FP, and FN rate, and the results are visualized in Table 4.4. Accuracy and precision are the most used measurements in comparing performance. Table 4.4 and Figure 4.5 show the accuracy and precision value of the selected classifiers. The result implies Decision Tree

Table 4.3: Confusion Matrices of Classifiers for Predicting Software Complexity

Classifier Names→	**Naive Bayes**		**Logistic Regression**		**Decision Tree**		**Random Forest**		**Ada Boost**	
Predicted→ Actual ↓	Low	High	Low	High	Low	High	Low	High	Low	High
Low	6416	475	6766	125	6832	59	6820	71	6813	78
High	330	434	173	591	223	541	58	706	82	682

& Random Forest classifier has the highest accuracy and precision than other classifiers. We also observe Random Forest has the highest recall & F_1 score.

Table 4.4: Performance of Machine Learning Models

Serial	**Classifier Name**	**Accuracy**	**Precision**	**Recall**	**F1 Score**	**FP Rate**	**FN Rate**
1	Naive Bayes	89	71	75	73	6.88	42.11
2	Logistic Regression	96	91	86	88	1.44	26.08
3	Decision Tree	98	95	96	96	0.90	7.53
4	Random Forest	98	95	99	97	1.00	1.95
5	Ada Boost	97	94	93	94	1.13	12.27

However, we evaluate the classifiers with another set of metrics: false positive rate and false negative rate. The higher the FN rate, the model generates more false alarms. This implies high complex classes are detected as low complex classes which are very risky. Figure 4.6 shows the relative performance of classifiers in terms of false positive rate and false negative rate. One may have to tolerate many false positives to ensure a reduced number of complex classes left undetected. As such, if the target is to predict a larger percentage of high-complexity class files, then the Naive Bayes classifier can be evaluated favorably although, in overall prediction, Random Forest and Decision Tree classifier performance are better.

On the other hand, if the target is to predict a fewer percentage of highly complex files as low to avoid risk, then obviously Random Forest might be a good choice as it has

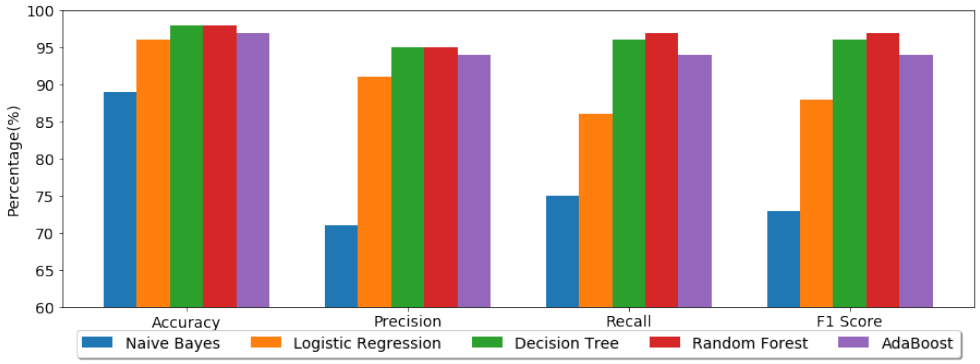


Figure 4.5: Relative Performance of ML Classifiers

the lowest false negative rate. We focus more on false negative rates to reduce the risk of detecting high complex classes as low. RF results indicate that it is a much better model for the prediction of complexity because of its bootstrapping random re-sample technique and working with significant elements. On the other hand, DT is working with all elements as a result it creates more false alarms than RF. Therefore, Random Forest is the best complexity predictor among selected ML techniques.

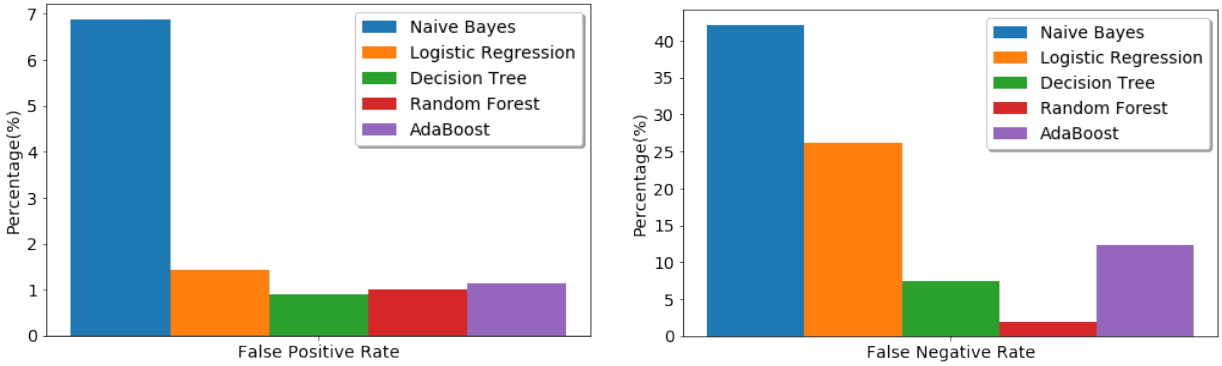


Figure 4.6: Relative FP and FN Rate of ML Classifiers

4.4 Conclusion

In this study, we analyze the software source code metrics which are most impacted the class complexity. It is undoubtedly necessary to take proper action before classes become more complex. Otherwise, it will become more expensive to test and fix if many classes become highly complex. To reduce such risk and cost, it is necessary to build a complexity predictor.

We start with extracting 38,778 classes of the dataset with 18 source code metrics, we use five different machine learning approaches to train the dataset to classify high or low-complex classes. In evaluation, we compare the performance of the approaches using the evaluation metrics. The result shows that the RF classifier predicts high complexity classes with an accuracy of 98% and also has the lowest FN rate of 1.95. Therefore, Random Forest is considered the best classifier to detect class complexity. In summary, we have made the following observation from our study. First, cross-validation implies a low variance of performance metrics detecting software complexity. Second, the FN rate needs to be reduced as much as possible to avoid the risk of detecting a high-complex class as a low-complex class.

Finally, the observations and results from this study can be useful in software quality research. Using ML automatic prediction on code quality will allow quality managers and practitioners to take preventive actions against bad quality, faults, and errors. Such proactive actions will allow software redesign and maintenance, ensuring better software quality during development.

Chapter 5

Evaluating the Accuracy of Machine Learning Algorithms for Code Smells Detection

This chapter presents research on code smells of handwritten code (HC) in model-driven engineering (MDE) software repositories, finds code smells and ML approaches in recent studies, and compares the performance of neural network-based ML approaches with traditional ML approaches for code smell detection. The chapter examines the hypothesis that the quality of HC developed in the context of MDE is negatively affected by unique constraints, such as the need to integrate with automatically generated code and artifacts. This chapter is related to published research in the 9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2021) [7] and submitted research in the WILEY Journal of Software: Practice and Experience (WILEY SPE).

In model-driven engineering (MDE) software projects, large portions of the executable code are automatically generated from designs and models. This generated code may or may not be edited by the developers to achieve their development objectives. MDE projects also include a significant amount of handwritten code (HC). This handwritten code is developed under unique constraints, as it must integrate with generated artifacts and code elements that are not directly developed by the engineers. These constraints adversely affect codebase quality and maintainability. This case study aims to investigate the hypothesis pertaining to the handwritten code quality developed in the context of MDE. The study analyzes these unique code fragments and compares their characteristics

to handwritten code in repositories where code generation is not present. The study finds that handwritten code quality in the MDE context suffers from elevated technical debt and code smells. We observe key code smells that are particularly evident in this handwritten code. These findings imply that code generators must optimize for human comprehension, prioritize extensibility, and must facilitate integration with handwritten code elements.

Recent studies on code smells and machine learning approaches have shown promising results in identifying and addressing software quality issues. Code smells are indicators of potential problems in code that can lead to bugs and technical debt. This chapter investigates the uses of different types of code smells and ML approaches in recent studies and reports all the details. The result helped to find the most used code smells and ML approaches for code smell detection. We compare the performance of the most used neural network-based ML approaches with the most used traditional ML approaches for code smell detection. The results show that neural network approaches are performing better than traditional ML approaches on average in classifying code smell.

5.1 Introduction

Model Driven Engineering (MDE) envisions software development teams that focus primarily on developing models that would generate all executable artifacts. This vision seems to have been realized only in organizations that have invested in infrastructures to support domain-specific modeling languages and custom code generators that produce all or most of the required executables. These organizations can afford the overhead to support the development of compilers, code generators, and custom-built design languages. Software modeling is undoubtedly a core activity in software development. The precise form of modeling varies from whiteboard sketches to models that support code generation. Further, modeling in some form is a fundamental part of designing, understanding, communicating, and analyzing software-heavy systems [115].

Today, many MDE practitioners generate only a portion of the required executable

artifacts. In these cases, engineers often write code that integrates with and extends the generated code. This handwritten code is unique for many reasons. The code must integrate with generated artifacts that may not be well-suited for integration. Code generators often do not follow coding conventions and frequently generate counter-intuitive code that may not be comprehensible [116]. Moreover, the originating models and their code generators may not be designed to prioritize extensibility; further complicating the engineers' tasks [97].

In addition to the generated code and the handwritten code categories in MDE projects, developers often modify code that was originally generated from models. This modified code category is also unique; the code is neither written from scratch nor purely generated. Software engineers are often constrained in the way they manipulate this code.

The goal of this study is to understand the quality characteristics of handwritten code. Specifically, the study aims to characterize the maintainability of handwritten code fragments in MDE projects. We investigate the hypothesis that handwritten code in MDE contexts suffers from unique deficiencies that have a significant impact on its maintainability.

The rest of the chapter is organized as follows. A background pertaining to MDE projects and code smells is discussed in Section 5.2. The study design is presented in Section 5.3. Detailed results and discussion are presented in Sections 5.4 and 5.5 respectively. The Threats to Validity are discussed in Section 5.6 and we conclude the chapter in Section 5.7.

5.2 Background

Model Driven Engineering (MDE) is a software development approach that emphasizes the use of models to capture software design and implementation details. In MDE, models serve as the primary artifacts throughout the software development lifecycle, from requirements specification to implementation and testing. MDE has several benefits, including increased productivity, better quality software, and improved maintainability. By using models to

capture design and implementation details, MDE reduces the amount of time and effort needed to develop software systems. Additionally, MDE helps ensure that software is of high quality by allowing developers to verify and validate models before implementing them. Finally, MDE makes software easier to maintain by making it easier to modify and update models as needed.

The benefits of MDE are clear; models are much easier to comprehend and provide a better platform to support collaborations. Models tend to be more visual and can support designs at variable levels of abstractions [117]. Moreover, there is significant potential in improving software engineers' productivity and the quality of the code they develop by automatically generating executable artifacts. Today, only a few organizations have succeeded in achieving this vision. Many MDE adopters generate some artifacts and rely on software developers to extend the generated code. This handwritten code often consumes the majority of the maintenance efforts [118]. As such, understanding this code quality is fundamental to understanding the MDE value proposition.

The handwritten code in MDE projects is subject to unique constraints that can affect code quality both positively and negatively. First, integrating with generated artifacts is a negative impact on MDE. But on the other hand, having well-formed unambiguous designs that are part of MDE artifacts would affect code quality positively [119]. Therefore, in this study, we analyze the handwritten code in the MDE context with comparable code from two sets of repositories; those that include designs and those that do not. In this study, we collect Graphical Modeling Framework (GMF) and Eclipse Modeling Framework (EMF) based MDE projects because both of these categories are popular, mature, and stable MDE platforms with extensive code generating engines and customized templates [119]. The Graphical Modeling Framework (GMF) is a framework within the Eclipse platform. It provides a generative component and runtime infrastructure for developing graphical editors based on the Eclipse Modeling Framework (EMF) [120]. EMF's purpose is to allow data models to be created and then stored in an 'ecore' file. However, GMF's purpose is to translate existing EMF models and utilize GEF (Graphical Editing Framework) to build

a graphical editor automatically based on the content [121]. Projects that are developed using GMF/EMF platforms include three unique classes of code. 1) Generated, code that is generated exclusively from models. 2) Generated and Modified, code that is generated but then later modified by engineers. 3) Handwritten code, this is code developed manually by engineers that either extend or integrate with the previous two classes of code.

In this study, we hypothesize that code quality characteristics such as Code Smells (CS) and Technical Debt (TD) are elevated in the MDE environment. CS is any surface symptom in the source code that suggests deficiencies related to maintainability [122]. CS appears because of bad software design and programming practices and indicates that code refactoring may be required [123, 124]. TD is a metaphor that provides short-term benefits but may hurt long-term software maintainability. TD has both positive and negative impacts on software systems. When TD is incurred intentionally to achieve short-term benefits can be beneficial if the cost associated with TD is made visible and kept under control. However, unintentional TD could be detrimental to the maintenance of the software systems [125, 126].

5.3 Study Design

This chapter presents the research questions, study design, data collection process for code smells in handwritten code, and recent studies and the selection of ML approaches for comparison.

5.3.1 Research Questions

The research is motivated by the following research questions.

RQ1: What are the quality characteristics of handwritten code in the MDE context? How do these characteristics compare to handwritten code in Non-MDE contexts?

RQ2: What are the key code deficiencies in handwritten code in MDE projects? What are the most prevalent code smells and their severity?

RQ3: How does the Technical Debt accumulated in Handwritten code in the MDE context compare to Non-MDE contexts?

RQ4: What are the predominant code smells discussed in the recent literature on machine learning-based code smell detection?

RQ5: Which machine learning algorithms are used to identify code smells in the recent literature on code smell detection?

RQ6: How do neural network-based machine learning techniques for code smell detection perform when compared to traditional machine learning approaches?

5.3.2 Code smell Characteristics of Handwritten Code in MDE Projects

This study identifies 15 sub-systems (sources are listed in Table 5.1), 5 identified as MDE repositories (based on GMF/EMF framework), and 10 identified as Non-MDE repositories that are further classified under two classes. The repository selection process is visualized in Figure 5.1.

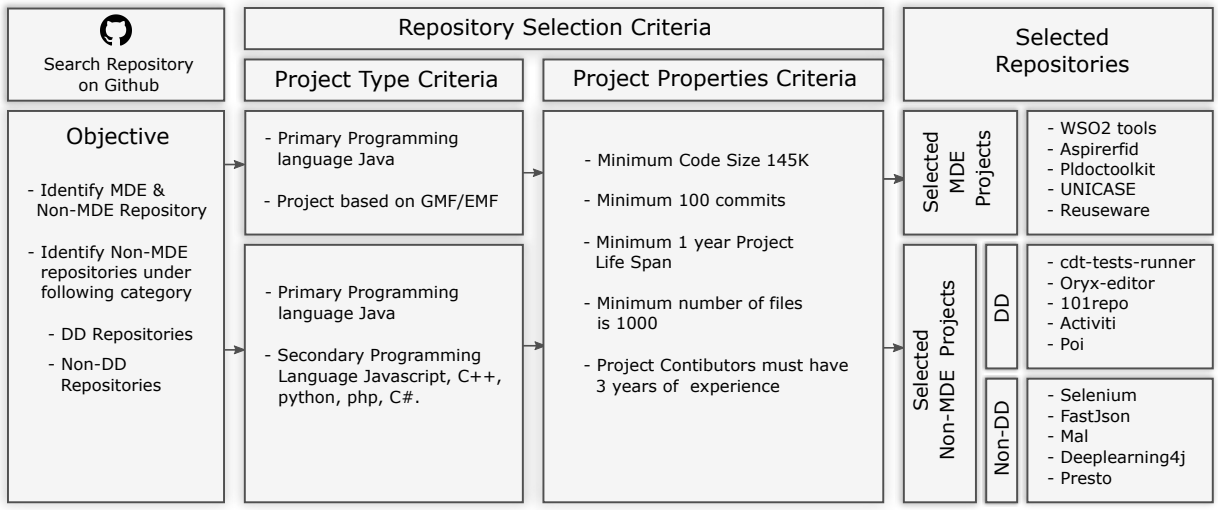


Figure 5.1: Repository Selection Process

The first five MDE sub-systems are selected from a pool of 16 MDE repositories that

Table 5.1: Selected Repositories Name, its Type, and URL

Type	**Project**	**URL**
MDE	WSO2 Tools	https://www.github.com/wso2-attic/tools.git
	aspirerfid	https://www.github.com/mouillerart/aspirerfid.git
	pldoctoolkit	https://www.github.com/spbu-se/pldoctoolkit.git
	UNICASE	https://www.github.com/unicase-ls1/unicase.git
	Reuseware	https://www.github.com/DevBoost/Reuseware.git
DD	cdt-tests-runner	https://github.com/xgsa/cdt-tests-runner
	Oryx-editor	https://github.com/andreaswolf/oryx-editor
	101repo	https://github.com/101companies/101repo
	Activiti	https://github.com/Activiti/Activiti
	Poi	https://github.com/apache/poi
Non-DD	Selenium	https://github.com/SeleniumHQ/selenium
	Fastjson	https://github.com/alibaba/fastjson
	Mal	https://github.com/kanaka/mal
	Deeplearning4j	https://github.com/deeplearning4j/deeplearning4j
	Presto	https://github.com/prestodb/presto

are reported in the study by He et. al. [119]. We select these repositories that meet the following criteria, each repository is GMF/EMF framework-based, code size greater than 145k lines of code (LoC), predominantly written in the Java object-oriented programming language, and the number of commits in GitHub is at least 100. This last condition is meant to exclude trivial projects.

To determine whether a project lies within the GMF/EMF category, we checked whether the project includes files with the extension *gmfgen*. The *gmfgen* extension is the generator model of GMF and from which the source code is derived. Since a GMF project may contain many sub-projects, only the sub-projects that are based on GMF/EMF are included in this

study. The details of the selection criteria of these 16 repositories are described in [119].

The next five repositories are identified as Design Driven sub-systems (DD) which are selected from a pool of 4,650 identified in [127] to be model-heavy repositories. These 4,650 repositories are selected by mining all GitHub repository artifacts that include UML and modeling elements [128]. From this list, we select the top 5 repositories that meet the following criteria: code size is greater than 145K lines of code, written predominantly in the Java object-oriented language, and have at least 100 commits in the GitHub repository.

The third set of 5 repositories are selected as reference repositories. These repositories are identified as Non-Design Driven (Non-DD). They are selected from the study by Badreddin et al. [94]. These repositories include similar object-oriented code size, number of commits, and similar programming languages and contributors' profiles. We ensure that the average expertise of the active contributors in this set is comparable to the expertise of the contributors of the identified repositories. For this, we collect profiling information of active contributors such as the history of their edits, and years of contribution in GitHub. Table 5.2 lists all 15 subject code repositories and the number of their identified files, commits, code size, and analyzed Lines of Code (LoC). The analyzed LoC column lists the lines of code that were analyzed in this study. This excludes non-object-oriented code and documentation.

5.3.3 Code Smells in Recent Studies

Article Search Strategy

A successful literature review synthesizes the existing research in a fair and seems fair manner. Kitchenham et al.[129] provided evidence of the importance of search strategy in a systematic literature review and how it affects the completeness of search results. We compiled a search process to collect all the published literature relevant to our study focus. Our search process is based on search keyword identification, resources to be searched, and article selection criteria. Figure 5.2 depicts a bird's eye view of our article selection process.

Table 5.2: Basic Information of Subject Software Repositories

Type	Repository	Commits	Code Size	File Category	No. of Files		Analyzed LoC	
					Count	%	Count	%
MDE	WSO2 Tools	2,609	1,009,000	GF	3,025	35.3	311,422	30.8
				MGF	615	7.2	149,801	14.8
				HC	4,934	57.5	550,709	54.6
	aspirerfid	341	145,000	GF	397	25.7	37,657	25.9
				MGF	55	2.8	3,124	2.15
				HC	1,105	71.5	99,147	68.4
	pldoctoolkit	493	182,000	GF	587	58.2	68,636	37.7
				MGF	102	10.1	14,537	7.9
				HC	320	31.7	30,076	16.5
	UNICASE	8,506	289,000	GF	3,202	54.6	406,819	59.7
				MGF	464	7.9	111,792	16.4
				HC	2,196	37.5	161,789	23.7
	Reuseware	104	526,000	GF	4,193	80.6	598,755	85.8
				MGF	107	2.1	25,414	3.6
				HC	903	17.4	73,912	10.6
DD	Cdt-tests-runner	19,589	1,003,261	HC	8,122		982,425	97.9
	Oryx-editor	2,022	640,127		2,887		543,704	84.9
	101repo	2,312	183,083		1,421		154,437	84.4
	Activiti	7,741	207,339		3,078		192,812	93.0
	Poi	9,157	450,906		3,575		427,326	94.8
Non-DD	Selenium	21,788	875,267	HC	4,150		775,268	88.6
	Fastjson	2,673	168,880		2,537		149,186	88.3
	Mal	2,249	178,870		1,567		166,296	93.0
	Deeplearning4j	9,301	283,711		2,062		221,711	78.1
	Presto	15,786	716,021		5,632		716,021	100

The following sections discuss the database, data query, and paper selection criteria in further detail.

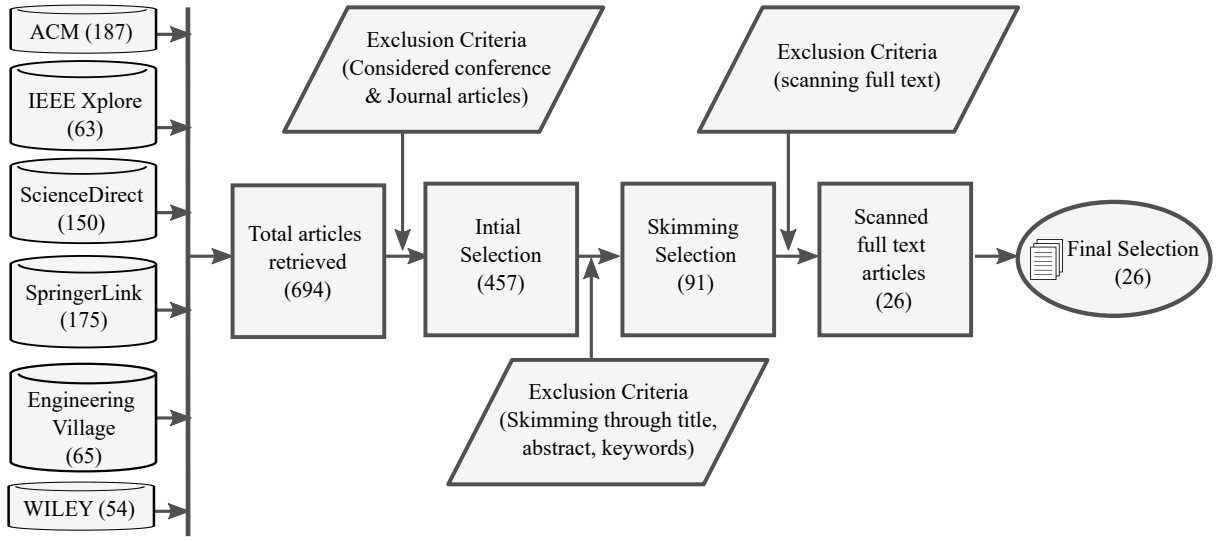


Figure 5.2: Article Selection Process

Identification of query keywords

To find relevant articles, we look for keywords related to our topic and synonyms of the keywords. Such a strategy was applied in similar research [79]. The alternative spelling and synonyms are given below:

- **Code Smells:** ("code smells" OR "code bad smells" OR "bad smells" OR antipatterns OR "design defect" OR "design-smells" OR "design flaw")
- **Machine Learning:** ("machine learning" OR "supervised learning" OR "unsupervised learning")
- **Detection:** (detection OR identification OR "prediction model")
- **Software:** (software OR "software engineering")

Due to the number of boolean term restrictions in some data sources, we have tweaked the number of keywords and created a smaller query string named *Query2*. As a result,

two different query strings have been devised to search. After combining all the keywords mentioned above, we created a query string named *Query1*, which is reported in Table 5.3.

Table 5.3: Query Strings Applied to Extract Articles

Serial	**Query Identifier**	**Query String**
1	Query1	(“code smells” OR “code bad smells” OR “bad smells” OR antipatterns OR “design defect” OR “design-smells” OR “design flaw”) AND (“machine learning” OR “supervised learning” OR “unsupervised learning”) AND (detection OR identification OR “prediction model”) AND (software OR “software engineering”)
2	Query2	(“code smells” OR “antipatterns” OR “design flaw”) AND (“machine learning”) AND (detection OR identification OR “prediction model”) AND (software OR “software engineering”)

Data sources

Selection of proper data sources is crucial for conducting a systematic literature review [130]. We consider six different databases as our data source. The detail of data sources with their URLs is shown in B.1. The query identifiers and their query string are listed in Table 5.3. Raw query strings applied as search queries are used to extract the potential literature from the data sources. Table 5.4 shows the number of articles found in data sources in each stage. Note that, the data sources are popular and recognized for software engineering research and are used in many SLRs [79].

Article Selection Criteria

We take four steps for identifying primary studies. The steps described below are taken to collect articles.

Table 5.4: Data Sources and Search Results

Serial	**Data source**	**Query identifier**	**Total results found**	**Skimming selection**	**Scanning selection**	**Final selection**
1	ACM Digital Library	Query1	187	151	25	7
2	IEEE Xplore Digital Libray	Query1	63	47	13	9
3	ScienceDirect	Query2	150	105	6	3
4	SpringerLink Digital Library	Query2	175	52	6	1
5	Engineering Village Digital Library	Query1	65	62	32	5
6	Wiley Online Library	Query1	54	40	9	1
Total			**694**	**457**	**91**	**26**

1. Step 1: Query string from Table 5.3 is applied to search relevant articles. We have added the date constraint(2015-2021) in our search process. The search results produced 694 articles in total against the query strings. The detailed search results for each data source are reported in Table 5.4.

2. Step 2: Starting from the entire 694 lists, we consider the articles from the journal, conference, symposium, and workshops, which resulted in a total of 457 publications in the list of articles. Reports and white papers are excluded from this study.

3. Step 3: Having the previous list, we skim through the title, abstract, and keywords. We excluded the articles that did not mention machine learning or reported code smells. The output of this exclusion compressed the list into 91 research articles.

4. Step 4: In this step, we scrutinize all 91 articles and attempt to find out the following criteria: code smell, dataset, independent variables, machine learning approaches, and evaluation metrics considered or not. We finalize the list of 26 research articles that cover the criteria mentioned.

Table 5.5 shows 26 primary studies considered in this study. In summary, all these selected primary studies are from six data sources, published between 2015 and 2021, and published in conferences, journals, symposiums, or workshops. To make it easy for the future researcher, we list all the exclusion and inclusion criteria in the following sections.

Table 5.5: Reviewed Articles in the Literature

Code	**Source**	**Article Title**	**Author**	**Year**	**Type**
S01	ACM	Detecting bad smells with machine learning algorithms: an empirical study	Cruz et al.	2020	Conference
S02		Machine learning techniques for code smells detection: an empirical experiment on a highly imbalanced setup	Luiz et al.	2019	Conference
S03		Sniffing Android code smells: an association rules mining-based approach	Rubin et al.	2019	Conference
S04		Comparing heuristic and machine learning approaches for metric-based code smell detection	Pecorelli et al.	2019	Conference
S05		Smells are sensitive to developers! on the efficiency of (un)guided customized detection	Hozano et al.	2017	Conference
S06		Deep semantic-Based Feature Envy Identification	Guo et al.	2019	Symposium
S07		On the role of data balancing for machine learning-based code smell detection	Pecorelli et al.	2019	Workshop

Code	Source	Article Title	Author	Year	Type
S08	IEEE Xplore	An Empirical Framework for Code Smell Prediction using Extreme Learning Machine	Gupta et al.	2019	Conference
S09		Code Smells Analysis Mechanisms, Detection Issues, and Effect on Software Maintainability	Lafi et al.	2019	Conference
S10		Assessment of Code Smell for Predicting Class Change Proneness Using Machine Learning	Pritam et al.	2019	Journal
S11		Comparison of Machine Learning Methods for Code Smell Detection Using Reduced Features	Hadziabdic et al.	2018	Conference
S12		A Support Vector Machine Based Approach for Code Smell Detection	Kaur et al.	2017	Conference
S13		Comparison of Multi-Label Classification Algorithms for Code Smell Detection	Kiyak et al.	2019	Symposium
S14		Comparing Heuristic and Machine Learning Approaches for Metric-Based Code Smell Detection	Pecorelli et al.	2019	Conference
S15		Identification of Code Smell Using Machine Learning	Jesudoss A. et al.	2019	Conference

Code	Source	Article Title	Author	Year	Type
S16		Detecting code smells using machine learning techniques: Are we there yet?	Di Nucci et al	2018	Conference
S17	Science Direct	Code smell severity classification using machine learning techniques	Fontana et al.	2017	Journal
S18		A large empirical assessment of the role of data balancing in machine-learning-based code smell detection	Pecorelli et al.	2020	Journal
S19		A machine-learning based ensemble method for anti-patterns detection.	Barbez et al.	2020	Journal
S20	Springer Link	Comparing and experimenting machine learning techniques for code smell detection	Fontana et al.	2015	Journal
S21	Engineering Village	Using developers' feedback to improve code smell detection	Hozano et al.	2016	Symposium
S22		Predicting Code Smells and Analysis of Predictions: Using Machine Learning Techniques and Software Metrics	Mhawish et al.	2020	Journal
S23		Evaluating the Accuracy of Machine Learning Algorithms on Detecting Code Smells for Different Developers	Hozano et al.	2017	Conference

Code	Source	Article Title	Author	Year	Type
S24		Applying Machine Learning to Customized Smell Detection: A Multi-Project Study	Oliveira et al.	2020	Conference
S25		Improving performance with hybrid feature selection and ensemble machine learning techniques for code smell detection	Jain et al.	2021	Journal
S26	Willey	MARS: Detecting brain class/method code smell based on metric–attention mechanism and residual network	Zhang et al.	2021	Journal

Article Exclusion

This section presents how we exclude articles that are not directly relevant to this literature review. There are some constraints reported as part of our exclusion criteria. The criteria are listed below.

- Articles that were published outside the following areas: journal, conference, symposium, and workshop.
- Articles that were not written in English.
- Articles that did not have the full text on the web.
- Articles that were based on machine learning but not related to software engineering.
- Articles that were related to code smell detection, but no ML techniques were applied.

Article Inclusion

This section presents how we include articles relevant to this literature review. The criteria for inclusion of papers are listed as follows:

- Articles that were published between 2015 and 2021.
- Articles that were purely written in English.
- Articles that reported evaluation metrics of machine learning techniques for code smell detection.

The reason behind choosing articles between 2015 and 2021 is to consider missing papers addressed in SLR by Azeem et al.[79] who conducted reviews between 2000 and 2017 on the code smell domain.

Metadata collection

After collecting all the articles, we perform metadata analysis on the selected articles to answer the research questions. The metadata we collect from the data extraction is listed in Table B.2. We ensure the data extraction process with some questions. It provides better data quality assessment during data extraction [131, 132]. The questions we set are as follows:

- Are the code smell considered in the research article clearly mentioned?
- Is the dataset information clearly stated?
- Are the independent variables clearly stated?
- Are the machine learning approaches clearly defined?
- Are the evaluation metrics for selected machine learning clearly reported?

The process is applied for each article and finalized the data to answer research questions. Each of these questions was clearly answered with either "Yes" (1), "No" (0), or "Somewhat" (0.5). We evaluate the "somewhat" in cases where the results can be derived methodologically from the given results, even if they were not clearly stated. For example, a research article clearly mentioned accuracy, precision, and recall, not F-score. In those cases, F-score is possible to derive from the accuracy, precision, and recall values.

5.3.4 Data Collection for Code Smells in MDE Projects

MDE repositories contain three types of files: Generated Files (GF), Modified Generated Files (MGF), and Handwritten Code (HC). In this study, we extract handwritten code files from the selected MDE repositories by carefully excluding GF and MGF. This process is achieved by a script [133] whose results were independently verified.

Study Variables

For each project, we consider 12 variables that directly relate to our research questions. The variable's description and relation with the research question are listed in Table 5.6. The first six variables ($\#F_{MDE}$, $\#F_{DD}$, $\#F_{NDD}$, $\#LOC_{MDE}$, $\#LOC_{DD}$, $\#LOC_{NDD}$) are selected under file & code metrics to compare the relationship between MDE & Non-MDE repositories code quality.

The variables ($\#CS_{MDE}$, $\#CS_{DD}$, $\#CS_{NDD}$) represents CS value for each MDE, Design Driven (DD) and Non-Design Driven (Non-DD) repositories. These variables provide total occurrences of all CS in handwritten code to help answer the second research question.

The last three variables ($\#TD_{MDE}$, $\#TD_{DD}$, $\#TD_{NDD}$) are related to the third research question and refer to TD in the selected repositories in MDE, DD, and Non-DD repositories respectively.

We construct two complex variables related to density for further analysis in this study. These variables are ($\#CSD_{MDE}$, $\#CSD_{DD}$, $\#CSD_{NDD}$, $\#TDD_{MDE}$, $\#TDD_{MDE}$,

$\#TDD_{MDE}$) and are described as Code Smell Density (CSD) and Technical Debt Density (TDD) in MDE, DD, and Non-DD repositories respectively. The variables are constructed by using the equations below.

$$\#CSD_X = \frac{\#CS_X}{\#LOC_X} \tag{5.1}$$

$$\#TDD_X = \frac{\#TD_X}{\#LOC_X} \tag{5.2}$$

Where X represents MDE or DD or Non-DD repositories.

Metrics and Thresholds

Metrics and thresholds are uniform for all subject repositories as listed in Table 5.6 and 5.8. We develop a custom program [133] that can read all the files & folders from the MDE repositories iteratively using java program extension and constructs an array of files and directories by filtering *.JAVA* or *.java* extension. To identify handwritten code files from previous filtered results, we determine which files are GF and which files are MGF. We classify the files that do not belong to Generated or Modified Generated as handwritten code files. The classification process of the files is followed by some search criteria which are shown in table 5.7.

Code Quality Metrics

This section describes code smells and technical debts that asses the code quality of the subject code repositories.

Code Smell: We use PMD [134] a source code analysis tool to identify Code Smells. We select six types of CS as listed in Table 5.8, which includes *God Class*, *Excessive Class Length*, *Excessive Method Length*, *Duplicate Code*, *Cyclomatic Complexity*, and *Excessive Imports*. The details of these CS can be found in PMD tool documentation [134]. These CS are selected because they are frequently used in literature [125] [135] as TD indicators. For

Table 5.6: Variables Description

Variables	**Description**	**Research Question (RQ)**
$\#F_{MDE}$	Total number of handwritten code files in Model Driven Engineering repositories	RQ1
$\#F_{DD}$	Total number of files in Design Driven repositories	RQ1
$\#F_{NDD}$	Total number of files in Non-Design Driven repositories	RQ1
$\#LOC_{MDE}$	Total number of lines of code in Model Driven Engineering repositories	RQ1
$\#LOC_{DD}$	Total number of lines of code in Design Driven repositories	RQ2
$\#LOC_{NDD}$	Total number of lines of code in Non-Design Driven repositories	RQ2
$\#CS_{MDE}$	Total Code Smells in Model Driven Engineering repositories	RQ3
$\#CS_{DD}$	Total Code Smells in Design Driven repositories	RQ3
$\#CS_{NDD}$	Total Code Smells in Non-Design Driven repositories	RQ3
$\#TD_{MDE}$	Total Technical Debt in Model Driven Engineering repositories	RQ3
$\#TD_{DD}$	Total Technical Debt in Design Driven repositories	RQ3
$\#TD_{NDD}$	Total Technical Debt in Non-Design Driven repositories	RQ3

instance, *God Class*, *Duplicate Code*, and *Cyclomatic Complexity* are related TD, which influence the maintainability of source code [119].

In this study, all the CS are measured by the PMD tool except for *Duplicate Code*. Duplicated Code Smell and its density are measured by SonarQube by identifying duplicated block counts of a project divided by physical lines of code. Other CS densities are measured by the CS counts divided by analyzed lines of code and multiplied by 100. This

Table 5.7: File Search Criteria

File Category	**Search Criteria**
Generated Files (GF)	Search in all files by these strings: '@generated', '@Generated'
Modified Generated Files (MGF)	Search in all files by these strings: '@generated NOT', '@generated not', '@Generated not', '@Generated NOT'
Handwritten Code (HC)	The files that do not belong to Generated or Generated & Modified are considered as handwritten code files.

density refers to the number of CS per line of code.

Table 5.8: Detected Types of Code Smells

No.	**Code Smell**	**Threshold**
1	Large Class	1000 LOC
2	Large Method	100 LOC
3	Excessive Imports	30 imports
4	God Class	N/A
5	Cyclomatic Complexity	10
6	Duplicate Code	100 duplicated blocks

Technical Debt: TD of subject software repositories are measured using the source code analysis tool SonarQube. SonarQube computes TD based on the Software Quality Assessment which is based on Lifecycle Expectations methodology (SQALE) [136]. The SQALE is a methodology that organizes non-functional requirements related to code quality. Non-functional requirements are realized in terms of coding rules and issues in the SonarQube implementation of the SQALE method. The details of this TD calculation by SonarQube can be found in SonarQube documentation [137].

We perform similar calculations to measure TD density by dividing the TD counts by

Table 5.9: Code Smells & Technical Debt Results

Type	Repository	Analyzed LoC	Code Smells							Technical Debt (Days)
			Large Class	Large Method	Code Clone	Excessive Imports	God Class	Cyclomatic Complexity	Total	
MDE	WSO2 Tools	550,709	78	636	5,600	371	414	1,872	**8,971**	741
	aspirerfid	99,147	10	103	1,000	40	40	257	**1,450**	166
	pldoctoolkit	30,076	1	24	349	24	15	97	**510**	50
	UNICASE	161,789	8	49	1,281	122	49	297	**1,806**	148
	Reuseware	73,912	5	44	794	18	62	217	**1,140**	100
	Total	**915,633**	**102**	**856**	**9,024**	**575**	**580**	**2,740**	**13,877**	**1,205**
	Average	**183,127**	**20**	**171**	**1,805**	**115**	**116**	**548**	**2,775**	**241**
DD	Cdt-tests-runner	982,425	150	477	9,535	69	319	1,619	**12,169**	1,200
	Oryx-editor	543,704	16	28	16,991	30	62	277	**17,404**	486
	101repo	154,437	1	15	2,475	0	6	43	**2,540**	386
	Activiti	192,812	19	77	890	40	66	302	**1,394**	122
	Poi	427,326	96	311	1,628	131	238	1,362	**3,766**	322
	Total	**2,300,704**	**282**	**908**	**31,519**	**270**	**691**	**3,603**	**37,273**	**2,516**
	Average	**460,141**	**56**	**182**	**6,304**	**54**	**138**	**721**	**7,455**	**503**
Non-DD	Selenium	775,268	3	8	10,104	71	11	92	**10,289**	217
	Fastjson	149,186	23	103	1,955	10	25	341	**2,457**	196
	Mal	166,296	0	2	3,075	0	4	27	**3,108**	415
	Deeplearning4j	221,711	79	374	2,699	184	160	1,381	**4,877**	720
	Presto	716,021	57	181	2,536	744	114	693	**4,325**	420
	Total	**2,028,482**	**162**	**668**	**20,369**	**1,009**	**314**	**2,534**	**25,056**	**1,968**
	Average	**405,696**	**32**	**134**	**4,074**	**202**	**63**	**507**	**5,011**	**394**

analyzed lines of code and multiplying by 100. This density refers to the number of TD per line of code. In other words, the number of TD is the total number of days it will take to fix an issue per line of code.

In Figure 5.3 and 5.4, R1, R2...R5 represents a set of 3 types of the repository that includes MDE, Design Driven (DD), and Non-Design Driven (Non-DD) respectively. This repository set selection process for R1, R2...R5 has been conducted sequentially. For instance, the MDE repository *WSO2 Tools* is selected with *Cds-test-Runner* and *Selenium* from DD and Non-DD repository lists respectively.

5.3.5 Machine Learning Approaches in Code Smell Detection

In the selection of machine learning approaches, we consider 5 traditional and 5 neural network-based approaches.

Traditional ML-Based Code Smells Detection

While several traditional ML classifiers have been previously used for code smell detection, it is still unclear which of the traditional MLs represents the best model for code smell detection. For this reason, in this work, we have used the following traditional ML algorithms: Naive Bayes, Decision Tree, Random Forest, Support Vector Machine, and Logistic Regression with a recently extracted data set deployed between 2016 to 2021.

Neural Network ML-Based Code Smells Detection

As there is a growing trend of applying neural network-based models in code smell research, we have decided to apply some of the models to detect code smells. In this research, we have applied the following neural network-based models: Multilayer Perceptron, Convolutional Neural Network [138], Long Short-Term Memory, Recurrent Neural Network, and Artificial Neural Network. To perform a fair comparison, we applied the same balancing configuration, pre-processing and training strategies to all the machine learning models.

5.4 Results

Our assessment criteria are based on two primary measurements; measurements of CS and measurements of TD. In the following, we report on these two measurements.

5.4.1 Results Based on Code Smells (RQ1 & RQ2)

The total number of Code Smells in handwritten code in the MDE context are significantly reduced as shown in Table 5.9. The total number of CS increases with code size metrics

Table 5.10: Selected Machine Learning Approaches

Serial	**Machine Learning Algorithm**	**Type**
1	Naive Bayes (NB)	Traditional
2	Decision Tree (DT)	
3	Random Forest (RF)	
4	Support Vector Machine (SVM)	
5	Logistic Regression (LR)	
6	Multilayer Perceptron (MLP)	Neural-Network
7	Convolutional Neural Network (CNN)	
8	Long Short-Term Memory (LSTM)	
9	Recurrent Neural Network (RNN)	
10	Artificial Neural Network (ANN)	

in any type of repository. we found that handwritten code in MDE contexts is associated with reduced CS ($\#CS_{MDE} < \#CS_{DD}$ & $\#CS_{MDE} < \#CS_{NDD}$). Since the number of CS are associated with elevated values when the code size increases, we calculate the frequency of CS for each repository, and we formulate normalized CS metrics as CS density.

Figure 5.3 illustrates the results of six CS densities of MDE HC and Non-MDE repository code. Figure 5.3(A) to Figure 5.3(F) illustrates Large Class, Large Method, Duplicate Code, Excessive Imports, God Class, and Cyclomatic complexity CS density respectively. In addition, we report on pairwise comparative analysis of CS in MDE handwritten code and Non-MDE repository code.

Figure 5.3 shows that 60% of the HC from selected MDE repositories have elevated Large method, Duplicate code, and Cyclomatic complexity CS densities. In the case of Excessive imports and God class CS densities, 80% of the HC (MDE) have more CS density than Non-MDE repository code. However, we observed the opposite results in Large class CS density in 80% of the HC in selected MDE repositories. We also found that all MDE HC

to be associated with elevated CS density on average compared to Non-MDE repositories ($\#CS^{D}_{MDE} > \#CS^{D}_{DD}$ & $\#CS^{D}_{MDE} > \#CS^{D}_{NDD}$).

In normalized CS, we found that Large method, Excessive imports, and Cyclomatic Complexity are the top 3 CS that are introduced in MDE HC. However, the God class and Large class are the least introduced CS in the MDE environment.

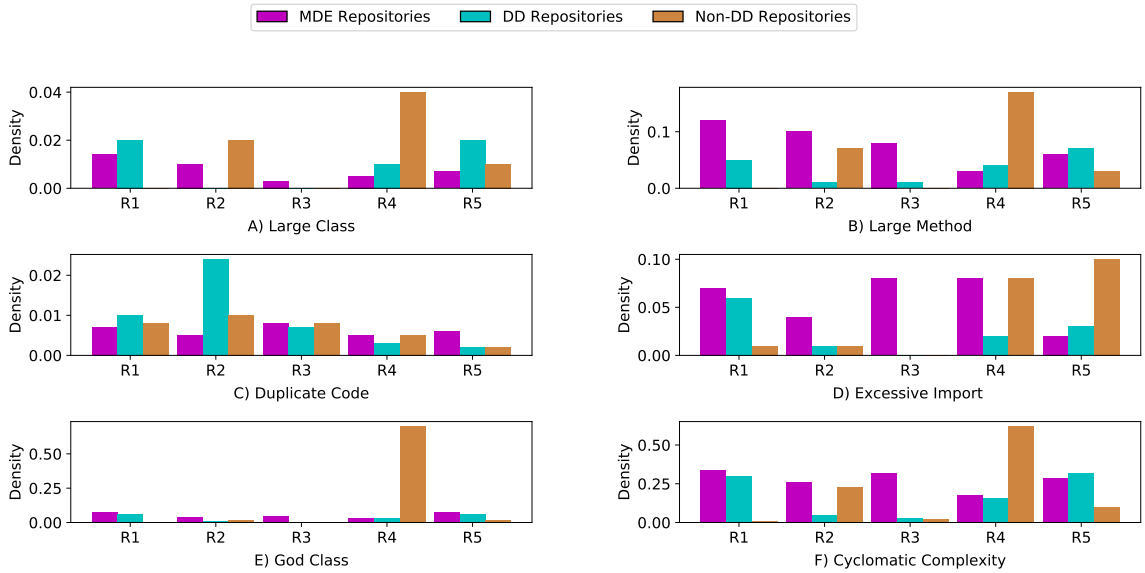


Figure 5.3: Code Smells in MDE, DD & Non-DD Repositories

5.4.2 Results Based on Technical Debt (RQ3)

Table 5.9 shows the total number of Technical debts in MDE handwritten code and Non-MDE repository code. We found that total TD in MDE HC is associated with reduced TD. In other words, HC in an MDE environment introduces less TD than Non-MDE environment code. To normalize the total number of TD in an MDE handwritten code base, we compute TD density.

Figure 5.4 illustrates TD density results for MDE HC and Non-MDE repository code. Overall, 80% of HC from selected MDE repositories have higher TD density than Non-MDE repository code.

We also calculate TD elevation between MDE handwritten code bases and Non-MDE repositories. There is a 13% TD density elevation in all 5 MDE HC compared to the DD repository code ($\#TD_{MDE} > \#TD_{DD}$). However, TD elevation in all MDE HC compared to Non-DD repositories is insignificant(2% elevation) ($\#TD_{MDE} > \#TD_{DD}$).

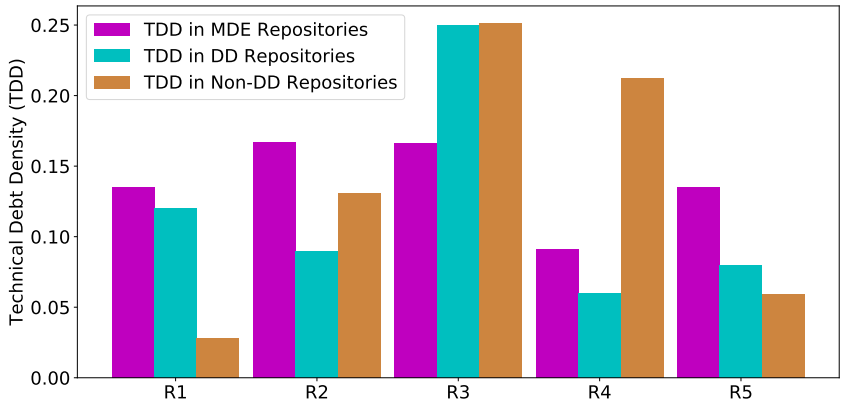


Figure 5.4: Technical Debt (TD) Result

5.4.3 Code Smells Considered in Recent Studies (RQ4)

In the preliminary result, we found a total of twenty-five various code smells from the primary studies. Table 5.11 presents the frequency of uses of code smells in the primary studies. One of the top uses of code smell is God class, 19 primary studies analyzed it. Similar results were achieved before by another research [139].

Table 5.11: Code Smells and Their Frequencies Identified in the Primary Studies

Serial	**Code Smell**	**Frequency**	**Primary Studies**
1	God Class	19	[S01], [S04], [S05], [S07], [S08], [S11], [S12], [S13], [S14], [S16], [S17], [S18], [S19], [S20], [S21], [S22], [S23], [S24], [S25]

Serial	Code Smell	Frequency	Primary Studies
2	Long Method	19	[S01], [S02], [S04], [S05], [S07], [S08], [S09], [S11], [S12], [S13], [S14], [S16], [S17], [S18], [S20], [S21], [S22], [S23], [S25]
3	Feature Envy	16	[S01], [S02], [S06], [S09], [S11], [S12], [S13], [S16], [S17], [S18], [S19], [S20], [S21], [S22], [S23], [S25]
4	Data Class	11	[S05], [S09], [S11], [S12], [S13], [S16], [S17], [S20], [S22], [S23], [S25]
5	Complex Class	6	[S04], [S07], [S08], [S14], [S18], [S24]
6	Spaghetti Code	4	[S07], [S14], [S18], [S24]
7	Speculative Generality	3	[S09], [S18], [S24]
8	Long Parameter List	3	[S09], [S18], [S21]
9	Refused Parent Bequest	3	[S01], [S09], [S18]
10	Lazy Class	2	[S09], [S24]
11	Parallel Inheritance	2	[S02], [S09]
12	Middle Man	2	[S09], [S18]
13	Large Class	2	[S02], [S09]
14	Shotgun Surgery	2	[S02], [S09]
15	Inappropriate Intimacy	2	[S09], [S18]
16	Divergent Change	2	[S02], [S09]

Serial	Code Smell	Frequency	Primary Studies
17	Class Data Should be Private	2	[S18], [S24]
18	Primitive Obsession	2	[S05], [S09]
19	Swiss Army Knife	1	[S08]
20	Brain Class	1	[S26]
21	Brain Method	1	[S26]
22	Duplicate Code	1	[S09]
23	Dead Code	1	[S09]
24	Data Clump	1	[S09]
25	Switch Statement	1	[S09]

Other top code smells are as follows: Long Method, Feature Envy, Data Class, and Complex Class are the most popular studied code smells. Whereas, Swiss Army Knife, Brain Class, Duplicate Code, Dead Code, Data Clump, and Switch Statement are the least studied code smells in the literature. A visualization of the top fifteen code smells is provided in Figure 5.5.

5.4.4 ML Approaches Considered in Recent Studies (RQ5)

This research explores popularly used machine learning algorithms for detecting code smells. Seventeen ML techniques are considered in primary studies and listed in Table 5.12 with their number of uses.

Naïve Bayes, Decision Tree, and Random Forest have widely used ML algorithms for code smell detection. Nearest Neighbor based techniques (KNN) [78] are also getting attention. However, we did not exploit which neighboring setting (i.e., the number of neighbors) produces the best result. We further observed that neural network-based techniques such

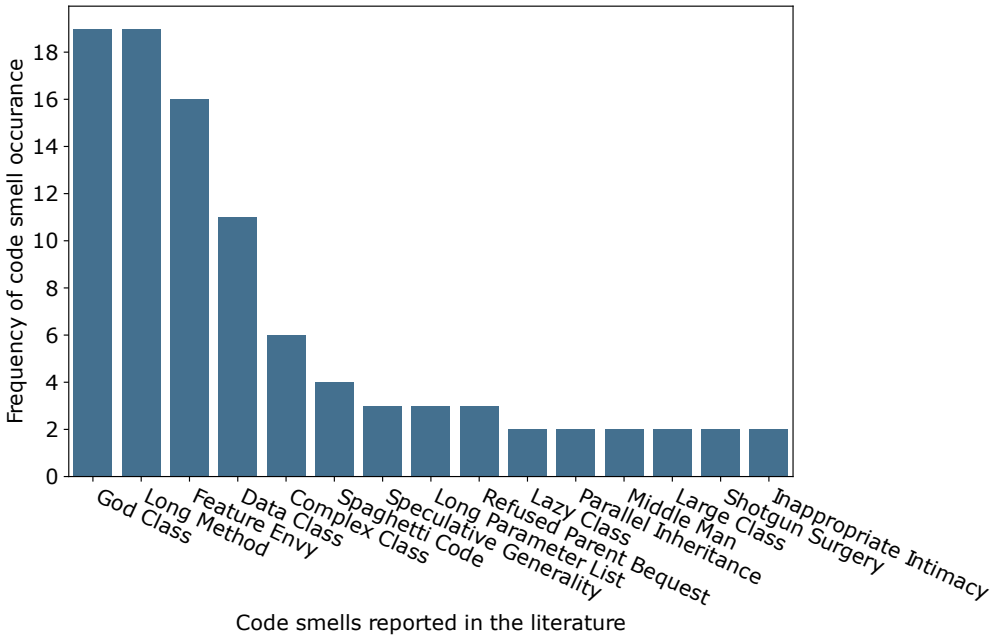


Figure 5.5: Code smells Reported in the Primary Studies

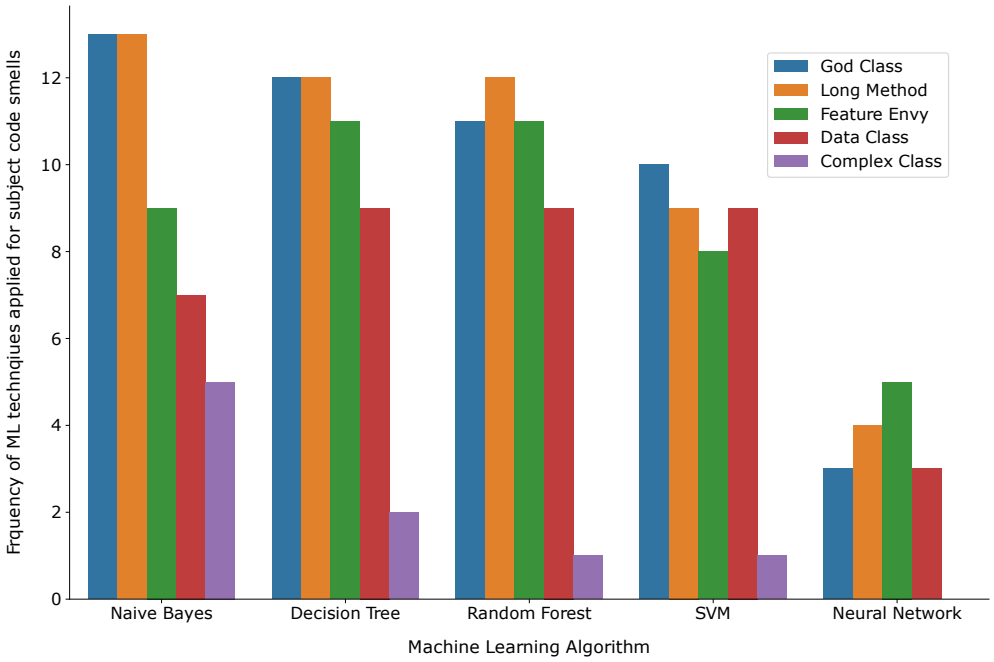


Figure 5.6: Frequency of Machine Learning Models in Code Smells Detection

as Artificial Neural Network (ANN) [78] and deep learning[67] algorithms have recently emerged too. We have compiled such kinds of algorithms to provide an overview of how neural network-based algorithms perform in general.

We further investigate the uses of machine learning algorithms based on code smells, which are visualized in Figure 5.6. The result depicts that Naive Bayes, Decision Tree, and Random Forest are popular machine learning techniques. Results also show that three machine learning techniques are frequently used to identify God class, Long method, and feature envy code. For example, to identify God class code smells approximately fourteen times Naive Bayes approach is used, whereas the Decision tree is used twelve times. Further, it is visible in Figure 5.6 that there is an emerging trend of using Neural Networks [S02], [S06], [S11], [S13], [S22], [S26] and Ensemble [S19], [S21] methods which might drive the researcher community to focus on exploring such algorithms in the future.

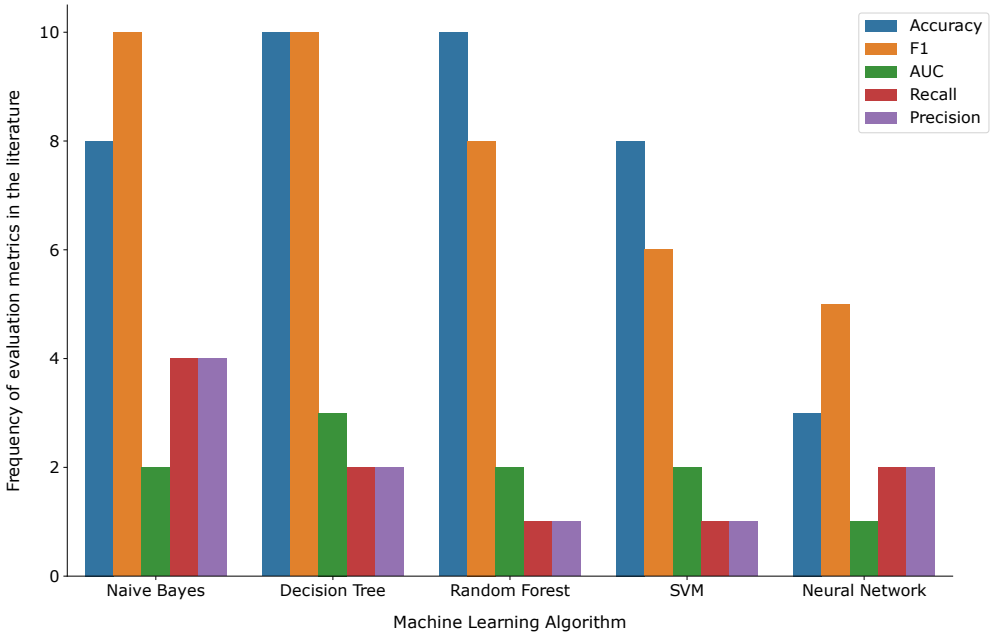


Figure 5.7: Frequency of Evaluation Metrics Used for ML Algorithms in Primary Studies

Note that, to apply machine learning approaches, datasets, and independent variables are important. Table B.3 demonstrates the list of datasets considered in the recent studies. We observed that the most popular dataset for the code smell research community is

Table 5.12: Machine Learning Algorithms Reported in Primary Studies

Serial	**Machine Learning Algorithm**	**Primary Studies**	**Frequency**
1	Naive Bayes	[S01], [S02], [S04], [S05], [S07], [S11], [S13], [S14], [S16], [S18], [S20], [S23], [S24], [S25]	14
2	Decision Tree	[S01], [S02], [S05], [S06], [S08], [S11], [S13], [S16], [S17], [S20], [S22], [S23], [S24], [S25]	14
3	Random Forest	[S01], [S02], [S05], [S11], [S13], [S16], [S17], [S20], [S22], [S23], [S24], [S25]	12
4	SVM	[S05], [S11], [S12], [S13], [S16], [S20], [S22], [S23], [S24], [S25]	10
5	Neural Network	[S02], [S06], [S11], [S13], [S22], [S26]	6
6	SMO	[S05], [S11], [S20], [S23], [S24]	5
7	Logistic Regression	[S01], [S02], [S08], [S25]	4
8	KNN	[S02], [S11], [S25]	3
9	Ensemble	[S19], [S21]	2
10	Linear Regression	[S08]	1
11	Decision Table	[S11]	1
12	Multi-Objective Search-Based	[S09]	1
13	Gradient Boosting	[S25]	1
14	One Rule	[S24]	1
15	Polynomial Regression	[S08]	1
16	GBT	[S22]	1
17	Adaboost	[S25]	1

Qualitas Corpus [140] which currently consists of 112 open-source Java systems. Regarding independent variables use, Table B.4 list the variables with the recent studies and Source code metric names along with their definition names can be found in Table B.5. One of the notable studies is [S25] which study used 55 different source code metrics in a machine-learning approach to detect code smells. We also found that three studies ([S05], [S21], [S23]) used the lowest number of source code metrics.

5.4.5 Performance Comparison of ML Approaches (RQ6)

This section presents the results of our study comparing the performance of traditional machine learning approaches and neural network-based models using accuracy as the evaluation metric.

Class-level Code Smell Results

Table 5.13 presents the accuracy of each machine learning approach we used in detecting God classes and data classes. The support vector machine approach achieved the highest accuracy of 81% in detecting God classes, while Naive Bayes had the lowest accuracy among the traditional machine learning approaches. On average, traditional machine learning approaches achieved an accuracy of 68.4%. In contrast, the convolutional neural network achieved the best accuracy of 76%, while the long short-term memory (LSTM) had the lowest accuracy of 60% among neural network-based approaches. On average, neural network-based approaches achieved an accuracy of 69.4%.

In terms of data classes, logistic regression had the highest accuracy of 70%, while Naive Bayes had the lowest accuracy among the traditional machine learning approaches. On average, traditional machine learning approaches achieved an accuracy of 61.4%. Among neural network-based approaches, the recurrent neural network (RNN) achieved the highest accuracy of 76% compared to others, with an average accuracy higher than that of traditional machine learning approaches.

Table 5.13: Accuracy Comparison of Traditional and Neural Network-based Machine Learning Approaches

Code Smell	**Traditional Machine Learning**						**Neural Network Based ML**					
	NB	**DT**	**RF**	**SVM**	**LR**	**AVG.**	**MLP**	**CNN**	**LSTM**	**RNN**	**ANN**	**AVG.**
God Class	50	69	70	81	72	68.4	70	76	60	75	66	**69.4**
Data Class	35	66	68	68	70	61.4	51	67	64	76	73	**66.2**
Long Method	51	73	75	85	78	**72.4**	64	71	68	80	74	71.4
Long Parameter List	48	79	79	83	73	72.4	66	79	69	76	77	**73.4**

Method-level Code Smell Results

For the long method code smell, we found that support vector machine and RNN achieved better accuracy in traditional machine learning and neural network-based machine learning, respectively. On the other hand, Naive Bayes and multi-layer perceptron (MLP) had the lowest accuracy in detecting long methods.

For the long parameter list code smell, support vector machine and convolutional neural network provided the best accuracy in traditional machine learning and neural network-based machine learning, respectively. Naive Bayes and MLP had the lowest accuracy in detecting this code smell.

Overall, our results indicate that neural network-based machine-learning approaches perform better than traditional machine-learning approaches in detecting code smells. Support vector machines and convolutional neural networks are the most accurate classifiers for identifying code smells.

5.5 Discussion & Analysis

The study results demonstrate that both code smells and technical debt are significantly elevated in handwritten code in MDE repositories. There are smells that were largely unique to this handwritten code, namely, large methods, duplicate code, and excessive imports. Interestingly, this code also had a significantly low number of large class code smells. This

suggests that refactoring for large methods would be relatively straightforward. Another key finding is that TD density was the largest in HC code in the MDE context. Based on our sample, we found evidence that designs by themselves tend to reduce TD, as evident in the TD density counts for design-driven repositories. Further, this would suggest that the elevated TD counts in MDE repositories are largely due to the unique constraints that software engineers face in developing this code. Overall, this confirms the hypothesis that handwritten code in the MDE context is subject to unique constraints that adversely affect its quality and sustainability.

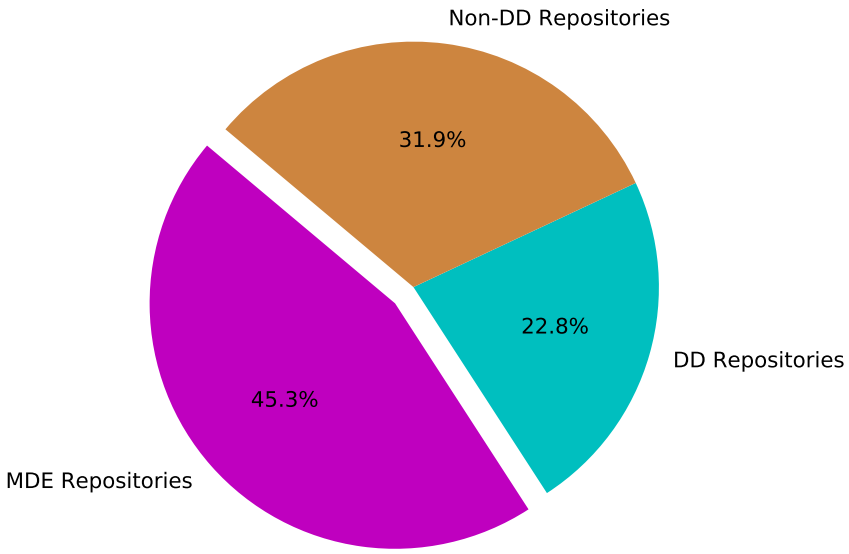


Figure 5.8: Average Code Smell Density Results

The first research question investigates the code quality characteristics. For that, we found that code smell density and TD density are elevated in HC in MDE repositories. We also observe that Cyclomatic complexity is elevated in HC MDE repositories. The second research question focuses on investigating unique deficiencies in the handwritten code in MDE contexts. This study finds that Large Method code smell density is the highest

overall in HC code, followed by duplicate code and excessive imports. Large method code smells are often associated with Large class smells, but this was not the case in this study. This suggests that classes in the HC MDE context have few numbers of methods but a significantly large number of lines of code within each method. This is potentially due to how these methods grow over time, or how these methods extend and/or integrate with generated artifacts. The third research question investigates Technical debt measures. Often, TD follows code smells as is the case in this study. TD count and density are elevated in HC code in MDE contexts in all five subject MDE repositories.

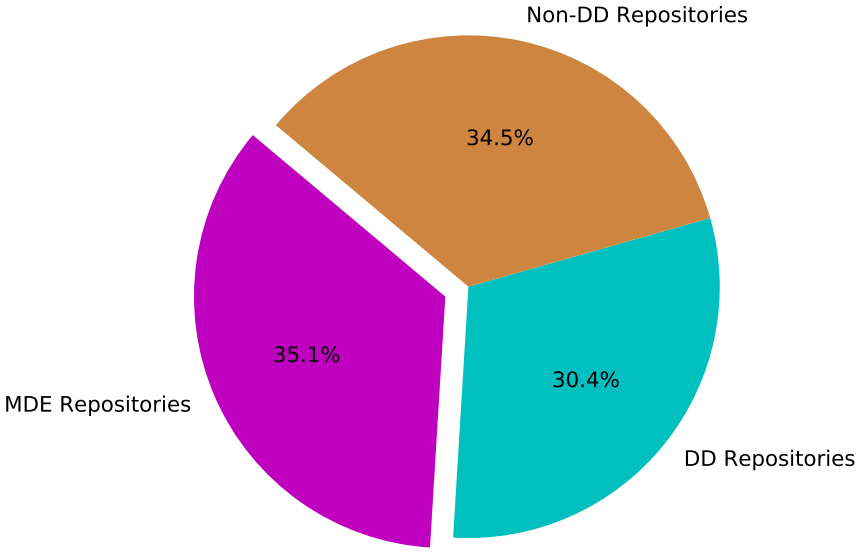


Figure 5.9: Average Technical Debt Density Results

For RQ4, we have found that God Class [141], Long Method, Feature Envy [78], Complex Class, and Data Class are the most studied code smells detected in recent studies shown in Figure 5.5. Interestingly, all of these code smells are related to design. One important lesson from this study is that the identified code smells are essential because they may trigger or hurt the maintenance of software systems. Identifying these crucial code

smells enables us to use various machine-learning algorithms to classify further these code smells toward answering RQ6. Literature from other studies[142, 143] also reported similar kinds of code smells in their studies.

The recent literature only used a few code smells, while the capabilities of machine learning in the detection of the other code smells in this area by Pecorelli et al. [143] and Zhang et al. [144] are not assessed adequately or only preliminary evaluated. Some of the minimal used code smells, e.g., Class Data Should be Private [S18], [S24], Inappropriate Intimacy [S09], [S18], Refused Parent Bequest [S01], [S09], [S18] are harmful to software by several studies [145, 146, 147]. We recommend doing the empirical investigations using machine learning approaches on the mentioned code smells, which create issues related to harmfulness to codebases.

In regarding RQ5, we explore machine learning algorithms for code smells detection in recent studies. The result shows that Decision Tree, Naive Base, Support vector Machine, and Random Forest are the top four machine learning algorithms that are used in the literature from 2015 to 2021 (Figure 5.6). However, this article depicts that most machine learning algorithms can not provide accurate results in identifying code smells (Figure 5.10) and lack implementation of ensemble & transfer techniques for code smell classification. The interpretation is that providing the better performance of these algorithms requires a handful of data sets to train the machine learning models. Several studies [78, 60, 148] agree with this interpretation. We infer that some other novel machine learning models such as ensemble [149, 150] and transfer learning [89, 151] need to be explored. It will create an opportunity for future researchers to improve the code smell detection algorithms.

5.6 Threats to Validity

There are some threats to validity in this study that we categorize as construct threats to validity and external threats to validity. Constructs threats refer to threats to which the study measures what it claims to be measuring. However, external threats refer to whether

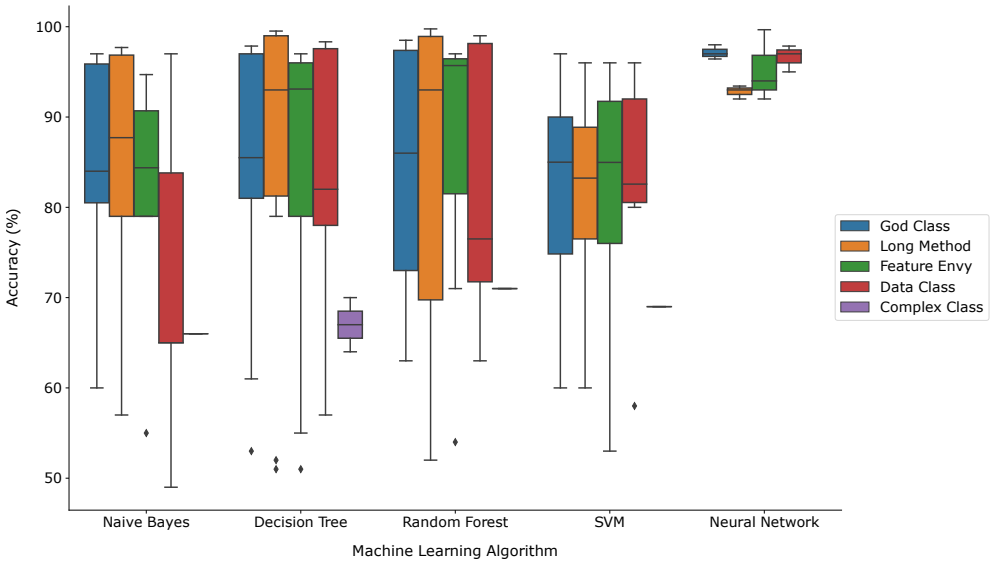


Figure 5.10: Comparative Accuracy Analysis of Machine Learning Models

we can generalize this study with different settings. This study does not deal with internal threats to validity and we exclude it.

5.6.1 Construct Validity

The selected types of code smells are a subset of all the code smells that are predominantly found in the code-base and indicate maintenance needed in the code-base. We do not claim that the selected types of CS are a complete set for TD. Furthermore, we do not claim that other CS that are not included in this study can not be TD indicators. However, it is an open question to investigate which code smells are more suitable than others as TD indicators. In the future, we plan to repeat this study with other code smells.

The second threat of this study is the precision of measuring CS by PMD and SonarQube tool. We do not claim that PMD and SonarQube are the best tools to measure CS. There are many source code analysis tools available to measure CS. We use SonarQube and PMD tools for their popularity as source code analysis tools [152, 153]. We plan to minimize this threat by using multiple code analysis tools and synthesizing the results.

The third threat is artifacts identification in MDE and Non-MDE projects which verifies

whether a selected project is MDE or Non-MDE. We conducted a semi-automated process to identify MDE elements in the code base. We do not claim that our identification process is the most appropriate one. This threat can be minimized by regenerating the code from models and comparing the current version with the regenerated version. However, this requires a lot of effort that can not be spent in this explanatory study.

The fourth threat is the comparison of TD in handwritten code (MDE environment) with TD in non-MDE code whether reasonable or not. We argue that this is reasonable to compare because we compare non-model-generated code quality with the non-MDE code. This is comparable since the coding is similar in terms of code bases which are handwritten.

5.6.2 External Validity

There is the risk that the selected 15 repositories are not the best representation of the general practices and other open-source repositories. This risk is introduced in the selection process. To minimize this risk, we selected repositories of sizes close to the median repository size in GitHub. We also excluded repositories that are trivial. We defined trivial repositories which have less than 100 commits and code size is less than 145k.

The second threat of this study is identifying MDE projects considering the GMF/EMF framework. There are some other modeling frameworks such as Xtext that also can generate executable code from the model. We use GMF/EMF because these are the most popular modeling framework in the MDE context [154].

The third type of threat is identifying comparable repositories by their code size, the number of commits in GitHub, and the primary programming language. We do not argue that these criteria are the best criteria to select comparable MDE and Non-MDE repositories. In the future, we plan to include other criteria such as software domain and software technology to identify comparable repositories.

The fourth external threat is we undermine the importance of assessing some confounding factors such as the developer's expertise. Assessing these types of confounding factors will not give us a perfect perception of the MDE environment, but may provide us with

the developers' circumstances under which the MDE environment could be beneficial.

Moreover, all the code repositories were selected from the GitHub open-source platform, and conclusions from this study should be comprehended within the context of open-source software.

5.7 Conclusion

This study aims to analyze code smell characteristics in Model-Driven Engineering (MDE) repositories, recent research studies, and a comparison of machine learning approaches for code smell detection.

First, we investigate the quality of handwritten code in MDE projects by comparing it to codebases in non-MDE environments. The handwritten code in the MDE context is unique because it must integrate with extent code that is automatically generated from models. The study reveals that the handwritten code in MDE projects has elevated levels of code smells and technical debt, resulting in poor design compared to similar non-MDE codes. The code smells identified include God class, excessive imports, large methods, and cyclomatic complexity are more prevalent in handwritten code in MDE repositories. In addition, measures of Technical Debt were also elevated in this code.

Secondly, we examine recent literature on code smell research using machine learning techniques from 2015 to 2021. The study highlights the predominant code smells and the most commonly used machine learning algorithms for code smell detection. The code smells identified as the most studied include God class, long method, feature envy, complex class, and data class. Furthermore, decision tree, naive base, support vector machine, and random forest were found to be the top four machine learning algorithms used in recent literature.

Thirdly, we compare the performance of neural network-based approaches with traditional machine learning-based approaches for code smell detection. Our findings reveal that neural network-based approaches achieved an accuracy of 69.4%, while traditional machine

learning approaches achieved an accuracy of 68.4%.

We reported key code smells that tend to be more prevalent in this unique handwritten code and recent studies; namely, large method, excessive imports, and duplicate code smell. We attribute this to the constraints that are unique to the handwritten code in MDE projects. These constraints include integrated and extended generated artifacts. MDE repositories often use code generators that may produce code that is not intuitive or comprehensible. Such factors, among others, contribute to the degraded code quality. And since this handwritten code tends to consume a significant portion of the maintenance effort, its degraded quality may cancel out or overshadow the benefits of automated code generation. We also report key code smells in recent studies and highlights the need to optimize code generators for human comprehension, and shows which machine learning approaches are receiving higher accuracy in detecting code smells.

In future research, we plan to investigate the impact of code smells on software maintainability in MDE environments and develop recurrent neural network approaches for code smell detection. We also plan to explore the relationship between code smells and software security. By exploring these areas, we can gain a deeper understanding of the impact of code smells on software quality in MDE environments and identify new approaches for detecting and addressing code smells.

Chapter 6

Issue Label Identification: Towards A Machine Learning-based Approach

This chapter presents research on software issues reported during community-based software development. The research focuses on analyzing software issue-related artifacts to understand the behavior of the software and improve its quality. The paper investigates the performance of the proposed issue-related artifacts mining tool "G-Issue" with other state-of-the-art tools. It also investigates the performance of software issue label classification and compares it with existing research.

Software developers or contributors report issues related to bugs, errors, and missing documentation during community-based software development. These issues are treated as feedback and are crucial to enhancing software's new features, documentation, and quality. If software issues are not being addressed with the correct developer, software quality degrades and is unable to use in the end. Hence, it is essential to analyze the software issue-related artifacts and classify them into correct labels to understand the behavior of the software. However, there is a misclassification of labeling with these issues. Some researchers used machine learning approaches to improve the classification but had low accuracy due to small dataset training. This paper investigates the performance of the proposed issue-related artifacts mining tool G-Issue with other state-of-the-art tools and proposes a deep learning approach to classify labels from software issues. We also investigate issue lifetime and evolution of issues over time among well-known and maintained repositories. The results show that G-Issue is faster in mining issue-related artifacts but takes more memory than general Python API during mining issue mining and the proposed

approach can classify labels with good accuracy. The results depict that we can prioritize issues based on issue labels, lifetime, and evolution. Such results may provide a new horizon about issues that can help in issue management, developer assignment, and quality management.

6.1 Introduction

Software development becomes distributed nowadays, and developers from anywhere can contribute towards the software development [155]. Towards this development, some software manages technical artifacts like commits, issues, and milestones which enables a social community that attracts many developers to work on and deliver projects within timeline [156, 157]. BitBucket [158], GitHub[159], GitLab [160] is the leader in distributed version control and source code management (SCM), which combines the ability to develop, secure, and operate software in a single application.

Source code management software is growing in features that allow faster development through bug identification, error reporting, or other issues. One of the features is an issue tracking system, often used to get user feedback related to proposed features, bugs, errors, and problems. Also, the service allows the developers to assign an issue to a developer [25] and automatic labeling issues to prioritize it better [30]. One example is shown in Figure 6.1. In summary, this tracking system enhances the code quality and increases the software's lifetime.

Software maintenance is a costly and largely unpredictable human-intensive activity in the software development life cycle. High maintenance efforts and expertise often eclipse the cost and sometimes become the reason for unsustainable software [161]. Moreover, if issues are not well managed during this maintenance, the software becomes smelly and may introduce bugs, and obsolete in the long run [28]. To solve such issues, developers worldwide may provide feedback on an issue and can contribute to fixing that. Therefore, source code management with issue tracking can provide collaborative pathways to manage

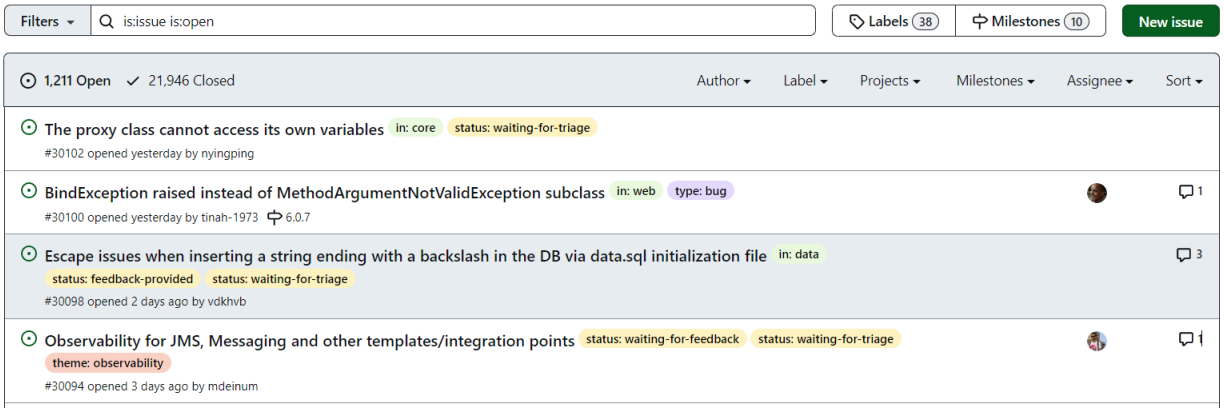


Figure 6.1: Example of Labels Attached to Issues for the Spring-framework Project in GitHub.

software, reduce software failures and improve software quality.

Very few research efforts have been conducted on mining [18, 162, 163], analyzing [164] and visualizing [165] issues in open source communities. These efforts include issue title prediction [21], automatic issue labeling [166, 167] and sentiment analysis of issues [168, 26]. However, there is a missing effort on mining issues faster, classifying labels from issues to identify the correct category.

In this paper, we investigate the performance of issue mining of an in-house developed tool, called G-Issue, and compare performance with other state-of-the-art tools [15, 128] in terms of execution time and memory usage. Moreover, we investigate the performance of the deep learning approach in classifying labels from issues to see the behavior of each repository.

This chapter is structured as follows: Section 6.2 discusses the study design with research questions. Section 6.3 shows results against each research question and discusses elaborately and finally, we conclude in Section 6.4.

6.2 Study Design

The study aims to analyze issue-related artifacts from open-source repositories with the purpose of mining, pre-processing, and visualizing the issues which can be effectively used in practice. The perspective is of both researchers and practitioners who are interested in analyzing the issues in terms of issue expectancy and evolution of issues. Specifically, we aim to address the following research question:

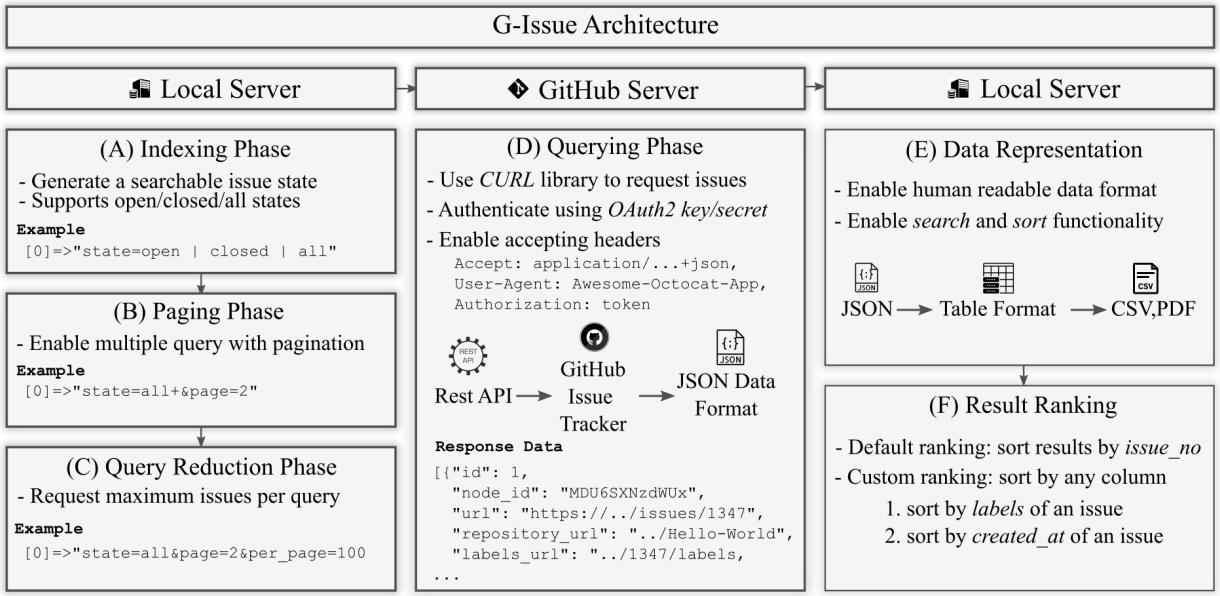


Figure 6.2: Architecture of G-Issue Tool

6.2.1 Research Questions

This section discusses the research questions we used and how we plan to answer these research questions. We are motivated to find the answer to the following research questions:

RQ1. What is the performance of the G-Issue tool compared to the state-of-the-art tools in mining issue-related artifacts?

The RQ focuses on the performance evaluation of G-Issue and is motivated by the fact that issue-related artifacts are crucial in repositories compared to the code itself and tend

to be significantly larger in terms of text size and issue comments. This often translates to complexity in identifying and extracting issue-related artifacts. For reference, we compare ModelMine with state-of-the-art tools Python API [18], GHTorrent [128, 10], PyDriller [15], G-Repo [22] for mining issues from GitHub. To answer this research question, we choose three individual tasks that are common for the majority of mining research with available support in mining tools. The tasks are as follows:

1. **Task 1 (Size related):** Retrieve the list of 1000 issues that include at least one open state issue, and the total number of issues is more than 1000.

2. **Task 2 (Time-related):** Retrieve the list of 1000 issues that include at least one open state issue and were created before January 2019.

3. **Task 3 (State related):** Retrieve the list of 1000 issues now in a closed state.

These tasks are implemented using the following frameworks/tools: (1) G-Issue, (2) Python API, (3) GHTorrent, (4) PyDriller, and (5) G-Repo. To compare the tools, we use two performance metrics: (1) Execution Time and (2) Max Memory (MM). Such performance metrics are used in evaluating different software artifacts mining tools [101, 15, 8]. The evaluation checks how fast and how much memory the tool takes to mine issue-related artifacts.

RQ2. What is the average issue lifetime among different repositories?

This RQ describes the analysis of the time it takes to solve an issue for each repository in our dataset on average. After collecting issues using G-Issue, we will find closed state issues, their created time, and when it is closed. We reveal the average issue lifetime among different repositories based on those data.

RQ3. What is the evolution of issues over time among repositories?

This RQ shows the evolution of open and closed state issues among repositories. To prepare the results, we need to extract yearly issues and their state from all the issues. We also plan to show Kernel Density Estimation (KDE) as a part of the probability density function on our ongoing issue of creating time variables.

RQ4. How accurately the proposed deep learning approach can classify labels from software issues?

This RQ reports our experiences with the application of a deep learning algorithm to classify issues from open-source repositories into multi-label categories in a completely automated way. The issue lists were selected from ten open-source repositories. We propose a deep-learning classification algorithm for this multi-label classification problem.

With these research questions, we aim to provide a more profound knowledge of the capabilities of G-Issue in mining and analysis of issue-related artifacts. The following subsections report the architecture of G-Issue and the steps that we conducted to collect the dataset.

6.2.2 G-Issue Architecture

In this section, we discuss the architecture of the issue mining tool G-Issue that we built in-house lab setup and hosted on the online platform. The tool adopts several approaches (indexing, paging, query reduction, querying, data representation, and results ranking) to mine issue-related artifacts of repositories from open source repositories. The overall architecture of the G-Issue tool is visualized in Figure 6.2.

In G-Issue, we provide a user interface with the mining capability to request GitHub for issue-related artifacts and process that data. This service is under the parent tool called ModelMine [8]. This tool provides a simple, extensible user interface to mine issue-related artifacts of repositories. It has a different way of searching to ensure the possibility of different mining types of datasets for MSR research.

Software issues have multiple types of artifacts, including state, milestone, assignee, and G-Issue, allowing researchers to investigate specific state-based issue searches. This feature allows researchers to analyze the different states of the issues in repositories and the behavior of software code issues of different projects. The user interface of the G-Issue tool is visualized in Figure 6.3.

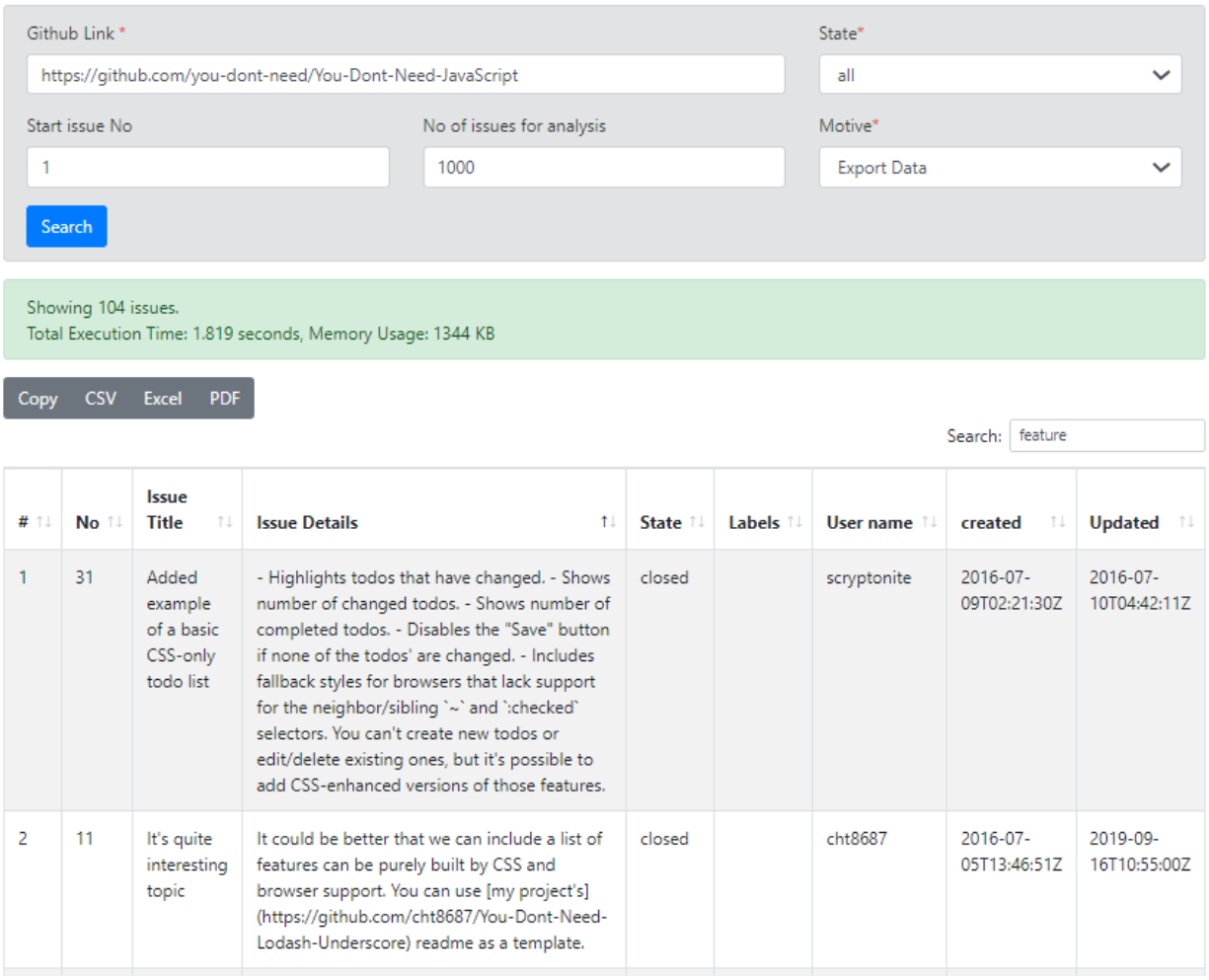


Figure 6.3: Search & Result Screenshot of G-Issue Tool

6.2.3 Proposed Deep Learning Approach

Figure 6.4 illustrates the structure of our deep neural network-based classifier for detecting issue labels. The model takes a vector of the issue's full text as input, which is then preprocessed using popular NLP techniques like stopword removal and stemming. Next, we use the word2vec technique to convert the preprocessed text into numerical vectors, which are then fed into our proposed deep-learning approach.

Our proposed model utilizes several layers such as embedding layers, dropout, and LSTM with specific settings, such as $recurrent\_dropout = 0.2$ and $activation = sigmoid$.

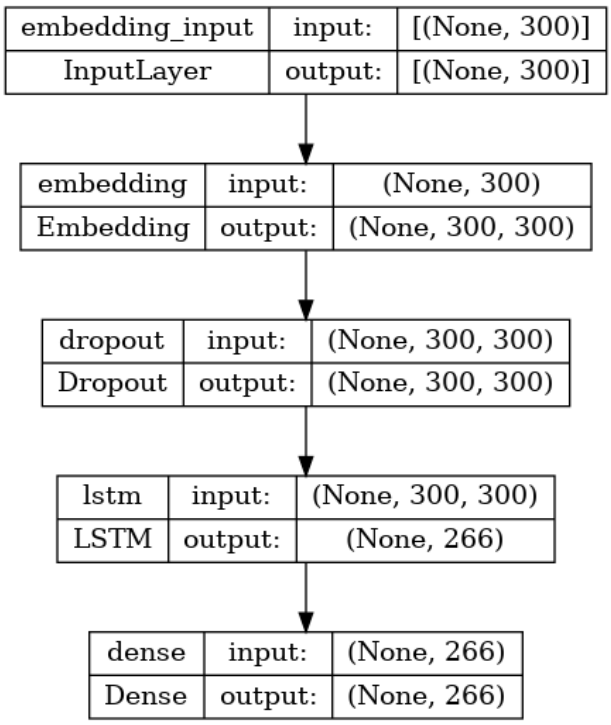


Figure 6.4: Classifier for Proposed Deep Learning Approach

The use of NLP techniques is well-suited for our task due to recent advancements in NLP, which enhance the capacity and flexibility of machine learning. Additionally, powerful neural network layers can learn deep semantic relationships among input vectors, enabling the classification of labels. Furthermore, NLP is ideal for parallel computation on modern GPUs, which significantly reduces training time.

The output of the NLP is then passed through a dense layer that transforms the input into a multi-dimensional vector.

6.2.4 Data Collection

One of the challenges in software research is identifying code repositories that have been actively maintained for an extended period. We identify some characteristics that may give us actively maintained repositories to search such code repositories. The characteristics are as follows: a repository with a minimum of 5000 commits, at least 100 active contributors,

Table 6.1: Selected Repositories with Metadata Information

Serial	Repository name	Commits	Contr.	Time Selection	No. Open Issues	No. Closed Issues	Total Issues
1	Spring framework	22,208	531	2004-05 to 2022-08	1,391	24,593	25,985
2	Junit-5	6,621	161	2015-01 to 2022-08	135	2,828	2,963
3	Apache kafka	8,590	762	2012-08 to 2022-08	1,002	11,477	12,479
4	Apache lucene-solr	34,789	232	2016-01 to 2022-08	255	2,411	2,666
5	Dropwizard	5,702	361	2011-03 to 2022-08	26	5,518	5,544
6	Checkstyle	9,922	254	2013-09 to 2022-08	697	11,287	11,984
7	Hadoop	24,612	339	2014-09 to 2022-08	681	3,681	4,362
8	Selenium	26,532	558	2013-01 to 2022-08	117	10,597	10,715
9	Skywalking	6,242	315	2015-11 to 2022-08	62	8,525	8,587
10	Signal android	7,015	223	2011-12 to 2022-08	242	10,049	10,291

a minimum of 3000 stars, and 500 forks. We use the ModelMine tool [8] which is capable of retrieving repositories with the mentioned criteria. A high number of stars and forks imply the popularity of the repositories, and a high number of commits imply maintenance throughout the software development life cycle. We choose the top ten repositories from the results provided by the ModelMine tool. Overall, the selected repositories have code changes in commits that will help us to extract the different source code metrics to reduce threats to the generalizability of this study. In this study, we have mined repositories and created a dataset composed of ten open-source repositories. Then we use the G-Issue tool to mine issue-related artifacts. The whole dataset is now published in GitHub `https://github.com/sayedmohsinreza/CSIQ` and available online [169]. The detailed summary of the ten open source repositories and issues in each repository are reported in Table 6.1.

6.2.5 Terminology

The software issues have some particular terminology we need to discuss to understand the results. Occasionally, issue-related artifacts include reporting bugs, requesting new features, refactoring code, and enhancement ideas. Also, the artifacts are typically created by anyone with title & details and consist of the person's information, created time, and labels associated with the issues. If the issue is closed or modified, that record is also documented in the specific issue.

Here are the details of some terminologies used in this study.

- **Issue lifetime -** Time from the first opening of the issue to the first closing of the issue.
- **Opened issue -** Newly created issue. Each issue is opened only once during its lifetime.
- **Closed issue -** issue that is marked closed in the issue tracking system. In practice, an issue might be reopened and closed again, but here we use only the last closing event.

6.3 Results & Discussion

In this section, we report the results and analysis of the research questions mentioned in Section 6.2.1.

6.3.1 Performance Evaluation

This section discusses the results of the performance of G-Issues compared to other state-of-art-tools. The performance evaluation results among the tools are visualized in Table 6.2. Such a result provides an idea of which tool performs better during mining issue-related

artifacts and how much fast and memory the tool takes to mine selected repositories. All these results are produced with the setup to mine 1000 issues from repositories.

Table 6.2: Performance Comparison of Different Tools

Tasks	**Metrics**	**G-Issue**	**Python API**	**GHTorrent**	**PyDriller**	**G-Repo**
Task 1*	*ET***	**12.1s**	18.2s	46.2s	Not	Not
(Size)	*MM****	18223KB	**10211KB**	67033KB	supported	supported
Task 2	ET	**30.22s**	41.7s	88.3s	Not	Not
(Time)	MM	20340KB	**16547KB**	74031KB	supported	supported
Task 3	ET	**11.8s**	15.5s	102.3	Not	Not
(Issue-related)	MM	19967KB	**14566KB**	63654KB	supported	supported

***** Task details are listed in Section 6.2.1

****** ET - Execution Time

******* MM - Max Memory

Table 6.2 shows that in each task, G-Issue mines a list of 1000 issues with the lowest execution time while GHTorrent mines with the highest execution time. Python API has the lowest memory utilization during mining, and GHTorrent has the highest utilization. Among state-of-the-art tools, PyDriller and G-Repo have focused on mining software repositories and have no feature to mine issue-related artifacts.

6.3.2 Analysis of Issue Lifetime

In this section, the results of issue lifetime among repositories are discussed and portrayed in Table 6.3. The table shows the average days it takes to solve an issue among repositories. Here "Average days to solve" means how many days it takes to close the issue by developers since the issue creation date.

From Table 6.3 results, we can see that *Spring Framework* project has the highest average of 1220 days to solve an issue where *skywalking* developers use only 37 days. *spring-framework* commit count is less than *haddop* project but average issue lifetime in

Table 6.3: Statistics on Days it Takes to Solve an Issue

Project Name	**Mean (days)**	**Minimum (days)**	**Maximum (days)**
1. spring-framework	1220	0	5491
8. selenium	551	0	2574
10. signal-android	215	0	3010
2. junit-5	162	0	2144
3. apache-kafka	104	0	2467
5. dropwizard	101	0	3221
7. hadoop	98	0	2158
4. apache-lucene-solr	91	0	2030
6. checkstyle	57	0	2496
9. skywalking	37	0	1840

haddop is twelve time less than *spring-framework*. For each repository, the minimum issue lifetime day is zero, which implies that within the issue created date, developers solve the issue and close that.

However, Figure 6.5 visualizes the boxplot of issue lifetime among repositories. From the figure, it is noticeable that *spring-framework* and *selenium* has the highest mean of days to solve an issue. Among all repositories, one issue from *spring-framework* has taken more than 5000 days / 13 years to solve. Here, we need to keep in mind that some issues are closed and reopened later on to receive more feedback on that issue.

6.3.3 Evolution of Issues

In this section, we discuss the evolution of issues among repositories. The results of the evolution of issues are visualized in Figure 6.6 showing a histogram of issue count per year in terms of open or closed state among repositories.

In every case, the graph implies that new issues are increasing in number during the

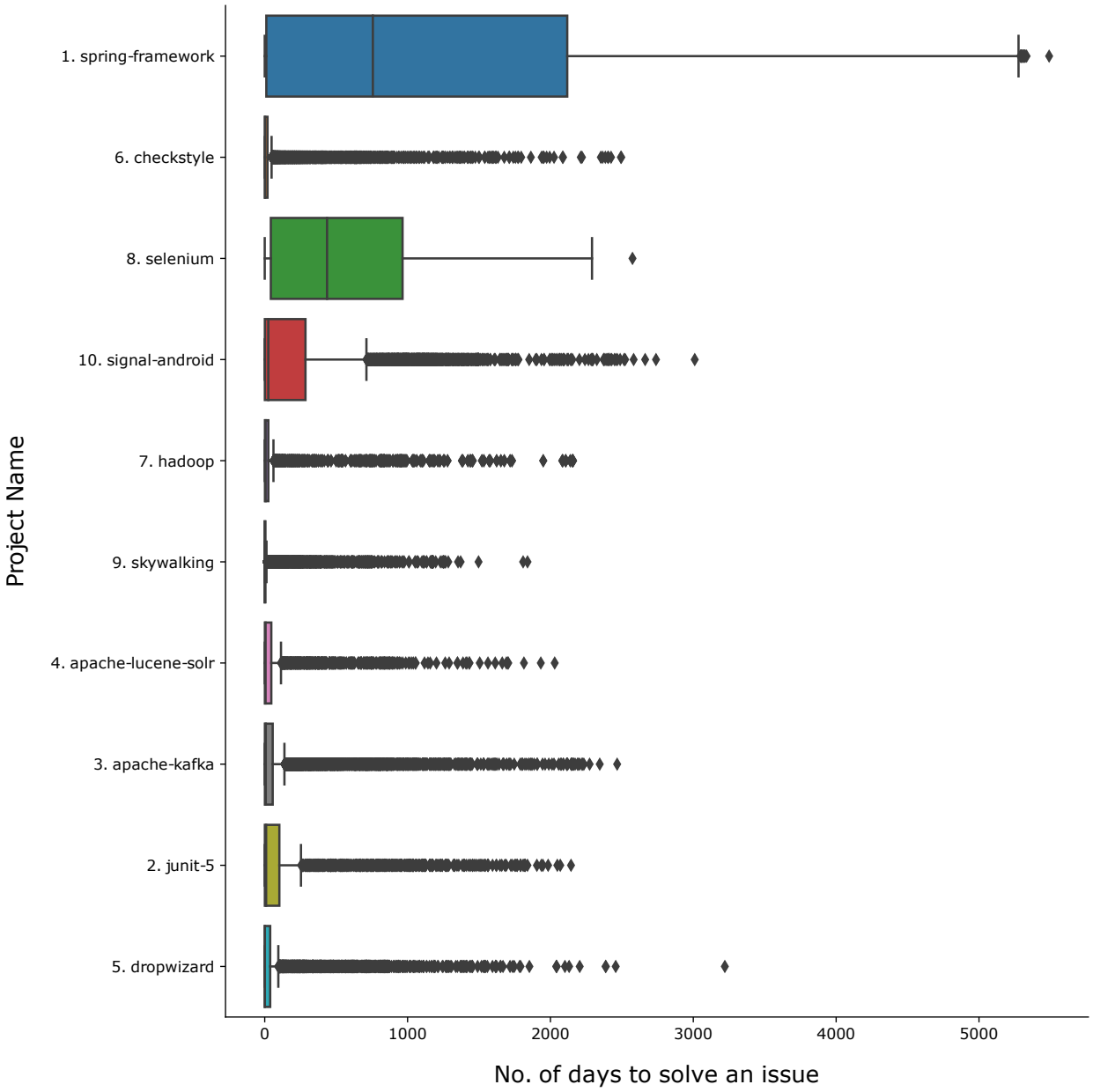


Figure 6.5: Box Plot of Days it Takes to Solve Issues among Repositories

software evolution. This number increases and becomes higher when the close-state issue rate declines. *spring-framework*, *junit-5*, *checkstyle* and *signal-android* show a recent decline in the rate of closed-state issues and an upward trend of new issues. The KDE density value represents an increasing number of issues reported by developers or contributors.

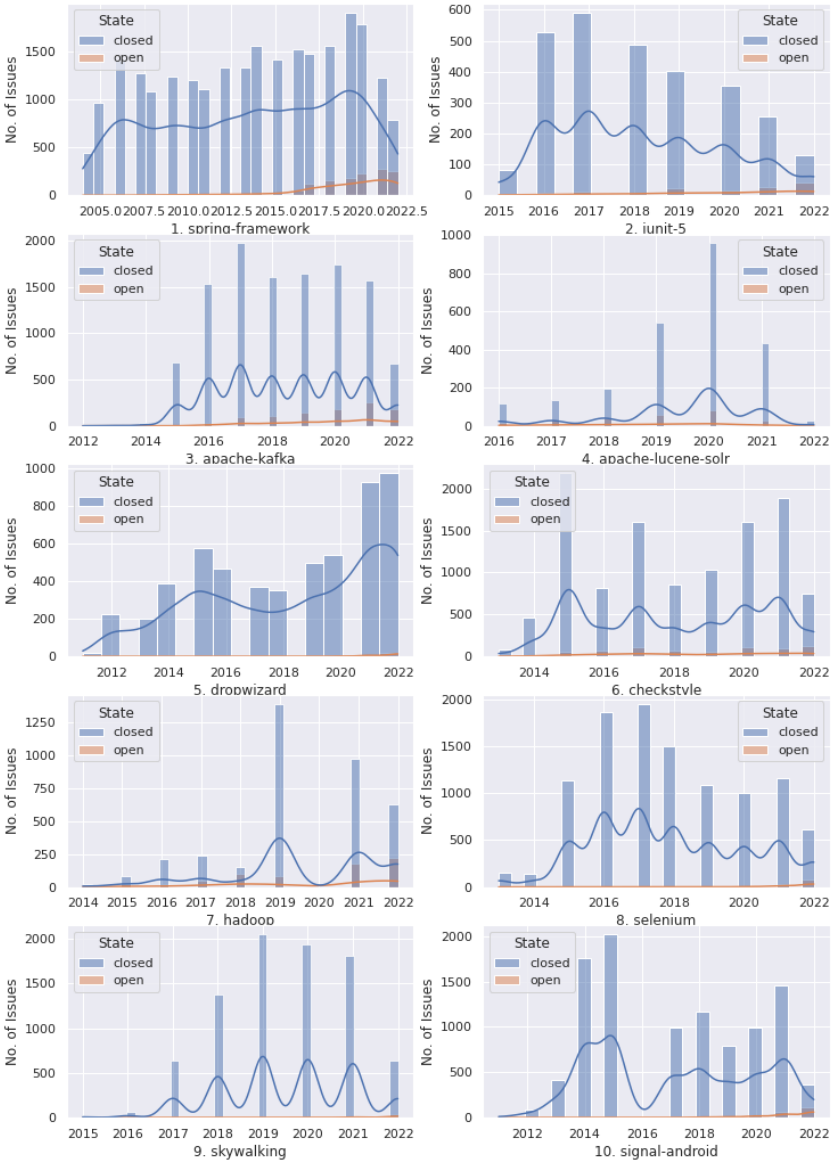


Figure 6.6: Evolution of Issue-related Artifacts Over Time among Repositories

Also, we have seen a pattern of the zigzag move of issues over the years among the repositories. It implies that when new issues are introduced within that year, it tries to be solved and closed the issue. Hence, continuous maintenance through issue-related artifact analysis prepares software for subsequent releases with improved software quality and minimized bugs in reporting.

6.3.4 Performance of Proposed Deep Learning Approach

In this section, we present the performance results of our proposed deep learning approach and compare it with existing research. We provide a detailed performance comparison between the two approaches in Table 6.4 and highlight the superior accuracy achieved by our proposed method.

Our proposed deep learning approach outperforms the existing approach in terms of accuracy. Specifically, it achieves an accuracy of 81.29% in classifying issue labels, while the existing approach achieves an accuracy of 76.8%. It's worth noting that our proposed approach can classify multi-labels with a higher level of accuracy (266 unique labels), while the existing approach can only classify bug-related labels only.

Table 6.4: Performance of Proposed Approach Compared with Existing Research

Serial	**Evaluation Metrics**	**Proposed Approach**	**Random Forest Algorithm***
1	Accuracy	81.29%	76.8%(Average)
2	F-measure	79.11%	70.7%(Average)

* Pandey, N., Sanyal, D.K., Hudait, A. and Sen, A., 2017. **Automated classification of software issue reports using machine learning techniques: an empirical study.** *Innovations in Systems and Software Engineering*, 13, pp.279-297.

6.4 Conclusion

Software issue maintenance is crucial during software development and source code management in GIT. To do software maintenance, developers need feedback in the form of issues. If the issue maintenance is not handled properly, the issues will not be solved and in the long run, the code repository will generate more issues. Most source code management software nowadays provides issues to report bugs and share ideas for new features.

In this study, we investigated the process of mining, analyzing, and visualizing issue-related artifacts through a tool called G-Issue and developed a deep-learning approach to classify the issue labels. The study primarily compares the performance of the G-Issue tool with state-of-the-art tools. Moreover, we investigate the lifetime and evolution of issues in well-known open-source projects. Finally, we compare the performance of the proposed approach with existing research.

The results show that the G-Issue tool performs a minimum of 33% faster than other state-of-the-art tools. However, in memory management, G-Issue is higher than the Python API but lower than other tools. Besides, the results show that highly popular & forked repositories have more issues; on average, it takes more days to solve an issue. In terms of evolution, if the rate of the closed issue is declining, there is a high chance of introducing new issues. And finally, the performance results demonstrate the effectiveness of our proposed deep learning approach and its potential to improve the accuracy of issue label classification. Our proposed approach provides a more comprehensive and accurate classification of multi-label issues, which is critical for effective issue management and resolution. Such results may provide new knowledge about issues-related artifacts and help team leaders with issue assignments for better software development.

In future research, we plan to analyze the issue text and apply more sophisticated natural language processing approaches to identify issue labels, improving the automatic issue label tracking system.

Chapter 7

Conclusion

Software systems continue to become more complex and have large code bases. Maintaining code quality and ensuring long-term functionality has become increasingly critical for large software companies. Refactoring and redesign activities should consider both short-term and long-term implications and aim to predict the future evolution of the code base. However, the process of software maintenance or evaluating code quality often involves managing the repository through code quality tools, which can take time and be challenging when dealing with large code bases and making changes are done so frequently. Current tools lack modern techniques like machine learning-based classification capabilities for maintaining software. One of the obstacles to the process is data related to software artifacts due to the lack of user-friendly tools and manual data organization, which is time-consuming. Also, limited machine learning techniques are applied toward code quality classification, code smell detection, and issue label classification.

To address this gap, this dissertation presents a comprehensive study of the use of sophisticated techniques in software mining repositories and applied machine learning approaches to code quality classification and code smell detection to enable software maintenance activities. Additionally, natural language processing techniques are implemented to classify the issue's label. For data collection for machine learning approaches, I investigate what makes a mining tool that can extract software artifacts from open-source repositories faster and compare the performance with a state-of-the-art tool.

7.1 Contributions

The first contribution of this dissertation is the development of ModelMine, a novel mining tool designed to extract repositories, codes, model-based artifacts, and designs from open-source repositories. The tool supports phase-by-phase caching of intermediate results to speed up the processing and enable efficient data mining. I compare ModelMine with a state-of-the-art tool in terms of performance and usability, and our results demonstrate that ModelMine has the potential to become an instrumental tool for mining software repositories. Using the ModelMine tool, I create a synthesized dataset from open-source repositories, containing code quality characteristics, code smell information, and issues from selected open-source

repositories. The collected dataset is created from several popular and quality-maintained repositories and a significant number of source code metrics. The dataset provides a valuable resource for data-driven approaches to the early detection of software quality degradation and code smell detection. The second contribution of this dissertation is the application of machine learning approaches to classify code quality attributes using source code metrics and the comparison of their performance. The results showed that the Random Forest ML technique significantly improves accuracy without generating false negatives or false positives. The third contribution is the investigation of unexpected code smell generation in software repositories, where I analyzed the unique characteristics of handwritten code developed in the context of model-driven engineering and discovered that MDE handwritten code quality is impacted by a higher level of technical debt and code smells. Also, I investigated the recent development in ML techniques for code smell detection and compared the traditional and neural network-based machine learning approaches in code smell detection. The final contribution is the development of a deep learning approach for issue label classification. The results indicate that the proposed deep learning technique can classify the issue label with an accuracy of 81% which outperforms the existing approaches in issue label classification.

7.2 Future Research Plan

Overall, the results of this dissertation have practical implications for software quality assurance, maintenance, and issue management, and provide a foundation for continuous software maintenance and re-engineering. In the future, I want to continue my dissertation research with an extensive study of software maintenance activities. My future research plan is as follows:

1. **Apply Natural Language Processing (NLP) techniques on code to understand the quality aspects of a repository.**

   This research plan proposes to investigate the application of NLP techniques to software code analysis, with a particular focus on understanding the quality, bugs, and defects of a code repository. The goal of this research is to develop a methodology for using NLP techniques to extract meaningful insights from the vast amount of textual data associated with software code, including code comments, documentation, and issue reports. The research will involve designing and implementing a prototype NLP-based tool and conducting experiments to evaluate its effectiveness in identifying and addressing software bugs and defects. The results of this research could have significant implications for the field of software engineering, particularly in the areas of software quality assurance and testing.

2. **Do research on reinforcement learning in software maintenance activities.** Another future plan is to conduct research on the use of reinforcement learning techniques in software maintenance activities, specifically focusing on using Abstract Syntax Tree (AST) paths generated from code. AST paths are a way of representing the structure of code, which can be used to analyze and understand its behavior.

   The goal of this research is to investigate how reinforcement learning can be applied to AST paths in order to automate and improve software maintenance tasks. By training an agent to learn from AST paths, it may be possible to automatically identify and fix bugs, optimize code, and perform other maintenance tasks. This approach has the potential to make software maintenance more efficient and effective, as well as reduce the need for human intervention.

3. **Do research on virtual reality-based software data visualizations for software maintenance.**

   Virtual reality (VR) has the potential to transform the way we visualize and analyze complex software systems. This research plan proposes to investigate the use of VR-based data visualizations for software maintenance. The goal of this research is to develop a novel approach to software maintenance that leverages the immersive nature of VR to provide developers with a more intuitive and efficient way to analyze and understand software systems. The research will involve designing and implementing a prototype VR-based data visualization tool and conducting user studies to evaluate its effectiveness. The results of this research could have significant implications for the field of software engineering, particularly in the areas of software maintenance and debugging.

4. **Do research on transfer learning applied to software code with and without adding pre-trained models.**

   Transfer learning is a popular technique in machine learning that has the potential to improve the effectiveness of software code analysis [170, 171]. This research plan proposes to investigate the application of transfer learning to software code analysis, both with and without pre-trained models. The goal of this research is to explore the effectiveness of transfer learning for tasks such as code classification, code similarity, and code clustering. The research will involve designing and implementing a prototype transfer-learning-based tool and conducting experiments to evaluate its effectiveness. The results of this research could have significant implications for the field of software engineering, particularly in the areas of code analysis and optimization.

All of my contributions and future plans will improve the accuracy of automated software maintenance activities.

References

[1] J. Bogner, S. Wagner, and A. Zimmermann, "Automatically measuring the maintainability of service- and microservice-based systems: a literature review," in *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, 2017, pp. 107–115.

[2] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture*, vol. 57, no. 3, pp. 294–313, 2011.

[3] J. Ludwig, S. Xu, and F. Webber, "Static software metrics for reliability and maintainability," in *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE, 2018, pp. 53–54.

[4] A. Shaheen, U. Qamar, A. Nazir, R. Bibi, M. Ansar, and I. Zafar, "Oocqm: Object oriented code quality meter," in *International Conference on Computational Science/Intelligence & Applied Informatics*. Springer, 2019, pp. 149–163.

[5] K. Rahad, O. Badreddin, and S. Mohsin Reza, "The human in model-driven engineering loop: A case study on integrating handwritten code in model-driven engineering repositories," *Software: Practice and Experience*, 2021.

[6] J. Tan, D. Feitosa, P. Avgeriou, and M. Lungu, "Evolution of technical debt remediation in python: A case study on the apache software ecosystem," *Journal of Software: Evolution and Process*, vol. 33, no. 4, p. e2319, 2021.

[7] K. Rahad, O. Badreddin, and S. M. Reza, "Characterization of software design and collaborative modeling in open source projects," in *9th Int Conf Model-Driven Eng and Soft Dev*, 2021, pp. 254–261.

[8] S. M. Reza, O. Badreddin, and K. Rahad, "Modelmine: a tool to facilitate mining models from open source repositories," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2020, pp. 1–5.

[9] S. M. Reza, M. M. Rahman, H. Parvez, O. Badreddin, and S. Al Mamun, "Performance analysis of machine learning approaches in software complexity prediction," in *Proceedings of International Conference on Trends in Computational and Cognitive Engineering*. Springer, 2021, pp. 27–39.

[10] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman, "Lean ghtorrent: Github data on demand," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 384–387.

[11] G. Gousios, "The ghtorent dataset and tool suite," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 233–236.

[12] P. B. Goes, "Editor's comments: big data and is research," 2014.

[13] S. Bayati and A. Tripathi, "Designing a knowledge base for oss project recommender system, using big data analytics," in *Twenty Fourth European Conference on Information Systems (ECIS)*, 2016.

[14] J. Noten, J. G. Mengerink, and A. Serebrenik, "A data set of ocl expressions on github," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 531–534.

[15] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 908–911.

[16] F. Z. Sokol, M. F. Aniche, and M. A. Gerosa, "Metricminer: Supporting researchers in mining software repositories," in *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2013, pp. 142–146.

[17] G. Robles, T. Ho-Quang, R. Hebig, M. R. Chaudron, and M. A. Fernandez, "An extensive dataset of uml models in github," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 519–522.

[18] F. Jurado and P. Rodriguez, "Sentiment analysis in monitoring software development processes: An exploratory case study on github's project issues," *Journal of Systems and Software*, vol. 104, pp. 82–89, 2015.

[19] R. Kikas, M. Dumas, and D. Pfahl, "Using dynamic and contextual features to predict issue lifetime in github projects," in *2016 ieee/acm 13th working conference on mining software repositories (MSR)*. IEEE, 2016, pp. 291–302.

[20] T. F. Bissyandé, D. Lo, L. Jiang, L. Réveillere, J. Klein, and Y. Le Traon, "Got issues? who cares about it? a large scale investigation of issue trackers from github," in *2013 IEEE 24th international symposium on software reliability engineering (ISSRE)*. IEEE, 2013, pp. 188–197.

[21] T. Zhang, I. C. Irsan, F. Thung, D. Han, D. Lo, and L. Jiang, "itiger: An automatic issue title generation tool," *arXiv preprint arXiv:2206.10811*, 2022.

[22] S. Romano, M. Caulo, M. Buompastore, L. Guerra, A. Mounsif, M. Telesca, M. T. Baldassarre, and G. Scanniello, "G-repo: a tool to support msr studies on github," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 551–555.

[23] M. Rees-Jones, M. Martin, and T. Menzies, "Better predictors for issue lifetime," *arXiv preprint arXiv:1702.07735*, 2017.

[24] S. Transue, S. M. Reza, A. C. Halbower, and M.-H. Choi, "Behavioral analysis of turbulent exhale flows," in *2018 IEEE EMBS International Conference on Biomedical & Health Informatics (BHI)*. IEEE, 2018, pp. 42–45.

[25] Z. Liao, D. He, Z. Chen, X. Fan, Y. Zhang, and S. Liu, "Exploring the characteristics of issue-related behaviors in github using visualization techniques," *IEEE Access*, vol. 6, pp. 24 003–24 015, 2018.

[26] J. Ding, H. Sun, X. Wang, and X. Liu, "Entity-level sentiment analysis of issue comments," in *Proceedings of the 3rd International Workshop on Emotion Awareness in Software Engineering*, 2018, pp. 7–13.

[27] B. Yang, X. Wei, and C. Liu, "Sentiments analysis in github repositories: An empirical study," in *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*. IEEE, 2017, pp. 84–89.

[28] G. Rodríguez-Pérez, J. M. Gonzalez-Barahona, G. Robles, D. Dalipaj, and N. Sekitoleko, "Bugtracking: A tool to assist in the identification of bug reports," in *IFIP International Conference on Open Source Systems*. Springer, 2016, pp. 192–198.

[29] M. Golzadeh, A. Decan, D. Legay, and T. Mens, "A ground-truth dataset and classification model for detecting bots in github issue and pr comments," *Journal of Systems and Software*, vol. 175, p. 110911, 2021.

[30] R. Kallis, A. Di Sorbo, G. Canfora, and S. Panichella, "Ticket tagger: Machine learning driven issue classification," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 406–409.

[31] S. Bharadwaj and T. Kadam, "Github issue classification using bert-style models," in *2022 IEEE/ACM 1st International Workshop on Natural Language-Based Software Engineering (NLBSE)*. IEEE, 2022, pp. 40–43.

[32] T. B. Alakus, R. Das, and I. Turkoglu, "An overview of quality metrics used in estimating software faults," in *2019 International Artificial Intelligence and Data Processing Symposium (IDAP)*. IEEE, 2019, pp. 1–6.

[33] S. Moshtari, A. Sami, and M. Azimi, "Using complexity metrics to improve software security," *Computer Fraud & Security*, vol. 2013, no. 5, pp. 8–17, 2013.

[34] S. Rahman, T. Sharma, S. Reza, M. Rahman, M. Kaiser *et al.*, "Pso-nf based vertical handoff decision for ubiquitous heterogeneous wireless network (uhwn)," in *2016 International Workshop on Computational Intelligence (IWCI)*. IEEE, 2016, pp. 153–158.

[35] K. A. Rahad and S. M. Reza, "A study on network security services with cryptography and an implementation of vigenere-multiplicative cipher," 2013.

[36] J. Moreno-León, G. Robles, and M. Román-González, "Comparing computational thinking development assessment scores with software complexity metrics," in *2016 IEEE global engineering education conference (EDUCON)*. IEEE, 2016, pp. 1040–1045.

[37] L. C. Briand, J. Wüst, J. W. Daly, and D. Victor Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *Journal of Systems and Software*, vol. 51, no. 3, pp. 245–273, May 2000. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121299001028

[38] M. Gegick, L. Williams, J. Osborne, and M. Vouk, "Prioritizing software security fortification throughcode-level metrics," in *Proceedings of the 4th ACM workshop on Quality of protection*, ser. QoP '08. New York, NY, USA: Association for Computing Machinery, Oct. 2008, pp. 31–38. [Online]. Available: https://doi.org/10.1145/1456362.1456370

[39] A. S. Nuñez-Varela, H. G. Pérez-Gonzalez, F. E. Martínez-Perez, and C. Soubervielle-Montalvo, "Source code metrics: A systematic mapping study," *Journal of Systems and Software*, vol. 128, pp. 164–197, 2017.

[40] A. Janes, M. Scotto, W. Pedrycz, B. Russo, M. Stefanovic, and G. Succi, "Identification of defect-prone classes in telecommunication software systems using design metrics," *Information sciences*, vol. 176, no. 24, pp. 3711–3734, 2006.

[41] J. Al Dallal, "Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics," *Information and Software Technology*, vol. 54, no. 10, pp. 1125–1141, 2012.

[42] Y. Shin and L. Williams, "Is complexity really the enemy of software security?" in *Proceedings of the 4th ACM workshop on Quality of protection*, 2008, pp. 47–50.

[43] ——, "An empirical model to predict security vulnerabilities using code complexity metrics," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, 2008, pp. 315–317.

[44] S. K. Dubey, A. Rana, and A. Sharma, "Usability evaluation of object-oriented software system using fuzzy logic approach," *International Journal of Computer Applications*, vol. 43, no. 19, pp. 1–6, 2012.

[45] S. Watanabe, H. Kaiya, and K. Kaijiri, "Adapting a fault prediction model to allow inter languagereuse," in *Proceedings of the 4th international workshop on Predictor models in software engineering*, 2008, pp. 19–24.

[46] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the" imprecision" of cross-project defect prediction," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[47] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *2013 35th international conference on software engineering (ICSE)*. IEEE, 2013, pp. 382–391.

[48] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," *Information and Software Technology*, vol. 54, no. 3, pp. 248–256, 2012.

[49] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 434–443, 2013.

[50] K. Gao, T. M. Khoshgoftaar, H. Wang, and N. Seliya, "Choosing software metrics for defect prediction: an investigation on feature selection techniques," *Software: Practice and Experience*, vol. 41, no. 5, pp. 579–606, 2011.

[51] I. Chowdhury and M. Zulkernine, "Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?" in *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010, pp. 1963–1969.

[52] S. Moshtari and A. Sami, "Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016, pp. 1415–1421.

[53] M. D'Ambros, M. Lanza, and R. Robbes, "On the relationship between change coupling and software defects," in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 135–144.

[54] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of mde in industry," in *Proceedings of the 33rd international conference on software engineering*. ACM, 2011, pp. 471–480.

[55] A. M. Fernández-Sáez, M. R. Chaudron, and M. Genero, "An industrial case study on the use of uml in software maintenance and its perceived benefits and hurdles," *Empirical Software Engineering*, vol. 23, no. 6, pp. 3281–3345, 2018.

[56] A. Nugroho and M. R. Chaudron, "The impact of uml modeling on defect density and defect resolution time in a proprietary system," *Empirical Software Engineering*, vol. 19, no. 4, pp. 926–954, 2014.

[57] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, "Understanding code smells in android applications," in *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2016, pp. 225–236.

[58] F. Palomba, R. Oliveto, and A. De Lucia, "Investigating code smell co-occurrences using association rule learning: A replicated study," in *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE, 2017, pp. 8–13.

[59] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "The scent of a smell: An extensive comparison between textual and structural smells," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 977–1000, 2017.

[60] F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 93–104.

[61] F. A. Fontana, V. Ferme, M. Zanoni, and A. Yamashita, "Automatic metric thresholds derivation for code smell detection," in *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*. IEEE, 2015, pp. 44–53.

[62] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda, "Towards a prioritization of code debt: A code smell intensity index," in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. IEEE, 2015, pp. 16–24.

[63] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[64] U. Azadi, F. A. Fontana, and M. Zanoni, "Poster: machine learning based code smell detection through wekanose," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2018, pp. 288–289.

[65] N. Pritam, M. Khari, R. Kumar, S. Jha, I. Priyadarshini, M. Abdel-Basset, H. V. Long *et al.*, "Assessment of code smell for predicting class change proneness using machine learning," *IEEE Access*, vol. 7, pp. 37 414–37 425, 2019.

[66] T. Lin, X. Fu, F. Chen, and L. Li, "A novel approach for code smells detection based on deep leaning," in *EAI International Conference on Applied Cryptography in Computer and Communications*. Springer, 2021, pp. 171–174.

[67] H. Liu, J. Jin, Z. Xu, Y. Bu, Y. Zou, and L. Zhang, "Deep learning based code smell detection," *IEEE transactions on Software Engineering*, 2019.

[68] S. Fakhoury, V. Arnaoudova, C. Noiseux, F. Khomh, and G. Antoniol, "Keep it simple: Is deep learning good for linguistic smell detection?" in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 602–611.

[69] J. Kreimer, "Adaptive detection of design flaws," *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 4, pp. 117–136, 2005.

[70] S. Vaucher, F. Khomh, N. Moha, and Y.-G. Guéhéneuc, "Tracking design smells: Lessons from a study of god classes," in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 145–154.

[71] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *2009 Ninth International Conference on Quality Software*. IEEE, 2009, pp. 305–314.

[72] F. Khomh, S. Vaucher, Y. G. Gueheneuc, and H. Sahraoui, "Bdtex: A gqm-based bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, 2011.

[73] L. Amorim, E. Costa, N. Antunes, B. Fonseca, and M. Ribeiro, "Experience report: Evaluating the effectiveness of decision trees for detecting code smells," in *2015 IEEE 26th international symposium on software reliability engineering (ISSRE)*. IEEE, 2015, pp. 261–269.

[74] F. A. Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowledge-Based Systems*, vol. 128, pp. 43–58, 2017.

[75] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mäntylä, "Code smell detection: Towards a machine learning-based approach," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 396–399.

[76] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[77] D. Cruz, A. Santana, and E. Figueiredo, "Detecting bad smells with machine learning algorithms: an empirical study," in *Proceedings of the 3rd International Conference on Technical Debt*, 2020, pp. 31–40.

[78] F. C. Luiz, B. R. de Oliveira Rodrigues, and F. S. Parreiras, "Machine learning techniques for code smells detection: an empirical experiment on a highly imbalanced setup," in *Proceedings of the XV Brazilian Symposium on Information Systems*, 2019, pp. 1–8.

[79] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–138, 2019.

[80] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: research and practice*, vol. 23, no. 3, pp. 179–202, 2011.

[81] A. Gupta, B. Suri, and S. Misra, "A systematic literature review: code bad smells in java source code," in *International Conference on Computational Science and Its Applications*. Springer, 2017, pp. 665–682.

[82] S. S. Rathore and S. Kumar, "Towards an ensemble-based system for predicting the number of software faults," *Expert Systems with Applications*, vol. 82, pp. 357–382, 2017.

[83] M. Lafi, J. W. Botros, H. Kafaween, A. B. Al-Dasoqi, and A. Al-Tamimi, "Code smells analysis mechanisms, detection issues, and effect on software maintainability," in *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*. IEEE, 2019, pp. 663–666.

[84] K. Karauzović-Hadžiabdić and R. Spahić, "Comparison of machine learning methods for code smell detection using reduced features," in *2018 3rd International Conference on Computer Science and Engineering (UBMK)*. IEEE, 2018, pp. 670–672.

[85] G. Rasool and Z. Arshad, "A review of code smell mining techniques," *Journal of Software: Evolution and Process*, vol. 27, no. 11, pp. 867–895, 2015.

[86] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A cooperative parallel search-based software engineering approach for code-smells detection," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, 2014.

[87] F. L. Caram, B. R. D. O. Rodrigues, A. S. Campanelli, and F. S. Parreiras, "Machine learning techniques for code smells detection: a systematic mapping study," *International Journal of Software Engineering and Knowledge Engineering*, vol. 29, no. 02, pp. 285–316, 2019.

[88] S. Dewangan, R. S. Rao, A. Mishra, and M. Gupta, "A novel approach for code smell detection: An empirical study," *IEEE Access*, vol. 9, pp. 162 869–162 883, 2021.

[89] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, "Code smell detection by deep direct-learning and transfer-learning," *Journal of Systems and Software*, vol. 176, p. 110936, 2021.

[90] K. K. Chaturvedi, V. Sing, and P. Singh, "Tools in mining software repositories," in *2013 13th International Conference on Computational Science and Its Applications*. IEEE, 2013, pp. 89–98.

[91] S. M. Reza, M. M. Rahman, M. H. Parvez, M. S. Kaiser, and S. Al Mamun, "Innovative approach in web application effort & cost estimation using functional measurement type," in *2015 International Conference on Electrical Engineering and Information Communication Technology (ICEEICT)*. IEEE, 2015, pp. 1–7.

[92] S. M. Reza, "Activity based new technique of effort & cost estimation using functional measurement type for web application," Ph.D. dissertation, Jahangirnagar University, 2016.

[93] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Review participation in modern code review: An empirical study of the android, qt, and openstack projects (journal-first abstract)," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 475–475.

[94] O. Baddreddin and K. Rahad, "The impact of design and uml modeling on codebase quality and sustainability," in *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, 2018, pp. 236–244.

[95] M. C. Paul, S. Sarkar, M. M. Rahman, S. M. Reza, and M. S. Kaiser, "Low cost and portable patient monitoring system for e-health services in bangladesh," in *2016 International Conference on Computer Communication and Informatics (ICCCI)*. IEEE, 2016, pp. 1–4.

[96] S. M. Reza, M. M. Rahman, and S. Al Mamun, "A new approach for road networks-a vehicle xml device collaboration with big data," in *2014 International Conference on Electrical Engineering and Information & Communication Technology*. IEEE, 2014, pp. 1–5.

[97] O. Badreddin, R. Khandoker, A. Forward, O. Masmali, and T. C. Lethbridge, "A decade of software design and modeling: A survey to uncover trends of the practice," in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2018, pp. 245–255.

[98] M. M. Rahman and C. K. Roy, "An insight into the pull requests of github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 364–367.

[99] D. Fett, R. Küsters, and G. Schmitz, "A comprehensive formal security analysis of oauth 2.0," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1204–1215.

[100] P. Maheshwari, H. Tang, and R. Liang, "Enhancing web services with message-oriented middleware," in *Proceedings. IEEE International Conference on Web Services, 2004.* IEEE, 2004, pp. 524–531.

[101] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *2013 35th International Conference on Software Engineering (ICSE).* IEEE, 2013, pp. 422–431.

[102] A. H. Altalhi, J. M. Luna, M. Vallejo, and S. Ventura, "Evaluation and comparison of open source software suites for data mining and knowledge discovery," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 7, no. 3, p. e1204, 2017.

[103] N. Bevan, "Classifying and selecting ux and usability measures," in *International Workshop on Meaningful Measures: Valid Useful User Experience Measurement*, vol. 11, 2008, pp. 13–18.

[104] E. E. Ogheneovo *et al.*, "On the relationship between software complexity and maintenance costs," *Journal of Computer and Communications*, vol. 2, no. 14, p. 1, 2014.

[105] S. Yu and S. Zhou, "A survey on metric of software complexity," in *2010 2nd IEEE International Conference on Information Management and Engineering.* IEEE, 2010, pp. 352–356.

[106] Z. Durdik, B. Klatt, H. Koziolek, K. Krogmann, J. Stammel, and R. Weiss, "Sustainability guidelines for long-living software systems," in *2012 28th IEEE International Conference on Software Maintenance (ICSM).* IEEE, 2012, pp. 517–526.

[107] S. M. Reza, M. M. Rahman, M. M. Mahmud, and S. Mamun, "A new approach of big data collaboration for road traffic networks considering path loss analysis in context of bangladesh," *JU Journal of Information Technology*, vol. 3, pp. 1–5, 2014.

[108] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *2012 34th International Conference on Software Engineering (ICSE).* IEEE, 2012, pp. 419–429.

[109] G. Singh, D. Singh, and V. Singh, "A study of software metrics," *IJCEM International Journal of Computational Engineering & Management*, vol. 11, pp. 22–27, 2011.

[110] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *IEEE Transactions on software engineering*, vol. 29, no. 4, pp. 297–310, 2003.

[111] A. Munappy, J. Bosch, H. H. Olsson, A. Arpteg, and B. Brinne, "Data management challenges for deep learning," in *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2019, pp. 140–147.

[112] Y. Zhang, D. Lo, X. Xia, B. Xu, J. Sun, and S. Li, "Combining software metrics and text features for vulnerable file prediction," in *2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2015, pp. 40–49.

[113] M. Jimenez, R. Rwemalika, M. Papadakis, F. Sarro, Y. Le Traon, and M. Harman, "The importance of accounting for real-world labelling when predicting software vulnerabilities," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 695–705.

[114] S. Yadav and S. Shukla, "Analysis of k-fold cross-validation over hold-out validation on colossal datasets for quality classification," in *2016 IEEE 6th International conference on advanced computing (IACC)*. IEEE, 2016, pp. 78–83.

[115] J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE software*, vol. 31, no. 3, pp. 79–85, 2013.

[116] D. C. Schmidt, "Model-driven engineering," *COMPUTER-IEEE COMPUTER SOCIETY-*, vol. 39, no. 2, p. 25, 2006.

[117] A. Forward, O. Badreddin, and T. C. Lethbridge, "Perceptions of software modeling: a survey of software practitioners," in *5th workshop from code centric to model centric: evaluating the effectiveness of MDD (C2M: EEMDD)*, 2010.

[118] D. Lucredio, E. S. de Almeida, and R. P. Fortes, "An investigation on the impact of mde on software reuse," in *2012 Sixth Brazilian Symposium on Software Components, Architectures and Reuse*. IEEE, 2012, pp. 101–110.

[119] X. He, P. Avgeriou, P. Liang, and Z. Li, "Technical debt in mde: a case study on gmf/emf-based projects," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. ACM, 2016, pp. 162–172.

[120] M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth, "Language evolution in practice: The history of gmf," in *International Conference on Software Language Engineering*. Springer, 2009, pp. 3–22.

[121] (2019) Eclipse foundation,. [Online]. Available: https://www.eclipse.org/articles/article.php?file=Article-Integrating-EMF-GMF-Editors/index.html

[122] D. Cedrim, L. Sousa, A. Garcia, and R. Gheyi, "Does refactoring improve software structural quality? a longitudinal study of 25 projects," in *Proceedings of the 30th Brazilian Symposium on Software Engineering*. ACM, 2016, pp. 73–82.

[123] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* IEEE, 2002, pp. 97–106.

[124] N. Zazworka, C. Izurieta, S. Wong, Y. Cai, C. Seaman, F. Shull *et al.*, "Comparing four approaches for technical debt identification," *Software Quality Journal*, vol. 22, no. 3, pp. 403–426, 2014.

[125] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya *et al.*, "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP workshop on Future of software engineering research.* ACM, 2010, pp. 47–52.

[126] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka, "A case study in locating the architectural roots of technical debt," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 179–188.

[127] T. Ho-Quang, R. Hebig, G. Robles, M. R. Chaudron, and M. A. Fernandez, "Practices and perceptions of uml use in open source projects," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP).* IEEE, 2017, pp. 203–212.

[128] G. Gousios and D. Spinellis, "Ghtorrent: Github's data from a firehose," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR).* IEEE, 2012, pp. 12–21.

[129] B. Kitchenham, "Procedure for undertaking systematic reviews," *Computer Science Depart-ment, Keele University (TRISE-0401) and National ICT Australia Ltd (0400011T. 1), Joint Technical Report*, 2004.

[130] J. C. Carver, E. Hassler, E. Hernandes, and N. A. Kraft, "Identifying barriers to the systematic literature review process," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.* IEEE, 2013, pp. 203–212.

[131] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering–a systematic literature review," *Information and software technology*, vol. 51, no. 1, pp. 7–15, 2009.

[132] Y. Xiao and M. Watson, "Guidance on conducting a systematic literature review," *Journal of Planning Education and Research*, vol. 39, no. 1, pp. 93–112, 2019.

[133] (2019) Handwritten code classification,. [Online]. Available: shorturl.at/otxMN

[134] (2019) Pmd documentation,. [Online]. Available: https://pmd.github.io/latest/index.html

[135] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM Journal of Research and Development*, vol. 56, no. 5, pp. 9–1, 2012.

[136] J.-L. Letouzey, "The sqale method for evaluating technical debt," in *2012 Third International Workshop on Managing Technical Debt (MTD)*. IEEE, 2012, pp. 31–36.

[137] (2019) Sonarqube documentation,. [Online]. Available: https://docs.sonarqube.org/latest/

[138] S. M. Reza, M. A. M. Bhuiyan, and N. Tasnim, "A convolution neural network with encoder-decoder applied to the study of bengali letters classification," *Big Data and Information Analytics*, vol. 6, no. bdia-06-004, pp. 41–55, 2021.

[139] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, 2016, pp. 1–12.

[140] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies," in *2010 Asia pacific software engineering conference*. IEEE, 2010, pp. 336–345.

[141] A. Kaur, S. Jain, S. Goel, and G. Dhiman, "Prioritization of code smells in object-oriented software: A review," *Materials Today: Proceedings*, 2021.

[142] J. Rubin, A. N. Henniche, N. Moha, M. Bouguessa, and N. Bousbia, "Sniffing android code smells: An association rules mining-based approach," in *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2019, pp. 123–127.

[143] F. Pecorelli, D. Di Nucci, C. De Roover, and A. De Lucia, "A large empirical assessment of the role of data balancing in machine-learning-based code smell detection," *Journal of Systems and Software*, vol. 169, p. 110693, 2020.

[144] Y. Zhang and C. Dong, "Mars: Detecting brain class/method code smell based on metric–attention mechanism and residual network," *Journal of Software: Evolution and Process*, p. e2403, 2021.

[145] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 101–110.

[146] E. V. de Paulo Sobrinho, A. De Lucia, and M. de Almeida Maia, "A systematic literature review on bad smells–5 w's: which, when, what, who, where," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 17–66, 2018.

[147] D. Taibi, A. Janes, and V. Lenarduzzi, "How developers perceive smells in source code: A replicated study," *Information and Software Technology*, vol. 92, pp. 223–235, 2017.

[148] F. Pecorelli, D. Di Nucci, C. De Roover, and A. De Lucia, "On the role of data balancing for machine learning-based code smell detection," in *Proceedings of the 3rd ACM SIGSOFT international workshop on machine learning techniques for software quality evaluation*, ser. MaLTeSQuE 2019. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3340482.3342744

[149] A. Panichella, R. Oliveto, and A. De Lucia, "Cross-project defect prediction models: L'union fait la force," in *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 2014, pp. 164–173.

[150] A. Alazba and H. Aljamaan, "Code smell detection using feature selection and stacking ensemble: An empirical investigation," *Information and Software Technology*, vol. 138, p. 106648, 2021.

[151] K. M. Ahmed, A. Imteaj, and M. H. Amini, "Federated deep learning for heterogeneous edge computing," in *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2021, pp. 1146–1152.

[152] P. Louridas, "Static code analysis," *IEEE Software*, vol. 23, no. 4, pp. 58–61, 2006.

[153] V. Lenarduzzi, A. Sillitti, and D. Taibi, "A survey on code analysis tools for software maintenance prediction," in *International Conference in Software Engineering for Defence Applications*. Springer, 2018, pp. 165–175.

[154] A. Evans, M. A. Fernández, and P. Mohagheghi, "Experiences of developing a network modeling tool using the eclipse environment," in *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 2009, pp. 301–312.

[155] A. Begel, J. Bosch, and M.-A. Storey, "Social networking meets software development: Perspectives from github, msdn, stack exchange, and topcoder," *IEEE software*, vol. 30, no. 1, pp. 52–66, 2013.

[156] J. A. Teixeira and H. Karsten, "Managing to release early, often and on time in the openstack software ecosystem," *Journal of Internet Services and Applications*, vol. 10, no. 1, pp. 1–22, 2019.

[157] D. Bertram, A. Voida, S. Greenberg, and R. Walker, "Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams," in *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, 2010, pp. 291–300.

[158] J. Fisher, D. Koning, and A. Ludwigsen, "Utilizing atlassian jira for large-scale software development management," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2013.

[159] J. D. Blischak, E. R. Davenport, and G. Wilson, "A quick introduction to version control with git and github," *PLoS computational biology*, vol. 12, no. 1, p. e1004668, 2016.

[160] J. C. C. Ríos, K. Kopec-Harding, S. Eraslan, C. Page, R. Haines, C. Jay, and S. M. Embury, "A methodology for using gitlab for software engineering learning analytics," in *2019 IEEE/ACM 12th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 2019, pp. 3–6.

[161] L. Hatton, D. Spinellis, and M. van Genuchten, "The long-term growth rate of evolving software: Empirical results and implications," *Journal of Software: Evolution and Process*, vol. 29, no. 5, p. e1847, 2017.

[162] S. Dueñas, V. Cosentino, G. Robles, and J. M. Gonzalez-Barahona, "Perceval: software project data at your will," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 1–4.

[163] X. Sun, B. Li, H. Leung, B. Li, and Y. Li, "Msr4sm: Using topic models to effectively mining software repositories for software maintenance tasks," *Information and Software Technology*, vol. 66, pp. 1–12, 2015.

[164] M. A. de F. Farias, R. Novais, M. C. Júnior, L. P. da Silva Carvalho, M. Mendonça, and R. O. Spínola, "A systematic mapping study on mining software repositories," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016, pp. 1472–1479.

[165] A. Fiechter, R. Minelli, C. Nagy, and M. Lanza, "Visualizing github issues," in *2021 Working Conference on Software Visualization (VISSOFT)*. IEEE, 2021, pp. 155–159.

[166] J. Wang, X. Zhang, and L. Chen, "How well do pre-trained contextual language representations recommend labels for github issues?" *Knowledge-Based Systems*, vol. 232, p. 107476, 2021.

[167] J. Wang, X. Zhang, L. Chen, and X. Xie, "Personalizing label prediction for github issues," *Information and Software Technology*, vol. 145, p. 106845, 2022.

[168] E. Guzman, D. Azócar, and Y. Li, "Sentiment analysis of commit comments in github: an empirical study," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 352–355.

[169] S. M. Reza, S. U. Mahmud, K. Rahad, and O. Badreddin, "Csiq: A synthesized dataset of code smells, issues and quality related artifacts from open source repositories," 2022. [Online]. Available: https://data.mendeley.com/datasets/77p6rzb73n/5

[170] Z. Chen, S. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in c code," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 147–165, 2022.

[171] M. H. Parvez, M. M. Khatun, S. M. Reza, M. M. Rahman, and M. F. K. Patwary, "Prediction of potential future it personnel in bangladesh using machine learning classifier," *Global Disclosure of Economics and Business*, vol. 6, no. 1, pp. 7–18, 2017.

[172] S. Lujan, F. Pecorelli, F. Palomba, A. De Lucia, and V. Lenarduzzi, "A preliminary study on the adequacy of static analysis warnings with respect to code smell prediction," in *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation*, 2020, pp. 1–6.

[173] S. M. Reza, O. Badreddin, K. Rahad, and S. U. Mahmud, "Software code quality and source code metrics dataset," 2021. [Online]. Available: https://data.mendeley.com/datasets/77p6rzb73n

[174] H. Keuning, B. Heeren, and J. Jeuring, "Code quality issues in student programs," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, 2017, pp. 110–115.

[175] J. DIBattista, "I have been merging pandas dataframes completely wrong," 2022. [Online]. Available: https://towardsdatascience.com/the-most-efficient-way-to-merge-join-pandas-dataframes-7576e8b6c5c

Appendix A

CSIQ: A Synthesized Dataset of Software Artifacts

This appendix presents CSIQ, a synthesized dataset of software data including code quality attributes, code smells, and issues extracted from open-source repositories. The dataset is developed through the use of ModelMine [8], CodeMR [4], PMD [172] tools, and python programming language. This chapter is related to a dataset that was published in Mendeley Dataset [173] and planned to publish an extended version in the Data in Brief journal.

A.1 Dataset Overview

The dataset contains synthesized code smells, issues, quality, and source code metrics information of 60 versions under 10 different repositories. The dataset is extracted into 3 levels: (1) Class (2) Method (3) Package. The dataset is created upon analyzing 9,420,246 lines of code and 173,237 classes. The provided dataset contains the following folders: code smells, issues, quality attributes, synthesized, and four associated Comma Separated Values (CSV) files: repositories.csv, versions.csv, codesmells.csv, and attribute-details.csv.

The first file (repositories.csv) contains general information (repository name, URL, number of commits, stars, forks, etc.) to understand the size, popularity, and maintainability.

File versions.csv contains general information (version unique ID, number of classes, packages, external classes, external packages, version repository link) to provide an overview of versions and how over time the repository continues to grow.

File attribute-details.csv contains detailed information (attribute name, attribute short form, category, and description) about extracted static analysis metrics and code quality attributes. The short form is used in the real dataset as a unique identifier to show value for packages, classes, and methods.

File codesmells.csv provides the information (Rule, Code Smell, Rule Description) of code smells analyzed from each version.

Table A.1: Dataset Specifications

Subject	Software
Specific subject area	The provided dataset lies in the area of software engineering, specifically in the code smell, software quality, maintenance, and evolution.
Type of data	Table
How data were acquired	Repositories were searched using the ModelMine tool [8] and were downloaded based on the conditions of popularity (minimum of 3,000 stars and 500 forks) and well-maintained (minimum of 5,000 commits). After repository collection, code quality and source code metrics data were extracted using the CODEMR tool [4], and code smell data were extracted using the PMD tool [172]. We use Python scripts to create a combined dataset by synthesizing both the source code metrics and the code smell dataset.
Data format	1) Raw 2) Analyzed 3) Synthesized
Parameters for data collection	Upon selecting the most popular & well-maintained repositories using the ModelMine tool, we filtered and selected the repositories with the following criteria; a repository with primary language *Java*, a minimum of 5,000 commits, at least 100 active contributors, a minimum of 3,000 stars, and 500 forks. We consider the high number of stars and forks as a proxy for the popularity of repositories and the high number of commits as a proxy for maintenance. On the other hand, code smells information was extracted using the PMD tool [174] and filtered using rules of code smells.
Description of data collection	The source code of the repositories were retrieved using the ModelMine tool [8]. The selected repositories are downloaded from their source location with the versions between 2016-2021. We have conducted the static analysis for each version at the class, package, and method level using the CODEMR static analysis tool [4] and collected code smell data using the PMD code analyzer tool [172]. Both code smell and quality metrics are then analyzed, synthesized, and reported in this paper.
Data source location	Software Engineering Laboratory, Department of Computer Science University of Texas at El Paso, Texas, USA.

Data accessibility	Repository name: CSQ - Code smells and quality dataset [173] DOI: `http://dx.doi.org/10.17632/77p6rzb73n`
Related research article	Sayed Moshin Reza, Md. Mahfujur Rahman, Hasnat Parvez, Omar Badreddin, and Shamim Al Mamun. Performance Analysis of Machine Learning Approaches in Software Complexity Prediction. In 2020 International Conference on Trends in Computational and Cognitive Engineering (TCCE), Springer, 2020 [9]. DOI: `https://doi.org/10.1007/978-981-33-4673-4_3`

A.2 Significance of the Data

- The data primarily benefit software engineering researchers interested in various source code analysis or code smell tasks over versions and predict future software quality using a data-driven or machine learning approach.
- The provided dataset contains forty-seven unique static analysis metrics that can be used to detect code smell and thus can be used as the information basis for the software maintainability.
- The provided dataset contains code quality metrics that are directly related to various open research questions in the areas of software quality, maintainability, and sustainability. The data is capable of constituting a valuable ground truth for researchers and practitioners in the field of software quality estimation.
- The data contains information at different software component levels (package, class, and method) of sixty versions of ten repositories varying both in terms of size and functionality and thus covers a wide range of data variations.

A.3 Data Description

The provided dataset contains three folders: *codesmells*, *quality_attributes*, *synthesized* and four associated Comma Separated Values (CSV) files: *repositories.csv*, *versions.csv*, *codesmells.csv* and *attribute − details.csv*. The first file (*repositories.csv*) contains general information (repository name, URL, number of commits, stars, forks, etc.) to understand the size, popularity, and maintainability. Table A.2 presents the information included in *repositories.csv*.

Table A.2: Selected Repositories with Metadata Information

Serial	Repository name	Commits	Contributors	Stars	Forks
1	Spring framework	22,208	531	41,400	28,800
2	Junit-5	6,621	161	4,400	991
3	Apache kafka	8,590	762	18,000	9,600
4	Apache lucene-solr	34,789	232	4,100	2,700
5	Dropwizard	5,702	361	7,900	3,300
6	Checkstyle	9,922	254	5,800	7,700
7	Hadoop	24,612	339	11,300	7,000
8	Selenium	26,532	558	19,800	6,200
9	Skywalking	6,242	315	16,100	4,700
10	Signal android	7,015	223	19,800	4,700

File *versions.csv* contains general information (unique version identification, number of classes, packages, external classes, external packages, version repository link, etc.) to provide an overview of versions and how the repository continues to grow over time. Table A.3 shows the metadata information related to each version of the repositories provided in *versions.csv*.

Table A.3: Version & Metadata Information of Selected Repositories

No	Repository name	Version	Commits	Lines of code	Classes	Packages
1	Spring framework	2021-1	22,029	237,160	5,743	442
2		2020-1	20,155	225,059	5,450	433
3		2019-1	17,649	205,268	5,025	394
4		2018-1	15,800	194,646	4,864	388
5		2017-1	13,862	178,744	4,658	383
6		2016-1	11,454	175,826	4,440	368
7	Junit-5	2021-1	6,529	17,372	688	70
8		2020-1	6,025	15,676	569	64
9		2019-1	5,244	13,540	500	61
10		2018-1	3,987	8,636	318	40
11		2017-1	2,821	6,575	243	33
12		2016-1	871	2,843	137	19

No	Repository name	Version	Commits	Lines of code	Classes	Packages
13	Apache kafka	2021-1	8,348	130,620	2,674	137
14		2020-1	6,943	111,332	2,375	124
15		2019-1	5,728	88,166	1,993	114
16		2018-1	4,514	71,958	1,682	89
17		2017-1	3,014	38,961	942	62
18		2016-1	1,857	26,423	661	56
19	Apache lucene-solr	2021-1	34,550	600,393	8,746	421
20		2020-1	32,842	588,425	8,713	394
21		2019-1	31,221	527,217	7,891	355
22		2018-1	29,186	497,523	7,420	343
23		2017-1	26,390	432,647	6,393	333
24		2016-1	23,932	389,276	5,786	308
25	Dropwizard	2021-1	5,643	14,460	514	88
26		2020-1	5,220	14,059	506	88
27		2019-1	4,681	14,057	554	93
28		2018-1	4,356	13,002	521	87
29		2017-1	4,044	11,463	477	81
30		2016-1	3,484	10,208	407	69
31	Checkstyle	2021-1	9,832	27,751	483	35
32		2020-1	8,936	25,791	445	26
33		2019-1	8,162	25,315	430	26
34		2018-1	7,469	24,992	416	26
35		2017-1	6,248	22,770	389	24
36		2016-1	5,507	20,416	366	23
37	Hadoop	2021-1	24,479	709,632	10,676	677
38		2020-1	23,378	687,561	10,462	687
39		2019-1	20,879	686,289	10,592	743
40		2018-1	17,547	594,562	9,066	573
41		2017-1	15,188	489,835	7,711	460
42		2016-1	12,619	431,280	6,978	431

No	Repository name	Version	Commits	Lines of code	Classes	Packages
43	Selenium	2021-1	26,202	41,483	1,290	138
44		2020-1	24,579	32,347	1,079	105
45		2019-1	23,328	32,416	1,045	100
46		2018-1	21,649	27,274	910	72
47		2017-1	19,817	24,648	835	67
48		2016-1	18,306	33,234	929	71
49	Skywalking	2021-1	6,145	72,800	3,001	765
50		2020-1	5,393	50,974	2,224	540
51		2019-1	4,503	29,773	1,238	269
52		2018-1	3,152	23,363	1,113	247
53		2017-1	1,287	8,544	379	98
54		2016-1	282	7,194	253	80
55	Signal android	2021-1	6,654	136,947	3,338	209
56		2020-1	4,849	91,578	2,239	141
57		2019-1	3,930	63,589	1,357	85
58		2018-1	3,339	46,727	1,075	67
59		2017-1	2,601	47,528	1,078	71
60		2016-1	2,299	42,098	950	56
Total 60 version of repositories			700,938	9,420,246	173,237	12,849

File *attribute − details.csv* contains detailed information (attribute name, attribute short form, category, and description) about extracted static analysis metrics and code quality attributes. The short form is used in the real dataset as a unique identifier to show value for packages, classes, and methods. Table A.4 lists the source code metrics and code quality attributes used in our dataset.

Table A.4: List of Source Code Metrics and Code Quality Attributes

No	Category	Code	Full name	Description
1	Package, Class, Method	Coupling	Coupling	Measures coupling value between two classes A and B.

No	Category	Code	Full name	Description
2	Package, Class, Method	Lack of Cohesion	Lack of Cohesion	Measure how well the methods of a class are related to each other.
3	Package, Class, Method	Complexity	Complexity	Implies being difficult to understand and describe the interactions between a number of entities.
4	Package, Class, Method	Size	Size	Measured by the number of lines or methods in the code.
5	Package, Class	LOC	Lines of Code	The number of all nonempty, noncommented lines of the body of the class.
6	Package, Class	WMC	Weighted Method Count	The weighted sum of all class methods.
7	Class	DIT	Depth of Inheritance Tree	The position of the class in the inheritance tree.
8	Class	NOC	Number of Children	The number of direct subclasses of a class.
9	Class	CBO	Coupling Between Object Classes	The number of classes that a class is coupled to.
10	Class	CBO_LIB	CBO Lib	The number of dependent library classes.
11	Project	CBO_APP	CBO App	The number of dependent classes in the application.
12	Class	RFC	Response For a Class	The number of the methods that can be potentially invoked in response to a public message received by an object of a particular class.
13	Class	SRFC	Simple Response For a Class	The number of the methods that can be potentially invoked in simple response to a public message received by an object of a particular class.
14	Class	LCOM	Lack of Cohesion of Methods	Measure how methods of a class are related to each other.

No	Category	Code	Full name	Description
15	Class	LCAM	Lack of Cohesion Among Methods	CAM metric is the measure of cohesion based on parameter types of methods. LCAM = 1-CAM.
16	Class	NOF	Number of Fields	The number of fields (attributes) in a class.
17	Class	NOM	Number of Methods	The number of methods in a class.
18	Class	NOSF	Number of Static Fields	The number of static fields in a class.
19	Class	NOSM	Number of Static Methods	The number of static methods in a class.
20	Class	SI	Specialization Index	Measures the extent to which subclasses override their ancestors' classes.
21	Class	CMLOC	Class-Methods Lines of Code	The total number of all nonempty, non-commented lines of methods inside a class.
22	Package	EC	Efferent Coupling	Outgoing Coupling. The number of classes in other packages that the classes in the package depend upon is an indicator of the package's dependence on externalities.
23	Package	AC	Afferent Coupling	Incoming Coupling. The number of classes in other packages that depend upon classes within the package is an indicator of the package's responsibility.
24	Package	#(C&I)	Number of Classes & Interfaces / Entities	Total number of Classes & Interfaces.
25	Package	#I	Number of Interfaces	Total number of Interfaces.
26	Package	#C	Number of Classes	Total number of classes.
27	Class	NORM	Number of Overridden Methods	The number of Overridden Methods.
28	Class	C3	C3	The max value of Coupling, Cohesion, Complexity metrics.
29	Project	nofP	Number of Packages	Number of Packages in the project.
30	Project	nofPa	Number of External Packages	Number of External Packages referenced by the project.

No	Category	Code	Full name	Description
31	Project	nofEE	Number of External Entities	Number of External classes and interfaces referenced by the project.
32	Project	NoPC	Number of Problematic Classes	Number of classes with high coupling, high complexity or low cohesion in the project.
33	Project	NoHPC	Number of Highly Problematic Classes	Number of classes with high coupling, high complexity, and low cohesion in the project.
34	Class	LTCC	Lack of Tight Class Cohesion	Measures the lack of cohesion between the public methods of a class.
35	Class	ATFD	Access to Foreign Data	The number of classes whose attributes are directly or indirectly reachable from the investigated class.
36	Package	Ins	Instability	Measure the relative susceptibility of class to changes.
37	Package	Abs	Abstractness	Measure the degree of abstraction of the package.
38	Package	ND	Normalized Distance	Normalized Distance metric is used to measure the balance between stability.
39	Class	InDegree	InDegree	In-degree of corresponding graph vertex of the class.
40	Class	OutDegree	OutDegree	Out-degree of corresponding graph vertex of the class.
41	Class	Degree	Degree	Degree of corresponding graph vertex of the class.
42	Method	MCC	McCabe Cyclomatic Complexity	McCabe Cyclomatic Complexity
43	Method	NBD	Nested Block Depth	Number of statement blocks that are nested due to the use of control structures (branches, loops).
44	Method	LOC.2	Method Lines of Code	Number of Lines of Code under a method.

No	Category	Code	Full name	Description
45	Method	#Pa	Number of Parameters	Number of Parameters.
46	Method	#MC	Number of Methods Called	Number of Methods Called.
47	Method	#AF	Number of Accessed Fields	Number of Accessed Fields.

File *codesmells.csv* provides the information (Rule, Code Smell, Rule Description) of code smells analyzed from each version. Table A.5 lists the code smells and their associated rules.

In addition to these files, the dataset contains a folder having ten sub-folders named against each repository name. Each sub-folder has six CSV files named according to the version. The CSV files contain analyzed static analysis metrics and code smell information.

A.4 Experimental Design, Materials, and Methods

The dataset construction process involves the following distinct steps:

The first step involves the selection of the repositories to be analyzed. Towards this direction, we retrieved information regarding the most popular repositories using the ModelMine tool [8] with the following criteria; a repository with primary language *Java*, a minimum of $5,000$ commits, at least 100 active contributors, a minimum of $3,000$ stars and 500 forks. We have selected ten diverse repositories out of 47 searched and filtered repositories. The selected repositories are open-source and can be downloaded from the GitHub link provided in A.5.

To validate the variations of the repositories, we consider a high number of stars and forks as a proxy for the popularity of repositories and a high number of commits as a proxy for maintenance. Also, we consider repository size as follows: low (1–1,000 classes), medium (1001–5,000 classes), and high (more than 5,000 classes) in the latest version of each repository.

After selecting repositories to be analyzed, the next step involves acquiring their version and source code. We use the ModelMine tool again to find the version link of the repositories each year between 2016-2021. The selected versions are open-source and can be downloaded from GitHub by visiting the GitHub tree link provided in A.6.

After collecting the source code of each version of the selected repositories, the next step involves analyzing the source code and extracting source code metrics and code quality attributes from each version.

Table A.5: List of Code Smells and Their Associated PMD Rule Names

No	Code Smell	Description	Rule
1	God Class	A class does too many things, is very big, and is overly complex.	GodClass
2	Long Method	A method does more than its name/signature suggests.	ExcessiveMethodLength
3	Data Class	A class holds simple data, which reveal most of their state without complex functionality.	DataClass
4	Long Parameter List	A method with numerous parameters challenged to maintain.	ExcessiveParameterlist
5	Large Class	A class may be burdened with many responsibilities provided by external classes or functions.	ExcessiveClassLength
6	Complex Class	A class that concentrates too much decisional logic in methods makes its behavior hard to read and change.	CyclomaticComplexity
7	Switch Statement	A switch statement implies a method with a switch statement is overloaded.	SwitchDensity
8	Class Data Should Be Private	A class with large numbers of public methods and attributes.	ExcessivePublicCount

We perform static analysis using the CODEMR, static analysis tool [4], which enables the computation of a series of metrics that quantify forty-seven different source code metrics and code quality attributes. The analysis was performed at package, class, and method levels. A list of source code metrics and code quality attributes, along with the description and acronyms, are shown in Table A.4.

Besides collecting the code quality attributes, we extracted design-related code smells information using the PMD tool [172] from the repositories. The code smell information for each repository is stored as CSV files in the *codesmells* folder. To extract the design-related code smells, we used a PMD command shown in Table A.6 and applied the parameters: (1) directory (-d), (2) rule (-R), and (3) output format (-f).

Table A.6: PMD Command & Parameters to Extract Code Smells

Command	pmd.bat **-d** "path_to_repository" **-R** category/java/design.xml **-f** csv >> "output_file_name.csv"
Parameters	-d <path>: Root directory for the analyzed sources. -R <refs>: Comma-separated list of ruleset or rule references. -f <format>: Output format of the analysis report.

Finally, we created a synthesized version of source code metrics, code quality, and code smell information by class and method name of each repository version. This process is done using a Python script and takes longer as we merge the data over a non-indexed column: Class/method name. Generally, By simply merging using the index column, the speed increases from 10 to 15% over a non-indexed column [175].

A.5 Repository Links

The table A.7 shows the repository link information that we mine from the GitHub repositories of interest.

Table A.7: Selected Repository Links

Serial	**Repository name**	**Repository link**
1	Spring framework	`https://github.com/spring-projects/spring-framework`
2	Junit-5	`https://github.com/junit-team/junit5`
3	Apache kafka	`https://github.com/apache/kafka`
4	Apache lucene-solr	`https://github.com/apache/lucene-solr`
5	Dropwizard	`https://github.com/dropwizard/dropwizard`
6	Checkstyle	`https://github.com/checkstyle/checkstyle`
7	Hadoop	`https://github.com/apache/hadoop`
8	Selenium	`https://github.com/SeleniumHQ/selenium`
9	Skywalking	`https://github.com/apache/skywalking`
10	Signal android	`https://github.com/signalapp/Signal-Android`

A.6 Version links

The following table shows the version link information we mine repositories of interest from GitHub. We download a total of sixty versions of ten repositories.

Table A.8: Version Link of Selected Repositories

No	Repository name	Version	Version link
1	Spring framework	2021-1	https://github.com/spring-projects/spring-framework/tree/44b29b6ecfdf77aec59409601301ca8e21452f0e
2		2020-1	https://github.com/spring-projects/spring-framework/tree/d5f0bb23aed29578cb45653f14d291d1bfe291e7
3		2019-1	https://github.com/spring-projects/spring-framework/tree/72fbbb24035c79f0c95e698aa26f913497b26323
4		2018-1	https://github.com/spring-projects/spring-framework/tree/0f1f95e0909c5d32bbc9305ae85c57312a491058
5		2017-1	https://github.com/spring-projects/spring-framework/tree/badde3a479a53e1dd0777dd1bd5b55cb1021cf9e
6		2016-1	https://github.com/spring-projects/spring-framework/tree/d681f77d625d4815563511e6abb96827c8aec2a5
7	Junit-5	2021-1	https://github.com/junit-team/junit5/tree/03a79c13f612099a8fc328aa2bcff23e4f8dcc2c
8		2020-1	https://github.com/junit-team/junit5/tree/20a9fe00e795b9234b6e9abb2e9a172d347625ff
9		2019-1	https://github.com/junit-team/junit5/tree/56ac9fde793903fe6fd36ec2d3b09e3fbed91f7d
10		2018-1	https://github.com/junit-team/junit5/tree/4b3b36e1b2b2069600f2fcf8fcf5385b0278b2c1
11		2017-1	https://github.com/junit-team/junit5/tree/73f3eeb436e78226f19554f2c2b5dab63e103670
12		2016-1	https://github.com/junit-team/junit5/tree/13d39cfbd9a42d3e6d6679e869caf98b5e3386de
13	Apache kafka	2021-1	https://github.com/apache/kafka/tree/2515bf236864c04cc75d8fd136b048625663e16c
14		2020-1	https://github.com/apache/kafka/tree/f610f9ff1f59f90434c3614f00c95001f50100e4
15		2019-1	https://github.com/apache/kafka/tree/b16afbb77bc1a497096815e64ed9e97df1edf92d
16		2018-1	https://github.com/apache/kafka/tree/96df93522f84173ff47f47ec78ec408991140b65
17		2017-1	https://github.com/apache/kafka/tree/ce1cb329d5aa788968e47d7dfe307128f2ddc2ff
18		2016-1	https://github.com/apache/kafka/tree/b905d489188768ba1c55226857db9713b9272918
19	Apache lucene-solr	2021-1	https://github.com/apache/lucene-solr/tree/beb163c9160bf983ac94c33eeee9659dc061ff6a
20		2020-1	https://github.com/apache/lucene-solr/tree/1e0471a2476d66ff64e866b354253b8e76bcd7c7
21		2019-1	https://github.com/apache/lucene-solr/tree/5016959ce8c1eed9d354822f01edc4b509e4aa9d
22		2018-1	https://github.com/apache/lucene-solr/tree/2da4ed17bae07593233f4e5610ce40a6a07f7c10
23		2017-1	https://github.com/apache/lucene-solr/tree/93562da610bf8756351be7720c69872bc1cea727
24		2016-1	https://github.com/apache/lucene-solr/tree/c9b7af045085bb44fada61cfa877f0238769ff72
25	Dropwizard	2021-1	https://github.com/dropwizard/dropwizard/tree/dda5bb6990607f6b2e9ca00e8156c3bf2cf480d6
26		2020-1	https://github.com/dropwizard/dropwizard/tree/1ae6ea6a62329d3715fa62d3f85b620178f8da9b
27		2019-1	https://github.com/dropwizard/dropwizard/tree/f000627a307d12280e39875da99caf685d1950d3
28		2018-1	https://github.com/dropwizard/dropwizard/tree/6e5c9c5aef82fe7d25097d837854c6d0778f86c7
29		2017-1	https://github.com/dropwizard/dropwizard/tree/a43e4b96df5b36a8cfd795bd5fd382e40a6733b3
30		2016-1	https://github.com/dropwizard/dropwizard/tree/3d78e02c3daef8d4acd78c570a4363db07eedeea
31	Checkstyle	2021-1	https://github.com/checkstyle/checkstyle/tree/c3e5dfff8ea8ec8ef5ed1ea12f86c1110445707a
32		2020-1	https://github.com/checkstyle/checkstyle/tree/5e7809c096db317143f38cf7fbe4d2a535836b85
33		2019-1	https://github.com/checkstyle/checkstyle/tree/a262bad94bb4aa5786a2c47582021dc1189208ec
34		2018-1	https://github.com/checkstyle/checkstyle/tree/327c0bc843612486ab4ded32a2f01038e1271fd0
35		2017-1	https://github.com/checkstyle/checkstyle/tree/16aadebeee91937b3fadaa6f911e6ce0a97863b1
36		2016-1	https://github.com/checkstyle/checkstyle/tree/af047afee8216a4b0db9027dd013a1aad2be9494
37	Hadoop	2021-1	https://github.com/apache/hadoop/tree/1448add08fcd4a23e59eab5f75ef46fca6b1c3d1
38		2020-1	https://github.com/apache/hadoop/tree/b6dc00f481189821e5d982083eba6d01f108b3de
39		2019-1	https://github.com/apache/hadoop/tree/cb26f154289ed065a967886b8eac04794907d643
40		2018-1	https://github.com/apache/hadoop/tree/dfe0cd86553bd2688603ea382ea593171d520471
41		2017-1	https://github.com/apache/hadoop/tree/b31e1951e044b2c6f6e88a007a8c175941ddd674
42		2016-1	https://github.com/apache/hadoop/tree/f9e36dea96f592d09f159e521379e426e7f07ec9

No	Repository name	Version	Version link
43	Selenium	2021-1	https://github.com/SeleniumHQ/selenium/tree/4af354bb99abf3191473fa32f636063311a378d5
44		2020-1	https://github.com/SeleniumHQ/selenium/tree/58249b7943198e92ce083f42052380fa2dbcfd61
45		2019-1	https://github.com/SeleniumHQ/selenium/tree/d25b01eca78ae9982e3c1fed7f2294a91c186f54
46		2018-1	https://github.com/SeleniumHQ/selenium/tree/f3eafa022fe42e3aa200821f3175350b611209c8
47		2017-1	https://github.com/SeleniumHQ/selenium/tree/9a39af7619b7165af80bd6d7b688369479baeeed
48		2016-1	https://github.com/SeleniumHQ/selenium/tree/4c35228399ed9b0610d41e5b8c758563c129efbc
49	Skywalking	2021-1	https://github.com/apache/skywalking/tree/61011635135cfe777370db59f0988d5a3c546dd2
50		2020-1	https://github.com/apache/skywalking/tree/568c2e53f09855199884f65f38a9e4771a5a6467
51		2019-1	https://github.com/apache/skywalking/tree/8506f8f3c5afb1af762bcd7b3d221121bf242ea7
52		2018-1	https://github.com/apache/skywalking/tree/e5ea6cf33154f4319a82029038ca533e4e593384
53		2017-1	https://github.com/apache/skywalking/tree/a944427df8f85f9881169ed3f342e36091f4d3e8
54		2016-1	https://github.com/apache/skywalking/tree/c1a90c9b7f26cce27e6fa13ecf055dd78eff1163
55	Signal android	2021-1	https://github.com/signalapp/Signal-Android/tree/ccd405fdce5f3d0a3f934e9ac02a4f0e33c9ed10
56		2020-1	https://github.com/signalapp/Signal-Android/tree/fe5fca8eaf3f73ab5b350d7d6b3c17b27729d92a
57		2019-1	https://github.com/signalapp/Signal-Android/tree/1c3052a580f8d40f235f0e3d2da9c4cda2f3860e
58		2018-1	https://github.com/signalapp/Signal-Android/tree/b307980d8ca0da588213d2cd448474ffa925d1c2
59		2017-1	https://github.com/signalapp/Signal-Android/tree/57cdbaedd646529ede41749fce0f16e1fe0b5ea3
60		2016-1	https://github.com/signalapp/Signal-Android/tree/df27fa47ed92f941e887e40974ae5f217f999294

Appendix B

Machine Learning in Code Smell Detection

This appendix presents different results regarding recent studies on machine learning algorithms for code smell detection considering the research articles published between 2015 and 2021.

B.1 Data Sources Links

Table B.1: Data Sources and Links

Serial	**Data source**	**Data source URL**
1	ACM Digital Library	https://dl.acm.org/
2	IEEE Xplore Digital Libray	https://ieeexplore.ieee.org/
3	ScienceDirect	https://www.sciencedirect.com/
4	SpringerLink Digital Library	https://link.springer.com/
5	Engineering Village Digital Library	https://www.engineeringvillage.com/
6	Wiley Online Library	https://onlinelibrary.wiley.com/

B.2 List of Questions for Data Collection

Table B.2: List of Questions & Metadata for Data Collection

Serial	Metadata	Description	Rationale
1	Code smells studied	What code smells are reported in the papers?	To answer research questions
2	Datasets used to train models	What datasets are used in the papers to train machine learning models?	
3	Independent variables	The classifier used to measure the proneness of code smell?	
4	Machine Learning algorithms applied	Which machine learning algorithms are considered in the research articles?	
5	Evaluation metrics used	What evaluation metrics have been used to assess the accuracy of the model?	
6	Demographic information of the authors	Which regions are employing machine learning in code smell-related research?	To conduct exploratory data analysis
7	Publication venues	In which conferences or journals are the papers published?	
8	Publication year	In which year were the articles published?	

B.3 Dataset Used in the Primary Studies

This section shows the results related to the reported dataset in recent studies.

Table B.3: Datasets Reported in the Primary Studies

Serial	Dataset	Primary Studies	Dataset Type
1	Qualitas Corpus	[S01], [S06], [S11], [S13], [S17], [20] [S22], [S25]	Open Source
2	Code Smell 13 repository dataset	[S04], [S07], [S14], [S18]	Open Source
3	Self generated dataset	[S15], [S23], [24]	Industrial
4	Two open source java projects [GanttProject, Apache Xerces]	[S05], [S21]	Open Source
5	Landfill	[S02]	Open Source
6	Android Smell Dataset (Created Manually)	[S03]	Industrial
7	Code Smell Github Dataset	[S08]	Open Source
8	Seven open source projects	[S09]	Open Source
9	Fourteen Industrial Software Systems	[S10]	Industrial
10	Two projects [ArgoUML, Apache Xerces]	[S12]	Open Source
11	Eight Open Source Projects	[S19]	Open Source
12	BrainCode Dataset	[S26]	Industrial

B.4 Independent Variables Considered in the Primary Studies

Table B.4: Independent Variables Considered in the Primary Studies

Article ID	**Metrics**	**Number of Metrics**
S01	CBO, CYC, DAC, DIT, ILCOM, LCOM, LD, LEN, LOC, LOD, MPC, NAM, NOC, NOM, RFC, TCC, WMC, MNB, uniqueWordsQty, assignmentsQty, comparisonsQty, loopQty, parenExpsQty, variables, parameters, startLine	26
S02	CLOC, LOC, WMC	3
S03	NoM, DoI, NoII, NoA, CLC, LoC, isAbstract, isStatic, isInnerClass, isInterface, isActivity, isBroadcastReceiver, isAsyncTask, ownOnLowMemory, NoP, NoI, NoDC, NoC, CC, NatureOfClass, callExternalMethod, callMethod, useVariable, callInit	24
S04	ELOC, LCOM, LOC_METHOD, NOA, NOM, NOPA, NP, NMNOPARAM, WMC	9
S05	LOC, CC	2
S06	LOC, LOCNAMM, NOPK, NOCS, NOM, NOMNAMM, NoA, WMC, WMCNAMM, AMW, AMWNAMM, CLNAMM, NOP, NOAV, ATLD, NOLV, FANOUT, ATFD, FDP, RFC, CBO, CFNAMM, NMCS, CC, CM, NOAM, DIT, NOC, NoII	29
S07	ELOC, LCOM, LOC_METHOD, NOA, NOM, NOPA, NP, NMNOPARAM, WMC	9
S08	NOC, IPM, WMC, CC, NOCH, NBI, NOM, DIT, PPIV, LCOM, LOCL, APD, XML, BSMC, NTO, WKL, GPS, BMAP, SQL, NET, I/O	22
S09	LOC, CBO, DIT	3
S10	LCOM, DIT, IFANIN, CBO, NOC, RFC, NIM, NIV, WMC, Cyclo	10
S11	RFC, PM, AFD, LOC, LOCNA, TCC, WMC, WMCNA, NPA, parameters, CC	11
S12	NSM, TLOC, CA, RMD, NOC, SIX, RMI, NOF, NOP, MLOC, WMC, NORM, NSF, NBD, NOM, LCOM, VG, PAR, RMA, NOI, CE, NSC, DIT	23

Article ID	**Metrics**	**Number of Metrics**
S13	LOC, LOCNAMM, NOPK, NOCS, NOM, NOMNAMM, NoA, CC, WMC, WMCNAMM, AMW, AMWNAMM, CLNAMM, NOP, NOAV, ATLD, NOLV, FANOUT, ATFD, FDP, RFC, CBO, CFNAMM, NMCS, CC, CM, NOAM, DIT, NOC, NoII	30
S14	ELOC, LCOM, LOC_METHOD, NOA, NOM, NOPA, NP, NMNOPARAM, WMC	9
S15	WMC, NOV	2
S16	CBO, CYC, DAC, DIT, ILCOM, LCOM, LD, LEN, LOC, LOD, MPC, NAM, NOC, NOM, RFC, TCC, WMC, maxNestedBlocks, uniqueWordsQty, assignmentsQty, comparisonsQty, loopQty, parenExpsQty, variables, parameters, startLine	27
S17	LOC, LOCNAMM, NOPK, NOCS, NOM, NOMNAMM, NoA, CC, WMC, WMCNAMM, AMW, AMWNAMM, CLNAMM, NOP, NOAV, ATLD, NOLV, FANOUT, ATFD, FDP, RFC, CBO, CFNAMM, NMCS, CC, CM, NOAM, DIT, NOC, NoII	30
S18	ATFD, ELOC, FanIn, FanOut, LCOM, LOC_METHOD, McCabe, MC, NOA, NOC, NOM, NOPA, NP, NMNOPARAM, PDM, PRM, WMC, LOC	18
S19	ATFD, LAA, FDP	3
S20	LOC, LOCNAMM, NOPK, NOCS, NOM, NOMNAMM, NoA, CC, WMC, WMCNAMM, AMW, AMWNAMM, CLNAMM, NOP, NOAV, ATLD, NOLV, FANOUT, ATFD, FDP, RFC, CBO, CFNAMM, NMCS, CC, CM, NOAM, DIT, NOC, NoII	30
S21	LOC, CC	2
S22	WMC, CFNAMM, NOAM, NIM, ATFD, LAA, Cyclo	7
S23	LOC, CC	2
S24	LOC, CBO, NOM, CC, FO, FI, LCOM	7

Article ID	**Metrics**	**Number of Metrics**
S25	LOC, LOCNAMM, NOPK, NOCS, NOM, NOMNAMM, NOA, CYCLO, WMC, WMCNAMM, AMW, AMWNAMM, MNB, CLNAMM, NOP, NOAV, Fanout, ATFD, FDP, RFC, CBO, CFNAMM, CINT, MaMCL, MeMCL, NMCS, CC, CM, CDISP, NOAM, NOPA, LAA, DIT, NoII, NOC, NMO, NIM, NoII, NOCS, WOC_Tyoe, NODA, NOPVA, NOPRA, NOFA, NOFSA, NOFNSA, NONFNSA, NOSA, NONFSA, NOABM, NOCM, NONCM, NOFM, NOFNSM, NONSM	55
S26	WMC, NOC, NOM, NONM, NOIM, TLOC, MLOC, CIS, NOPM	28

B.5 List of Source Code Metrics and Definitions

Table B.5: Source code metric names along with their definitions

Name	Definition	Name	Definition
AMW	Average Method Weight	NMCS	Number of Message Chain Statements
APD	Access to Private Data	NOA	Number of Anchestors
ATFD	Access to Foreign Data	NOABM	Number of Abstract Methods
ATLD	Access to Local Data	NOAV	Number of Accessed Variable
CBO	Coupling Between Objects	NOC	Number of Children
CC	Class Complexity	NOCM	Number of Construction Methods
CFNAMM	Called Foreign Not Accessor or Mutator Methods	NOCS	Number of Classes
CINT	Coupling Intensity	NODA	Number of Default Attributes
CIS	Class Interface Size	NOF/NOA	Number of Fields/Attributes
CLOC	Class Lines of Code	NOFA	Number of Fields Accessed
CM	Changing Methods	NOFM	Number of Final Methods
CYC / CLC	Cyclomatic complexity	NOII	Number of Implemented Interfaces
DAC	Data Abstraction Coupling	NOIM	Number of Inherited Methods
DIT	Depth Inheritance Tree	NOLV	Number of Local Variables
DOI	Depth of Inheritance	NOM	Number of Methods
ELOC	Estimated Lines of Code	NONM	Number of Normal Methods
FANIN/FI	Max number of references to the subject class to another class	NOPA	Number of Public Attributes
FANOUT/FO	Number of methods and fields used by one entity	NOPVA	Number of Private Attributes
FDP	Foreign Data Provider	NOPM	Number of Polymorphic Methods
LAA	Locality of Attribute Accesses	NOSA/NOSF	Number of static attributes
LCOM	Lack of Cohesion in Methods	NOV	Number of Variable
LD	Locality of Data	NOPK, NP	Number of Packages
LEN	Length of Class Names	NPA	Number of Protected Attributes
LOC	Lines of Code	NSC	Number of static Classes
LOD	Lack of Documentation	NSM	Number of Static Methods
MC/MCC	Mccabe Cyclomatic Complexity	PDM	Percentage of Delegate Methods
MLOC/LOCM	Method Lines of Code	PM	Number of Private Methods
MN	Max Nesting Level	PRM	Percentage of Refused Methods
MNB	Max Nested Block	RFC	Responses for a Class
MPC	Message Passing Coupling	TCC	Tight Class Cohesion
NOAM	Number of Accessor Methods	TLOC	Total Lines of Code
NIM	Number of Inherited Methods	WMC	Weighted Method Call

Biosketch

Sayed Mohsin Reza is a Ph.D. candidate in the field of Computer Science at the University of Texas at El Paso. He received his Master of Science in Computer Science from the same institution in 2021 and a Bachelor of Science in Information Technology from Jahangirnagar University in Bangladesh in 2014. His research interests lie at the intersection of Software Engineering, Machine Learning, Software Quality, and Data Science.

His current research focuses on exploring the application of machine learning techniques in identifying code quality, code smells, and refactoring techniques. He has developed a tool called "ModelMine" which enables efficient mining of models and designs from open-source repositories and ranks repositories based on the presence of models and designs.

In addition to his research in software engineering, Mr. Reza is also interested in data science-based CS curriculum design that is culturally relevant. He draws inspiration from the foundational work of Gloria Ladson Billings in framing learner experiences that are connected to personal interests, cultural backgrounds, and the sociopolitical landscape.

Mr. Reza's undergraduate research work was on the development of a conference management system named PROCONF aimed at improving the submission and review process. This system is currently in use by multiple international conferences and journals and is hosted on the website, `http://www.proconf.org/`. He serves as the CEO of the software company that developed the system.

Contact Information: smrezaiit@gmail.com