Empirical studies on managing code clone evolution using machine learning techniques

By Osama Ehsan

A thesis submitted to the Graduate Program in Electrical And Computer Engineering in conformity with the requirements for the Degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
June, 2023

Abstract

The duplication of code snippets with or without minor modifications creates a code clone. The number of code clones is expected to increase as the software expands in size. The maintenance of code clones can be challenging and costly for software maintainers as software evolves. For example, changes made to one copy of a code snippet may require updating the other copies. Prior studies show that there can be up to 21% code clones in a software project and the inconsistent nature of code clone evolution can induce software bugs in the future. Poorly maintained software projects are often closed or abandoned. Software developers have limited resources to keep track of code clones and fix the code clones that can be harmful. Prior studies focus on the detection of code clones and analysis of the clone patterns. However, there is limited support provided to developers to work on more harmful clones. In this thesis, we leverage the information available (e.g., issue reports, commits, and pull requests) from open-source software projects. We apply AI and machine learning approaches to help the software developers monitor the evolution of clones and provide intelligent approaches to help developers maintain the code clones better. In particular, we conduct four studies: (1) a study to rank the code clones in software projects based on the bug-proneness of clones to have fewer bugs in the future; (2) a study to estimate the effort needed to propagate code clone changes to clone siblings

and conduct a user survey to identify the usefulness of our approach; (3) an empirical study to predict late propagation and the associated bugs in software projects; and (4) a study to predict the degree of inconsistency in a clone group, the lifetime of bugs to survive in a software project, and suggest if inconsistent changes in a clone group should be propagated or not at the pull request level. Overall, the research in this thesis provides intelligent approaches to aid developers to better maintain the evolution of the code clones.

Acknowledgments

First of all, I would like to thank Almighty Allah that he has given me the strength and ability to complete this thesis. I would like to take this opportunity to express my heartfelt gratitude to everyone who has contributed to the completion of this thesis.

Next, I would like to thank my supervisor, Dr Ying Zou, for her invaluable guidance, support, and encouragement throughout my research. Her expertise, insights, and feedback have been critical to the success of this thesis. I thank her for being an amazing supervisor.

I am also deeply grateful to my examining committee members, Dr. Chanchal Roy, Dr. Jianbing Ni, Dr. Yuan Tian, and Dr. Ning Lu for their insightful comments and suggestions, which have greatly improved the quality of this thesis. Their constructive criticism and challenging questions have pushed me to think more deeply and critically about my research. I would like to thank my intelligent colleagues at Software Evolution Analytics Lab (SEAL) including Dr. Daniel Alencar da Costa, Dr. Safwat Hassan, Dr. Stefanos Georgiou, Dr. Mariam El Mezouar, Dr. Yu Zhao, Dr. Guoliang Zhao, Dr. Taher Ghaleb, Shayan Noei, Chunli Yu, Omar El Zarif, Maram Assi, Jieyeon Woo, Aidan Yang, Tanghaoran Zhang, Fangjian Lei, Zitong Su, and Bihui Jin who shared the PhD experience with me.

I am also grateful to my family and friends, especially my parents for their unwavering support, encouragement, and patience throughout this long and challenging journey. My parents have put in immense hard work throughout my education journey, they deserve all the praise. Their love and understanding have been a constant source of strength and motivation for me.

Lastly, I extend my sincere gratitude to everyone who has contributed to this thesis.

Statement Of Originality

The following works are my own and I hereby certify the intellectual content of this thesis is the product of my own work. The research papers published in the context of this dissertation are co-authored with Dr. Ying Zou, Dr. Foutse Khmoh, Dr. Haoxiang Zhang, and Dr. Dong Qiu. Dr. Ying Zou supervised all the research work conducted in this dissertation. The co-authors have contributed to the success of my work through valuable feedback and suggestions to improve the research quality.

The publications related to this thesis are listed as follows.

- Ehsan, O., Khomh, F., Zou, Y., Qiu, D. Ranking Code Clones to Support Maintenance Activities. Journal of Empirical Software Engineering (EMSE), accepted October 2022

- Ehsan, O., Barbour, L., Khomh, F., Zou, Y. (2021). Is Late Propagation a Harmful Code Clone Evolutionary Pattern? An Empirical Study. Code Clone Analysis: Research, Tools, and Practices, 151-167.

Contents

Abstract i

Acknowledgments iii

Statement Of Originality v

Table of Contents vi

List of Tables xii

List of Figures xv

Chapter 1: Introduction 1

1.1 Introduction 1

1.1.1 Code Clones 2

1.1.2 Software Repositories 4

1.2 Research Problems 5

1.3 Thesis Statement 8

1.4 Thesis Objectives 8

1.5 Outline 11

Chapter 2: Background 13

2.1 Code Clone Detection . . . 13
2.2 Bug-Proneness of Code Clones . . . 16
2.2.1 Bug Introduction and Propagation . . . 16
2.2.2 Code Change Bug Analysis . . . 17
2.3 Analysis on Code Clones . . . 18
2.3.1 Analysis of Clone Genealogies . . . 19
2.3.2 Code Clone Consistency. . . . 20
2.4 Machine Learning Approaches for Code Change . . . 22
2.4.1 Effort Estimation for Code Changes . . . 22
2.4.2 Commit Message Analysis . . . 23
2.4.3 Bug Resolution Duration . . . 25
2.5 Summary . . . 26

Chapter 3: Ranking Code Clones to Support Maintenance Activities 28

3.1 Problem and Motivation . . . 28
3.2 Background . . . 31
3.2.1 Motivating Example . . . 31
3.2.2 Learning-to-Rank Algorithms . . . 33
3.2.3 Classification Approaches . . . 36
3.2.4 Regression Analysis . . . 37
3.2.5 Machine Learning Frameworks . . . 39
3.2.6 Cross Validation . . . 40
3.3 Study Setup . . . 40
3.3.1 Project Selection . . . 41
3.3.2 Building Clone Genealogies . . . 42

3.3.3 Detecting Buggy Clones . . . 46
3.3.4 Collecting Features . . . 48
3.4 Study Results . . . 50
3.4.1 RQ3.1. Can we use learning-to-rank (LtR) algorithms to effectively rank bug-prone code clones? . . . 50
3.4.2 RQ3.2. How well can classification algorithms rank bug-prone clones? . . . 55
3.4.3 RQ3.3. Can we use regression algorithms to predict the proportion of buggy changes in code clones and effectively rank bug-prone clones? . . . 59
3.4.4 RQ3.4. Which metrics are significantly correlated to the risk of bugs in code clones? . . . 65
3.5 Discussion . . . 70
3.6 Threats to Validity . . . 72
3.7 Summary . . . 74

Chapter 4: An Empirical Study on Effort of Propagating Code Changes in Code Clones **76**

4.1 Problem and Motivation . . . 76
4.2 Study Setup . . . 79
4.2.1 Projects Selection . . . 80
4.2.2 Collecting Features . . . 81
4.2.3 User Survey . . . 81
4.2.4 Localizing the Changes in a Clone Group . . . 85
4.3 Study Results . . . 87

4.3.1 RQ4.1. How well can we predict the effort needed to maintain code clones? . . . 87
4.3.2 RQ4.2. What are the efforts for propagating different types of code clone changes at a commit level? . . . 98
4.3.3 RQ4.3. What are efforts for propagating a bug-prone code change that may cause inconsistency among siblings? . . . 106
4.4 Discussion . . . 114
4.5 Threats to Validity . . . 117
4.6 Summary . . . 119

Chapter 5: Is Late Propagation a Harmful Code Clone Evolutionary Pattern? An Empirical Study 121

5.1 Problem and Motivation . . . 121
5.2 Study Setup . . . 124
5.2.1 Project Selection . . . 125
5.2.2 Classification of Genealogies. . . . 126
5.2.3 Detecting Buggy Clones. . . . 127
5.3 Study Results . . . 128
5.3.1 RQ5.1: Are there different types of late propagation? . . . 128
5.3.2 RQ5.2: Are some types of late propagation more bug-prone than others? . . . 130
5.3.3 RQ5.3: Which type of late propagation experiences the highest proportion of bugs? . . . 134
5.3.4 RQ5.4: Can we predict whether a clone pair would experience late propagation? . . . 136

5.4 Discussion . . . 139
5.5 Threats to Validity . . . 140
5.6 Summary . . . 141

Chapter 6: Predicting the change propagation of code clones at a pull request level 143

6.1 Problem and Motivation . . . 143
6.2 Study Setup . . . 147
6.2.1 Selecting Projects . . . 148
6.2.2 Extracting Pull Requests . . . 149
6.2.3 Extracting Bug Information . . . 151
6.2.4 Collecting Features . . . 152
6.3 Study Results . . . 154
6.3.1 RQ6.1: How well can we predict the degree of code clone consistency in a pull request? . . . 154
6.3.2 RQ6.2: How well can we predict the lifetime of a bug at a pull request level? . . . 161
6.3.3 RQ6.3: How well can we predict whether a code clone change in a pull request should be propagated to the siblings? . . . 171
6.4 Discussion . . . 178
6.5 Threats to Validity . . . 182
6.6 Summary . . . 184

Chapter 7: Conclusion 187

7.1 Contributions . . . 187

7.2 Future Work 190

Bibliography **192**

Appendix A: Supporting Data **209**

List of Tables

3.1 The calculated features for the clone pair genealogies 49

3.2 Results of evaluation metrics for LtR algorithms for Java and C projects 54

3.3 10-fold cross-validation results of three classification algorithms and two frameworks, evaluation metrics precision, recall, accuracy, AUC, and F1-score. Java and C projects results presented separately. . . . 58

3.4 10-fold cross-validation results of four regression algorithms and two frameworks, evaluation metrics R-squared, RMSE, MAE. Java and C projects results presented separately. 63

3.5 Results of the mixed-effect model for Model_{all}, Model_{mature}, and Model_{early}. Sorted by χ^2 in decreasing order 69

4.1 The extracted features for the clone group genealogies 82

4.2 Effort estimation metrics used in the study. 90

4.3 10-fold cross-validation results of four regression algorithms and two frameworks, evaluation metrics R-squared, RMSE, MAE. 95

4.4 Results of the commit classification for different types of code changes 101

4.5 Manual labeling results on the statistical sample of perfective commit messages. 104

5.1 Description of selected projects for our study 126

5.2 Summary of late propagation types for the studied open-source projects 128
5.3 Contingency table, chi Square tests results for clone genealogies with and without late propagation. The table shows the values for all the combinations of late propagations and bugs. 131
5.4 Contingency table, chi square tests results for clone genealogies with and without late propagation. 132
5.5 Contingency table with the chi square test for different late propagation types. 133
5.6 Proportion of bugs for each type of late propagation. 134
5.7 Description of clone genealogies features from [1] used to build the models 137
5.8 Evaluation metrics for the machine learning algorithms 138

6.1 The calculated features for the clone group genealogies 153
6.2 10-fold cross-validation evaluation metrics results of three classification algorithms and two frameworks in terms of precision, recall, accuracy, AUC, and F1-score. 159
6.3 Top eight importance scores for XGBoost Model for the prediction of the degree of inconsistency, sorted by importance score descendingly. 161
6.4 10-fold cross-validation results of four regression algorithms and two frameworks, evaluation metrics R-squared, RMSE, MAE. 167
6.5 Top eight importance scores for LighGBM Model for the lifetime of bugs, sorted by importance score descendingly. 169
6.6 10-fold cross-validation results of three classification algorithms and two frameworks, evaluation metrics precision, recall, accuracy, AUC, and F1-score. 176

6.7 Results of the mixed-effect model for prediction of the code clone propagation, Sorted by χ^2 descendingly 179
6.8 A summary of previous studies which predict consistency maintenance. 182

1 Chapter 3: Details of the selected JAVA projects in our study 210
2 chapter 3: Details of the selected C projects in our thesis 211
3 Chapter 3: Results of the mixed-effect model for Model_{all}, Sorted by χ^2 descendingly 212
4 Chapter 3: Results of the mixed-effect model for Model_{mature}, Sorted by χ^2 descendingly 213
5 Chapter 3: Results of the mixed-effect model for Model_{early}, Sorted by χ^2 descendingly 214
6 Chapter 4: Description of the questions in our survey. 215
7 Chapter 6: Details of the selected JAVA projects in our study. 216
8 Chapter 6: The importance scores for XGBoost Model for the prediction of degree of inconsistency, sorted by importance score descendingly. 217
9 Chapter 6: The importance scores for LighGBM Model for the lifetime of bugs, sorted by importance score descendingly. 218

List of Figures

1.1 Types of code clones as compared with original source code. 2

1.2 An example of Type 4 code clone [2]. 3

1.3 Overview of this thesis. 9

3.1 An example of clone pair. The code inside the red boxes are clone pairs from checkstyle project. 32

3.2 Overview of our code clone ranking approach 41

3.3 An example of the code clone in the source file. The start and end lines of the code clone can be affected by changes inside and outside of the code clones. 44

4.1 Overview of our approach for effort estimation for propagating code clone changes. 80

4.2 The distribution of the participant's education background (Q2 in the survey) 83

4.3 The distribution of the participant's software development experience (Q3 in the survey) 83

4.4 The evaluation result of (Q7) In your opinion, how are the complexity/churn/size metrics better than LOC to determine effort estimation? 97

4.5 The evaluation result of (Q5): Do you perceive the type of code changes differently, for example, corrective (bug-fixes), perfective (refactoring changes), new features etc? 106
4.6 The evaluation result of (Q6): What do you think of the usefulness of providing information regarding the likelihood that changes made to cloned code may cause bugs before issue reports are submitted? . . . 114

6.1 Overview of our approach 147
6.2 Annotated screenshot of a pull request in GitHub 150
6.3 Distribution of the number of inconsistencies in a clone group in our dataset. 157

Chapter 1

Introduction

1.1 Introduction

In modern software development, the software environment and requirements are constantly changing, which leads to hardware and software upgrades to keep pace with the changes. Similarly, the benefits of software reuse urge software developers to use the already developed code rather than investing time and effort in rewriting the code. During the development process, a developer often copies code snippets from one part of the software and pastes them in other parts to save time and to avoid implementing the code from scratch. The duplication of code snippets can result in two or more copies of the code which are known as code clones. Over time, the number of code copies may increase in the collaborative development environment, which makes it difficult to keep track of all the copies. As a result, the code copies may become inconsistent and prone to bugs [1]. There are four different types of code clones. The type of code clone may change over time from syntactically identical (Type I or Type II) to semantically identical (Type III or Type IV) [3]. Moreover, it is important for software maintainers to get feedback from the software development

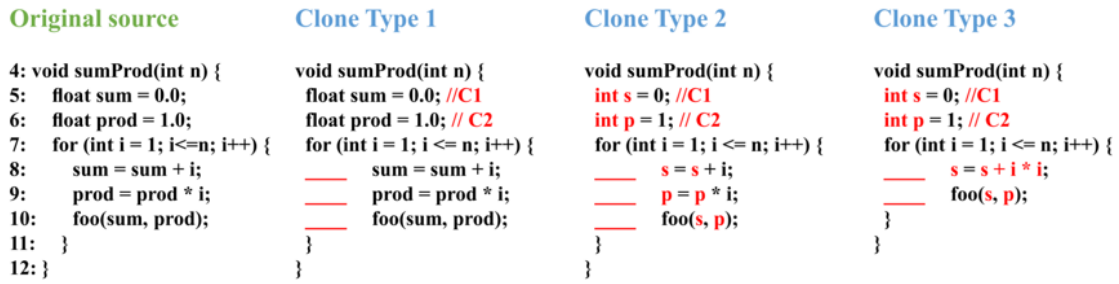


Figure 1.1: Types of code clones as compared with original source code.

community and efficiently use their resources in maintaining the software project. Below, we provide a brief background on code clones, clone genealogy (clone pair evolution over time), and clone groups (clone copies in software).

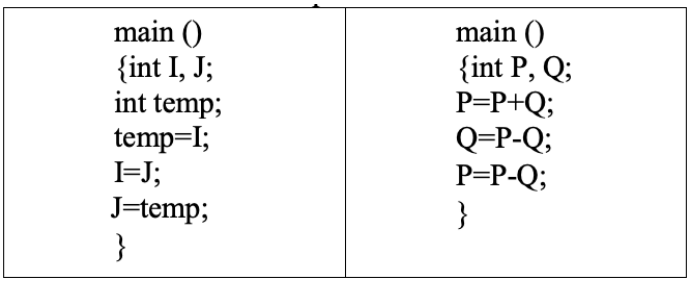
1.1.1 Code Clones

A code fragment is a continuous section of source code with a well-defined *start* and *end* implementing certain functionality. For example, a method is implemented to add two numbers. A code clone is a code fragment or sub-string of code appearing at least twice in code. Code clones can be either syntactically or semantically similar [4]. The degree of similarity of two similar code fragments is described as follows.

- **Type 1.** (Exact) The token sequences of both code fragments are identical, disregarding white spaces and comments. Figure 1.1 demonstrates an example of Type 1 code clone.

- **Type 2.** (Renamed) A type 1 clone pair but names of identifiers and literals may be different. Figure 1.1 shows an example of Type 2 code clone.

- **Type 3.** (Near-miss) A type 2 clone pair with gaps where some tokens exist in only one of the fragments. Figure 1.1 represents an example of Type 3 code



```
main ()
{int I, J;
int temp;
temp=I;
I=J;
J=temp;
}
```

```
main ()
{int P, Q;
P=P+Q;
Q=P-Q;
P=P-Q;
}
```

Figure 1.2: An example of Type 4 code clone [2].

clone.

- **Type 4.** (Semantic) Both code fragments are syntactically different but semantically similar (i.e., similar in logic/functionality) [5] [6]. Figure 1.2 shows an example of Type 4 code clone.

Type 1 and Type 2 code clones are usually easier to identify than Type 3 code clones. Therefore, it is important to study the current approaches and tools to properly maintain open-source software and improve the current practices using the latest approaches. It is important to note that in this thesis, we will focus only on Type 1, Type 2, and Type 3 code clones.

A code can have more than one copy across the software. The multiple copies of the clone are referred to as **clone groups** or clone classes. Two copies of a clone in a clone group are called a **clone pair**. The clone pair may exist in the same code file or separate files.

Clones undergo multiple changes and evolves. If the copies of a clone are changed together, and the changes are similar, then the clone pair is considered in a **consistent** state. However, if only one copy is changed or multiple copies are changed with different changes, then the clone pair is in an **inconsistent** state. An inconsistent

clone pair can be later re-synchronized to make the clone pair consistent. A consistent clone pair can also diverge to an inconsistent state. The set of states and the changes between states of a clone pair for multiple versions of the software project are known as a **clone genealogy**.

1.1.2 Software Repositories

A software repository is a central storage location for a source code used by developers to maintain the source code of a software system. The most popular software repository used by developers is GitHub. GitHub provides developers with the ability of a version control system and collaboration among multiple developers. Following are some basic concepts in software repositories.

Commit

Commit is a snapshot of a repository that a developer can save at any point in time to make sure that the changes that have been made locally are available in the software repository as well. A GitHub commits show the files changed between two commits and the actual source code changes performed. A commit has an associated unique identifier that is able to distinguish between different commits.

Issue

An issue is generated in a software repository when a part of the source code is not performing an accurate action. The issue includes the relevant details about the problems initiated by a software component that can help the developer fix the problems with the software component. Once a fix is performed, the developer submits a

commit confirming the fix for the issue.

Pull Request

A pull request is a collection of one or more commits that are submitted by the developer when a software component is ready to be integrated into the production software. A reviewer then reviews the code changes in the pull request and then either merges the code with the production software or rejects the code changes and sends it back to developers with the feedback to improve the code changes.

1.2 Research Problems

Developers make frequent changes in software systems. Multiple developers often need to collaborate on the same module of a software system. Developers may use the existing code snippets to speed up the development process by copying and pasting code activities, in turn creating code clones. It is time-consuming for developers to run clone detection tools and then review the reports of the clone detection tools. Developers need to prioritize the code clones that should be maintained or fixed first because of the limited resources. The code clones that are consistent and maintained in time can reduce the maintenance cost as well as introduce fewer bugs related to code clones in the software system. Therefore, the need for intelligent approaches to maintain code clones increases as more and more code clones are introduced into the software system.

We envision the following research problems related to code clone software maintenance and evolution.

Problem I: Developers are unaware of the riskiness of the bugs in code

clones. Software maintainers need to identify harmful clones before merging code changes to the code base. To speed up coding activities while maintaining software quality, developers often duplicate code to create code clones. Prior work proposes approaches to analyze code clones in an evolving software system for the early detection of harmful clones that can induce bugs in the future [7]. However, existing approaches focus on the available data to gain insights (e.g., code clones increase over time) into the problems related to software maintenance. Developers may have to run clone detection tools and explore the results to identify clone pairs which is a tedious and time-consuming task. Instead of running a clone detection tool at every commit, developers may prefer to keep track of code clones that are more bug-prone.

Problem II: There is limited support for developers to estimate the effort for propagating code clone changes to clone siblings. In the software development process, developers often create code clones by copying and pasting code snippets. As the code size of a software project continues to increase over time, it is challenging for developers to be aware of all the clones and track their changes. Existing studies [1] have reported that inconsistent code changes to the clone siblings can be risky as they may introduce buggy code and increase maintained costs. However, not all inconsistent changes should be propagated as the code changes could be potentially buggy. The code changes should be tested for bugginess before propagation. The overall effort to propagate the changes to the clone siblings is also crucial for developers. To aid developers to decide whether the propagation of inconsistent changes to the clone siblings is beneficial, they must be provided with all the relevant metrics before they decide to propagate the changes to all siblings.

Problem III: Developers have limited knowledge about the bugs in the

late propagation pattern. Late propagation is a specific pattern of clone evolution. In late propagation only, one of the clone in the clone pair is modified, causing the clone pair to become inconsistent. The other clone in the clone pair is re-synchronized in a later revision. Existing work has established late propagation as a clone evolution pattern, and suggests that the pattern is related to a high number of bugs and that some specific cases of late propagation could be more harmful than others. The information about the different types of late propagation is not available to the developers as well as their bug-proneness. With such information about harmfulness of late propagations, developers can make changes in the code clones with late propagation.

Problem IV: It is difficult for developers to identify the risky code clones at the pull request level.

Developers perform maintenance of the software project by making changes to the source code of the projects and submitting pull requests for review. Existing approaches focus on the commit level, which can have more transient changes. The code changes at the pull request level are more mature and ready to be merged into the main source code. If the changes are not propagated to all the siblings, the clone group may become inconsistent. The inconsistent clone groups are highly likely to introduce bugs in the system. No information regarding the bug-proneness of the code changes that are present in the pull request is available. It becomes imperative for developers to make in-time changes to such clone siblings before increasing the maintenance cost of the clone groups.

1.3 Thesis Statement

Code clones are introduced by developers in the software repositories to improve the development time and refrain from writing the code from scratch. In large collaborative software repositories, it becomes difficult to manage different types of code clones as multiple developers are making changes in the software repositories. Even for a developer working alone on a project, it can become difficult to track all the code clones as the size of code increases. Earlier studies focus more on clone detection in the software repositories and less focused on identifying the bug-prone, faulty and problematic code clones. This thesis aims to provide developers with intelligent approaches using machine learning that can help them maintain code clones better and reduce maintenance costs. In particular, we conduct the following studies (1) ranking code clones based on the bug-proneness, (2) providing effort estimation for propagating code clone changes, (3) identifying harmful code clones associated with late propagation, and (4) predicting the clone maintenance at the pull request level. Our studies can help the developers in (1) properly maintaining the code clones across large software repositories, (2) providing in-time identification for the bugs associated with the code clones, and (3) improving the development time related to code clones.

1.4 Thesis Objectives

In this thesis, we focus on addressing the aforementioned problems faced by software maintainers. Figure 1.3 shows high level objectives and outline of our thesis and and are described in detail as follows.

- **Ranking code clones based on the bug-proneness of the clones.** The number of code clones at any particular commit can be enormous. Developers

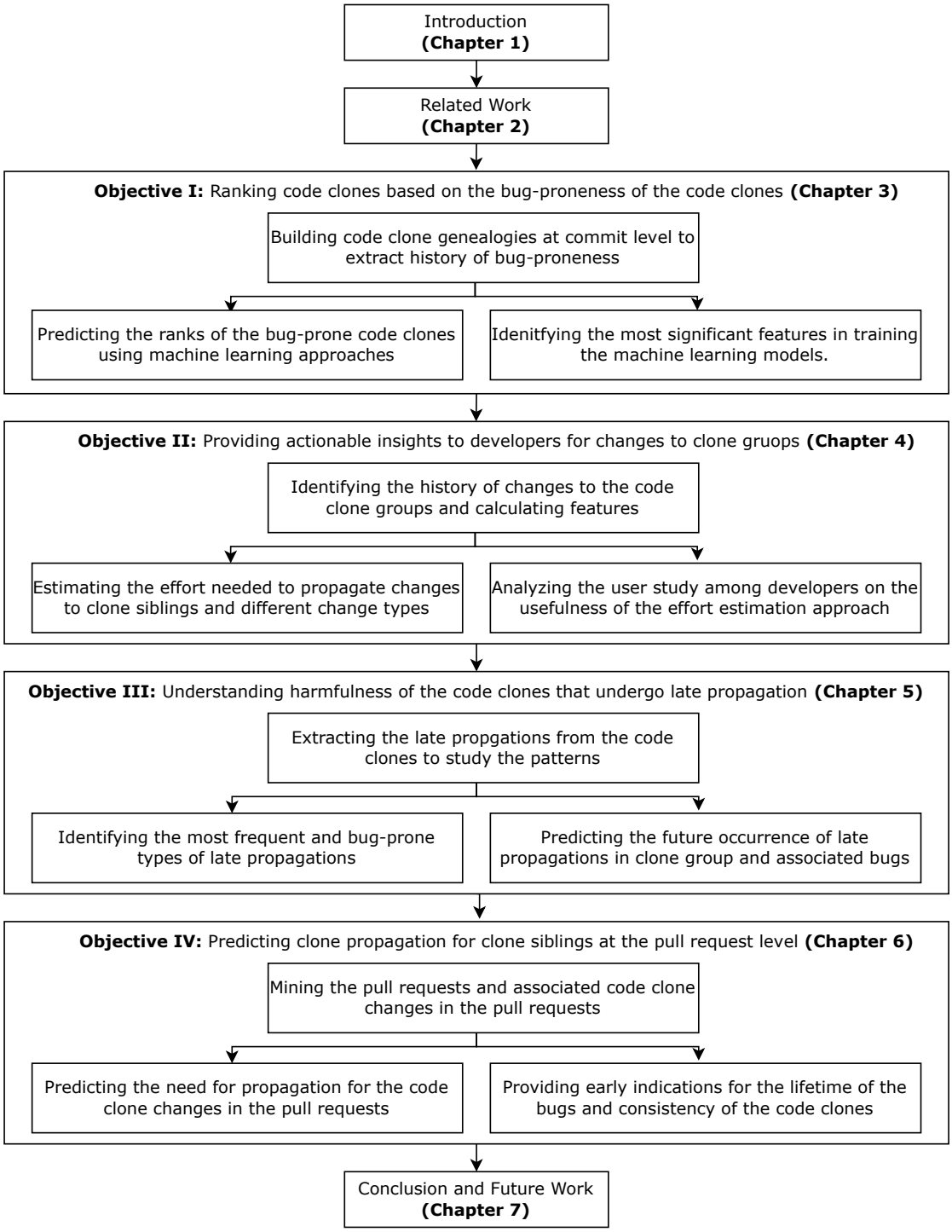


Figure 1.3: Overview of this thesis.

may be unable to (a) identify all the code clones and (b) prioritize the clones that are more risky. We rank the code clones based on the probability of inducing bugs using multiple machine learning approaches to code changes (i.e., learning-to-rank, classification, regression). The clone ranking helps developers address the most harmful clones timely to avoid any possible bugs in the future.

- **Providing actionable insights to developers for changes to clone group.** The developer makes frequent commits while making changes to the code. While performing the code changes, developers are often unaware of the code siblings present in the system. We propose a clone propagation effort estimation approach that can estimate the effort of code clone changes, identify the type of code clone changes, and predict the bug-proneness of the code changes in order for the developers to propagate the changes to all the clone siblings. The proposed approach can assist the developers in making an informed decision while propagating changes to the clone siblings.

- **Understanding harmfulness of the code clones that undergo late propagation.** Different patterns exist while developers make changes to the code clones. Late propagation (a clone group that remains in an inconsistent state before undergoing a consistent change) is one of the patterns, which has different types as well. Understanding the harmfulness of the clone's late propagation pattern can reduce the number of bugs in the system. Existing study [1] focuses on only four projects and provide eight different types of code clones. We replicate the study with a larger dataset of 10 open-source software projects and identify the frequency of eight different types of late propagation as well as the types which are highly bug-prone. In addition, we also use machine learning

algorithms to identify whether a code clone would experience a late propagation pattern. Finally, we provide information about the bug-proneness of the late propagation pattern using the clone evolution history.

- **Predicting clone propagation for clone siblings at the pull request level.** Developers submit pull requests after making changes in the code base for the code reviewers to merge into the main code. It is the responsibility of the developer to make sure that the changes are of quality before the code reviewer merges them, to limit the probability of bugs and reduce maintenance costs. We use machine learning approaches to identify the long-standing bugs in the codebase within code clones. We also predict the consistency maintenance issues of the clone groups. Finally, we provide developers with a prediction for the changes in the clone groups regarding propagation. The information can significantly help developers in identifying which clone groups need propagation before the code changes are merged into the main source code.

1.5 Thesis Outline

The remainder of this thesis is organized as follows:

Chapter 2, Related Work: We present the prior studies related to the analysis of the code clones and differentiate them from our work.

Chapter 3, Ranking Code Clones to Support Maintenance Activities: We provide a ranking approach that can provide a ranked list of bug-prone code clones for the developers.

Chapter 4, Predicting Effort Estimation of Propagation in Code Clones: We present an effort estimation approach that can assist developers in making the

clone siblings consistent by providing information related to effort, type, and bug-proneness when a change is made in a clone group.

Chapter 5, An Empirical Study of Late Propagation Pattern on the Harmfulness During Clone Evolution? We describe the steps to identify different types of late propagation patterns and identifying the most bug-prone types. In addition, we present the results of the machine learning approaches to predict late propagation in the clone group.

Chapter 6, Predicting the Change Propagation of Code Clones at the Pull Request Level: We outline a prediction approach for developers at a pull request level which can predict whether a change should be propagated to the clone group to reduce maintenance costs.

Chapter 7, Contributions and Future Work: We summarize the contributions of our thesis and illustrate the future directions of our research work.

Chapter 2

Literature Review

In this chapter, we provide an overview of the prior approaches related to the work conducted in this thesis. In particular, we focus on fours aspects related to code clones and machine learning approaches: (1) code clone detection, (2) analysis of code clones, (3) bug-proneness in code clones, and (4) machine learning approaches for code changes.

2.1 Code Clone Detection

In this section, we focus on the code clone detection approaches available in the literature. There are many code clone detection approaches presented in the literature, our goal is to identify the best clone detection approaches and use them in our thesis.

Prior studies propose state-of-the-art clone detection tools that are able to identify code clones from large codebases. Göde *et al.* [8] propose an incremental clone detection tool (iClones) which detects clones based on the analysis of the previous revisions. iClones creates a mapping between the multiple revisions of code clones.

iClones effectively provides the add/delete information of the code clones. Incremental detection of clones is helpful in evolutionary clone analysis and is fast as compared to other similar clone detection tools. Kamiya *et al.* [9] propose transformation rules and a token-based comparison clone detection technique that is optimized to achieve better performance. The technique converts the source code into tokens and, based on the pre-defined rules, performs the comparison. The clone detection technique is evaluated on four large-scale projects, and the results show that the tool is able to identify clones. Cordy *et al.* [10] propose a clone detection technique that uses a language-sensitive parsing and language-independent similarity analysis to identify near-miss clones. The approach is available as a simple command-line tool, and code directories are used as input while the output is available in a HTML and XML format. The approach is able to achieve high precision and high recall and is scalable to extensive systems.

Schwarz *et al.* [11] propose a set of lightweight techniques that can detect the code clone from the large codebase as well as across projects. The approach used a bad hashing concept to index the code clones. Squeaksource, a dataset of multiple projects with the project history, is used to evaluate the proposed lightweight techniques. The analysis shows that 22% of all type-3 clones could be missed if the analysis is performed on only the latest versions. Saha *et al.* [12] focus on the extraction and classification of near-miss code clone genealogies. Clone genealogy extraction starts by accepting multiple versions of a program, maps clone classes between the consecutive versions, and extracts how the cloned fragments are changed throughout the period. Finally, the approach identifies change patterns (i.e., added, deleted, modified) using a three-pass algorithm. Yang *et al.* [13] use machine learning models

to classify the true clones based on the code clones detected by the clone detection tools. A web-based tool (FICA) is provided as a proof of concept to identify true clones. A user study is performed on interviewing 32 participants to evaluate the usefulness of the tool. The authors report 70% accuracy for the classification tool. Roy *et al.* [14] focus on identifying near-miss clones (i.e., where small to large changes have been made to the copied fragments of code). A hybrid approach is presented to detect clones, followed by the meta model of the clone types. An empirical study of cloning in more than 20 open-source systems is performed for current clone detection techniques and tools. Svajlenko *et al.* [15] evaluates the performance of the 11 code clone detection techniques. The study compared the 11 code clone detection tools on the recall of the results presented by the tools. Every clone detection tools is employed to detect code clones from the BigCloneBench dataset of the code clones. The results of the evaluation shows that iClones and NICAD achieve the highest recall among the studies code clone detection tools.

There are number of clone detection approaches available in the literature and there are studies to evaluate different clone detection tools as well. The evaluation studies suggests that for the code clone detection of the code bases where multiple revisions are available, iClones and NICAD perform well.

Summary

The summary of the clone detection tool shows that iClones and NICAD perform better in identifying code clones when there are multiple revisions (i.e., commits, and pull requests) of the projects. Therefore, we use iClones and NICAD to identify the clones from our studied projects in this thesis.

2.2 Bug-Proneness of Code Clones

In this section, we present prior studies related to the bugs in the code clones. In particular, we discuss bug introduction and propagation and code change bug analysis.

2.2.1 Bug Introduction and Propagation

Inconsistent changes of code clones can lead to the introduction of bugs in the project. It is vital to effectively make changes to the copies of clones to avoid the risk of bugs. The following studies focus on the impact of bug propagation and the bug-proneness of different types of clones.

Mondal *et al.* [7] present an empirical study to understand the intensity of bug-propagation through code cloning. The empirical analysis shows that up to 33% of the code clones can be related to bug prediction, provided that code clones experience bug fix changes. Near-miss clones (i.e., Type 2 and Type 3 clones) have a higher chance of being involved in the bug-propagation than the identical clones (i.e., Type 1). Xie *et al.* [16] present a study of bug-proneness of Type-3 code clones in the evolving software. The study analyzes three long-lived software systems APACHE-ANT, ARGOUML, and JBOSS, written in JAVA. The clone genealogies are build using the NICAD clone detection tool to examine two evolutionary processes: 1) how clone types are mutated in a system; and 2) how code clones are migrated in a project. Li *et al.* [17] propose a keyword based approach to identify the buggy code clones. The bug reports are evaluated and are linked with the code clones using the keyword approach to identify the buggy code clones. The bugs related to code clones comprise of 4% of overall bugs in the studied systems. The approach is available as a tool that

can be used to identify buggy code clones. Saha *et al.* [18] perform an exploratory study about the evolution of Type-1, Type-2, and Type-3 clones in six open source projects written in two different languages. Results show that a considerable number of type-1 and type-2 clones change to type-3 clones during evolution. Although the life span of type-3 clones is similar to type-1 and type-2 clones, it is essential to manage the type-3 clones to limit their negative impact. Barbour *et al.* [3] examine the characteristics of late propagation of code clones in two software systems. Late propagation introduces the concept of making changes to one clone, and the other is changed afterward in the next revisions; this evolutionary pattern can introduce bugs. The study defined eight different types of late propagation and compared them with other forms of clone evolution.

2.2.2 Code Change Bug Analysis

The commit records code changes to a software system. Existing studies focus on the changes in commits to identify bugs introduced by code changes and use commits to predict the bugs that can be introduced in the future. Shivaji *et al.* [19] review the existing bug prediction approaches for the code changes and reduced the number of features to 25% as compared to prior studies. The substantial reduction of the features produced better results by improving the F-measure by 21%. The approach achieves more than 0.8 accuracy for 10 out of 12 projects in the study when SVM is used to predict bug-proneness. Di Nucci *et al.* [20] propose an approach for predicting the bugs using the scattering changes of the developers. The study suggests that more focused developers who are working on closely related modules of the project are less likely to introduce bugs as compared to the developers who are working on different

modules regularly. The approach proposed by Di Nucci *et al.* shows better results when compared with four existing prediction models. Moreover, the hybrid approach which combines the proposed and existing approach is even better. [21] propose a harmful code change detection approach using CodeBert. CodeBert is a pre-trained bimodal for a programming language (PL) and natural language (NL) [22] which can learn a general-purpose representation of PL-NL and can be used in natural language code search and documentation. The approach proposed by [21] is able to detect if the change introduced in a software system is harmful to security or DDOS attack in future. The results of the approach show an accuracy of 0.62 when predicting harmful code changes. [21] recently use CodeBert to perform harmful code detection.

Summary

To the best of our knowledge, prior studies are unable to provide the approaches related to bug-proneness of the code clones that can rank the buggy code clones, identify the bug-prone code changes, and predict the lifetime of bugs. In this thesis, we aim to provide that are able to perform ranking of the risky code clones, early identification of the code changes that are likely to introduce bugs and predict the lifetime of the bugs associated to the code clones.

2.3 Analysis on Code Clones

In this section, we present the existing studies that perform analysis on the code clones. In particular, we discuss studies related to the analysis of the clone genealogies and code clone consistency.

2.3.1 Analysis of Clone Genealogies

The clone genealogies need to be built first using the history of code clones before the analysis of code clones. Barbour *et al.* [1] investigate six different evolutionary patterns using the clone genealogies extracted from four open-source Java systems. The analysis uses the clone genealogy information to identify the bug-prone nature of clone pairs based on the evolutionary patterns of the clone pairs. The results show that including the clone genealogy information can increase the identification power of bug prediction models. Zhang *et al.* [23] use clone genealogy information to predict the consistency-maintenance of code clones. The study identifies the code clones with consistency issues earlier in the lifetime in order to improve the maintenance of such code clones. The approach can predict the consistency-maintenance of the code clones. Thongtanunam *et al.* [24] investigate the life of code clones using the clone genealogy information from the code clones. The approach used multiple features to train a Random Forest classifier to determine whether a code clone would be short-lived. Garg *et al.* [25] propose a clone ranking approach for better clone management. The paper prioritizes the results of clone detection tools by finding out the maintenance overhead using the size of code clone, MCC complexity and frequency of clone results. Different weights are provided to the three selected features. The CCFinder tool [9] is used to detect clone fragments and a score is given to each clone fragments based on the pre-defined weights. The approach is able to achieve high AUC and emphasize that the code churn, complexity, and the size of the newly-introduced code are highly influential in determining the life of the code clones.

To summarize, code clone genealogy information is used in different contexts (e.g.,

consistency-maintenance, evolutionary patterns, the life of a clone pair) in the software project to improve multiple aspects of clone maintenance.

2.3.2 Code Clone Consistency.

The state of the code clones can change and if the siblings of the clone groups are not changed together, the bugs can be introduced. A number of studies in the literature provide approaches to predict the consistency of the code clones. Bettenburg *et al.* [26] perform an empirical study on the inconsistent changes to the code clones at the release level. The study is conducted on three open source java projects that include approximately 4,500 code clone instances. The results of the study show that 4% of the defects related to the code clones are caused by inconsistent changes in the code clones. The study also shows that the weekly defects are four times higher than the release-level bugs. Wang *et al.* [27] propose a novel machine learning approach that is able to predict the consistent maintenance of the code clones at the point of copy and paste the code snippets. The paper uses the Bayesian Networks approach to predict the consistency of the code clones relating to the context of copy-paste operation. A total of approximately 6,000 clone instances from four projects (two Java and two C projects) are part of the experiment. The approach is able to recommend developers to perform more than 50% of the cloning operations and avoid 37% to 72% of the code clones that may require consistency maintenance. The approach only provides a consistency maintenance solution at the creation of the code clones in the software project. Zhang *et al.* [28] present the results of five different machine learning algorithms at the clone-creating and clone-changing operation. The experiment is performed on eight open-source projects comprising around 20,000

code clone instances for both creation and changing operations. The five machine learning algorithms used in the study include Bayesian Network, Naive Bayes, Support Vector Machine, K-nearest Neighbors, and Decision Tree. The clone consistency prediction for the clone changing operation achieves the F-measure ranges from 66% to 76% while for the clone creation operation, F-measure ranges from 68% to 77%. The approach provides the prediction for both the creation and changing operation of the code clones. Nguyen *et al.* [29] introduces a novel clone management tool named JSync that is able to make the developers aware of the clone creation and clone update operations. The approach uses Abstract Syntax Trees (ASTs) to represent the code clones and the source code and measures the code similarity based on the structural characteristic vectors. The empirical study on five real-world systems includes approximately 5,000 code clone instances from 900 revisions and achieves a precision value of 83% for change consistency of the code clones. The results indicate that the proposed tool is accurate to identify the clone creation and update operation and also provides accurate information on the defects that originate from inconsistent code clones.

The aforementioned approaches are focused on the consistency maintenance of the code clones. For the prediction of the clone consistency maintenance, the amount of data used is low (the highest number of code clones used is 20,000) and the number of the project studied is fewer (the highest number of projects is eight).

Summary

The code clone analysis is focused on the clone genealogies and the consistency maintenance of the code clones with less amount of data. In this thesis, we aim to provide intelligent approaches using extended dataset and use code clone genealogies to predict the consistency of the code clones and aid the bug prediction in code clones.

2.4 Machine Learning Approaches for Code Change

In this section, we focus on the machine learning approaches available in the literature related to code changes in the software projects. In particular, we focus on the machine learning approaches related to (1) effort estimation for code changes, (2) commit message analysis, and (3) bug resolution duration.

2.4.1 Effort Estimation for Code Changes

Several studies propose approaches to estimate the effort from the perspective of code change and identify features and metrics to develop an effort estimation model. Fedotova *et al.* [30] perform a study in software development organization for the most commonly used effort estimation techniques. The organization replaces the manual effort estimation with the step-wise multiple linear regression techniques. The results of the study by Fedotova *et al.* suggest that the step-wise multiple linear regression techniques provide better effort estimation than the manual effort estimation by an expert developer with knowledge of the code base. Qi *et al.* [31] provides an approach

to collect the data to solve the issue of lack of data for building effort estimation models. The approach calculates the diverse features including input, output, functional points, files, LOC, and team's experience. The results of the approach, named ABCART shows that the personnel factor is most helpful while building effort estimation models. Minhaz *et al.* [32] for conflict aware optimal scheduling of the code clone refactoring. The approach estimate the effort to refactor the code clones for object oriented and procedural source code. The authors propose a constraint aware approach followed by a qualitative survey among the developer that shows that the model is complete and useful. Shihab *et al.* [33] perform an empirical study on four open-source projects to show that lines of code alone are not the best measure of effort. Shihab *et al.* introduce more than 20 metrics to estimate efforts that belong to three main categories: size, churn, and complexity. Shihab *et al.* show that the LOC underestimates the effort by 66% when compared with the proposed effort estimation approach. To the best of our knowledge, there are no existing studies that perform effort estimation to propagate code clone changes to clone siblings for code clones specifically. Shihab *et al.* [33] propose multiple metrics across the software repository to estimate effort. In our work, we identify the metrics related to the code changes in the code clones to estimate the effort for clone propagation.

2.4.2 Commit Message Analysis

The commit messages describe a developer's intent for making changes in the code base. The stakeholders use the commit messages to determine the next actions (such as closing an issue report if a bug is fixed). Sabetta *et al.* [34] propose an approach that uses machine learning to identify the commits that are relevant to

security. The approach treats commit messages and the changed source code as a document and uses natural language processing approaches to classify the changes related to security. The approach is able to get 80% precision and 43% recall when determining the security-related changes. Levin *et al.* [35] propose an automatic commit classification method that can classify commits into different maintenance activities. Levin *et al.* experiment on 11 projects and 1,151 commits are manually labelled and 85% of the data is used for training the machine learning models. The machine learning model achieves 76% accuracy for classifying the commit messages into maintenance activities. Mauczka *et al.* [36] provide the dataset of developer-labeled commit messages. Seven developers label their changes with meta-information that would help in determining the intent of the commits more clearly. A total of 967 commits are classified from six projects to label; perfective, adaptive, corrective, and non-functional tasks. Dos Santos *et al.* [37] propose a natural language-based commit classification approach that provides the context of the change from the commit message. A dataset of 5,631 manually labelled commits is gathered from previous studies and labels include corrective, features, perfective, non-functional, and others. The approach uses FastText [38], an NLP approach to train the models and achieve an F1 score of 91%.

To the best of our knowledge, no existing approach has a better accuracy score than Dos Santos. Also, the corrective, features, and perfective labels accumulates to approximately 98% of the labels for the commits in our dataset.

2.4.3 Bug Resolution Duration

The duration of the bugs is studied in several studies in the literature that try to predict how long it would take for a bug to be resolved. Gomez *et al.* [39] conducts an analytical and comparative study on six open-source projects. The approach shows that the number of long-lived bugs is significantly higher in popular open-source projects. The paper uses five popular machine learning algorithms and techniques that include K-nearest neighbours, Naive Bayes, Neural Networks, Random Forest, and Support Vector Machines. The machine learning algorithms are trained on eight features identified from the issue reports and only looked at the number of days until 365 days. The results indicate that the Neural Networks classifier is able to provide an accuracy of 70%. The approach provides a binary outcome for a bug whether it would be long-lived or not. Habayeb *et al.* [40] propose an approach to predict the time to fix the bugs using the hidden Markov model. The hidden Markov model is coupled with the temporal sequences to aid the prediction of when the bug report would be closed. The experiment is conducted on the eight years of bug reports from the eclipse project. The results indicated that the Hidden Markov Model is able to achieve higher accuracy than the frequency-based classification approaches. The approach achieves an accuracy of 71% while trained on seven features from the issue reports. Giger *et al.* [41] propose a prediction-based approach that is able to determine if the bug is resolved slow or fast. The approach uses the decision trees and by performing the 10-fold cross-validation the paper is able to build a recommender system. The experiments are conducted on six open-source projects and the results indicate that the approach's results are better than the random classification. The approach uses 17 features related to the issue reports to train the decision tree model.

The approach is able to achieve a precision of 0.65 and a recall of 0.69. Weiss *et al.* [42] provide an approach that is able to provide a prediction of the time needed for the developers to resolve the bugs once it is created. The approach is based on the textual analysis of the bug reports and incorporates K-nearest neighbour to identify similar bug reports in the system. The approach predicts the number of hours the developer would take to close or resolve the specific bug report. The approach is evaluated on the data from JBoss project and the results show that on average in most cases the prediction is off by two hours and approximately the accuracy is around 50% of the effort needed by developers.

The research studies available focus mostly on the binary level classification (i.e., short or long-lived bugs). One of the approach focuses on the prediction of the number of hours that a developer would take to resolve a bug. However, there is a lack of prediction for how long a bug would remain in the system.

Summary

There are different studies available in the literature related to effort-estimation, commit-message analysis, and bug resolution duration. None of the studies are focused on the code clones specifically. In this thesis, we propose approaches to estimated effort for propagating code clone changes, providing classification of the commit messages, and prediction the lifetime of the bugs.

2.5 Summary

A number of research studies are focused on detecting code clones and performing analysis on the detected code clones. However, still developers find it challenging to

manage to code clones in large software system. In this thesis, we provide developers with the intelligent code clone management approaches that can help in organization of the code clones better, saving them time, and reducing the number of bugs related to code clones.

Chapter 3

Ranking Code Clones to Support Maintenance Activities

In this chapter, we present the study conducted to rank the code clones based on the bug-proneness to improve the maintenance of the code clones. Section 3.1 describes the problem and motivation of our study. Section 3.3 presents the study setup of our work. Section 3.4 discusses the results for the three research questions in our study. Section 3.5 emphasize on the results and how they can be used by developers. Section 3.6 provides the threats to validity for our study. Finally, Section 3.7 summarizes our study and provide directions for the future work.

3.1 Problem and Motivation

Over the lifetime of a project, the number of code clones may increase. For example, at the start of project for nd4j[1], there are 109 clones. As the project evolved, the number of code clones increased to 9,550. With such a large amount of code clones in a system, it becomes crucial to maintain them, since (1) multiple,

[1] https://github.com/deeplearning4j/nd4j

possibly unnecessary duplication of code can increase maintenance cost [43], and (2) inconsistent changes to the cloned code can introduce bugs leading to incorrect product behaviours [4].

Existing approaches provide clone detection tools that can identify a large number of code clones. However, it can be tedious and time-consuming for the developers to examine all the code pairs in a project manually to identify the risky clone pairs. For example, there can be multiple code clones in any commit, and it is difficult for a developer to recognize which code clones should be tracked first to prevent future bugs.

Recent studies have used code clone evolution history to analyse the risk of different clone evolution patterns [1], predict the consistency-maintenance of code clones [23], and identify short-lived code clones [24]. Although these previous work can help improve the maintenance of the code clones, they do not support developers in assessing the risk of all the clones in a software system. To the best of our knowledge, no prior studies aimed at ranking the code clones in a software system, at the commit level, based on their bug-proneness.

In this study, we examine the potential of various machine learning techniques at providing an accurate ranking of code clones based on their bug-proneness. We are interested to rank code clones at the commit level. The commit is the smallest unit of code change in a repository, and code clones can be refactored as soon as they are introduced or modified if they are tracked at the commit level [44]. We use 52 projects (i.e., 34 Java and 18 C) with an average of 893k SLOC per project to train the machine learning models. Using the history of changes of a software system, we detect code clones and build clone genealogies. Then, we leverage the

SZZ algorithm [45] to identify buggy commits in the clone genealogies. Next, we calculate 28 clone-related metrics on the detected clones and train learning-to-rank (LtR), classification, and regression machine learning models, to rank the code clones for bug-proneness. We also conduct an analysis of the most important features of the models, to understand the main factors affecting the bug-proneness of a code clone. We assess the performance of the models and answer the following research questions:

RQ1: Can we use learning-to-rank (LtR) algorithms to effectively rank bug-prone code clones? In this research question, we train multiple learning-to-rank (LtR) algorithms to assess their effectiveness at ranking clones. We experiment with both early projects and mature projects to allow developers to rank code clones during the early stages of the development process of their projects. Our results show that the best performing approach (i.e., LightGBM) is able to achieve a precision of 0.72 (for Java projects). Learning-to-rank algorithms achieve moderate performance due to the unavailability of the labeled ranked data for code clones.

RQ2: How well can classification algorithms rank bug-prone clones? In this research question, we use classification algorithms to rank clones based on the probability of being buggy (i.e., bug-proneness of code clones). To support clone ranking in early development phases as well as mature development phases, we constructed specialized models for the projects that have limited clone history in the early phase as well as projects that have more longer clone history in the mature phase. We performed 10-fold cross-validation of the trained models and found that Random Forest achieves the highest AUC among the studied classification approaches. Our analysis also shows that the performance of the models increases as more information about the history of code clones is available.

RQ3: Can we use regression algorithms to predict the proportion of buggy changes in code clones and effectively rank bug-prone clones? In this research question, we rank code clones based on their predicted proportion of buggy changes, using regression algorithms. Similar to the previous research question, we also train two specialized models for early stage and mature stage respectively. Our 10-fold cross-validation results show that LightGBM and XGBOOST are able to identify the bug-prone code clones with a high R-squared score (0.89 for Java projects). Similar to RQ3.2, the performance of the models increase as more information about the history of code clones is available.

RQ4: Which metrics are significantly correlated to the risk of bugs in code clones? In this research question, we investigate the most influential features in determining the rank of the code clones. We use a mixed-effect model to determine the most significant features when ranking code clones. Results suggest that developers should closely monitor the code clones that have more changes and the code clones that are changed by multiple developers.

3.2 Background

In this section, we describe the motivating example and the machine learning algorithms/frameworks used in this study.

3.2.1 Motivating Example

We use an example to illustrate the need to rank the buggy commits. In one of the commits in project nd4j[2], there are 87 code clones identified from the clone detection

[2] https://github.com/eclipse/deeplearning4j/commit/0ec1c8f

```
43          super(aIndentCheck, "operator new", aAST, aParent);
44      }
45
46      @Override
47      public void checkIndentation()
48      {
49          final DetailAST type = getMainAst().getFirstChild();
50          if (type != null) {
51              checkExpressionSubtree(type, getLevel(), false, false);
52          }
53
54          final DetailAST lparen = getMainAst().findFirstToken(TokenTypes.LPAREN);
55          final DetailAST rparen = getMainAst().findFirstToken(TokenTypes.RPAREN);
56          checkLParen(lparen);
57
58          if ((rparen == null) || (lparen == null)
59              || (rparen.getLineNo() == lparen.getLineNo()))
60          {
61              return;
62          }
63
64          // if this method name is on the same line as a containing
65          // method, don't indent, this allows expressions like:
66          //    method("my str" + method2(
67          //        "my str2"));
68          // as well as
69          //    method("my str" +
70          //        method2(
71          //            "my str2"));
72          //
73
74          checkExpressionSubtree(
75              getMainAst().findFirstToken(TokenTypes.ELIST),
76              new IndentLevel(getLevel(), getBasicOffset()),
77              false, true);
78
79          checkRParen(lparen, rparen);
80      }
81
82      @Override
83      protected IndentLevel getLevelImpl()
84      {
```

NewHandler.java

```
147         return new IndentLevel(indentLevel);
148     }
149
150     @Override
151     public void checkIndentation()
152     {
153         final DetailAST methodName = getMainAst().getFirstChild();
154         checkExpressionSubtree(methodName, getLevel(), false, false);
155
156         final DetailAST lparen = getMainAst();
157         final DetailAST rparen = getMainAst().findFirstToken(TokenTypes.RPAREN);
158         checkLParen(lparen);
159
160         if (rparen.getLineNo() == lparen.getLineNo()) {
161             return;
162         }
163
164         // if this method name is on the same line as a containing
165         // method, don't indent, this allows expressions like:
166         //    method("my str" + method2(
167         //        "my str2"));
168         // as well as
169         //    method("my str" +
170         //        method2(
171         //            "my str2"));
172         //
173
174         checkExpressionSubtree(
175             getMainAst().findFirstToken(TokenTypes.ELIST),
176             new IndentLevel(getLevel(), getBasicOffset()),
177             false, true);
178
179         checkRParen(lparen, rparen);
180     }
181
182     @Override
183     protected boolean shouldIncreaseIndent()
184     {
185         return false;
186     }
187 }
```

MethodCallHandler.java

Figure 3.1: An example of clone pair. The code inside the red boxes are clone pairs from checkstyle project.

tool (one of the clone pairs is shown in Figure 3.1). Suppose that there are 48 buggy code clones. Currently, in practice, it is a manual process to check 87 code clones in order to identify the potential buggy code clones and fix them. Development teams are likely to have limited time and resources, so they would need to prioritize which code clones to examine and test first. However, some commits have a higher number of code clones identified by the clone detection tools (e.g., a commit in nd4j has more than 600 code clones identified). In such scenarios, it becomes imperative for the developers to have a ranked list, so they can focus on the most risky code clones first by making optimal use of their resources.

3.2.2 Learning-to-Rank Algorithms

Learning-to-Rank (LtR) algorithms stem from the application of machine learning models (either supervised or semi-supervised) or reinforcement learning to solve ranking tasks. The significant difference between LtR and classical regression/classification problems is that LtR focuses on a list of items to predict the rank, while regression/classification problems focus on one item at a time. The most common application of LtR algorithm is web search, while it is also used in product recommendation and candidate selection.

Types of LtR Models. In general, there are three types of LtR algorithms: pointwise, pairwise, and listwise. To understand the working of different types, let us suppose to have the following ranking task. Given a list of documents D= d_1, d_2,....d_n, and a query q, we would like to learn a function f such that f(q,D) will predict the relevance of the documents with respect to the query. In our context, the list of clone pairs at a commit is the documents and our query to identify bug-prone clone pairs. The three types of LtR algorithms are different in terms of loss function formulation in the machine learning task.

- **Pointwise.** The documents are scored individually. It is useful in cases where the relevance of the given documents is binary.
- **Pairwise.** The training example is constituted of pairs and are given relevancy scores ranging from lowest to highest. For example, 0 is irrelevant and 5 is the most relevant.
- **Listwise.** The training examples consist of all the documents available to rank. This method is costly; however, it covers the full range of documents that need

to be ranked.

The pairwise and listwise algorithms are reported to outperform the point-wise algorithms [46]. Therefore, in this chapter, we experiment only with pairwise and listwise algorithms using the following two popular frameworks; i.e., XGBOOST and LightGBM. Specifically, we use the pairwise algorithms (RankBoost, and RankNet), the listwise algorithms (LambdaRank, LambdaMart, and Random Forest), and the frameworks XGBOOST, LightGBM to rank code clones.

Evaluation of LtR Models.

Several metrics have been proposed to evaluate the LtR models. They are generally divided into two categories: binary relevance and graded relevance.

(1) Binary Relevance. Binary relevance evaluation is used when the ranking task only considers the relevance or irrelevance of documents.

(a) Mean Average Precision (MAP). MAP is a measure based on the binary label of relevancy where precision needs to be calculated at every ranking position. The average ranking is calculated at every relevant position while every irrelevant position is penalized. Equation 3.1 is used to calculate the MAP.

$$MAP = \frac{\sum_{q=1}^{Q} AP(q)}{Q} \tag{3.1}$$

where

$AP(q)$ = Average precision given at query q

Q = Total number of queries

(2) Graded Relevance. Graded relevance metrics are used to evaluate the LtR

algorithms when the ranking task gives different scores to the documents based on their relevancy to the query.

(a) Normalized Discounted Cumulative Gain (NDCG). Discounted cumulative gain prefers the higher relevant item to be ranked higher. In Equation 3.2, the numerator is an increasing function of relevance while the denominator is a decreasing function of ranking position. Therefore, higher relevance gains more points.

$$DCG@k = \sum_{i=1}^{k} \frac{2^{l_i} - 1}{log_2(i+1)} \tag{3.2}$$

where

DCG = Discounted cumulative gain

l = relevance score

Normalized discounted cumulative gain (NDCG) is calculated using Equation 3.3. If DCG@k is calculated by re-sorting the list by correct relevance labels, then the IDCG@k is the maximum possible value of DCG@k that can be achieved given a ranking list.

$$NDCG@k = \frac{DCG@k}{IDCG@k} \tag{3.3}$$

where

DCG = Discounted cumulative gain

$IDCG$ = Ideal discounted cumulative gain

3.2.3 Classification Approaches

Classification approaches use selected machine learning algorithms to learn how to assign a class label to examples from a problem domain. The class labels can be binary (i.e., only two classes to chose from, e.g., 0 or 1) or multiple (i.e., more than two classes, e.g., Types of grade for course A, B, C, or F). From a modeling perspective, classification requires a training dataset that has many example inputs (i.e., features) and outputs.

Evaluation of Classification Techniques.

The following four metrics are commonly used to evaluate a classification technique.

Precision is the fraction of relevant instances among the retrieved instances [47]. As shown in Equation 6.3, we use precision to identify the total number of correctly identify bug-prone code clones (true positives) over the the number of the wrongly identified bug-prone code clones (false positives).

$$P = \frac{TP}{TP + FP} \tag{3.4}$$

Recall is defined as a probability that a relevant object is returned by a system [47]. As shown in Equation 3.5, we use recall to identify the number of bug-prone clone pairs that are correctly identified (true positives) over the number of the missed bug-prone clone pairs (false negatives).

$$R = \frac{TP}{TP + FN} \tag{3.5}$$

F1-score is the weighted average of the precision and the recall that includes the impact of the false positive as well as false negatives. Hence, we use F1-score to evaluate the overall accuracy of our approach for our manually labeled dataset (Sample_{tuning}), as shown in Equation 3.6.

$$F1 = 2\frac{P * R}{P + R} \tag{3.6}$$

AUC (Area under the ROC Curve). The discriminative power of the model measures the ability of the model to distinguish value 0 and 1 of the dependant variable (i.e., predicting the probability of the code clones being buggy). We use Area Under Curve (AUC) [48] to determine the discriminative power of the model. We use Receiver Operator Curve (ROC) to plot the true positives against the false positive for different thresholds. The value of AUC ranges from 0 to 1, 0 being the worst performance, 0.5 being the random guessing performance, and 1 being the best performance. [48]

3.2.4 Regression Analysis

Regression analysis is a type of statistical method used to determine the strength of a relationship between a dependent variable (usually denoted as Y) and a number of independent variables. In contrast to only two possible outcomes in the classification problem, a regression problem can have multiple outcomes. For example, predicting the price of a house or the interest rate over a period of time. There are multiple types of regression analysis that include linear regression, logistic regression, ridge regression, and lasso regression.

Evaluation of Regression Approaches

Multiple metrics have been introduced in the literature to evaluate the performance of regression models. Two of them are most commonly used by the practitioners: (1) R-Squared (R^2) and (2) Root mean square error (RMSE)

(1) R-Squared (R^2). R-squared is a proportional improvement in the prediction of a regression model as compared to the mean model. The scale of R^2 ranges from zero to one, with zero indicating that the regression model is not able to improve than the mean model while one depicts that the regression model performs perfectly in terms of prediction. Equation 6.1 is used to calculate R^2.

$$R^2 = \frac{\text{Error from regression model}}{\text{Simple average model}} \tag{3.7}$$

(2) Root Mean Square Error (RMSE). The RMSE is the square root of the variance of the residuals that indicates the absolute fit of the model. In simple words, it calculates how close are the predicted values and the actual values. Lower values of RMSE are desirable for regression models. Equation 6.2 is used to calculate RMSE.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(Predicted_i - actual_i)^2}{N}} \tag{3.8}$$

(3) Mean Absolute Error (MAE). Mean absolute error indicates the magnitude of the different the predicted observation and the actual observation. MAE takes the average of the absolute errors from the group of predictions. Equation 6.3 is used to calculate MAE.

$$MAE = \frac{|Y_i - Y_p|}{N} \tag{3.9}$$

where Y_i is the actual value, Y_p is the predicted value and N is the total number

of observations.

3.2.5 Machine Learning Frameworks

This section briefly describes the machine learning frameworks used in this chapter.

XGBOOST. XGBOOST is a scalable machine learning system for tree boosting proposed by Chen and Guestrin [49]. XGBOOST combines the original model (i.e., gradient boosting) with weak base learning models in an iterative manner to generate a robust learning model. The residual in each iteration of the boosting is used to improve the previous predictor (i.e., optimizing the loss function). The algorithm is engineered for time efficiency and memory optimization. One of the most important features of the XGBOOST is the sparse awareness that makes it useful even when some of the data values are missing. XGBOOST offers a block structure that enables the algorithm to make use of parallelization for tree construction. The models in XGBOOST are constructed by computing the gradient descent using an objective function.

XGBOOST has been used in prior studies for cross-device identification [50], intrusion detection [51], and bug localization [52].

LightGBM. LightGBM is a gradient boosting framework that uses tree-based algorithms [53]. The most distinctive feature of LightGBM among other tree-based algorithms is that it grows trees vertically. In other words, LightGBM grows tree leaf-wise while other tree-based algorithms grow trees level-wise. This allows it to reduce more loss than a level-wise algorithm. LightGBM is highly effective for large scale data. LightGBM supports GPU learning, but it is sensitive to the size of

datasets; there is a risk of overfitting if used on a relatively smaller dataset. Similar to XGBOOST, LightGBM computes gradient descent using an objective function.

LightGBM has been used in prior studies for intrusion detection [54] and malware detection [55].

3.2.6 Cross Validation

Cross-validation is a resampling technique used to evaluate machine learning models on limited data samples. Essentially, there are multiple folds (known as k-folds) where the k value is the number of folds. We choose the value of k as 10. The data is randomly divided into ten groups, and first nine folds are used for training and validation, and the last fold is used for testing. This process is repeated ten times using different folds each time for testing. In each fold for training, data is divided into a training set (t1) and validation set (v1). The model is trained on the data from the group's training set (t1) and then validated within the same group's validation set (v1). The performance results of each fold are aggregated.

3.3 Study Setup

This section describes the experiment setup used in this study to examine the effectiveness of machine learning techniques at ranking code clones. Figure 6.1 presents the overview of our approach.

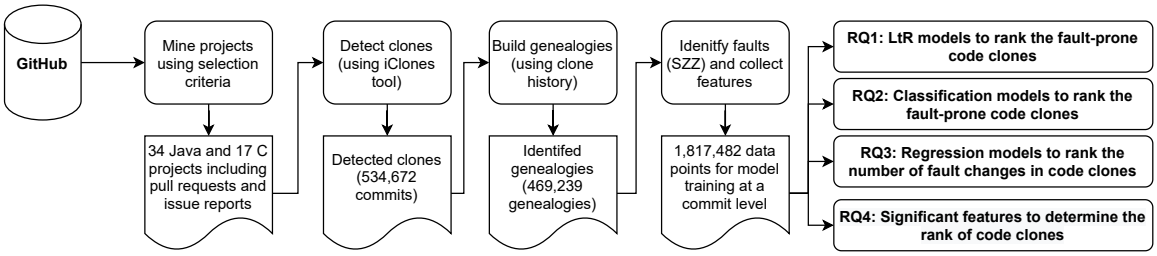


Figure 3.2: Overview of our code clone ranking approach

3.3.1 Project Selection

We use GHTorrent on the Google cloud[3] to extract all projects that have more than 1,000 commits, 1,000 issues, and 1,000 pull requests. We use such a high number of commits, pull requests, and issues to ensure that we have enough history of clone genealogies. We limit our study to Java and C projects. We focus on these two programming languages because code clones studies often use Java and C projects [1] [3]. Our selection criteria provide us with 66 Java and 29 C projects. Then, we discard the projects that are younger than five years (i.e., created after June 2015). A recent study on clone genealogies [1] suggests including projects with more than 100K source lines of code (SLOC). We remove the projects with less than 100K SLOC using the GitHub project SLOC calculator extension[4]. Furthermore, we remove the forked projects and the projects with less than 70% of code files (i.e., Java or C). The percentage of code files is calculated using the language information for each project in GitHub. After applying all the selection criteria, we obtain 34 Java projects and 18 C projects that are used in this study. The following steps are applied to both the Java and C projects. The details of the selected projects are provided in Table 7 and Table 2 available in appendix section of this thesis.

[3] https://ghtorrent.org/gcloud.html
[4] https://github.com/artem-solovev/gloc

3.3.2 Building Clone Genealogies

The selected projects are all Git-based projects. Git provides multiple functions to extract the history of the projects. The history includes the renamed files, changed files, and changes made to each file using the `blame` function. We perform the following steps on each of the projects in our dataset. After downloading the repositories, we extract identifiers, committer emails, commit dates, and messages of each commit using the following command.

git log – pretty=format:"%h,%ae, %ai, %s"

Detecting Code Clones. We use the latest version of the iCLONES and NICAD clone detection tool [8] to identify the clones from the projects. We select iCLONES and NICAD because it is recommended by Svajlenko *et al.* [56] who have evaluated the performance of 11 different clone detection tools. iCLONES uses a hybrid approach that includes the suffix tree and multiple revisions of the program to perform incremental clone detection. We use the settings recommended by Svajlenko *et al.* [56] since such settings are reported to achieve higher precision and recall values. For Nicad, we use the following settings: Min length 6 lines, blind identifier normalization, identifier abstraction, min 70% similarity. We use the git checkout command to extract a snapshot of a project at a specific commit. We sort all commits chronologically and run the clone detection on each commit.

Extracting Clone Genealogies. To identify all the genealogies, we perform the following steps.

The clone detection tools provide us with a list of clones that are present at a specific commit (i.e., a source code). To create a set of genealogies that correspond to the changes of a code clone, we collect the clone changes performed at a commit

level. If a change is made inside a code clone (between the *start* and *end* line of the corresponding clone as shown in Figure 3.3), it can change the size of the code clone (i.e., the number of clones SLOC). If a change is made outside of a code clone (above the *start* line of the corresponding clone as shown in Figure 3.3), then it can affect the location of the code clone. Therefore, we utilize the git diff[5] command to detect all changes to a specific file at a commit level. More specifically, we track the positional changes of a code clone that are caused due to a change in a non-clone part of a file. If a change is made inside a code clone only, then we update the state (i.e., consistent or inconsistent) of a code clone. However, a shift in the location of the code clone (in terms of start and end line) do not correspond to a state change as it means the change is made elsewhere in the code file (outside of the code clone). Additionally, we make sure that the code clone siblings are labelled as consistent only if the change made to the siblings is similar as reported by the clone detectors. If the clone siblings are changed independently but the changes are not identical, they remain in an inconsistent state. We use two code clone detection tools (i.e., NiCAD and iClones) to check the consistency of clone siblings in order to mitigate the possibility of identifying the clone siblings in the wrong states (i.e., consistent or inconsistent). Both the tools are evaluated [56], and perform the best among the compared tools in correctly identifying the states of the code clones (i.e., consistency of clones).

We build a clone genealogy for each clone pair detected by the iCLONES tool. We start by extracting the commit sequence of each project under study. We use the commit sequence to identify the modifications in the clone pairs of each commit. If a commit C2 changes a file that contains code in the clone pair, we use the diff

[5]https://git-scm.com/docs/git-diff

```
249
250         /**
251          * Implements the Contextualizable interface using bean introspection.
252          *
253          * @see Contextualizable
254          */
255         @Override
256         public final void contextualize(Context context)                    start
257                 throws CheckstyleException {
258             final Collection<String> attributes = context.getAttributeNames();
259
260             for (final String key : attributes) {
261                 final Object value = context.get(key);
262
263                 tryCopyProperty(key, value, false);
264             }
265         }                                                                   end
266
267         /**
268          * Returns the configuration that was used to configure this component.
269          *
270          * @return the configuration that was used to configure this component.
271          */
272         protected final Configuration getConfiguration() {
273             return configuration;
274         }
```

Figure 3.3: An example of the code clone in the source file. The start and end lines of the code clone can be affected by changes inside and outside of the code clones.

command to compare the changes to a previous commit C1. If a clone snippet is changed in C2, we update the start and end line numbers of the clone from C2. To generate the mapping and check the modifications, we used a third-party python patching parser, called whatthepatch [57]. If the start or the end of the clone snippet is deleted, we move the clone line numbers accordingly to reflect the deleted lines.

A clone pair is said to be changed if a clone snippet is modified or deleted. When a clone pair is changed, we assess whether such a change is consistent or inconsistent. We verify this by searching the list of clone pairs generated by iCLONES. For each clone fragment having start and end line numbers, if another clone of this fragment is found in the tool results, a consistent state is assigned to the clone genealogy at that specific change point. Otherwise, an inconsistent state is assigned to the clone genealogy. It is important to note that a clone pair may undergo many changes where each change may have a different state. We repeat the process until one or both of the clones are deleted or until we reach the latest commit in the project.

The names of code files may also change during the development of the project. For each commit, we extract a file pair between the current and previous commits where an old file is deleted, and a new file is added. If the code similarity between the deleted file and the new file is more than 99%, we consider this as a renaming operation. A similar approach to identify renamed files is used by Barbour *et al.* [1]. We use the following command to extract the file pairs to detect renaming operations:

git diff [old-commit] [new-commit] –name-status -M

3.3.3 Detecting Buggy Clones

We use the SZZ algorithm [45] to identify the changes that introduced bugs. First, we use the heuristic proposed by Fischer *et al.* [58], to identify bug fixing commits using a regular expression. The regular expression identifies the bug-ID in the commit messages. If a bug-ID appears in the commit message, we map the commit to the bug as a bug-fixing commit. Second, we mine the issue reports of each project from GitHub. For the issues that are closed, we identify if there are any pull requests associated with such issues. If there is a pull request associated with an issue, we identify all the commits included in the pull request and map the commits to the issue as a bug-fixing commit.

Once we have a list of all bug-fixing commits, we use the following command to identify all the modified files in each commit.

git log [commit-id] -n 1 –name-status

We consider only changes to code (i.e., Java and C) files in a commit. For all changes to a specific file of a bug-fixing commit, we use the following git blame command to identify all the commits when the same snippet is changed.

git blame -L startLineNumber,endLineNumber filePath

The *startLineNumber* and *endLineNumber* are the start and end of the code clone under consideration while *filePath* is the file name where code clone exists. We consider such commits as the "candidate buggy changes". We exclude the changes that are blank lines or comments. Finally, we filter the commits that are submitted before the creation date of the bug reports. We then check whether the commits identified as bug-inducing commits include clone pairs. If a clone snippet is included in the bug-inducing commits, we label the clone change as "buggy".

The buggy commit is a commit in which a bug introducing change is made. We only associate a commit to the genealogy if there is a change within the cloned lines of code. Once a buggy change is introduced via a buggy commit, a code clone becomes buggy at that specific change instance. If any future changes are made to fix the bug, the clone again becomes bug-free. To summarize, the clone genealogy can experience different state changes from buggy to bug-free and vice versa.

In our dataset, we treat each change in a code clone as a separate data point. For example, if there are 10 changes in a clone genealogy for any clone pair, there are 10 different data points for this instance. It essentially means that we are labelling the clone pair as buggy in any of the specific commit. Let us assume that a bug introducing change is in the third commit and bug fixing change happens in the fifth commit. The clone pair would be labelled as bugy in only the third and the fourth commit and would be bug-free in the fifth commit and on-wards unless a new bug happens in the genealogy.

During the lifetime of a clone genealogy, there can be buggy changes and subsequent bug-fixing changes later in the lifetime of the clone genealogy. In our dataset, we observe that more than 60% of the buggy clone changes are fixed at some points in the later stage of the project while the rest of the clone genealogies remain buggy. It is also interesting to note that while the clone genealogy is in a buggy state, more than 90% of the clone pairs are inconsistent. It means that both copies of the code clones are not changed together in such cases. To calculate the above-mentioned statistics, we identify the bugs in the code clones and match them with the state of the code clones (more details in the replication package). To identify the clone genealogies with the bug fixing changes, we measure if a code clone has a bug at any stage of the

lifetime and later the bug is fixed.

3.3.4 Collecting Features

For each instance in a clone pair genealogy, we extract multiple features that may help rank bug-prone code clones. Some of the computed features were used in a prior study by Barbour *et al.* [1]. The features are divided into four categories; product, process, genealogy, and user metrics. Table 3.1 presents the description of our collected features.

Removing Correlated and Redundant Features. We remove the redundant features to avoid the possibility of having correlated features interfering in the interpretation of our models [59]. We use the Spearman rank correlation test with a cut-off value of 0.7 to identify the redundant features [60]. To construct the hierarchical overview of inter-feature relationships, we run the varclus function from the R package Hmisc [61]. The results indicate that: 1) clone age in days (*CAgeDays*) and clone age in commits (*CAgeCommits*) are correlated; and 2) submitted pull requests (*NFixPR*), accepted pull requests (*NAccPR*), and rejected pull requests (*NRejPR) are correlated*. We therefore retained *CAgeCommits* and *NFixPR*, and removed the rest of the redundant features from our model training step.

This section presents background information on motivating example of ranking code clones and multiple machine learning ranking techniques.

Table 3.1: The calculated features for the clone pair genealogies

Metric	Description
	Product Metrics
CLOC	The number of cloned lines of code.
CPathDepth	The number of common folders within the project directory structure.
CCurSt	The current state of the clone pair (consistent or inconsistent).
FChgFreq	The average of the file changes experienced by clone pair.
CAgeCommits	The clone age in terms of the number of commits.
CAgeDays	The clone age in terms of the number days.
CSib	The number of siblings of the clone pair at any specific commit.
	Process Metrics
EFltDens	The number of bug fix modifications to the clone pair since it was created divided by the total number of commits that modified the clone pair.
TChurn	The sum of added and the changed lines of code in the history of a clone.
TPC	The total number of changes in the history of a clone.
NumOfBursts	The number of change bursts on a clone. A change burst is a sequence of consecutive changes with a maximum distance of one day between the changes.
SLBurst	The number of consecutive changes in the last change burst on a clone.
CFltRate	The number of buggy modifications to the clone pair divided by the total number of commits that modified the clone pair.
	Genealogy Metrics
EEvPattern	One of the *SYNC, DIV, INC, LP,* or *LPDIV* clone evolutionary patterns [1]
EConChg	The number of consistent changes experienced by the clone pair.
EIncChg	The number of inconsistent changes experienced by the clone pair.
EConStChg	The number of consistent change of state within the clone pair genealogy.
EIncStChg	The number of inconsistent change of state within the clone pair genealogy.
EFltConStChg	The number of re-synchronizing changes (i.e., *RESYNC*) that were a bug fix.
EFltIncSChg	The number of diverging changes (i.e., *DIV*) that were a bug fix.
EChgTimeInt	The time interval in days since the previous change to the clone pair.
UUsers	The number of unique committers in the clone genealogy.
	User Metrics
CommiterExp	The experience of committer (i.e., the number of previous commits submitted before a specific commit.)
NFixPR	The number of pull requests submitted by a specific committer.
NRejPR	The number of pull requests rejected for a specific committer.
NAccPR	The number of pull requests accepted for a specific committer.
Contributor	Core, if number of commits by a specific committer is higher than the average commits by contributors over the past months otherwise Casual.
CCurFile	The total number of changes to current file by a specific committer.
OChgRatio	The number of commits in the genealogy by a specific committer.

3.4 Study Results

3.4.1 RQ3.1. Can we use learning-to-rank (LtR) algorithms to effectively rank bug-prone code clones?

Motivation. Developers in software projects aim to make the best use of their available time and resources. The maintenance of code clones can be a hectic task, especially in large software systems, as the number of clones can be immense. To identify the code clones at each commit level, developers can run a clone detection tool that provides developers with the list of clones at a specific commit. To identify clones that need to be maintained in priority, they often have to browse through a long list of clone candidates, which is time-consuming. If the risk of each code clone could be assessed accurately through an automatic process, we could save developers some precious time. Although previous studies [1] have examined the bug-proneness of different clone evolutionary patterns, to the best of our knowledge, they did not prioritize clones for maintenance. In this research question, we examine the possibility of leveraging LtR algorithms to provide developers with a ranked list of code clones based on the bug-proneness of the code clones. This ranked list can help developers make informed decisions and better utilize their time and resources by focusing on the most bug-prone clones. We select LtR algorithms, because they have been successfully used in previous studies to rank bug reports [62], web services [63], and pull requests [64].

Approach. We use three different types of datasets to train and test the LtR models: a) data from the whole lifetime of the projects (Model_{all}); b) data from the early phase of the projects (Model_{early}); and c) data from the mature phase of the projects

(Model_{mature}). The definition of the Model_{early} and Model_{mature} are the same across all the research questions. Model_{early} is used to model the early phase that is a project tenure during the start of the project while Model_{mature} is used to model the mature phase that corresponds to the latest changes to a project.

To select the data for Model_{early} and Model_{mature}, we proceed as follows. We subtract the latest commit date from the initial commit date of each project. The difference in the number of days is then converted to four quantiles. We use code clone data mentioned in Table 2 of the commits from the first (1^{st}) quantile of each project for Model_{early} and code clone data mentioned in Table 2 of the commits from the fourth (4^{th}) quantile for Model_{mature}. A similar approach of time-based slicing of the data has been used in previous studies to slice Stack Overflow data [65], and Gitter Chatroom data [66].

To assess the effectiveness of LtR algorithms for clone ranking, we train the LtR models described in Section 3.2.2 on our dataset of clones extracted from 52 projects and assess their effectiveness at ranking the clones. A LtR model is trained using a set of documents (d_1, d_2, ... ,d_n) for a set of queries (q_1,q_2,...., q_n). In our case, the documents are the code clone instances at any commit level, and the query is ranking all the code clones in the commit. A commit $\mathbf{C}$ can contain multiple code clones, including their history, and a single code clone can be a part of multiple commits as well but at a different time in their lifetime. The LtR algorithm computes the relevance between the documents (i.e., code clones) and the query using the features (defined in Table 3.1), and outputs a ranked list of code clones at each commit level. The most risky code clones in terms of bug-proneness appear at the top of this ranking. Developers can therefore prioritize them for maintenance. To assess the performance

of the trained learning-to-rank models, we use binary relevance (MAP) and graded relevance (NDCG) evaluation metrics. We select these two metrics because we have binary labels in our models, i.e., 1 for the code clones that are bug-prone and 0 for the code clones that are not bug-prone. In addition to the aforementioned metrics, we also calculate the precision of the trained models through cross-validation.

The data related to the evolution of the software systems is time-sensitive [67], and this time sensitivity should be considered while evaluating the models trained on such data. Since code clones go through multiple changes during the evolution of the system, when building ranking models, we have to ensure that they are trained only on past code change data. To ensure this, we sorted our data based on the commit date, and for every project individually, we split the data into training and testing sets. It is important to note here that a code clone in the training set can have a commit date that is later than the commit dates of clones in the test set only if the clone is from a different project. In other words, we focus on the correct sorting of the clones within a project. The code clones that belong to the same project always have higher dates in the test set than in the training set. The time-sensitive nature of the data is carefully considered while conducting 10-fold cross-validation.

In this chapter, we only identify clones within projects and perform the analysis as each project in our dataset has its own uniqueness. For example, they originate from different application domains, have their own development style, and follow various management policies [68].

Results.

LightGBM and XGBOOST perform better among the learning-to-rank techniques when ranking code clones based on their bug-proneness. Table

3.2 summarizes the results for the five algorithms and the two frameworks used to perform ranking. The models are evaluated using three LtR evaluation metrics (i.e., precision, MAP, and NDCG). LightGBM and XGBOOST outperform the pairwise and listwise algorithms and achieve a precision of 0.72 (Model_{all}). Overall, we can conclude that learning-to-rank algorithms do not perform well on clone ranking tasks, since developers would have to stiff through a long list of false positives if they were to adopt any of these models. We attribute this weak performance to the nature of the objects that are being ranked. In fact, multiple code clones at the commit level can have the same risk of bug, while LtR models are known to perform well when documents have clear distinctive ranks [64]. In our models, the dependant variable is a Boolean variable indicating whether or not a clone pair experienced a bug-inducing change. Our query is to rank all the pairs in a commit. For example, if there are three different clone pair changes $(c1, c2, c3)$ in a specific commit T, there can be three possibilities.

- None of the changes is bug-prone, (0,0,0);
- all the changes are bug-prone, (1,1,1);
- there are buggy as well as non-buggy code clones, e.g., (0,1,0).

LtR performs well when the documents in the training data has a clear distinctive rank, e.g., (0,1,2). Moreover, the performance of the model increases with the availability of historical data. In the early phase of the projects (i.e., Model_{early}), the performance of the model is lower. When more clone history data is available (i.e., $\textbf{Model}_{mature}$), the models perform slightly better.

Table 3.2: Results of evaluation metrics for LtR algorithms for Java and C projects

Type	**Algorithms**	**Java Projects**			**C Projects**		
		Precision	**MAP**	**NDCG**	**Precision**	**MAP**	**NDCG**
Model$_{all}$							
Pairwise	RankBoost	0.56	0.05	0.17	0.55	0.06	0.15
	RankNet	0.58	0.07	0.20	0.60	0.05	0.17
Listwise	LambdaRank	0.61	0.09	0.22	0.63	0.09	0.20
	LambdaMart	0.60	0.11	0.23	0.61	0.12	0.21
	Random Forest	0.67	0.13	0.26	0.66	0.15	0.24
Framework	XGBOOST	0.71	0.13	0.27	0.67	0.17	0.26
	LightGBM	0.72	0.14	0.27	0.69	0.14	0.26
Model$_{early}$							
Pairwise	RankBoost	0.52	0.03	0.15	0.52	0.03	0.12
	RankNet	0.54	0.04	0.17	0.57	0.04	0.15
Listwise	LambdaRank	0.59	0.08	0.21	0.61	0.08	0.18
	LambdaMart	0.57	0.09	0.20	0.57	0.10	0.19
	Random Forest	0.64	0.11	0.22	0.63	0.14	0.21
Framework	XGBOOST	0.68	0.11	0.25	0.66	0.16	0.24
	LightGBM	0.69	0.13	0.26	0.67	0.11	0.24
Model$_{mature}$							
Pairwise	RankBoost	0.57	0.05	0.18	0.56	0.07	0.15
	RankNet	0.57	0.08	0.20	0.61	0.04	0.19
Listwise	LambdaRank	0.62	0.08	0.21	0.64	0.07	0.19
	LambdaMart	0.62	0.11	0.24	0.63	0.14	0.22
	Random Forest	0.68	0.14	0.24	0.67	0.13	0.23
Framework	XGBOOST	0.72	0.14	0.28	0.69	0.18	0.27
	LightGBM	0.73	0.19	0.29	0.71	0.17	0.28

Summary of RQ3.1

Among the studied learning-to-rank (LtR) algorithms/frameworks LightGBM and XGBoost achieve better results. Overall, learning-to-rank (LtR) algorithms/frameworks perform moderately due to the nature of the code clone representation at a commit level. The performance of the models increases with the level of maturity of the projects, possibly due to the availability of more information about the history of the code clones.

3.4.2 RQ3.2. How well can classification algorithms rank bug-prone clones?

Motivation. In RQ3.1, we used the learning-to-rank (LtR) machine learning algorithms to rank the clone pairs based on their bug-proneness. However, these algorithms only achieved moderate performance (i.e., the precision of 0.72 at best). Hence, in this research question, we examine the possibility of using classification algorithms to achieve better results. Using historical information about the clone pairs, we use classification algorithms to classify them as buggy or not buggy. The probability that a clone pair belongs to the buggy class is used for ranking. These ranks can be used by developers to prioritize clones for maintenance.

Approach. Similar to **RQ3.1**, we use three different types of the datasets to train and test the classification models: a) data from the whole lifetime of the projects (Model_{all}); b) data from the early phase of the projects (Model_{early}); and c) data from the mature phase of the projects (Model_{mature}). We use the 28 features (related to code clones) described in Table 3.1 to train the models. We train and evaluate the models as follows.

Step 1: Training Phase

a) Data for Training and Testing. The data for the three different models are selected in different ways. Data for Model$_{all}$ contains the data from the whole lifetime of the projects. There are 1,789,457 data points from the Java projects and 1,154,673 data points from the C projects. To select the data for Model$_{early}$ and Model$_{mature}$, we proceed as follows. We subtract the latest commit date from the initial commit date of each project. The difference in the number of days is then converted to four quantiles. We used code clone data of the commits from the first (1^{st}) quantile of each project for Model$_{early}$ and code clone data of the commits from the fourth (4^{th}) quantile for Model$_{mature}$. A similar approach of time-based slicing of the data has been used in previous studies to slice Stack Overflow data [65], and Gitter Chatroom data [66]. For Model$_{early}$, we obtained 395,256 rows of data for the Java projects and 156,781 rows of data for the C projects. For Model$_{mature}$, we obtained 725,158 rows of data for the Java projects and 475,317 rows of data for the C projects.

b) Problem Formulation. The dependent variable of our model takes the value 0 when a clone pair is not buggy and 1 when the clone pair is buggy. We use Naive Bayes, Logistic Regression, Random Forest algorithms to classify the clone pairs. We select these algorithms because they are commonly used in classification problems. In addition, we also use the LightGBM and XGBOOST's classifier functions to classify the code clones pairs.

Step 2: Evaluation Phase

We used 10-fold cross-validation to evaluate the performance of the models. We created time-consistent folds as described in Section 3.2.6. We sorted commit data using commit dates and ensured that folds on which models are tested always contain data that is posterior to the data on which the models were trained. Within each

fold, we ensure that validation sets always contain data that is posterior to the data on which the models were trained. This time consistency ensures that we are not predicting past data using future data. We assessed the performance of the models using the metrics described in Section 3.2.3.

Results

Random Forest achieves the highest AUC Score when classifying clone pairs based on their bug-proneness. Table 3.3 presents the results for three classification algorithms and two machine learning frameworks. For all the evaluation metrics, Random Forest achieves the highest scores for both Java and C projects. The reported values are the mean values obtained over the 10-fold cross-validation. The performance of Logistic Regression, LightGBM, XGBOOST, and Random Forest are superior to the maximum performance values obtained in **RQ3.1** when using LtR algorithms. In terms of the execution time, LightGBM is 5x faster than the Random Forest.

Random Forest also achieves the best performance when models are trained using only data from the early phase, or only data from the mature phase. The results of Model_{early} and Model_{mature} presented in Table 3.3 show that models trained exclusively with data collected during the early phase of the projects (i.e., Model_{early}) achieve lower performance than the models trained using data from all the evolutionary history of the projects (i.e., Model_{all}). However, models trained using only data from the mature phase of the project (i.e., Model_{mature}) achieve higher performance than Model_{all}. This result suggests that models are able to more precisely capture the risk of bugs of code clones when more information about the evolutionary history the project is available. However, the recency of the information

Table 3.3: 10-fold cross-validation results of three classification algorithms and two frameworks, evaluation metrics precision, recall, accuracy, AUC, and F1-score. Java and C projects results presented separately.

	Java projects					**C projects**				
Evaluation Metrics	**Precision**	**Recall**	**Accuracy**	**AUC**	**F1-Score**	**Precision**	**Recall**	**Accuracy**	**AUC**	**F1-Score**
Model$_{all}$										
Logistic Regression	0.80	0.58	0.80	0.75	0.67	0.78	0.62	0.79	0.74	0.66
Naive Bayes	0.49	0.79	0.64	0.67	0.60	0.52	0.78	0.64	0.66	0.63
LightGBM	0.91	0.78	0.90	0.87	0.84	0.90	0.75	0.89	0.87	0.85
XGBOOST	0.91	0.78	0.89	0.87	0.84	0.89	0.77	0.90	0.88	0.86
Random Forest	0.98	0.96	0.98	0.97	0.97	0.97	0.96	0.94	0.96	0.96
Model$_{early}$										
Logistic Regression	0.70	0.37	0.76	0.65	0.49	0.71	0.37	0.77	0.64	0.52
Naive Bayes	0.56	0.33	0.71	0.61	0.41	0.55	0.33	0.75	0.60	0.43
LightGBM	0.92	0.76	0.91	0.86	0.83	0.90	0.77	0.90	0.85	0.84
XGBOOST	0.91	0.78	0.91	0.87	0.84	0.88	0.77	0.88	0.86	0.82
Random Forest	0.96	0.90	0.96	0.94	0.93	0.95	0.89	0.94	0.94	0.92
Model$_{mature}$										
Logistic Regression	0.64	0.16	0.89	0.57	0.26	0.63	0.18	0.90	0.58	0.29
Naive Bayes	0.20	0.84	0.58	0.69	0.32	0.25	0.85	0.60	0.70	0.35
LightGBM	0.98	0.79	0.97	0.90	0.88	0.98	0.77	0.96	0.89	0.89
XGBOOST	0.99	0.83	0.98	0.92	0.90	0.97	0.82	0.96	0.90	0.91
Random Forest	0.99	0.95	0.99	0.97	0.97	0.98	0.94	0.97	0.96	0.97

is important (as shown by the high performance of Model$_{mature}$). During maintenance activities, development teams looking to prioritize clones should pay attention to give the models as much recent information about the clones as possible.

Summary of RQ3.2

Random Forest achieves the best AUC when classifying the clone pairs based on the bug-proneness of the code clones. Random Forest also achieves high scores of evaluation metrics in early and mature phase data, while the mature phase model performs better than the early and all phase models.

3.4.3 RQ3.3. Can we use regression algorithms to predict the proportion of buggy changes in code clones and effectively rank bug-prone clones?

Motivation. In this research question, we explore the possibility of using regression algorithms to rank code clones. Specifically, we use regression models to predict the ratio of future buggy changes for each clone pairs and use this information to rank the clone pairs. Clone pairs that are predicted to experience the highest ratio of buggy changes are ranked at the top, i.e., they are considered to be more risky than clone pairs predicted to experience a lower ratio of buggy changes. Similarly to **RQ3.2**, we predict the ratio of buggy changes at different stages of the development process (i.e., early and mature).

Approach. Similar to **RQ3.2**, we use three different types of the dataset to train the regression models: a) data from the whole lifetime of the projects (Model_{all}); b) data from the early phase of the projects (Model_{early});, and c) data from the mature phase of the projects (Model_{mature}).

To train the models, we computed the values of 28 features that belong to four categories of clone-related metrics: process, product, genealogy, and user. Table 3.1 provides a description of the features. One common problem in regression analysis

is multicollinearity, which occurs when two or more independent variables are highly correlated. Although multicollinearity may not affect the accuracy of the model much, it causes imprecise estimates of the coefficient values of the model, which prevents distinguishing precisely between the individual effects of the different independent variables on the dependent variable. Variance Inflation Factor (VIF) is used to determine the level of multicollinearity for the regression problems [69] [70]. As we have used the regression analysis in RQ3.3 and RQ3.4, the VIF approach is used to identify the level of multicollinearity among the features from the dataset. The varclus is used to extract the correlated features from the dataset for RQ3.1 and RQ3.2. Following standard guidelines [69] [70], we retained in our models only features for which the VIF is under 5. The following paragraphs provide more information about the training and evaluation phase of our models.

Step 1: Training Phase

a) Data for Each Model. As explained in the previous research question, there are 395,256 data points for the Java projects and 156,781 data points for the C projects in Model_{early}. For Model_{mature}, there are 725,158 data points for the Java projects and 475,317 data points for the C projects. Model_{all} includes all the data points collected by using our data selection criteria described in Section 3.3.

b) Problem Formulation. The independent variables (X) of our models are the features described in Table 3.1. The dependent variable (Y) is the ratio of buggy changes for a clone pair. In our case, the ratio of buggy changes is expressed as the ratio of buggy commits in which a code clone experienced a change. We calculate the ratio of bug-prone changes using the SZZ approach as explained in Section 3.3. The ratio of buggy changes is calculated using the equation 3.10.

$$Ratio of buggy Changes = \frac{\text{Number of bug-prone changes}}{\text{Total number of changes}} \tag{3.10}$$

It is important to note here that we omit the *CFltRate*(bug-prone modifications) feature in this research question because it is related to our dependant variable. Once we predict the ratio of buggy changes for each clone pair at the commit level, we rank the code clones from high to low based on their estimated ratio of future buggy changes. A ratio of buggy changes of 0 means that the clone pair is not expected to experience a bug-fixing change in the future, while a ratio of 1 means that all future changes on the clone pair are predicted to be buggy.

We use a mixed-effect model [71], to take into account the context of the project in our analysis of bug-prone code clones. A mixed-effect model presents the significant features while keeping in view the context during the model training. The mixed-effect model consists of two types of features, explanatory features, and context features. The explanatory features (i.e., process, product, genealogy, and user) are used to explain the data, while context features (i.e., project) are used to determine the effect of explanatory features. The mixed-effect model is able to show the relationship between the outcome (i.e., the ratio of buggy changes) and the explanatory features while taking into consideration the context features (i.e., project).

We use a linear regression (mixed-effect) as our baseline model (as used in prior studies [66] [60] for software engineering problems) to determine the ratio of buggy changes for a clone pair. We also use ridge regression and lasso regression algorithms; which are commonly used to train regression models. Finally, we use Random Forest, regressor function of XGBOOST, and regressor function of LightGBM. Overall, we

experimented with four regression algorithms and two frameworks. We use the implementation of scikit-learn [72] for the four algorithms and for two frameworks (i.e., XGBOOST[6] and LightGBM[7]), we use their open source implementation available on GitHub.

Step 2: Evaluation Phase

Similarly to **RQ3.2**, we used a 10-fold cross-validation approach to evaluate the models, constructing time-consistent folds as described in Section 3.2.6. We sorted commit data using commit dates and ensured that folds on which models are tested always contained data that are posterior to the data on which the models were trained. We also ensured that validation sets always contained data that are posterior to the data on which the models were trained. These time-consistent folds help ensure that we are not predicting past data using future data.

From the results of the cross-validation approach, we can interpret whether the model is over-fitting. If the training RMSE/MAE and test RMSE/MAE has significant differences, it indicates that the model is over-fitting.

Results

LightGBM outperforms the other algorithms/frameworks in ranking code clones based on their estimated ratio of future buggy changes. Table 3.4 presents the results of the cross-validation for the four algorithms and two frameworks. The models are trained separately for Java and C software projects. The LightGBM algorithm is able to achieve an R-Squared score ranging from 0.87 (for C projects) to 0.89 (for Java projects) for Model_{all}, which is much better than the baseline Logistic Regression (0.70 for C and 0.71 for Java projects). The test and

[6] https://github.com/dmlc/xgboost
[7] https://github.com/microsoft/LightGBM

Table 3.4: 10-fold cross-validation results of four regression algorithms and two frameworks, evaluation metrics R-squared, RMSE, MAE. Java and C projects results presented separately.

	Java projects					**C projects**				
Evaluation Metrics	**R-squared**	**Train RMSE**	**Test RMSE**	**Train MAE**	**Test MAE**	**R-squared**	**Train RMSE**	**Test RMSE**	**Train MAE**	**Test MAE**
Model$_{all}$										
Linear Regression (mixed-effect)	0.78	23.4	24.5	42	44.2	0.68	23.5	25.8	43.8	45.7
Ridge Regression	0.79	21.2	22.6	37.2	38.9	0.77	22.5	23.7	38.6	39.9
Lasso Regression	0.78	18.6	19.5	35.6	36.8	0.77	19.2	21.5	37.2	39.2
Random Forest	0.86	9.5	10.3	18.6	20.3	0.85	10.6	12.2	18.3	19.7
XGBOOST	0.86	5.2	5.8	13.8	15.1	0.86	5.4	6.3	14.2	15.7
LightGBM	0.87	4.7	4.95	11.1	12.3	0.86	4.9	5.6	11.4	12.8
Model$_{early}$										
Linear Regression (mixed-effect)	0.77	24	25.7	44	46.7	0.65	24.3	26.5	44.5	47.2
Ridge Regression	0.76	22.4	23.9	38.6	39.5	0.75	24.8	26.7	39.5	42.1
Lasso Regression	0.76	19.5	21.3	36.1	38.6	0.74	23.5	24.9	38.4	40
Random Forest	0.82	10.5	11.7	19.3	21.8	0.82	11.6	13.1	19.5	21.6
XGBOOST	0.84	5.9	7.1	14.8	16.4	0.83	6.2	7.6	15.3	16.9
LightGBM	0.84	5.8	7.3	14.5	16.5	0.82	6.1	7.9	16.5	17.7
Model$_{mature}$										
Linear Regression (mixed-effect)	0.80	22.7	23.9	41.6	44	0.0.71	22.7	25.3	43	45.2
Ridge Regression	0.79	20.9	21.7	37.1	39.2	0.78	21.9	23.4	38.1	39.5
Lasso Regression	0.80	19.3	21	36.3	37.5	0.79	20.2	21.8	37.5	39.8
Random Forest	0.87	8.8	10.1	17.5	19.3	0.86	9.7	11.3	17.9	19.2
XGBOOST	0.89	4.9	5.4	12.3	13.4	0.88	5.1	5.8	13.5	14.6
LightGBM	0.90	4.1	4.6	9.7	10.9	0.89	4.3	5	10.3	11.8

train RMSE and MAE are also better for LightGBM than the other studied algorithms. The performance metrics of XGBOOST and Random Forest show that they outperform the baseline and achieve a performance close to that of the top performer LightGBM. It is important to note here that the train and test RMSE and MAE scores are uniform, which is one of the indicators that the model is not overfitting. The regression results are significantly better than the learning-to-rank(LtR) results while classification models seem to be more effective. However, the classification approach is a more simplistic approach which only provides developer with the information on whether a clone pair would have a bug. The regression approach provides the ratio of buggy changes in a clone pair and achieve good results. Developers can use the approach based on their specific need and improve the maintenance of the code clones.

XGBOOST and LightGBM can be used in the early and mature phases of software projects to determine the rank of the code clones. Table 3.4 shows the evaluation metrics results for Model_{early} and Model_{mature}. XGBOOST achieved an R-squared value ranging from 0.83 (for C projects) to 0.84 (for java projects) for the Model_{early}. The evaluation metrics results for the early phase is lower than that of the Model_{all} which is understandable because the history of the code clone is not developed in the early phase. However, it is encouraging to see that Model_{mature} is able to outperform Model_{all}. This is similar to the results obtained with classification algorithms. A possible explanation for this phenomenon is the recency of information used in Model_{mature}. The occurrence of bugs in matured clone pairs is likely to be more predictable than the occurrence of bugs in newly created clone pairs, and recent information about the clone pairs are better predictors of bug occurrences than older

information. During maintenance activities, developers can select the appropriate model (by taking into account the level of maturity of the project) to predict the risk of bugs in code clones contained in their system. The top ranked code clones can be fixed first, to prevent future clone related bugs.

Summary of RQ3.3

LightGBM outperforms the studied algorithms/frameworks when trained for ranking code clones based on the ratio of buggy changes. However, LightGBM ad XGBOOST both can be used in the early and mature phases of the project as they perform well. The mature phase performance from the previous two research questions also holds in this context.

3.4.4 RQ3.4. Which metrics are significantly correlated to the risk of bugs in code clones?

Motivation. In RQ3.2 and RQ3.3, we trained machine learning models to rank code clones at the commit level, using 28 features from four different categories. The detection of code clones, the identification of clone genealogies, and the computation of the features of the models can be costly. In this research question, we examine the importance of each feature used in our models to identify the subset of features that are the most important for predicting bug occurrence in clone pairs. It is important to note here that we only identify significant features for our regression approach (RQ3.3),since it provides the ratio of buggy changes and achieves good performance. Developers can use similar steps to identify significant features in the learning-to-rank

(LtR) and classification approaches. The identified features can provide quick guidance (before using machine learning models trained in previous research questions) to developers, helping them to prevent bug-occurrence in code clones.

Approach. In this section, we describe the approach followed to identify the significant features in predicting the rank of the code clones, based on their bug-proneness. We computed features following the steps described in Section 3.3. The data for all the three models (i.e., Model_{all}, Model_{early}, and Model_{mature}) are collected using the approach described in RQ3.2. We also identify the level of multicollinearity using VIF as mentioned in RQ3.3.

Building and Analyzing the Mixed-Effect Model. Because the studied software projects belong to different domains and different programming languages, the behavior of code clones in these projects is likely to be different. To take into account the context of the project in our analysis of bug-prone code clones, we use a mixed-effect model [71] as already explained in RQ3.3.

It is important to note that we build three different mixed-effect models (i.e., Model_{all}, Model_{early}, and Model_{mature}) for each of our three datasets. We use the glmer function of the R package lmer [73] to construct mixed-effect models. The discriminative power of the model measures the ability of the model to distinguish values 0 and 1 of the dependant variable (i.e., predicting the number of buggy changes of the code clones). We use the Area Under Curve (AUC) [48] to determine the discriminative power of the model. We use Receiver Operator Curve (ROC) to plot the true positives against the false positive for different thresholds. The value of AUC ranges from 0 to 1, 0 being the worst performance, 0.5 being the random guessing performance, and 1 being the best performance. [48]

To understand the impact of explanatory features, we use Wald statistic [74] to estimate the relative contribution (X2). A higher value of (X2) shows the high impact of the feature on the performance of the model [60]. We use the R package Car [75] which provides the implementation of anova to calcualte wald X2.

Results

The increased involvement of developers in the clone genealogy can have an effect on the bug proneness of the clone pair. Table 3.5 presents the significance between the clone-related features and the number of buggy changes of the clone pairs. Due to the space constraint, Table 3.5 shows only the top five significant features for all three studied models. The AUC is measured to be 0.79 for the Model_{all}. However, detailed results of Model_{all} (Table 3) , Model_{mature}(Table 4), and Model_{early}(Table 5) are available in the Appendix of this thesis. The feature *unique users* accounts for the highest χ^2 in the mixed-effect model of Model_{all}, suggesting that the fewer is the number of developers changing a clone pair, the lower is the risk of bugs. This result is understandable because if more developers make changes to a clone pair, they might not know about the other copies of the clone. Therefore, the clone copies can become inconsistent, and the probability of introducing a bug becomes higher than that of preserving the consistency of clones. In addition, *age_commits* is the second most significant feature. This essentially means that as more changes are made on a clone genealogy, the risk for bug increases.

In mature projects, the age of code clones (in terms of the number of commits) is correlated with the risk of bugs. Table 3.5 shows that *age_commits* is the most significant feature for Model_{mature}. The AUC is measured to be 0.85 for the Model_{mature}, suggesting that, similarly to RQ3.2 and RQ3.3, the models achieve better

performance when more historical information about the clones is available. Some code clones are changed a lot, while other experience significantly fewer changes. If a software project has an extensive code clone history, it is better to use Model_{mature} to obtain better results. The *UnqUsers* (unique committers to a clone pair) is identified as the second most important feature in Model_{mature}. Hence, similarly to Model_{all}, the risk of bug introduction increases when multiple developers change a clone pair. Developers can focus on these two features (i.e., *age_commits* and *UnqUsers*) to reduce the risk of bug introduction in clone pairs.

The changes to code clones in the early phase of the software project should be performed by a fewer number of developers. As shown in Table 3.5 for Model_{early}, *UnqUsers* is the most significant feature in determining the rank of clone pairs based on bug-proneness. The AUC is measured to be 0.77 for Model_{early}. If a clone pair is changed by multiple developers in the early phase of a project, the risk of bug introduction increases. To prevent the introduction of bugs, developers can strive to keep the number of developers involved in changing cloned code to a minimum level. We suggest that any code clone changed by more than two developers should be carefully monitored. As observed in Model_{early} results, *age_commits* is also the second most important predictor of future bugs in clone pairs during the early phase of a project. A clone pair experiencing a higher number of changes at the start of the project should be monitored carefully. The feature *age_commits* has a higher importance in Model_{early} in comparison to Model_{all}; which emphasizes that a higher number of changes to a clone pair at the start of the project is not recommended.

Table 3.5: Results of the mixed-effect model for Model_{all}, Model_{mature}, and Model_{early}. Sorted by χ^2 in decreasing order

. Top 5 features only

Factor	Coef.	χ^2	Percentage	$Pr(<\chi^2)$	Sign.$^+$	Relationship
			Model_{all}			
(Intercept)	$-4.653e^{+00}$	1238		$<2.2e^{-16}$	***	↘
UnqUsers	$1.617e^{-01}$	300315	64.53	$<2.2e^{-16}$	***	↗
age_commits	$4.907e^{-01}$	117350	25.22	$<2.2e^{-16}$	***	↗
FCngFreq	$-2.470e^{-03}$	20924	4.50	$<2.2e^{-16}$	***	↘
UChgRto	$2.668e^{-03}$	5917	1.27	$<2.2e^{-16}$	***	↗
age_days	$2.823e^{-04}$	4352	0.94	$<2.2e^{-16}$	***	↗
			Model_{mature}			
(Intercept)	$-6.772e^{+00}$	324		$<2.2e^{-16}$	***	↘
age_commits	$5.230e^{-01}$	11639	52.24	$<2.2e^{-16}$	***	↗
UnqUsers	$1.228e^{-01}$	2152	9.66	$<2.2e^{-16}$	***	↗
UFileChg	$-1.668e^{-03}$	1318	5.92	$<2.2e^{-16}$	***	↘
FCngFreq	$-1.185e^{-03}$	1248	5.60	$<2.2e^{-16}$	***	↘
age_days	$5.290e^{-04}$	1119	5.02	$<2.2e^{-16}$	***	↗
			Model_{early}			
(Intercept)	$-4.400e^{+00}$	103		$<2.2e^{-16}$	***	↘
UnqUsers	$1.638e^{-01}$	40938	52.42	$<2.2e^{-16}$	***	↗
age_commits	$4.420e^{-01}$	26567	34.02	$<2.2e^{-16}$	***	↗
CPathDepth	$1.082e^{-01}$	3762	4.82	$<2.2e^{-16}$	***	↗
cloc	$5.170e^{-03}$	3736	4.78	$<2.2e^{-16}$	***	↗
SLBurst	$-1.163e^{-01}$	430	0.55	$<2.2e^{-16}$	***	↘

$^+$Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Summary of RQ3.4

When there are more developers changing a clone pair, the likelihood of the clone pair having bug in future increases significantly. As a project becomes mature, the age of code clones also show an association with the introduction of bugs in the clone pair. At the start of the project developers should make sure that code clones are changed mostly by the authors of the clone pairs.

3.5 Discussion

In this section, we discuss the implications of our results for the developers of open-source software projects. First, we will discuss the results of the machine learning approaches used to rank the code clones based on bug proneness. Then, we will specify specific guidelines for the stakeholders. The learning-to-rank(LtR) algorithms (RQ3.1) show moderate results due to the unavailability of the labeled ranked dataset. However, the classification algorithm (Random Forest) achieves better performance among the studied algorithms (RQ3.2). The developers can find this approach very effective in cases where they try to identify whether a clone would be bug-prone. In addition, if developers want to achieve a more accurate ranking by identifying the code clones that can have more bug-prone changes, they can use our regression approach (RQ3.3) that achieves higher evaluation scores using LightGBM. Based on the results from our research questions, we suggest the following guidelines to developers.

Developer can use the learning-to-rank (LtR) approach to rank code clones at a commit level. Our results suggest that LtR algorithms can be useful but have a lower accuracy than classification and regression algorithms. Developers who can label code clones with different ranks based on the associated bugs can obtain better results. The developers can use different rankings based on their project needs. For example, developers can give a higher rank to functional bugs, a middle rank to non-functional bugs, and a lowest rank to non-buggy clones. It can be useful to rank the code clones on a different scale rather than just 0 and 1. A recent study [64] on pull requests uses a distinctive rank (0,1,2) on pull requests based on their priority and shows promising results. Developers can extract clone genealogies and calculate the features provided in Table 3.1 to train the LtR model using XGBOOST and LightGBM.

Developers can use classification models to rank code clones. Classification algorithms, Random Forest in particular achieves high efficiency in determining the probability of a code clone having a bug. The rank of the code clones using these probability values can help identify the most risky code clones at a commit level.

Developers can predict the number of buggy changes of the code clones using regression techniques. We use multiple regression techniques to estimate the risk of future bugs in clone pairs. Our results show that LightGBM and XGBOOST achieve the best results. We ranked the code clones based on the number of buggy changes of the code clones, which developers can use to focus on top risky clones instead of going through all the code clones. Developers can choose the number of clones to refactor based on their resources and eventually, it would prevent future bugs.

We provided specialized models for the early and mature lifetime of open-source software projects. In order to assist the new and old software projects efficiently, we trained two different models by explicitly choosing the data from these two stages of the software projects from our dataset. The specialized models can help developers in making an informed decision related to maintaining code clones at different stages of the development process.

We have identified significant features for developers to look out for when maintaining code clones. If the developers want to get a quick knowledge about the specific features related to code clones that they can focus on (if they are short on time). Developers must be cautious of too many changes by their peers to a particular code clone as it affects the number of buggy changes of the code clones. Moreover, it is essential to keep an eye on the code clones that experience many changes as they tend to be buggy in the future. We suggest that any code clone that is above the average

age of code clones in the project should be monitored carefully. We also identify significant features (e.g., code clones in terms of the number of commit and unique committers in code clone history) for the new and old projects as well, that can aid the developers for their specific software projects.

3.6 Threats to Validity

In this section, we discuss the threats to the validity of our approach.

Threats to conclusion validity concern the relation between the treatment and the outcome. In our case, threats to conclusion validity concern the errors that occurred when processing the code clones. The accuracy of the clone detection is dependant on the clone detection tool used. To achieve a high accuracy, we use iClones recommended by Savalajeko *et al.* [56] who compare multiple code clone detection tools. We use the same setting as recommended by the results of the comparison.

SZZ is the de facto standard in identifying the bug-inducing commits and used by existing studies [76]. The identification of bug-prone commits and bug-prone clone pairs is performed using the SZZ approach. The approach considers that the bug is introduced before the creation of a bug report, and no commit between the bug report creation and the bug fixing commit is considered. The limitations of the SZZ approach are applicable to our data [77].

Threats to external validity concern the selection of projects and the analysis methods. To mitigate the issue of our results being biased towards a particular set of projects, we use well-defined selection criteria beforehand to include large-scale open-source software projects. The projects are selected from different domains of software

development. The clone detection is varied (i.e., the number of clones identified for each project is different) among the selected projects. We aim to show the diversity of the selected projects to ensure that our results are not biased towards a specific language. We include projects from two different languages.

We analyze all revisions of the projects from their creation date until September 2020. The code clones detected, issue reports, and features calculated are valid for this period. A selection of projects from a different time period can result in a different number of clones detected, different clone genealogies, and different values for the features identified.

Prior studies [78] [79] indicate that it is essential to identify different aliases used by developers in an open-source project. We fix the problem of disambiguation of identity (due to multiple aliases) as follows. Instead of the previous approaches that are valid for mailing lists and other similar datasets, we use the GitHub API to retrieve the GitHub account information of the committers. Each commit has an associated email for the committer; we are able to verify that every committer has a different GitHub account. Our approach is similar to the prior studies solving the disambiguation problem [80]. However, similar to existing approaches, we are unable to identify whether a developer has multiple GitHub accounts.

Threats to internal validity concern the possibility of generalizing our results. We select GitHub because it is the most popular platform for open-source projects, and in addition, commits, pull requests, and issue reports are readily available. We use iClones as it can achieve higher accuracy in detecting the code clones. The projects from Java and C programming languages are popular in open-source. Our study can be extended to software projects from other programming languages hosted on

different platforms, and other clone detection tools can be used to detect the clones in the projects.

3.7 Summary

Previous studies report that code clones should be adequately maintained to reduce maintenance costs and prevent future bugs. The occurrence of similar code fragments in the software project can be harmful, leading to the introduction of bugs in the software projects. In this chapter, we examine the possibility of ranking clone fragments based on the risk of future bug occurrences. Specifically, we identify code clones from 534,672 commits from 34 Java and 18 C open-source software projects from GitHub. We identify 469,239 clone genealogies from the studied projects and examined the bug association to the clone genealogies. We then calculate 28 different features related to process, product, genealogy, and users for the clone pairs identified. We experiment with different learning-to-rank (LtR), classification, and regression machine learning models. Our findings can be described as follows.

- We train LtR models to identify the effectiveness of LtR algorithms on our dataset. Our results show that LightGBM achieves a precision of 0.72 to rank the code clones for bug-proneness.

- We use classification approaches to predict the probability of a code clone having bug or not. Random Forest achieves the highest AUC (0.96) among the studied classification approaches.

- We use regression approaches to predict the number of buggy changes of the code clones. Our 10-fold cross-validation on six different regression approaches

shows promising results and indicates that LightGBM (0.87 R-squared) is useful in predicting the number of buggy changes of a code clone.

- We provide specialized models using early and mature phase data of the projects. Our analysis shows that as projects become mature, information about the history of the code clones can improve the performance of prediction models of future bug-proneness. Our specialized models can be used by developers for new projects as well as projects that are mature.

- We build a mixed-effect model to identify the significant features for predicting the proportion of buggy changes of the code clones. Our analysis identifies that *UnqUsers* (i.e., the number of unique developers making changes to a clone pair) and *age_commits* (i.e., the number of commits that a clone pair has changed) has a significant effect on the prediction. Developers can focus on the top significant features for a quick suggestion to prevent bugs in the future.

We attempt to provide all the details needed to replicate our study. [8]

[8] https://zenodo.org/record/7229977

Chapter 4

An Empirical Study on Effort of Propagating Code Changes in Code Clones

In this chapter, we present the empirical study conducted to study the effort of propagating code clone changes to code clone siblings. Section 4.1 discusses the problem and motivation of our study. Section 4.2 describes the study setup of our work. Section 4.3 presents the results for the three research questions in our study. Section 4.4 emphasize on the results and how they can be used by developers. Section 4.5 describes the threats to validity for our study. Finally, Section 4.6 summarizes our study and provide directions for the future work.

4.1 Problem and Motivation

During the maintenance phase of a project, code clones are continuously updated [23]. However, it is challenging for developers to identify all the siblings in a clone group as clone siblings may be spread from the original directory to other directories and the developer loses track of them. To find code clones in a commit, a

developer has to run a code clone detection tool on the snapshot of a project. However, clone detection tools generate long reports and make it hard for a developer to spot and examine the inconsistencies among code clone siblings. Additionally, there can be many siblings in a clone group and developers are unaware of all of them and, eventually, they may end up changing only some of them.

Our study shows that in almost 83% of the cases (in 32 Java projects), developers do not propagate their changes to all the siblings of a clone group. Prior studies have shown that it can be harmful to keep inconsistent clone groups [23]. The code clone change should not be propagated to its siblings when a change to a code clone causes a defect. If the changes require a large amount of effort to propagate to clone siblings, alternative solutions (e.g., clone refactoring) may need to be considered by developers. Therefore, developers need more information to decide if a code clone change should be propagated or not to its siblings and be aware of the effort for propagation. Nevertheless, the propagation of a clone change to its siblings is not a straightforward decision, since developers have to take into account different aspects before propagating changes. For example, a bug-fixing change would have a higher priority to propagate to clone siblings than a new feature.

Recent studies have used code *clone evolution history* to analyse the risk of different clone evolution patterns [1], graph-based change prediction [81], and identify short-lived code clones [24]. Although the above studies can help alleviate the maintenance effort of the code clones, they do not support developers in propagating inconsistent changes to siblings.

In this study, we extract code clones from 32 open-source Java projects and build clone genealogies. More specifically, we provide a machine learning approach that

estimates the effort to propagate a code change to clone siblings. We classify the code changes into types of changes by using the fastText, a natural language processing (NLP) approach for efficient text classification and representation learning [82]. The classification of change types can help developers prioritize code changes to code clones based on the effort of change propagation. Moreover, we locate the inconsistent siblings and use CodeBert (a deep learning model for source code) to predict the bug-proneness of a code change in code clones. To the best of our knowledge, we are the first to attempt to use source embeddings to predict the bug-proneness of code clone changes.

To verify the usefulness of the suggestion for change propagation, we perform a user survey among the GitHub developers who have contributed to the subject projects. A total of 141 participants provide their feedback on our work. To this end, we aim to answer the following research questions.

RQ4.1: How well can we predict the effort needed to maintain code clones? In this RQ, we estimate the effort needed to propagate code clone changes to the siblings of a clone group. We utilize six machine-learning models to estimate the propagation effort of the changes. Our results suggest that Random Forest (RF) can achieve better root mean square error (RMSE equal to 0.87) performance compared to its counterparts. The results of the survey show that 79% of the participants agree that the information (e.g., effort estimation) provided by our approach is very useful for the developers to decide whether a change to a code clone could be propagated to the clone siblings.

RQ4.2: What are the efforts for propagating different types of code clone changes at a commit level? In this RQ, we provide three main categories

according to the existing study [37]: (1) *perfective* (refactoring changes), (2) *corrective* (fixing bugs), and (3) *features* (new implementations). The results show that *perfective* type of changes is the most common, while *features* type of changes is the least among the code clone changes. By further investigating our results, we find 16 sub-types of *perfective* changes. The impact of effort on propagating different types of code clone changes shows that the corrective changes have the least effort involved among the types of code clone changes. The results of the survey indicate that 73% of the participants find the classification of different types of changes helpful to propagate the code clone changes to siblings.

RQ4.3: What are efforts for propagating a bug-prone code change that may cause inconsistency among siblings? In this RQ, we predict the bug-proneness of code changes to a code clone using CodeBert for developers to decide if it is safe to propagate clone changes to the siblings. We achieve a 0.7 AUC score, which is the highest AUC for code embedding-based bug-proneness predictor for code clone changes when training the CodeBert for the bug-proneness prediction in code clone changes. The results also show that the bug-prone changes have more effort involved in propagating the code clone changes to clone siblings. The results of the survey indicate that 72% of the participants find the bug-proneness prediction useful for the propagation of code clone changes to siblings.

4.2 Study Setup

In this section, we discuss the experiment setup we use in our study. Figure 4.1 shows the overview of our approach. First, we mine GitHub projects to select subject projects for our work and run clone detection tools to build clone genealogies for the

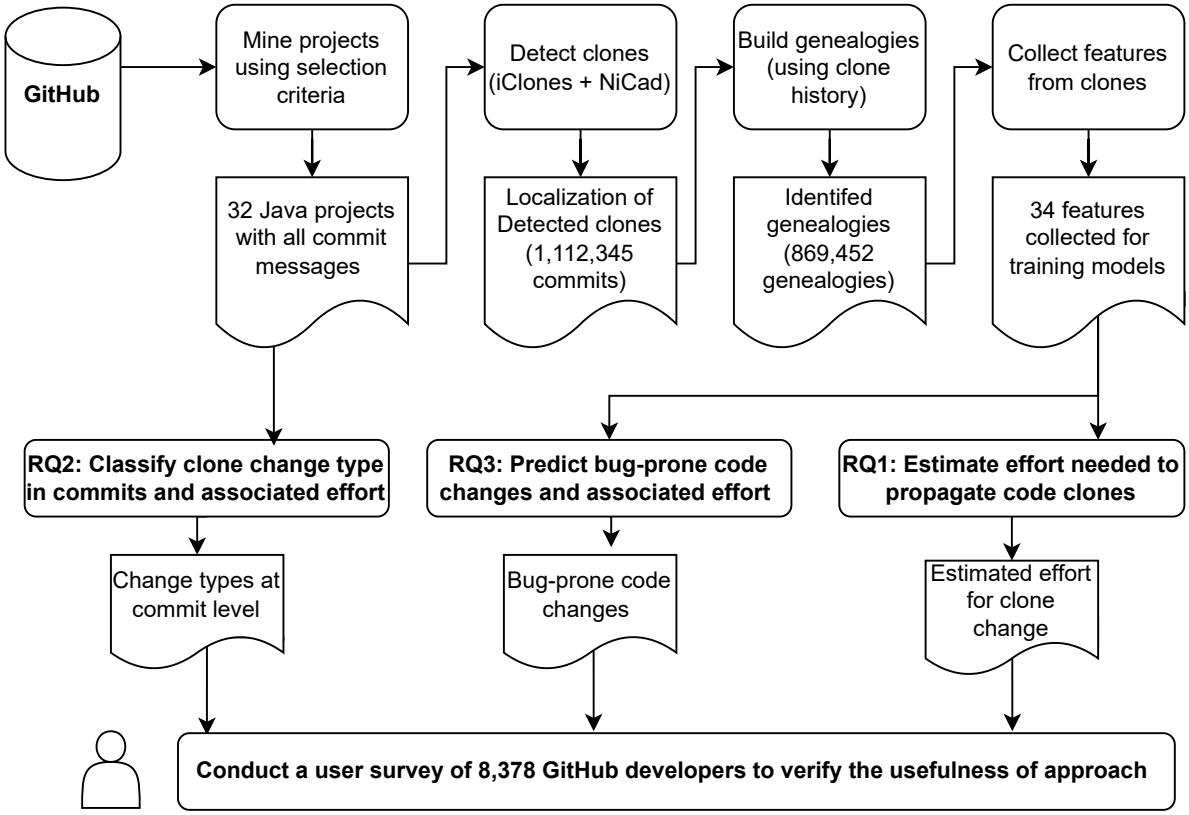


Figure 4.1: Overview of our approach for effort estimation for propagating code clone changes.

subject systems. To build clone genealogies, we use the same steps as mentioned in Section 3.3.2. We then extract features from the subject systems that are used in the research questions. We describe each of the steps in detail in the following subsections.

4.2.1 Projects Selection

To extract projects from GitHub, we use GHTorrent [83]. To ensure that we have enough historical data for generating clone genealogies and bug information, we select projects with at least 1,000 commits, 1,000 issues, and 1,000 pull requests. We

limit the scope of this study to only Java projects, similar to prior studies on code clones [1, 3]. Our initial selection criteria provide 66 Java projects. We include the long-lived projects that are created before June 2015. Furthermore, we discard any projects with fewer than 100k lines of source code (SLOC) as suggested in a prior study [1]. To identify the SLOC, we use GitHub project SLOC calculator extension[1] that can be added to the GitHub. Next, we remove any forked projects and projects with less than 70% of Java code. After applying all the above criteria, we obtain 32 Java projects that we utilize to perform our study. Table 7 illustrates descriptive statistics of the selected projects.

4.2.2 Collecting Features

To train machine learning models, we calculate multiple features from the collected projects. We collect features for all the individual instances (each change to a code clone) identified in the clone genealogy. We introduce new features as well as adapt features used by [1]. The features are divided into four categories: process, product, genealogy, and user. The detailed description of each feature and the rationale for selecting the feature are presented in Table 4.1.

4.2.3 User Survey

In this section, we present the design and distribution of our user survey.

Survey Design. We start the survey with the background information, types of changes, bug-proneness, and effort estimation of code clones. In addition, we provide the participants with a random sample of 10 code changes in the clone groups from our dataset. Each change provided to the participants includes the following details.

[1] https://github.com/artem-solovev/gloc

Table 4.1: The extracted features for the clone group genealogies

Metric	Description	Rationale
	Product Metrics	
CLOC	Number of cloned lines of code [1].	Bigger clones may have higher maintenance costs
CPathDepth	Number of common folders in the project directory structure [1].	Closer code clones might change more often.
CCurSt	Current state of a clone group (consistent or inconsistent) [1].	The states of the clone can have a different effect.
FChgFreq	Average of the file changes experienced by a clone group.	More changes to the file may have more maintenance.
CAgeCommits	Clone age in terms of the number of commits.	More changes to clone may mean more maintenance.
CAgeDays	Clone age in terms of the number of days.	Long-term code clones can have higher maintenance.
CSib	Number of siblings of the clone group at any specific commit.	Higher siblings can affect maintenance.
	Process Metrics	
EFltDens	Number of bug fix modifications to the clone group since it was created and divided by the total number of commits that modified the clone group [1].	Bugs need to fix fast, so it introduces more changes.
TChurn	Sum of added and changed lines of code in the history of a clone [1].	Higher total lines changed in the history can suggest more changes in future.
TPC	The total number of changes in the history of a clone [1].	More changes may mean more maintenance.
NumOfBursts	Number of change bursts on a clone. A change burst is a sequence of consecutive changes with a maximum distance of one day between the changes [1].	Many changes in a short period of time may mean that it might again have such changes.
SLBurst	Number of consecutive changes in the last change burst on a clone.	Number of changes in a short period can suggest how often it is changed
CFltRate	Number of bug-prone modifications to the clone group divided by the total number of commits that modified the clone group.	If a code clone is changed due to a bug, it can be changed again.
TFChanged	The total number of files changed in a commit.	The total number of files can indicate the overall contribution of the developer towards change.
CFChanged	The total number of files having clones and changed in a commit.	The more code clone file change can indicate more maintenance for code clones.
	Genealogy Metrics	
EEvPattern	One of the *SYNC, DIV, INC, LP,* or *LPDIV* clone evolutionary patterns [1].	Different evolutionary patterns have different maintenance affect.
EConChg	Number of consistent changes experienced by the clone group [1].	Consistent may be changed less often
EIncChg	Number of inconsistent changes experienced by the clone group [1].	Inconsistent may change more often.
EConStChg	Number of consistent changes of state in the clone group genealogy [1].	If the clone remains in a consistent state for a longer time, it might have fewer changes.
EIncStChg	Number of inconsistent change of state in the clone group genealogy [1].	If a clone remains in an inconsistent state for a longer time, it may have more changes.
EFltConStChg	Number of re-synchronizing changes (i.e., *RESYNC*) that is a bug fix [1].	When re-synchronized after a bug fix, it may mean fewer changes in future.
EFltIncSChg	Number of diverging changes (i.e., *DIV*) that were a bug fix [1].	If the clone diverges after the bug fix, it may again be changed.
EChgTimeInt	Time interval in days since the previous change to the clone group.	If a clone is not changed for a long time, it might not change soon
UUsers	Number of unique committers in a clone genealogy.	More committer may mean more changes in future.
	User Metrics	
CommiterExp	Experience of a committer (i.e., the number of previous commits submitted before a specific commit.)	Experienced committers might correspond to more stable code clones.
NFixPR	Number of pull requests submitted by a specific committer.	More submitted PRs can indicate more experience.
NRejPR	Number of pull requests rejected for a specific committer.	More rejected PRs can indicate inexperience.
NAccPR	Number of pull requests accepted for a specific committer.	More accepted PRs can show valuable committer.
Contributor	*Core*, if the number of commits by a specific committer is higher than the average commits of the project's contributors over the past months otherwise *Casual*.	Casual and core committers show active and inactive committer, the changes by core committers can be more stable.
CCurFile	The total number of changes to a file by a specific committer in which a clone group exists.	If the same file is changed by many committers, it may experience more changes.
OChgRatio	Number of commits in the genealogy by a specific committer.	If there are more committers in genealogy, more changes can be expected.

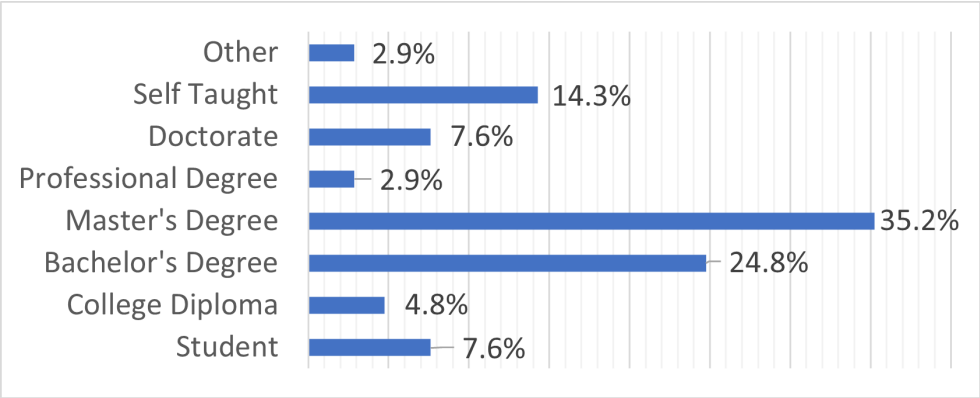


Figure 4.2: The distribution of the participant's education background (Q2 in the survey)

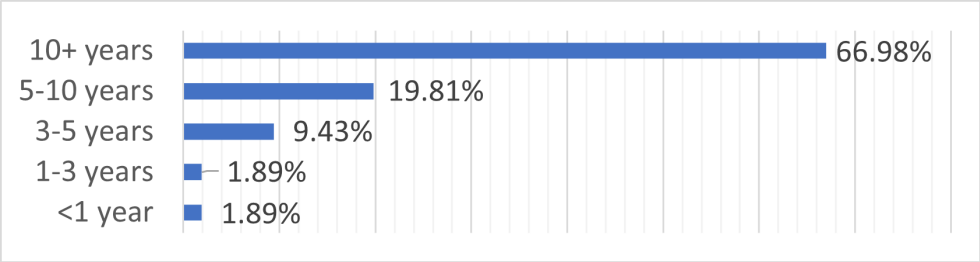


Figure 4.3: The distribution of the participant's software development experience (Q3 in the survey)

- **Commit.** A link to the GitHub commit where a change is made.
- **Clone siblings.** Path and line numbers of each of the clone siblings in a clone group.
- **Type of a commit.** The type of a commit is identified by our approach which includes corrective, perfective, or new features.
- **Probability of a bug.** A percentage value from 0-100% represents the probability of bugs introduced by the change predicted by our approach. A value of 0% represents no possibility of a bug while a value of 100% represents the

highest probability of a bug.

- **Effort estimation.** A real value of 0-3 is predicted by our effort-estimation approach. A value of 0 represents no effort while a value of 3 represents the highest effort.

The survey consists of 10 questions that are from two categories: (1) professional background, and (2) verification of our approach. Table 6 describes the details of the questions in our survey and the options for the answers from the participants as well. Our survey questions are designed to make sure that we identify the professional details (i.e., use of GitHub, education level, and years of experience). Next, we provide the details of each of our research questions in a separate section for participants to evaluate.

The survey was reviewed and approved by Queen's research ethics board[2] before distribution.

Survey Distribution. For each of the 34 projects in our dataset, we identify the commit changes and the emails associated with the commit information using the following git command.

git log - -pretty=format:"%h,%ae,%ai

The above command extracts the email address associated with each commit. In total, we extract 8,378 unique email addresses that are the developers who make at least one code change in at least one of the software projects included in our study. We contact all 8,378 developers using their email addresses by providing them with the survey link[3]. In total, we are not able to contact 1,172 participants via email due

[2] https://www.queensu.ca/planningandbudget/irp/surveys/conducting-survey
[3] https://queensu.qualtrics.com/jfe/form/$SV_dgv6lt6iP9dQ5y6$

to obsolete or non-accessible email addresses. Finally, we receive 168 responses, and 141 participants complete the survey (i.e., 84% completion rate). We only report the results of the completed surveys [4].

Participants Demographic. The first four questions of the survey are related to the demographics of the participants. 96% (135 out of 141) of the participants provided information that they use GitHub. We include the results for all the participants in our survey irrespective of their current use of GitHub. 35% of the participants have a Master's degree while 24% of the participants have Bachelor's degree. Overall, 71% of the participants have a Bachelor's or higher degree while 14% are self-taught. 68% of the participants have 10+ years of experience while 19% of the participants have work experience between five to ten years. Only 4% of the participants have less than 3 years of experience and 9% participants have experience between 3-5 years. 82% of the participants indicate that they do not use any kind of clone detection tools.

4.2.4 Localizing the Changes in a Clone Group

To provide developers with sufficient information to understand code clone changes, our approach can identify the location of the code change and all the associated clone siblings in a clone group. We identify the location of inconsistent siblings through clone detection tools (i.e., NiCAD and iClones) and their code differences by comparing the ASTs of code clone siblings. The localization of the code clone changes includes the following information.

- Identifying path and line numbers. Our goal is to suggest the location where

[4] https://ql.tc/6m4N7a

changes should be propagated to keep the siblings of a clone group in a consistent state. By utilizing NiCAD and iClones, we obtain data, such as the path and line number of each code clone of a project and we use it to find out the locations of all clones of a particular clone group. It is important to note that the starting and ending lines are provided for each clone sibling in a clone group.

- Identifying statements. We aim to identify the statements in a code clone that needs to be changed to have consistency in a clone group. First, we create an Abstract Syntax Tree (AST) for each of the siblings in the clone group. Then, we compare the ASTs of each unchanged sibling against the changed sibling using *GumTreeDiff* [84], a tool that generates the differences between ASTs and provides their distinguishing elements. If we find that any code clone siblings are different, we consider them as inconsistent, otherwise, we consider them as consistent. After finding all the inconsistent instances, we label the inconsistent clone siblings as potential candidates to propagate changes. We keep track of the nodes of the ASTs of two different siblings. This can support developers to identify the exact location in terms of lines of code where a change can be propagated in order to make the clone siblings consistent.

Our goal is to suggest the location where changes should be propagated to keep the siblings of a clone group in a consistent state. By utilizing NiCAD and iClones, we obtain information, such as the path and the line number of each code clone in a project. We use the results of the clone detection tools to find out the locations of all clones of a particular clone group. It is important to note that the start and end lines are provided for each clone sibling in a clone group. Figure 3.3 shows an example of how the location of the code clone is identified and represented in a code file. We can

find the code differences between siblings of a clone group in an inconsistent state by using ASTs to identify the abstract level of code changes among code clones. If a change to a code clone is inconsistent, we compare the ASTs of its clone siblings using *GumTreeDiff*.

4.3 Study Results

4.3.1 RQ4.1. How well can we predict the effort needed to maintain code clones?

Motivation. Developers often modify source code to fix a bug, address an issue, implement a new feature, refactor a function to increase its maintainability, and so on. Multiple features such as lines of code added, and files changed are used as indicators to estimate effort for making code changes in software systems [33]. It is common that a developer only makes changes to some of the clone siblings in a clone group, making the clone group inconsistent [1]. To improve developers' awareness regarding the inconsistent siblings and understand the amount of effort needed to propagate clone changes, we propose the use of a machine learning approach to predict the effort needed to propagate a code clone change to the inconsistent clone siblings. Based on the code changes made in the first clone, developers can use the results to determine whether the changes made to code clones should be propagated to the clone siblings or not.

Approach. In this section, first, we discuss the independent features collected to predict the amount of effort to propagate changes to inconsistent code clones. Then, we introduce the dependent features that are used as indicators for measuring effort. In the end, we describe our approach for training and evaluating the machine learning

models.

(A) Independent Features. Table 4.1 presents a collection of independent features that include process, product, genealogy, and user-related features. We extract code change features in a file level and code clone level. We parse commits using the `whatthepatch` [57] tool, which is a library for parsing patch files. The code in the commit is parsed to extract information that corresponds to the file name, start line, end line, and the actual change. We can determine if the change happens in a code clone as we extract the clone start and end lines from the clone detection tools. After processing the collected commit extracted from patches, we extract the independent features presented in Table 4.1.

Removing redundant independent features. To estimate the effort needed to propagate changes to the siblings of a clone group, we use regression analysis. Regression analysis is a type of statistical method used to determine the strength of a relationship between a dependent variable (usually denoted as y) and several independent variables (denoted as X). A common problem in regression analysis is multicollinearity, which occurs when two or more independent variables are highly correlated. Although multicollinearity may not affect the accuracy of a model, it might cause imprecise estimations of the coefficient values of a model. Therefore, it prevents distinguishing precisely among the individual effects of the different independent variables on the dependent variable. We use Variance Inflation Factor (VIF) [85] to assess the level of multicollinearity between the variables of a model. According to prior studies [69, 70], we retain in our models only features of which the VIF is under 5. Based on the evaluation, we leave out *NRejPR* (the number of pull requests rejected for a committer), *NAccPR* (the number of pull requests accepted

for a committer), *EConStChg* (the number of consistent changes of state in the clone pair genealogy), and *EIncStChg* (the number of inconsistent changes of state in the clone pair genealogy) as defined in Table 4.1.

(B) Dependant Feature. In our work, we wish to use the dependent feature to characterize the effort for propagating code clone changes to clone siblings. However, it is difficult to directly measure effort by monitoring the actual time that a developer has devoted to a programming task or indirectly measure effort using a single metric, such as code churn [33]. Shihab *et al.* [33] suggest that only using lines of code is not a good estimator of effort rather multiple metrics must be used. Shihab *et al.* propose more than 30 metrics for effort-aware bug prediction from different levels of granularity (i.e.,classes, functions, code snippets, statements and other code features). Inspired by Shihab *et al.* we only select the 13 out of 33 metrics presented in Table 4.2) that are closely related to the code clone changes at the commit level as we aim to identify the effort of propagating code clone changes to clone siblings. We categorize the metrics [33] into three categories (i.e., churn, size, and complexity). The details of the selected metrics are presented in Table 4.2.

Removing redundant dependent features. To avoid the possibility of the correlation among the metrics for estimating effort [59], we remove the correlated metrics using the Spearman rank correlation test with a cut-off value of 0.7 [60]. We run the varclus function from the R package Hmisc [61] to construct the hierarchical overview of inter-feature relationships. The results indicate that: (1) the total blank lines (*TBLOC*) and the total code lines (*TCLOC*) are correlated and (2) the total semicolons (*TNOS*) and the total statements (*TNOST*) are correlated. Therefore, we retain *TCLOC* and *TNOST* as the total code lines and the number of statements

Table 4.2: Effort estimation metrics used in the study.

ID	Name	Definition
		Complexity
FOUT	Fan out	The total number of external calls to functions or libraries.
PAR	Nested block depth	The total of nested block depth of the method.
NBD	Number of parameters	The total number of parameters of the method.
VG	McCabe Cyclomatic Complexity	The measure of the McCabe cyclomatic complexity of the method.
		Size
TBLOC	Total blank lines	The total number of blank lines in the code snippet.
TCLOC	Total code lines	The total number of code lines in the code snippet.
TNOS	Total semi-colons	The total number of semi-colons in the code snippet.
TNOST	Total statements	The total number of statements in the code snippet.
TNODST	Declarative statements	The total number of declarative statements.
TNOEST	Executable statements	The total number of executable statements.
		Churn
CHURN	Total Churn	The sum of the number of added lines of code and the number of deleted lines of code.
ADD	Added lines	The total number of added lines in a commit.
DEL	Deleted lines	The total number of deleted lines in a commit.

added can provide developers with a better indication of effort, and remove the rest of the correlated metrics from our model training step. We also perform the redundancy analysis after the completion of the correlation analysis by using redun of rms R package. Redundant metrics are not able to aggregate values for the models and can be explained by other metrics. Our analysis shows that there is no redundant metric in our dataset.

We aim to combine the effort estimation metrics listed in Table 4.2 in a way that each metric contributes to the overall estimation of effort. We first normalize each individual metric (a total of 11 metrics) in the three categories using Equation 4.1

between 0 and 1. As shown in Equation 4.2, the overall effort can be estimated from the three categories of the metrics. To ensure that each category has an equal contribution to effort estimation, we then normalize the value of each category to have a real value between 0 and 1. There are four metrics in complexity and size categories and each metric is divided by 0.25 to have an equal contribution while churn has three metrics and each metric is divided by 0.33. Therefore, the real value of the effort metric ranges from 0 to 3 where 0 corresponds to the least effort and 3 corresponds to the highest effort.

$$x_i = \frac{x_i - x_{min}}{x_{max} - x_{min}} \tag{4.1}$$

where x_i represents current value of the metric; x_{min} is the minimum value for the metric; x_{max} is the maximum value of the metric and $x \in$ {FOUT, PAR, NBD, VG, TCLOC, TNOST, TNODST, TNOEST, ADD, DEL, CHURN} The details of the metrics represented as x are presented in Table 4.2.

$$Effort = Complexity + Size + Churn \tag{4.2}$$

where complexity = 0.25(FOUT+PAR+NBD+VG) ; size = 0.25*(TCLOC+TNOST+TNODST+TNOEST) and churn = 0.33*(ADD+DEL+CHURN). Each metric in complexity, size and churn is presented in detail in Table 4.2.*

(C) Model Training and Evaluation. Our effort metric is composed of multiple features that belong to complexity, churn, and size. As a baseline model to determine the effort needed to propagate the change of a code clone to its siblings, we use linear regression (mixed-effect). A mixed-effect model includes two types of features, explanatory features, and context features. The explanatory features (independent features) are used to explain the data (i.e., dependent features), while context features refer to the project ID. A mixed-effect model shows the relationship between the outcome (i.e., dependant feature) and the explanatory features (independent features) while taking into consideration of the context variables (project).

Overall, we experiment with four regression algorithms and two gradient-boosting frameworks. In addition to Linear Regression, we use Random Forest, Ridge Regression, and Lasso Regression algorithms (by employing their SciKit-Learn [72] implementation). These regression algorithms are commonly used to train regression models according to prior studies [86,87]. Also, we use the regression function of XGBoost[5] and LightGBM[6] by utilizing their open-source implementations from GitHub.

We use a 10-fold cross-validation approach to evaluate our models. Cross-validation is a resampling technique used to evaluate machine learning models on limited data samples. We construct time-consistent folds to ensure that we are not predicting past data using future data. More specifically, we sort commit data using commit dates to ensure that our testing dataset is posterior to the training dataset. We ensure that test validation sets always contain data that are posterior to the data on which the models are trained. As we predict the effort to propagate the code clone changes to clone siblings, we only use the inconsistent code clone changes in our data. From the

[5]https://github.com/dmlc/xgboost
[6]https://github.com/microsoft/LightGBM

results of the cross-validation approach, we can interpret whether a model is over-fitting or not. If the training and testing RMSE/MAE are significantly different, it indicates that our model is over-fitting. We use R-squared, RMSE (root mean square error) and MAE (mean absolute error) as evaluation metrics for the regression models. R-squared is a proportional improvement in the prediction of a regression model as compared to the mean model, while RMSE calculates how close are the predicted values and the actual values. MAE measures the magnitude of the errors and a lower MAE value is associated with high accuracy for a model. The details of the evaluation metrics are presented in Section 3.2.4.

(D) Impact of inconsistent changes on the code clone change effort. We are interested in understanding the effort of propagating inconsistent changes in maintaining code clone groups. We identify whether inconsistent clone changes may require more effort in managing code clone changes. The analysis can help developers understand the effort needed for propagating inconsistent code clone changes to clone siblings. We test the following null hypotheses ($H0_1$) and ($H0_2$).

$H0_1$: *the effort to make a consistent change to a clone sibling is the same as making an inconsistent change to a clone sibling.*

To test the null hypothesis ($H0_1$), we collect two distributions from our clone change datasets (i.e., the effort for each consistent change in the clone groups and the effort for each inconsistent change in the clone groups). The effort for consistent and inconsistent changes is calculated using Equation 4.1 and Equation 4.2 using historical data. We apply the Kruskal-Wallis test [88] using the 5% confidence level (i.e., p-value < 0.05). The Kruskal-Wallis test is a non-parametric statistical test designed to assess whether two or more samples originate from different sources. We

calculate the average and median effort values of the two distributions to identify which distribution has a higher value (average and median) if they are significantly different.

$H0_2$: *the number of file changes in a consistent code clone change is the same as making an inconsistent code clone change.*

To test the null hypothesis ($H0_2$), we collect two distributions including (1) the number of file changes for each of the consistent code clone changes and (2) the number of file changes for each of the inconsistent code clone changes. We use the commit information of each of the code clone changes to identify the number of files changed in the commit. We apply the Kruskal-Wallis test [88] using the 5% confidence level (i.e., p-value < 0.05). We calculate the average and median number of file changes of the two distributions to identify which distribution has a higher value (average and median) if they are significantly different.

Results. Random Forest (0.87 Test RMSE) outperforms all the other regression models for estimating the effort needed to change a code clone sibling. Table 4.3 presents the result of our selected regression algorithms used to predict the needed effort. The Random Forest achieves the lowest train and test RMSE score and has the highest R-squared value. In addition, LightGBM and XGBoost obtain the best train and test RMSE scores against the other algorithms. Developers can use our approach to predict the effort needed to propagate changes to inconsistent siblings of a clone group.

The effort to make the consistent code clone changes is almost double than making inconsistent code clone changes. We further analyze whether the effort to propagate code clone change has any effect on the inconsistency of the clone

Table 4.3: 10-fold cross-validation results of four regression algorithms and two frameworks, evaluation metrics R-squared, RMSE, MAE.

Evaluation Metrics	**R-squared**	**Train RMSE**	**Test RMSE**	**Train MAE**	**Test MAE**
Linear regression (mixed-effect)	0.78	23.9	25.3	42.7	45.1
Ridge regression	0.79	21.8	23.4	38.3	40.3
Lasso Regression	0.77	19.2	20	36.3	37.6
Random Forest	**0.87**	9.9	10.8	19.4	20.9
XGBOOST	0.86	5.7	6.3	14.4	15.8
LightGBM	0.86	**4.9**	**5.3**	**11.5**	**12.2**

group. We observe that the average and the median number of code changes (for inconsistent changes) in terms of effort are 1.2 and 1.4 respectively. However, for consistent changes, the average and median effort of a code change are 2.3 and 2.4 respectively. From the analysis, we can determine that with lower effort for making inconsistent changes to code clones in a commit, the possibility of the clone group being inconsistent becomes higher. After running a Kruskal Wallis test, we achieve a p-value of $6.587e^{-03}$ (which is < 0.05), indicating that the state (i.e., consistent or inconsistent) of code clones has a statistically significant impact on the estimated effort for propagating code clone changes to clone siblings. Hence, we reject the null hypothesis ($H0_1$).

Inconsistent clone groups have a higher number of file changes. We investigate the commits with the code clones for all the selected projects and find that the commits associated with inconsistent changes have overall more file changes in a commit. Inconsistent changes have a median of 4.7 files changed per commit

while consistent changes have 3.5 files changed per commit. However, not all file changes are associated with code clone changes. One possible explanation for having more files changed for inconsistent code clones are that developers have to modify more files to maintain clone groups. Nevertheless, we analyse the code changes for all files associated with commits that cause code clone changes. The results of the Kruskal Wallis test achieve a p-value of $7.351e^{-04}$ (which is < 0.05) indicating that the number of file changes among consistent and inconsistent code clone changes is statistically significant. Hence, we reject the null hypothesis ($H0_2$)

More than 70% of the participants agree that the line of code (LOC) alone is not a good measure of effort. For the Q7 in our survey, we ask the participants about each type of effort estimation metric (i.e., complexity, churn, size) that are included in our approach. The results presented in Figure 4.4 show that on average only 5% participants disagree that the combination of several features (i.e., complexity, churn, size) that are used to estimate effort to propagate code clone changes to clone siblings, included in the study are better than LOC. In other words, 53%, 50%, and 69% of the participants agree that size, churn, and complexity metrics together are useful to estimate the effort needed to propagate the changes in code clones to clone siblings. Among the participants that agree to question 5, 73% of the participants have either a bachelor's, master's, or doctorate level of education in computer science and 78% of the participants have an experience of more than 5 years.

In question 9 of the survey, we ask participants about the feedback for the approach in the form of an open-ended question. Talking about the metrics for calculating effort, two of the comments from the participants include *"LOC is rarely useful"* and

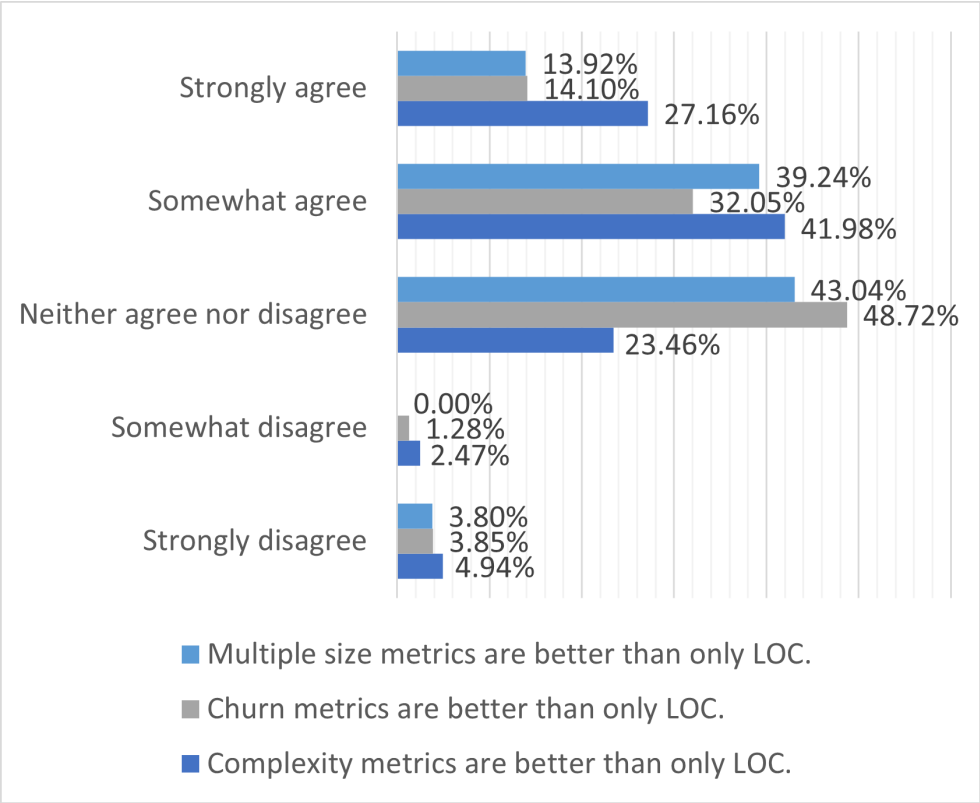


Figure 4.4: The evaluation result of (Q7) In your opinion, how are the complexity/churn/size metrics better than LOC to determine effort estimation?

"LOC is a terrible metric because less code is better". The comments indicate that LOC alone is not helpful and the use of additional metrics can be useful as well as shown in Figure 4.4.

Summary of RQ4.1

Random Forest achieves a higher performance for estimating the effort needed to propagate changes to clone siblings. Our results also show that the consistent changes have significantly more effort involved to make changes in the code clones. More than 70% of the participants agree that a combination of size, churn, and complexity to identify effort for propagation is better than LOC.

4.3.2 RQ4.2. What are the efforts for propagating different types of code clone changes at a commit level?

Motivation. Developers commit different types of changes to a repository. Classifying commits, based on their messages, can help developers know what kind of changes are made to a codebase. From the perspective of code clones, developers can prioritize the types of clone changes that need to be addressed first. We are interested in understanding the effort required for propagating different code types of code clone changes to clone siblings. We wish to use a machine learning approach to classify commit messages into different categories. The output of the approach can specifically identify the types of changes at a code clone level. We further manually investigate the most common change type to identify the discrete changes in that category. Our investigation of the clone change types can help developers identify the rationale of code changes (e.g., fixing a bug or adding a new feature) in the context of code clones and be aware of the effort required for maintaining different types of code clone changes.

Approach. In this section, we first present the approach used to classify commits using commit messages. Then, we investigate the effort required for maintaining

different types of code clone changes.

(A) Data Collection. [37] have performed a study on commits classification and gathered a dataset of 5,631 manually labelled commits marked as *corrective*, *features*, *perfective*, *non-functional*, and *others*. We use the above dataset to train our NLP model in order to classify commit messages.

For testing our model, we collect commit messages from the selected 32 Java projects (see Section 3.3). We check the clone genealogies found in Section 3.3 to make sure that the collected commit messages are either created or updated by code clones. In total, we extract 1,110,437 commit messages from the selected projects.

(B) Data Pre-processing. We pre-process more than a million commits to remove noise from the collected data. The natural language data can contain typos which can negatively affect our classification [89]. To correct spelling mistakes, we use spell checker [90]. We remove any automatic commit messages caused by merging commits, such as, "merge commit X with Y", since they do not offer any value in the training process of our model. For the training and testing datasets, we create a word vector for the sentences in a commit message by using TF-IDF [91], a state-of-the-art technique for creating word representations in a form of a vector. TF-IDF calculates the inverse document frequency of a word, from a particular document to the percentage of documents in which the corresponding word exists. This implies that words with higher TF-IDF scores have a strong association with the documents they appear to.

TF-IDF can represent an N-gram relationship between the words of our dataset. For example, if a commit message is "A bug is fixed", then uni-gram/1-gram would have "A", "bug", "is", "fixed" while bi-gram/2-gram would contain "A bug", "bug

is", "is fixed". The experiment by [37] shows that uni-grams and bi-grams seem to find a good relationship between labels and words. Due to this reason, TF-IDF vectors are created for our commit messages.

(C) Model Training and Evaluation. The commit classification using the widely known NLP model fastText (a library for efficient text classification and representation learning, proposed by Facebook Artificial Intelligence Research [82]) has proved to achieve good results [37]. Therefore, we use fastText to train our model on the manually labelled dataset and, then, use it to predict the labels of the commit messages extracted from the 34 Java projects. The best result with 91% F-measure is achieved using a learning rate of 0.1 with 25 epochs [37]. Therefore, we use the same settings to train and test our model in predicting commit message labels.

To further evaluate the results of the labels of the commit messages, we perform a qualitative study. To perform the study, we select a statistically representative sample using a confidence level of 95% and a confidence interval of 5%. The statistical sample for our dataset contains 384 commit messages. The first and third authors of the study manually label the samples using five different labels: perfective, corrective, features, non-functional, and others. Then, we use Cohen's Kappa [92] to access the agreement between the evaluators. Cohen's Kappa is used to measure the agreement between the evaluators for categorical items (e.g., identifying the change type of the commit by reviewing the commit message). The value of Cohen's kappa ranges from 0 to 1, one being the complete agreement between the evaluators. The inter-rater agreement between the evaluators of the statistical sample is 0.92 which represents a strong agreement.

(D) Impact of code clone change types on the effort to propagate code

Table 4.4: Results of the commit classification for different types of code changes

Change Type	Frequency	Percentage
Perfective	672,261	60%
Corrective	257,703	23%
Features	156,860	14%
Non-Functional	22,398	2%
Others	11,204	1%

clone changes to clone siblings. We study whether types of changes affect the effort of propagating code clone changes to clone siblings. We test the following null hypothesis ($H0_3$).

$H0_3$: *There is no difference in terms of effort to make different types of code clone changes to clone siblings.*

We want to examine if different types of code clone changes would have similar efforts to propagate the code clone changes to the clone siblings. To test $H0_3$, we collect three distributions including (1) the effort for each of the perfective changes to code clones, (2) the effort for each of the corrective changes to code clones and (3) the effort for each of the feature changes to code clones. The effort for different types of code changes is calculated using Equation 4.1 and Equation 4.2 using historical data. To test the null hypothesis ($H0_3$), we apply the Kruskal-Wallis test [88] using the 5% confidence level (i.e., p-value < 0.05). We measure the median and average efforts among the three distributions to identify which distribution has a higher value if they are significantly different.

Results. The majority of code changes to code clones are perfective (60%), while the second most are corrective (23%). Table 4.4 illustrates the predicted labels. Our results show that 97% of code changes to code clones belong to *perfective, features,* or *corrective* code changes. In particular, the most common label predicted

is *perfective* which makes up to 60% of the whole dataset. *Perfective* changes aim to increase code quality without modifying its functional requirements. Examples of perfective changes are: a change in a function's name, an update of a statement, a change in a variable type, etc. The second most common type of code changes for our code clones are the *corrective* changes which make up to 23% of the collected commits, where developers try to fix a bug or provide a solution to an existing issue.

The machine learning approach predicts the labels with an accuracy of 93% in our manually analyzed data. We investigate the similarity of labels between the machine learning approach and manual labelling and find that 93% of the labels are the same. We also perform error analysis on the remaining 7%, where the labels are misclassified, and find that the commit messages in such cases are in-comprehensive. For example, a commit message *"lots of Observer, super X"* was labelled as "Corrective" while it should be other, as it does not provide enough information.

Fixing bugs has a lower effort to propagate code clone changes to clone siblings than perfective and new feature changes. For each type of code clone change, we aggregate the effort and calculate the average among all types of changes. The average and median effort for corrective changes are 1.4 and 1.5, perfective changes are 1.8 and 1.9, and features are 2.1 and 2.2. The result of the Kruskal Wallis test states that we achieve a p-value of $5.824e^{-04}$ (which is < 0.05) suggesting that the distributions of efforts for different types of changes for effort are significantly different. Hence, we reject the null hypothesis ($H0_3$) which shows that different types of code clone change require different efforts to manage code clone groups. The above fact shows that developers need more effort when adding new features to a project

while the least effort is made when fixing a bug.

83% of the code changes in code clones are inconsistent. We extract the state of the code clone group (i.e., consistent or inconsistent). We then analyze the relationship between the state of a code clone group and the code changes in a code clone group. This implies whether a change has been made to all the copies of the code clones in a group. Interestingly, we find that 83% of the code clone changes at the commit level are inconsistent. In other words, it is a prevalent practice for developers to make changes to one or more siblings in a clone group and leave other siblings unchanged.

Corrective type of code changes, where a bug or an issue is addressed, have an even higher percentage of inconsistent code clone changes (i.e., 90%). Thus, the inconsistent changes in clone groups can contribute to inefficient bug fixing and, even more, such bug-prone code clones might be copied by a developer. We identify different sub-types of perfective changes as listed in Table 4.5. The identification of sub-types can help developers in determining the exact type of change in a commit. Once identified, developers can decide on the appropriate actions for the rest of the copies of code clones.

The most common type of changes in our dataset is the perfective changes, which are refactoring changes. We use the classification by [93] to label sub-types of the *perfective* changes in our dataset. To achieve this, we extract a statistically significant sample of 384 commit messages from our dataset using a confidence level of 95% and a confidence interval of 5%. Then, the first and third authors of this study are responsible to label the obtained samples of commit messages. In the end, we calculated the inter-rater agreement using the Cohen Kappa and obtained a

Table 4.5: Manual labeling results on the statistical sample of perfective commit messages.

Change Type	**Frequency**	**% of inconsistency**
Statement Update	132	87%
Statement Insert	96	89%
Statement Delete	61	90%
Attribute Type Change	37	87%
Additional Functionality	22	91%
parameter type change	15	84%
Condition Expression Change	14	70%
Method Renaming	8	-
Attribute Renaming	7	-
Parameter Renaming	4	-
Increasing Statement Insert	2	-
Else-Part Delete	2	-
Else-Part Insert	1	-
Parameter Delete	1	-
Parameter Insert	1	-
Others	13	-

score of 0.87, which suggests a strong agreement between the two annotators. After performing the manual investigation, we perform a manual analysis on the percentage of inconsistent changes among each change type. We observed that *Statement Delete* and *Additional Functionality* have more than 90% of inconsistent changes. *Statement Update*, *Statement Insert*, and *Attribute Type change* also have high percentage of inconsistent changes. Table 4.5 presents the frequency of the *perfective* changes sub-types found in the statistically significant sample of commit messages along with the percentage of inconsistent changes.

The results of the labelling show that our code clone dataset contains 16 different types of *perfective* changes. The most common refactoring change is the *statement update* which has almost 34% of instances in our dataset. *Statement update* is a type of change that revises operators, operands, or a statement expression.

Other common types of refactoring changes include *statement insert*, *attribute type change*, and *additional functionality*. The output of our labelling shows that we can further identify sub-types in the *perfective* changes. Our qualitative analysis of the sub-type of perfective changes shows that refactoring practices in code clones occur in diverse granularities (e.g., change the logic, change sub-statement code element) and different purposes (e.g., change attribute vs parameter). Future research can explore an automated approach to identify different types of perfective changes.

73% of the participants in our user survey agree or strongly agree that identification of the different types of changes at a commit level is useful. Figure 4.5 shows the results of question 5 in our user survey. Among the participants that agree to question 5, 83% of the participants have either a bachelor's, master's, or doctorate level of education and 87% of the participants have an experience of more than 5 years. In particular, 36% strongly agree, and 37% choose somewhat agree while only around 11% of the participants disagree that the information about the types of code commits might not be useful for propagating the changes of code clones to clone siblings. One of the participants added a comment *"New features sometimes lead to refactoring changes which exposed bugs."* when discussing the different types of changes presented at the commit level which indicates that it is helpful to know different kinds of changes performed at the commit level.

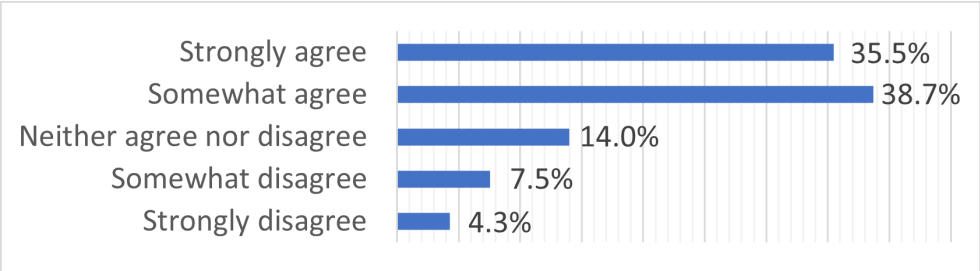


Figure 4.5: The evaluation result of (Q5): Do you perceive the type of code changes differently, for example, corrective (bug-fixes), perfective (refactoring changes), new features etc?

Summary of RQ4.2

The most common type of code changes to code clones are the perfective, corrective, and features code changes. Perfective changes count for more than 60% of the code changes in code clones. The comparison among efforts for making different types of code clone changes and effort indicates that the corrective changes have the least effort to propagate the code clone changes to clone siblings among all different types of code clone changes. 73% of the participants of the user survey find the classification of the types of code clone changes helpful to propagate changes to clone siblings.

4.3.3 RQ4.3. What are efforts for propagating a bug-prone code change that may cause inconsistency among siblings?

Motivation. Developers are unaware whether a particular change to a code clone can introduce a bug. If code clone changes would potentially cause bugs, developers should fix the bugs first before propagating the changes to the siblings. Therefore, it is important to provide the bug-proneness of a code clone change so that developers can

make an informed decision on whether to propagate a change. Such information aims to ease the maintenance effort of code clones and could potentially lead to fewer bugs associated with code clones in the project. Therefore, in this RQ, we are interested to understand the effort for maintaining a bug-prone change to a code clone.

Approach. In this section, we provide details of how we identified the bug-proneness of the code changes that are made to code clones.

[21] use CodeBert to detect the harmfulness of a code snippet. CodeBert is a pretrained bimodal for a programming language (PL) and natural language (NL) [22] which can learn a general-purpose representation of PL-NL and can be used in natural language code search and documentation. We use CodeBert to predict the bug-proneness in code changes to code clones and the following steps are involved.

(A) Data collection. For each commit where a clone sibling is changed, we extract the siblings' of the corresponding clone group. Then, we predict the bug-proneness of the code changes. To build a training dataset, we use the SZZ algorithm [45] to identify the changes that introduced bugs. The limitations of the SZZ approach are applicable to our data [77]. To assess the impact of the limitation in the SZZ algorithm, we perform a manual investigation of the precision of the SZZ algorithm. We collect a statistically significant sample of 96 commits using a confidence level of 95% and a confidence interval of 10% that are identified by the SZZ algorithm. We go through each commit one by one and identify whether the commit identified by the SZZ algorithm is associated with a bug fix commit and a bug-inducing commit. The author of the thesis and a Ph.D. student (expert in code clones) performed manual investigation. We use Cohen's Kappa [92] to access the agreement between the evaluators. Cohen's Kappa is used to measure the agreement between the evaluators

for categorical items (e.g., identifying whether bug-inducing and bug-fixing commits have related). The value of Cohen's kappa ranges from 0 to 1, one being the complete agreement between the evaluators. Our manual investigation shows that 84 out of 96 (i.e., 88%) of the commits are correctly associated with bug-fixing commits. The inter-rater agreement (Cohen's Kappa score) between the evaluators is 0.88 which represents a strong agreement.

We use the heuristic proposed by [58] that uses a regular expression to identify bug-fixing commits. The heuristic identifies the bug-ID in the commit messages. If a bug-ID appears in a commit message, we map the commit to the bug as a bug-fixing commit. Then, we extract the issue reports of each project from GitHub. For the closed issue reports, we identify if there are any pull requests associated with such issues. If there is a pull request associated with an issue, we identify all the commits included in the pull request and map the commits to the issue as a bug-fixing commit.

Once we have a list of all bug-fixing commits, we use the following command to identify all the modified files in each commit.

git log [commit-id] -n 1 –name-status

We consider only changes to Java files in a commit. For all changes to a file of a bug-fixing commit, we use the git blame command to identify all the commits when the same snippet was changed. We consider such commits as the "candidate buggy changes". We exclude the changes that are blank lines or comments. Finally, we filter the commits that are submitted before the creation date of the bugs. We then check whether the commits identified as bug-inducing commits include clone pairs. If a clone snippet is included in the bug-inducing commits, we label the clone change as "buggy".

(B) Model training and evaluation. After collecting the bug-proneness information of the code changes, we used 80% of the dataset to train the CodeBert model,[7] and 20% of the dataset to test the trained model. We use the same evaluation metric as used by [21] (i.e., accuracy) to report the performance of the trained model and AUC (Area under the ROC Curve) as an evaluation parameter.

Accuracy of the model is defined as the number of correct predictions divided by the total predictions made by the model. Accuracy is a way of assessing the performance of a model and it provides the best perspective on how well a model is performing on a given dataset.

AUC. The discriminative power of the model measures the ability of the model to distinguish value 0 and 1 of the dependant variable (i.e., predicting the probability of the code clones being buggy). The AUC value ranges from 0 to 1, 0 being the worst performance, 0.5 being the random guessing performance, and 1 being the best performance [48]. We use AUC [48] to determine the discriminative power of our model. We use Receiver Operator Curve (ROC) to plot the true positives against the false positives for different thresholds.

(C) Impact of inconsistency and type of code clone changes on bug-proneness. We perform an in-depth analysis of the impact of inconsistent changes and different types of changes classified in RQ4.2 on the bug-proneness of the code clone changes. To perform the evaluation, we have two null hypotheses: $H0_4$ and $H0_5$

$H0_4$: *there is no difference in terms of bug-proneness for making consistent and inconsistent code clone changes.*

Hypothesis $H0_4$ is two-sided and paired as we use the following data. For each of

[7] https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Defect-detection

the code clone changes in our dataset, we identify whether the code clone group is consistent or inconsistent. We use two distributions to test hypothesis $H0_4$ including (1) the bug-proneness for the consistent changes to the code clones and (2) the bug-proneness for the inconsistent changes to the code clones. We calculate the bug-proneness of the consistent and inconsistent code clone changes by checking if a code clone change is labelled as buggy or not as mentioned in the research question's approach section (data collection). To test the hypothesis, we use the paired Mann-Whitney U test [94] using a confidence level of 95%. The Mann-Whitney U test is a non-parametric test of the null hypothesis that the two distributions under the test are similar (i.e., consistent and inconsistent). A p-value < 0.05 shows that the distributions are significantly different and are not similar and hence reject the null hypothesis. We measure the percentage of bugs in the two distributions to identify which distribution has a higher percentage of bugs if they are significantly different. Each change is labelled as either bug-prone or bug-free, we identify the percentage of bug-prone or bug-free changes by dividing each of them by the total changes. We also calculate Cliff's δ [95] to determine the effect size that quantifies the difference between two distributions. A Cliff's $\delta \geq 0.474$ shows that the effect size is large [95].

$H0_5$: *there is no difference in terms of bug-proneness among different types of changes (i.e., perfective, corrective, features).*

To test hypothesis $H0_5$, we use three distributions including the bug-proneness for each of the three types (i.e., perfective, corrective, features) of code clone changes. We calculate the bug-proneness of different types of code clone changes. The bug-proneness of a code clone change is labelled as mentioned in the research question's approach section (data collection). Our goal is to test the null hypothesis that there is

no difference among the bug-proneness of the three types of code changes. To test the null hypothesis, we use a one-way ANOVA [96] test. One-way Anova is a statistical test that determines whether there are any statistically significant differences between the means of the three or more distributions (i.e., perfective, corrective, and features). A p-value < 0.05 shows that the difference among the distributions is statistically significant. We measure the percentage of bugs in the two distributions to identify which distribution has a higher percentage of bugs if they are significantly different. Each change is labelled as either of three types of code changes, we identify the percentage of each type of change by dividing each of them by the total changes.

(D) Impact of bug-prone changes on the code clone change propagation effort. We study the impact of bug-prone changes on the estimated effort to identify whether the inconsistent bug-prone changes affect the effort estimation of code clone changes. The analysis can help developers estimate the effort for propagating the code clone changes to clone siblings associated with bug-prone changes. We test the following null hypotheses ($H0_6$).

$H0_6$: *the bug-prone code clone changes and bug-free code clone changes attributes approximately similar effort to propagate code clone changes to clone siblings.*

We are interested in checking if the bug-prone and bug-free code clone changes have similar or different efforts to propagate the code clone changes to the clone siblings. To test $H0_6$, we use two distributions including (1) the effort of making bug-prone code clone changes and (2) the effort of making bug-free code clone changes. The effort for bug-prone and bug-free code clone changes is calculated using Equation 4.1 and Equation 4.2 using historical data. To test the null hypothesis ($H0_6$), we apply the Kruskal-Wallis test [88] using the 5% confidence level (i.e., p-value < 0.05). We

measure the median and average efforts of the two distributions to identify which distribution has a higher value (median and average) of effort if they are significantly different.

Results. CodeBert can predict the bug-prone code changes with a 0.71 accuracy score. After training and fine-tuning the model's parameters, we are able to achieve a 0.71 accuracy and 0.7 AUC. The results for the accuracy and AUC are the highest in the existing work based on the CodeBert model for the code clone changes. The bug-proneness detection can aid developers in determining whether a clone change should be propagated to its siblings.

Inconsistent changes have a higher probability of introducing bugs than consistent changes. We perform an in-depth analysis to identify whether there is a relationship between bugs and inconsistency. Our analysis shows that inconsistent changes are associated with 66% of bugs. However, for the consistent changes, only 42% of bugs are present. We apply the Mann-Whitney U test and achieve a p-value of $2.438e^{-04}$ (which is < 0.05). Therefore, we reject the null hypothesis ($H0_4$). The results show that the difference between the consistent and inconsistent bug-prone changes is statistically significant. We also achieve Cliff's δ of 0.754 (which is ≥ 0.474) which indicates the difference is large in terms of effect size. The results emphasize the fact that the possibility of having a bug becomes higher for inconsistent changes.

Adding new features can have a higher percentage of bug-prone changes than perfective and corrective. To investigate our results, we further compare the bug-proneness of different types of code clone changes. The percentage of the perfective changes (17%), corrective changes(12%), and features changes(23%) indicate that when new features are added, there is a high probability of introducing bugs.

We apply the one-way anova test and achieve p-value of $3.115e^{-03}$ (which is < 0.05) and we reject the null hypothesis ($H0_5$). The results show that the distribution of bug-prone code changes in different types of code changes is statistically significant.

Bug-prone changes involve more effort to propagate the code clone changes to the clone siblings than bug-free code changes. The average and median effort for bug-prone changes to code clones are 3.1 and 3.3 while the effort for bug-free changes is 2 and 2.1 respectively. The results suggest that, more effort increases the chance of introducing a bug. After running a Kruskal Wallis test, we achieve a p-value of $5.381e^{-05}$ (which is < 0.05), which indicates that the bug-prone code clone changes can have a statistically significant impact on the effort of propagating code clone changes to clone siblings. Hence, we reject the null hypothesis ($H0_6$).

72% participants agree that the bug-proneness prediction can help in the propagation of the code clone changes. Figure 4.6 shows the results of the Q6 of the survey where participants respond about the usefulness of the bug prediction for the propagation of the code clone change to clone siblings. A total of 72% of the participants agree to some level that the prediction of the bug-proneness of the code clone change is helpful. In particular, 24% participants strongly agree and 48% somewhat agree while only 6% participants answer that it would not be helpful to know the bug-proneness of the code change. Among the participants that agree to question 6, 81% of the participants have either a bachelor's, master's, or doctorate level of education and 84% of the participants have an experience of more than 5 years. To comment on the bug-proneness prediction, a participant added *"This could be used to e.g., require (additional) tests or code review on commits that have*

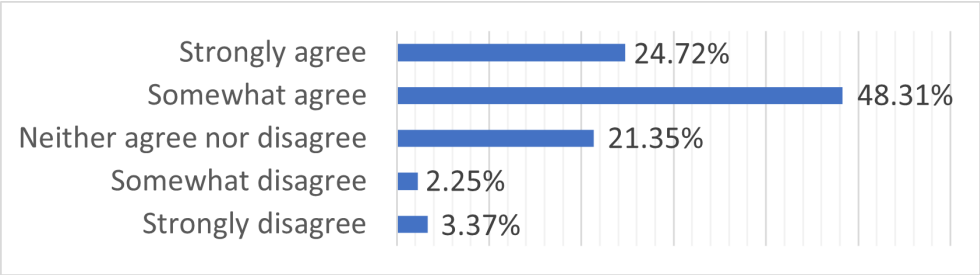


Figure 4.6: The evaluation result of (Q6): What do you think of the usefulness of providing information regarding the likelihood that changes made to cloned code may cause bugs before issue reports are submitted?

a bug likelihood above a certain threshold. I think it would take time for me to build confidence in the error likelihood predictions." while other participants commented *"the case of a bug being fixed in one case but remaining in similar cases are relevant".*

Summary of RQ4.3

We use CodeBert to predict the bug-proneness of the code clone change and achieve an AUC score of 0.7 and an accuracy of 0.71. The bug-prone code clone changes involve more effort than bug-free code clone changes for propagating the code clone changes to clone siblings. 72% of the participants of our survey agree that the prediction result of bug-proneness for the newly made change in the code clone can help in the propagation of changes to clone siblings.

4.4 Discussion

In this section, we discuss the implications of our results and we summarize and suggest to developers the following guidelines.

Developers can get an estimation of the effort needed to propagate a change in the clone group. Our approach provides developers with an effort

estimation in terms of size, churn, and complexity that is needed to propagate a change to the siblings of a clone group. The effort estimation includes the total number of modified lines of code for a specific code clone. The results can help in instances where there are multiple inconsistent code siblings.

Developers can use the change type identification approach to determine the type of code change in a commit. In RQ4.2, we classify a code clone changes based on commit messages. The FastText approach is able to label a code change as *perfective, features,* and *corrective*. Developers can use such labels to decide how to manage and prioritize a code clone change at a commit level.

The detailed information about the sub-types of perfective changes can point out the code level changes. Our results show that the most common type of changes to code clones are the *perfective* changes. We further investigated and identified 16 subtypes of code changes for the *perfective* type of changes. The classification includes changes like *statement insert, statement delete, new functionality added,* and *attribute type change*. Developers can get help in accessing the various sub-types of *perfective* changes while deciding on maintaining clone groups. The benefit of the types of changes is also shown by the results of the user survey as 73% of participants found the classification of the types of code changes very useful.

Developers can identify the location and changes of a code clone sibling. We use the results from the clone detection tools (i.e., NiCAD and iClones) and AST difference tool to find a code clone's location and code changes that would be needed from all its siblings to make them consistent. Identifying the location of inconsistent clone siblings can support developers in enhancing the maintenance of their code clones. Developers can make changes to clone siblings as well as review

the suggestions line-by-line.

Developers can identify whether a submitted code change to a code clone is buggy or not. We use CodeBert to figure out whether a change to a code clone is buggy or not. Pointing out the bug-proneness of code clones can help developers to take an immediate decision on whether or not to propagate code changes to the siblings of a code clone. This can help in avoiding bugs and vulnerability propagation in a project.

Our approach can support developers to decide whether to propagate a change of a clone to its clone group based on all the available information. 79% of the participants of the survey think that the information provided by the approach can help in making the propagation decision for code clone changes. For question 8 in our survey, we asked participants about the usefulness of our approach when making a decision for propagation. Only 21% of participants think that the information is not sufficient while 79% of the participants think that the information can be used effectively while propagation changes to clone siblings. In question 10, we ask participants about the comprehensive information provided by the approach on a scale of 1-10. 30% of the participants gave a score between 7-10 while 52% of the participants gave a score of 5 and 6. One interesting comment from one of the participant states *"My concern is that busy/uneducated/tired developer will act just the same as the "dumb" git rebase does - click Ok every time and think no bugs will be introduced. I'd give such reports to architects/managers, not devs."* We believe that there should be a more sophisticated approach for the architect, managers, and code reviewers as developers are making multiple changes quickly at the commit level but might not make a request to merge it in the code base. The approach for the

managers, architects, and code reviewers can be studied in future research and could contribute towards the better management of the code clones.

4.5 Threats to Validity

Threats to conclusion validity concern about the relationship between the treatment and the outcome. In our case, threats to conclusion validity concern the errors that occur when processing the code clones. The accuracy of the clone detection is dependent on the used clone detection tools. To achieve high accuracy, we use iClones and NiCAD as recommended by [56] who compare multiple code clone detection tools. Therefore, we use the same settings as recommended in the above study.

Threats to external validity concern the selection of projects and the analysis methods. To mitigate the issue of our results being biased towards a particular set of projects, we use well-defined selection criteria beforehand to include large-scale open-source software projects. The projects are selected from different domains (e.g., web applications, desktop applications, and APIs) of software development. The number of clones identified for each project is different among the selected projects. We also aim to show the diversity of the selected projects to ensure that our results are not biased towards a specific type of software project.

We analyze all revisions of the projects from their creation date. A selection of projects from a different time can result in a different number of clones detected, different clone genealogies, and different values for the features identified. To overcome the issue of the subset of projects, we include the data from the first commit of the project to the latest available commit.

The SZZ algorithm is the de facto standard in identifying the bug-inducing commits and is used by existing studies [76]. In our work, the SZZ algorithm is used to identify bug-prone commits and bug-prone clone pairs. The SZZ algorithm considers that a bug is introduced before the creation of a bug report, and no commit between the bug report creation and the bug fixing commit is considered. The limitations of the SZZ algorithm are applicable to our data [77]. To mitigate the limitations of the SZZ approach, we perform a manual evaluation and the details are presented in RQ4.3. The results show that in our dataset, SZZ have acceptable accuracy and the results would not be significantly affected by the bug data collected by SZZ.

We apply FastText to identify the types of code changes using the commit messages. Previous approaches [37] have achieved an accuracy of 0.91. Our results are dependent on the accuracy of the pre-trained model of FastText. To mitigate the issue of the accuracy of FastText, we perform a qualitative study on a statistically significant random sample (with a confidence level of 95%) of 384 commit messages and find that manual labels are 93% similar to predicted labels.

Prior studies [78, 79] indicate that it is essential to identify different aliases of developers in software systems. We fix the problem of disambiguation of identity (due to multiple aliases) as follows. However, the previous approaches are valid for mailing lists and other similar datasets. In our work, we use the GitHub API to retrieve the GitHub account information of the committers. Each commit has an associated email for the committer; we can verify that every committer has a different GitHub account. Our approach is similar to the prior studies solving the disambiguation problem [80]. Similar to existing approaches, we are unable to identify whether a developer has multiple GitHub accounts.

We use CodeBert to predict the bug-proneness of code change in the code clone. The accuracy of the CodeBert is 0.7 and developers need to keep in mind the accuracy of bug prediction for the code changes in the code clone. Although, CodeBert is not able to achieve a higher accuracy but in the user survey, developers still find it useful to get some indication that a code change in the code clone can introduce the bugs in the system with an accuracy of 0.7.

Threats to internal validity concern the possibility of generalizing our results. We select open-source projects on GitHub because GitHub is the most popular platform for open-source projects, and in addition, commits, pull requests, and issue reports are readily available. Moreover, we select Java projects because they are popular in open-source software development [1] [3]. However, our study can be extended to software projects from other programming languages hosted on different platforms, and other clone detection tools can be used to detect the clones in the projects.

4.6 Summary

In this study, we provide an approach that estimates the effort to propagate code clone changes to the inconsistent siblings of a clone group, identify types of code changes in the code clones, and predict bug-proneness of code changes in the code clones. We also study the impact of different change types and bug-proneness on propagating the code clone changes to clone siblings. Our work can be summarized as follows.

- Our effort prediction models can assist developers to identify the effort needed to propagate code clone changes to the clone siblings.
- We classify commit messages as *perfective*, *features*, and *corrective* and provide

effort estimation for each of the different types of code clone changes.

- We provide the effort of making bug-prone and bug-free code clone changes to propagate the code clone changes to clone siblings.
- To assist the developers, we pinpoint the locations of inconsistent siblings and suggest changes to keep code clone siblings consistent.
- The usefulness of our study is demonstrated by a user survey among the GitHub developers. In the user survey, 79% of the participants found our approach useful to manage the propagation of code clone changes to the siblings of the clone group.

We attempt to provide all the details needed to replicate our study. [8]

[8] https://zenodo.org/record/7769057

Chapter 5

Is Late Propagation a Harmful Code Clone Evolutionary Pattern? An Empirical Study

In this chapter, we present the extension study that includes more subject systems and machine learning models to predict the late propagation in code clones. Section 5.1 describes the problem and motivation of our study. Section 5.2 presents the study setup of our work. Section 5.3 discusses the results for the three research questions in our study. Section 5.4 emphasize on the results and how they can be used by developers. Section 5.5 provides the threats to validity for our study. Finally, Section 5.6 summarizes our study and provide directions for the future work.

5.1 Problem and Motivation

In a previous study on clone genealogies, Kim *et al.* [97] define two types of evolutionary changes that can affect a clone pair: a consistent change or an inconsistent change. During a consistent change, both clones in a clone pair are modified in parallel, preserving the clone pair. In an inconsistent change, one or both of the clones evolves independently, destroying the clone pair relationship. Inconsistent changes

can occur deliberately, such as when code is copied and pasted and then subsequently modified to fit the new context. For example, if a driver is required for a new printer model, a developer could copy the driver code from an older printer model and then modify it. Inconsistent changes can also occur accidentally. A developer may be unaware of a clone pair and cause an inconsistency by changing only one half of the clone pair. This inconsistency could cause a software bug. If a bug is found in one clone and fixed, but not propagated to the other clone in the clone pair, the bug remains in the system. For example, a bug might be found in the old printer driver code and fixed, but the fix is not propagated to the new printer driver. For these reasons, a previous study [97] has argued that accidental inconsistent changes make code clones more prone to bugs.

Late propagation occurs when a clone pair undergoes one or more inconsistent changes followed by a re-synchronizing change [98]. The re-synchronization of the code clones indicates that the gap in consistency is accidental. Since accidental inconsistencies are considered risky [99], the presence of late propagation in clone genealogies can be an indicator of risky, bug-prone code.

Many studies have been performed on the evolution of clones. A few (e.g., [98,99]) have studied late propagation and indicated that late propagation genealogies are more bug-prone than other clone genealogies. Thummalapenta *et al.* began the initial work in examining the characteristics of late propagation. The authors measured the delay between an inconsistent change and a re-synchronizing change and related the delay to software bugs. In our chapter, we examine more characteristics of late propagation to determine if only a subset of late propagation genealogies are at risk of bugs. Developers are interested in identifying which clones are most at risk of bugs.

Our goal is to support developers in their allocation of limited code testing and review resources towards the most risky late propagation genealogies. To achieve this goal, we first study the prevalence and bug-proneness of late propagation genealogies and secondly, we train multiple machine learning models to predict whether a clone pair would have late propagation. Early diagnosis of late propagation can help developers in addressing the clones with late propagation fast before they become buggy.

In this chapter, we replicate and extend the analysis of late propagation performed by Barbour *et al.* [3]. We study the characteristics of late propagation genealogies and estimate the likelihood of bugs. We used 10 open-source projects from GitHub instead of only two projects as in the original study. We also include an additional research question aimed at predicting occurrences of late propagation genealogies.

We address the following four research questions:

RQ5.1: Are there different types of late propagation? Late propagation has been defined by several researchers [98, 100] as an inconsistent change followed by a re-synchronizing change. We perform an exploratory study to examine several late propagation patterns and investigate whether inconsistent clone pairs ever re-synchronize without a propagation occurring. The results show that LP1, LP7, and LP8 are the most commonly occurring types of late propagations.

RQ5.2: Are some types of late propagation more bug-prone than others? Previous researchers have determined that late propagation is more prone to bugs than other clone genealogies [98]. Using the classification of late propagation clone genealogies, we evaluate late propagation in greater depth and examine if the risk of bugs remains consistent across all types of late propagation. The results show that the most commonly occurring late propagation types (i.e., LP7 and LP8) are less

bug-prone than less commonly occurring late propagation types (i.e., LP6) which are more bug-prone.

RQ5.3: Which type of late propagation experiences the highest proportion of bugs? In the previous question, we determine whether some types of late propagation are more bug-prone than others. For this question, we examine the overall number of bugs across each late propagation type, to determine which type of late propagation is responsible for the most bugs. The results show that overall LP7 and LP8 are most dangerous in terms of the proportion of bugs while the risk for other types is system-dependant.

RQ5.4: Can we predict whether a clone pair would experience late propagation? In this research question, we aim to use machine learning models to predict whether a clone pair would experience late propagation (M_{LP}) and whether the late propagation genealogy would have a bug (M_{BUG}). We use 18 features related to clone pairs and train four different machine learning models including logistic regression, SVM Classifier, random forrest, XGBOOST. The results show that our model achieves 0.80 F1-score for (M_{LP}) and 0.93 f1-score for (M_{BUG}). We also identify the most important predictors of late propagation genealogies, and bug occurrences in late propagation genealogies.

5.2 Study Setup

The goal of our study is to investigate the bug-proneness of clone pairs that undergo late propagation. The quality focus is to lower the maintenance effort and cost due to the presence of late propagated clone pairs in software systems. The perspective is that of researchers interested in studying the effects of late propagation

on clone pairs. The results may also be of interest to developers who perform development or maintenance activities. The results will provide insight in deciding which code segments are most at risk for bugs and in prioritizing the code for testing.

The context of this study consists of the change history of open-source software projects, which have different sizes and belong to different domains. This section describes the setup used to perform our study.

5.2.1 Project Selection

We use GHTorrent on the Google Cloud[1] to extract all projects that have more than 1,000 commits, 1,000 issues, and 1,000 pull requests. We use such a high number of commits, pull requests, and issues to ensure that we have enough history of clone genealogies. We limit our study to Java projects. Our selection criteria provide us with 66 Java projects. Then, we discard the projects that are younger than five years (created after June 2015). If a project has more source lines of code (SLOC), the probability of having code clones increases. A recent study suggests [1] to include projects with more than 100K source lines of code. We remove the projects with less than 100K SLOC by using the GitHub project SLOC calculator extension[2]. Furthermore, we remove the forked projects and the projects which have less than 70% of Java files. The percentage of Java files is calculated using GitHub's language information of each project. Finally, after applying all the selection criteria, we retain the top 10 projects used in this study. Table 5.1 provides the description of the selected projects. After selecting the projects, we detect the code clones using iCLONES clone detection tool and then use the same approach described in section 3.3.2 to extract

[1] https://ghtorrent.org/gcloud.html
[2] https://github.com/artem-solovev/gloc

Table 5.1: Description of selected projects for our study

Project Name	# of commits	# of issues	SLOC	% of java files	# of clone genealogies
Druid	10,496	1,657	1.2m	94.50%	61,718
Netty	9,910	4,174	476.2k	98.60%	6,576
Muikku	16,970	2,696	318.4k	50.4%	23,836
Framework	18,969	1,788	867.9k	95.50%	11,961
Checkstyle	9,454	2,198	457.4k	97.80%	7,705
Gatk	4,173	2,736	2.2m	93.70%	22,651
Realm	8,318	3,358	199.9k	83.80%	13,540
Nd4j	7,021	1,238	467.0k	99.80%	45,413
Rxjava	5,762	1,950	474.9k	99.90%	8,866
K	15,997	1,134	243.3k	83.50%	6,026

the clone genealogies.

5.2.2 Classification of Genealogies.

In the current state of the art, late propagation is defined as a clone pair that experiences one or more inconsistent changes followed by a re-synchronizing change [99]. For example, consider two clones that call a method. A developer modifies the call parameters of the method and updates one of the clones to reflect the change. This causes the clone pair to become inconsistent. Using all combinations of the inconsistent phases described by Barbour et al. [3], we identify eight possible types of late propagation (LP) genealogies. The detail of the eight types of late propagation are described in [3]. The 8 types are organized in three groups based on the occurrence or not of a change propagation : (1) propagation always occurs (three types named LP1, LP2, and LP3), (2) propagation may or may not occur (four types named LP4, LP5, LP6, and LP7), and (3) propagation never occurs (one type named LP8). In this study, we examine if the cases that always involve propagation (i.e., LP1, LP2,

and LP3) or never involve propagation (i.e., LP8) are more prone to bugs than the other types of late propagation. We made a slight modification in the definition of LP7 to include cases where during divergence either A or B is changed, instead of considering only instances in which both A and B are changed during divergence.

5.2.3 Detecting Buggy Clones.

We use the SZZ algorithm [45] to identify the changes that introduced bugs. First, we use the Fischer *et al.* [58] heuristic to identify bug fixing commits using a regular expression. The regular expression identifies the bug-ID in the commit messages. If a bug-ID appears in the commit message, we map the commit to the bug as a bug-fixing commit. Second, we mine the issue reports of each project from GitHub. For the issues that are closed, we identify if there are any pull requests associated with such issues. If there is a pull request associated with an issue, we identify all the commits included in the pull request and map the commits to the issue as a bug-fixing commit. Once we have a list of all bug-fixing commits, we use the following command to identify all the modified files in each commit.

git log [commit-id] -n 1 –name-status

We consider only changes to Java files in a commit. A commit is a set of changes to the file(s) in the software repository. For all changes to a specific file of a bug-fixing commit, we use the git blame command to identify all the commits when the same snippet was changed. We consider such commits as the "candidate buggy changes". We exclude the changes that are blank lines or comments.

Finally, we filter the commits that are submitted before the creation date of the bugs. We then check whether the commits identified as bug-inducing commits include

Table 5.2: Summary of late propagation types for the studied open-source projects

Propagation Category	Type	Projects										Sum	%
		Druid	Netty	Muikku	Framework	Checkstyle	Gatk	Realm	Nd4j	RxJava	K		
Propagation Always Occurs	LP1	102	15	78	46	3	102	39	74	10	21	490	13%
	LP2	23	-	4	5	5	3	3	5	-	4	52	1.5%
	LP3	195	22	-	24	14	11	57	67	3	58	451	12%
Propagation May or May Not Occur	LP4	18	5	10	7	6	1	4	15	1	6	73	2%
	LP5	49	3	-	3	10	3	19	24	3	6	120	3%
	LP6	207	12	-	12	10	7	28	76	6	19	377	10%
	LP7	714	62	-	29	81	21	143	297	78	102	1,527	40.5%
Propagation Never Occurs	LP8	102	18	210	29	3	195	28	50	2	53	690	18%
Total LPs		1,410	137	302	155	132	343	321	608	103	269	3,780	100%

clone pairs. If a clone snippet is included in the bug-inducing commits, we label the clone change as "buggy".

5.3 Study Results

This section reports and discusses the results of our study.

5.3.1 RQ5.1: Are there different types of late propagation?

Motivation. This question is preliminary to questions RQ5.2 and RQ5.3. It provides quantitative data on the percentages with which different types of late propagation occur in our studied systems.

Approach. We address this question by classifying all instances of late propagation as described in Section 5.2.2. For each type of late propagation, we report the number of occurrences in the systems. Table 5.2 lists each of the categories and the proportion of occurrences in our dataset, both as a numerical value and a percentage of the overall number of late propagation instances for the systems.

Results. As summarized in Table 5.2, four types of late propagation are dominant

across all systems when using the iClones clone detection tool (i.e., LP1, LP3, LP7, and LP8). The four dominant types represent the three late propagation categories. Only LP3 (instead of LP6) is more dominant as compared to the results of Barbour *et al.* [3]. As shown in Table 5.2, LP7 occurs in an average of 40.5% of instances of late propagation, so it is the most common form of late propagation across all systems. However, LP7 is also the least understood of the types of late propagation. Since both clones in LP7 clone pairs can be modified during all three steps of late propagation (i.e., experiencing a diverging change, a change during the period of divergence, a re-synchronizing change), it is unclear in which direction changes are propagated during the evolution of the clone pair. A few types of late propagation (i.e., LP2, LP4, and LP5) contribute minutely to the number of late propagation genealogies. Other than the one project (Muikku), all the other projects include almost all types of late propagation. Our further investigation shows that only 1% (297 out of 23,836) of the clone genealogies experience late propagation which is the lowest among all the projects and this might be the reason for the absence of half of LP types.

Overall, we conclude that there is representation from multiple types of late propagation and across all categories of late propagation. In the next two research questions, we examine the types in more detail to determine if some types are more risky than others.

Summary of RQ5.1

Late propagation types LP1, LP3, LP7, and LP8 are the most commonly occurring type of late propagation in the 10 studied open-source projects from GitHub. The results are consistent with the previous study except that LP3 is more frequent instead of LP6. Most of the projects include all types of late propagations.

5.3.2 RQ5.2: Are some types of late propagation more bug-prone than others?

Motivation. Previous researchers have determined that late propagation is more prone to bugs than other clone genealogies [98]. Using the classification of late propagation clone genealogies proposed by Barbour et al. [3], we evaluate late propagation in greater depth and examine if the risk of bugs remains consistent across all types of late propagation.

Approach. We compute the number of bug-containing and bug-free genealogies in each late propagation category. We compute the same values for non-late propagation clone genealogies that experience at least one change. For the remainder of this chapter, we use the abbreviation "Non-LP" for clone pairs that experience at least one change but are not involved in any type of late propagation. We test the following null hypothesis[3]

H_{02}: *Each type of late propagation genealogy has the same proportion of clone pairs that experience a bug fix.*

[3]There is no H_{01} because RQ5.1 is exploratory.

Table 5.3: Contingency table, chi Square tests results for clone genealogies with and without late propagation. The table shows the values for all the combinations of late propagations and bugs.

LP - bugs	**LP - no bugs**	**no LP - bugs**	**no LP - no bugs**	p**-value**	**OR**
1,851	1,929	42,526	48,928	0.05	1.8

We use the Chi-square test [101] and compute the odds ratio (OR) [101]. The Chi-square test is a statistical test used to determine if there are non-random associations between two categorical variables. The odds ratio indicates the likelihood of an event to occur. It is defined as the ratio of the odds p of an event (i.e., bug fixing change) occurring in one sample (i.e., experimental group), to the odds q of the event occurring in the other sample (i.e., control group): OR $= \frac{p/(1-p)}{q/(1-q)}$. An $OR = 1$ indicates that the event is equally likely in both samples; an $OR > 1$ shows that the event is more likely in the experimental group while an $OR < 1$ indicates that it is more likely in the control group. Specifically, we compute two sets of odds ratios. First, we select the clone pairs that underwent a late propagation as an experimental group. Second, we form one experimental group for each type LP_i of late propagation and re-compute the odds ratios. In both cases, we select the non-LP genealogies as the control group.

Results. Previous researchers [99] have studied the relationship between late propagation and bugs. In this research question, we first replicate the earlier studies and then extend the study to include the different categories of late propagation.

(a) Bug-proneness of late propagation. Table 5.3 summarizes the results of the tests described above for instances of late propagation compared to non-late propagation (LP) genealogies. The first and second columns show the number of LP-genealogies with and without bugs. The third and fourth columns in the table list the number of non-LP genealogies that experience bug fixes and the number that is free of bug fixes. The last column of the table lists the odds ratio test results

Table 5.4: Contingency table, chi square tests results for clone genealogies with and without late propagation.

Projects	LP - bugs	LP - no bugs	% of buggy LPs
Druid	970	440	67%
Netty	1	136	0.5%
Muikku	145	157	48%
Framework	3	152	2%
Checkstyle	78	54	60%
Gatk	134	209	39%
Realm	135	186	42%
Nd4j	283	325	47%
Rxjava	0	103	0%
K	102	167	38%

for each system. All of our results pass the Chi-square test with a p-value less than 0.05 and are therefore significant. Where there are few data points, we use Fisher's exact test to confirm the results from the Chi-Square test. The Fisher's exact test is more accurate than the Chi-Square test when sample sizes are small [101]. In this study, the Fisher test provides the same information as the Chi-Square test, so we do not present the Fisher test results in the tables. Table 5.4 shows the percentage of bug-prone late propagation in each of the studied projects. In all the significant cases, the odds ratio is greater than 1, indicating that late propagation genealogies are more bug-prone than non-LP genealogies. Overall, our results agree with previous studies [99] that found that late propagation is more at risk of bugs.

(b) Bug-proneness of late propagation types. We repeat the previous tests, dividing the instances of late propagation into their respective late propagation types. We compare each type of late propagation to genealogies with no late propagation. For each type of late propagation, Table 5.5 lists the number of instances that experience a bug fix, the number of instances with a no-bug fix, the result from the Chi-Square test, and the odds ratio using the control group composed of non-LP genealogies.

Table 5.5: Contingency table with the chi square test for different late propagation types.

Propagation Category	LP Type	Bugs	No Bugs	p-value	OR
	No LP	42526	48928	< **0.01**	1
Propagation Always Occurs	LP1	244	246	< **0.01**	3.953
	LP2	20	32	< **0.01**	0.672
	LP3	224	227	< **0.01**	0.922
Propagation May or May Not Occur	LP4	23	50	< **0.01**	2.256
	LP5	68	52	< **0.01**	1.765
	LP6	216	161	< **0.01**	6.179
	LP7	803	724	< **0.01**	1.277
Propagation Never Occurs	LP8	253	437	< **0.01**	3.2

An examination of the significant cases in Tables 5.5 reveals that the odds ratios are greater than 1, so each type of late propagation is more bug-prone than non-LP genealogies. There are two exceptions to this observation, LP2 and LP3 in Table 5.5. All exceptions belong to the 'propagation always occurs' category. Thus, in general, these late propagation types are not more bug-prone than non-LP genealogies. Our observation is consistent with the previous findings by Barbour *et al.* [3].

We conclude that there are many types that make up a small proportion of LP instances and have a very high odds ratio. Thus, when one of these LP types occurs, the risk of bug introduction is high. For example, LP6 has a high odds ratio (e.g., 6.17 in Table 5.5) but accounts for less than 5% of all late propagation instances in Table 5.2.

The two most common late propagation types in the previous research question, LP7 and LP8, in general, have low odds ratios in Table 5.5. This indicates that although they occur frequently, they are less bug-prone than other less common late propagation types (e.g., LP6). The result is consistent with the previous findings by Barbour *et al.* [3]. Overall, each type of late propagation has a different level of

Table 5.6: Proportion of bugs for each type of late propagation.

Propagation Category	**LP Type**	**# of Bugs**	**% of Bugs**
Propagation Always Occurs	LP1	244	13.2%
	LP2	20	1.1%
	LP3	224	12 %
Propagation May or May Not Occur	LP4	23	1.2%
	LP5	68	3.7%
	LP6	216	11.7%
	LP7	803	43.3%
Propagation Never Occurs	LP8	253	13.8%
	TOTAL	1851	100.00%

bug-proneness. Thus, we reject H_{02} in general.

Summary of RQ5.2

The most commonly occurring late propagation types (i.e., LP7 and LP8) are less bug-prone than the less commonly occurring late propagation (i.e., LP6). The result is consistent with the previous study and shows that each propagation type is different from others.

5.3.3 RQ5.3: Which type of late propagation experiences the highest proportion of bugs?

Motivation. In the previous research question (i.e., RQ5.2), we identify the bug-proneness of late propagation types as compared to the no-LP clone pairs. The results show that bug-proneness is not related to the frequency of LP type. In this research question, we want to identify which type of late propagation experiences the highest proportion of bugs. In other words, we examine if, when bugs occur, do they occur in large numbers?

Approach. We test the following null hypothesis

H_{03}: *Different types of late propagation have the same proportion of clone pairs that experience a bug fix.*

For each type of late propagation, we calculate the sum of all bugs experienced by instances of that type of late propagation. We use the non-parametric Kruskal Wallis test to investigate if the number of bugs for the different types of late propagation is identical.

Results. Table 5.6 presents the distribution of bugs for different types of late propagation. The 'Total' row represents the total numbers of bugs over all late propagation genealogies. To validate the results, we perform the non-parametric Kruskal Wallis test which compares the distribution of bugs between groups of different types of late propagation. The results of the Kruskal Wallis test is statistically significant with a p-value of 2.89^{-15}. Hence, there is a statistically significant difference between the distribution of bugs across all types of late propagations.

Examining the results in Table 5.6 for the significant cases, we see that in general, LP7 and LP8 contribute to a large proportion of the bugs. In the previous question, LP7 and LP8 have lower odds ratios. Although they are less prone to bugs, when they do experience bugs, the bugs are likely to occur in large numbers. The change causing the inconsistency may lead to bugs in the system, which may explain why the change is reverted instead of being propagated to the other clone in the clone pair. Overall, we can conclude that types LP7 and LP8 are the most dangerous. The level of bug-proneness of the other types is system-dependant. The proportion of bugs for each type of late propagation are therefore very different. Thus, we reject H_{03}. This result is consistent with the findings of Barbour *et al.* [3].

Summary of RQ5.3

In terms of the proportion of bugs, LP7 and LP8 are more risky and should be monitored carefully and-or refactored if possible. The risk for the other types of late propagation is system-dependant.

5.3.4 RQ5.4: Can we predict whether a clone pair would experience late propagation?

Motivation. In this research question, we use machine learning algorithms to train models that can help developers predict which clone pair will experience late propagation and have bugs in the future. Using these predictions, developers would be able to refactor risky clone pair early on and-or keep them in check before the clone pair become inconsistent or a bug is introduced. This information about risky clone pairs will help developers in making better use of their time and resources.

Approach. For each instance in the clone pair genealogy, we calculate multiple features that may help with training the models for predicting whether a clone pair would experience late propagation or not. The features are used in a prior study by Barbour *et al.* [1]. Table 6.1 presents the description of our collected features.

We train models for two different behaviours; (1) presence of late propagation (M_{LP}) and (2) bug-prone late propagation (M_{BUG}). For every change experienced by a clone pair, we calculate 18 features as described in Table 6.1. We also examined the bug-proneness of the clone pairs, as described in Section 5.2.

We use logistic regression, SVM classifier, Random Forrest, and XGBOOST to classify the clone pairs data. Logistic regression is a statistical model that uses a logistic function to model a binary dependant variable. Support vector machine

Table 5.7: Description of clone genealogies features from [1] used to build the models

Metric	Description
Product Metrics	
CLOC	The number of cloned lines of code.
CPathDepth	The number of common folders within the project directory structure.
CCurSt	The current state of the clone pair (consistent or inconsistent).
CommiterExp	The experience of committer (i.e., the number of previous commits submitted before a specific commit.)
Process Metrics	
EFltDens	The number of bug fix modifications to the clone pair since it was created divided by the total number of commits that modified the clone pair.
TChurn	The sum of added and the changed lines of code in the history of a clone.
TPC	The total number of changes in the history of a clone.
NumOfBursts	The number of change bursts on a clone. A change burst is a sequence of consecutive changes with a maximum distance of one day between the changes.
SLBurst	The number of consecutive changes in the last change burst on a clone.
CFltRate	The number of bug-prone modifications to the clone pair divided by the total number of commits that modified the clone pair.
Genealogy Metrics	
EConChg	The number of consistent changes experienced by the clone pair.
EIncChg	The number of inconsistent changes experienced by the clone pair.
EConStChg	The number of consistent change of state within the clone pair genealogy.
EIncStChg	The number of inconsistent change of state within the clone pair genealogy.
EFltConStChg	The number of re-synchronizing changes (i.e., *RESYNC*) that were a bug fix.
EFltIncSChg	The number of diverging changes (i.e., *DIV*) that were a bug fix.
EChgTimeInt	The time interval in days since the previous change to the clone pair.

(SVM) is a supervised model associated with learning algorithms that analyze data for classification. Random forrest is an ensemble learning method for classification that operates by constructing several decision trees. XGBOOST [49] is an optimized gradient boosting library designed to be highly efficient and flexible. Recent studies [102] [51] have used XGBOOST for training the models for classification problems. We split the data into training (70%) and testing (30%) to train and test the models.

Table 5.8: Evaluation metrics for the machine learning algorithms

ML algorithm	$\mathbf{M}_{LP}$			$\mathbf{M}_{BUG}$		
	Precision	**F1-score**	**AUC**	**Precision**	**F1-score**	**AUC**
Logistic Regression	0.81	0.68	0.76	0.78	0.71	0.75
SVM Classifier	0.87	0.72	0.80	0.78	0.72	0.76
Random Forrest	0.89	**0.80**	0.85	**0.94**	**0.93**	**0.93**
XGBOOST	**0.91**	0.72	**0.87**	0.91	0.75	0.90

We make sure that our data-splitting is time consistent i.e., we do not use future late propagations data to predict past late propagations.

Results. Table 5.8 shows the results of model training using the four machine learning algorithms. We evaluate the models using three performance metrics commonly used for assessing trained machine learning models; including precision, f1-score, and AUC. Precision is the fraction of relevant instances among the retrieved instances. F1-score is the harmonic mean between precision and recall. AUC provides an aggregate measure of performance across all possible classification thresholds. Results show that XGBOOST outperforms all the algorithms in terms of precision and AUC. However, Random Forrest achieves the highest value among the four models.

Furthermore, we analyze the most important predictors for both behaviours (i.e., late propagation occurrence and bug occurrence in late propagation). For M_{LP}, the number of consistent state changes (EConStChg)(37.5%), the number of consistent changes (EConChg)(32%), and the sum of added or changed lines (Tchurn)(23.2%) are the most significant features having more than 90% effect in the model. The number of consistent state changes (EConStChg) has a negative effect; meaning that if a genealogy experience more inconsistent changes than consistent changes, then it can be an indicator of late propagation introduction in clone genealogies. For M_{BUG}, number of bug-prone modifications in the history (CFltRate)(65%), number of previous commits by a specific developer (CommitterExp)(17%), and time interval in

days since last change (EChgTime)(8%) are the most significant features having more than 90% effect in the model. The number of buggy changes divided by the number of changes (CFltRate) has a positive effect. A higher number of erroneous changes in clone genealogy history is an indicator of future bug occurrences. Experience has a negative effect, which suggests that late propagation genealogies changed by less experienced developers are more bug-prone. Developers can benefit from these results as they can leverage the trained machine learning models to assess the risks of the clone pairs.

Summary of RQ5.4

For M_{LP}, XGBOOST achieves the highest precision (0.91) and AUC (0.87) with consistent state changes (EConStChg) being the most significant feature. For M_{BUG}, Random Forrest achieves the highest precision (0.94) and AUC (0.93) with the number of past bug-fixing changes (CFltRate) being the most significant feature.

5.4 Discussion

In this study, we provide the developers with the information about different types of late propagation code clones that exists in software systems. We also aid the developers with the information about the most frequent late propagation types and the most bug-prone late propagation types. Developers can use this information to keep an eye on the most frequent and bug-prone late propagation types and maintain the code clones appropriately. In addition, we have also employed machine learning approaches to predict which of the clone pair can experience a late propagation in

future. We also provide the developers with the prediction of the bug-prone late propagation in the code clones. Our results can help the developers reduce the cost of maintenance for the code clones and save time by making early changes in the code clones.

5.5 Threats to Validity

We now discuss the threats to the validity of our study.

Construct validity threats in this study are mainly due to measurement errors possibly introduced by our chosen clone detection tool. To reduce the possibility of misclassification of code fragment as clones, we chose the best configuration for clone detection tool that has been recommended by the recent evaluation of code clone tools [56]. The clone detection tool in this study can detect identical (i.e., type 1), near-identical clones (i.e., type 2), and near-miss clones (i.e., a type 3 clone). The addition or deletion of a line of code to one clone segment and not the other is an inconsistent change. The iClones tool is able to detect these changes as well. Another construct validity threat stems from the SZZ heuristics used to identify bug-fixing changes [45]. The results of this study are dependent on the accuracy of the results from SZZ. Although this heuristic does not achieve a 100% accuracy, it has been successfully employed and reported to achieve good results in multiple studies [103].

Threats to internal validity do not affect this study, as it is an exploratory study. Although we cannot claim causation, we do identify, in RQ5.2 and RQ5.3, a relation between late propagation and bug-proneness for clone pair genealogies. Furthermore, we have provided some qualitative explanation of our results based on the inspection of the source code of our studied systems.

Conclusion validity threats concern the relation between the treatment and the outcome. We pay attention to the assumptions of the statistical tests. Also, we mainly use non-parametric tests that do not require the normality of the distribution of the data.

Threats to external validity concern the possibility of generalizing our results. We examine 10 Java systems, all of them are different in size and belong to different domains. The results are validated on more systems instead of only two as in the previous study.

5.6 Summary

In this chapter, we replicate a previous study by Barbour *et al.* [3] to examine late propagation in more detail. We first confirm the conclusion from the previous study that late propagation is more risky than other clone genealogies. We then identify eight types of late propagation and study them in detail to identify which types of late propagation contribute the most to bugs in the systems. Overall, we find that two types of late propagation (i.e., LP7 and LP8) are riskier than the others, in terms of their bug-proneness and the magnitude of their contribution towards bugs. LP7 occurs when both clones are modified, causing a divergence and then at least one of the two clones in the pair is modified to re-synchronize the clone pair. LP8 involves no propagation at all and occurs when a clone diverges and then re-synchronizes itself without changes to the other clone in a clone pair. The contribution of other types of late propagation is found to be system-dependent. From this study, we can conclude that late propagation types are not equally risky. We train machine learning models to identify the clone genealogies with late propagation (M_{LP}) and

bug-prone late propagations (M_{BUG}) early on. We use 18 different clone genealogy-related features to train four different machine learning models. For the occurrence of late propagation (M_{LP}), XGBOOST achieves the highest precision (0.91) and AUC (0.87) with consistent state changes (EConStChg) being the most significant feature. For the bug-prone late propagations (M_{BUG}), Random Forrest achieves the highest precision (0.94) and AUC (0.93) with the number of bug-prone changes (CFltRate) being the most significant feature.

We attempt to provide all the details needed to replicate our study. [4]

[4] https://zenodo.org/record/7769033

Chapter 6

Predicting the change propagation of code clones at a pull request level

In this chapter, we present our study on predicting the change propagation of code clones at a pull request level. Section 6.1 describes the problem and motivation of our study. Section 6.2 presents the study setup of our work. Section 6.3 discusses the results for the three research questions in our study. Section 6.4 emphasizes on the results and how they can be used by developers. Section 6.5 provides the threats to the validity of our study. Finally, Section 6.6 summarizes our study and provides directions for future work.

6.1 Problem and Motivation

A software system may contain approximately 21% of its code in the form of code clones [4]. Once a code clone is created, it usually undergoes many changes over time [44]. Prior studies show that making inconsistent changes to the code clones can potentially lead to bugs and higher maintenance costs [104] [105] [44] [23]. Existing research has provided various approaches to analyze code evolution [3] [106], predict

bug fixing time in cloned code [23] and the inconsistent change propagation [44]. For example, prior research [23] has conducted the prediction of the bug resolution time by providing a binary result (e.g., longer or shorter bug resolution time). The lifetime of the bug is defined as the start date when the bug is reported and the end date when the bug is fixed. However, a more specific estimation for the lifetime of a bug in a clone is not provided. Mondal *et al.* [44] study the propagation of code clone changes to siblings dependent on the state of a clone group (i.e., consistent state or inconsistent state). The change propagation could be determined by considering the benefit to lower maintenance costs and reduce potential bugs. Moreover, the existing approaches are conducted at the commit-request level.

Existing studies [107] [108] indicate that the code clones can be introduced for experiment purposes and could be eventually removed from the code base before the developer submits the changes for review in the form of a pull request. The code submitted at the pull request level usually consists of multiple finalized commits. The changes to code clones could be short-lived at the commit level as the code clone changes could be removed before submitting a pull request. Therefore, code clone analysis in the pull request level can be more stable than the individual commit level as the code is to be approved to merge into the code base [109]. More specifically, the pull requests are reviewed by code reviewers and could be rejected and sent back to the developer for several reasons, such as the existence of clone instances, poor quality of code chunk, and lack of cohesion [110] [111] [112]. In our dataset, we find that 72% of the rejected pull requests that contain code clones have at least one inconsistent code clone group. To reduce the likelihood of pull request rejection from a code clone perspective, developers must conduct a code clone analysis before submitting a pull

request. However, the current research focuses on code clone analysis at a commit level which may produce obsolete information as inconsistent changes might be fixed before submitting a pull request.

In this chapter, we strive for providing an automated approach that can assist the developers in conducting clone analysis before submitting a pull request in order to better maintain the code clones. More specifically, we analyze the clones in the code snapshot before submitting the pull request, estimate the riskiness of clones, and give developers earlier warnings about the bugs in the clone code. We apply machine learning approaches to provide developers with rich information regarding the clones in the pull requests: (1) the degree of consistency that describes a clone group experiencing no inconsistency, a single inconsistency, and multiple inconsistencies in the evolution process. The degrees of consistency of a clone group could indicate the amount of attention needed from developers. No inconsistency in a clone group could require minimal effort from the developers while the developers should be more alert to the clone groups experiencing multiple inconsistent changes; (2) the lifetime of bugs in clones is measured by the number of days and the number of releases to indicate the priority of a bug to fix. For example, a long-lived bug could be more challenging to fix it fast before submitting the pull request; and (3) the likelihood of propagating code changes to the clone siblings which can be used to identify the potential riskiness of clone groups. Developers can focus more on the highly risky clone groups in order to reduce the number of clone groups that a developer needs to track for change propagation. Such predictions allow developers to examine the potential issues caused by code clones and obtain estimates for defect fixing before the code review.

We use 32 open-source software projects from GitHub to extract code clones and train machine-learning models. We evaluate the performance of the machine learning models and address the following research questions. Moreover, we conduct empirical studies on the characteristics of clones appearing in the pull requests.

RQ6.1. How well can we predict the degree of code clone consistency in a pull request? In this research question, we predict whether a clone group would experience an inconsistent change in the future. We use multi-class classification to predict the inconsistent changes and the results show that XGBOOST achieves the best evaluation metrics score (i.e., 0.86 F-score) for prediction. Moreover, we find that the churn of code changes and previously experienced inconsistent changes are the most important features in the prediction of the degree of inconsistency in code clone groups.

RQ6.2. How well can we predict the lifetime of a bug at a pull request level? In this research question, we aim at predicting the lifetime of a bug from two perspectives: 1) the number of days of bug resolution, and 2) the number of releases that a bug remains in the software system. We use a regression machine learning algorithm and the results show that lightGBM can achieve the best R-squared value among all algorithms for predicting the number of days for bug survival and the number of releases for bug survival. Additionally, our results show that inconsistent changes have more impact on the duration of bug resolution than consistent changes.

RQ6.3. How well can we predict whether a code clone change in a pull request should be propagated to the siblings? In this research question, we predict the clone propagation after a change has been made in one or more siblings of the clone group. The prediction of clone propagation can identify potentially

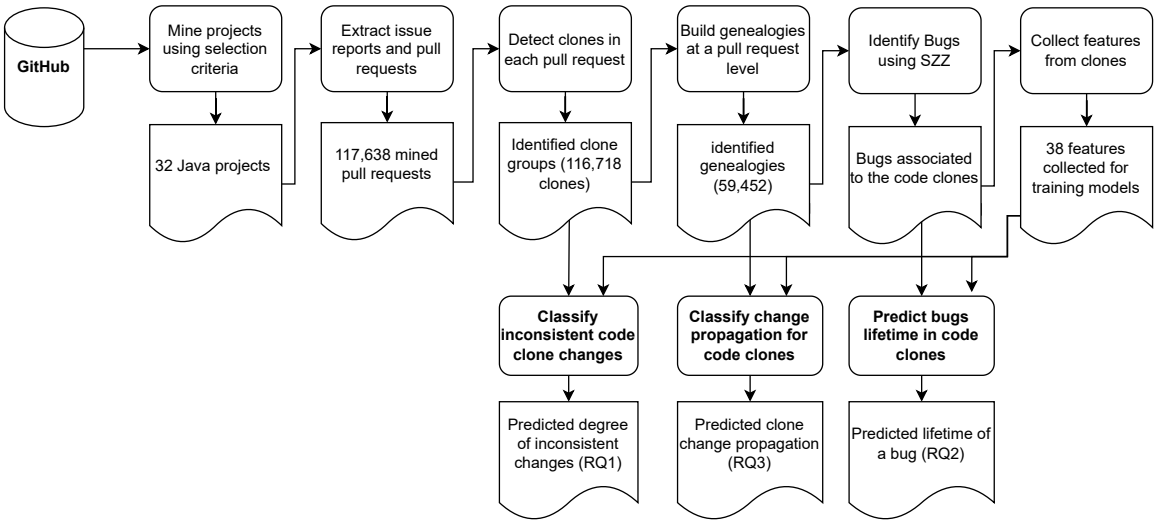


Figure 6.1: Overview of our approach

risky clone groups. By focusing more on such clone groups, developers can reduce the number of clone groups that need propagation. Our results of machine learning algorithms show that Random Forest is able to achieve the best AUC (0.88 F1-score) after validating the results with 10-fold cross-validation. Our results show that active siblings, the contribution level of the pull request submitter, and the size of the code change are significant features for clone propagation prediction. Finally, the comparison of the duration of bug resolution and clone propagation suggests that clone groups predicted to perform a propagation are more prone to contain bugs with a longer lifetime than the clone groups that are not predicted to be propagated.

6.2 Study Setup

This section describes the experiment setup. Figure 6.1 presents the overview of our approach. We start with the project selection and mine the selected projects. Then we run a clone detection tool, called iClones, to identify code clones, and build

genealogies. Next, we extract issue reports and pull requests for each project. We further calculate features for the extracted code clones to create a dataset that can be used for training the prediction models. The steps are described in detail in the following sections.

6.2.1 Selecting Projects

We use GHTorrent on the Google cloud[1] to extract all projects that have more than 1,000 issues, and 1,000 pull requests. We use such a high number of issues and pull requests to ensure that we have sufficient history of clone genealogies. We focus on Java programming language because code clone studies often use Java projects to experiment and evaluate code clone approaches [1] [3]. Our selection criteria provide us with 66 Java programming language-based projects. Then, we discard the projects that are younger than seven years (i.e., created after June 2015). A recent study on clone genealogies [1] suggests including projects with more than 100K source lines of code (SLOC). We remove the projects with less than 100K SLOC using the GitHub project SLOC calculator extension[2]. Furthermore, we remove the forked projects and the projects with less than 70% of source code files in Java. The percentage of source code files is calculated using the language information for each project in GitHub. Language information provides details about the different number of files present in the project related to each programming language. After applying all the selection criteria, we obtain 32 Java projects that are used in this study. The details of the selected projects are presented in Table 7 in the appendix. We use a similar approach presented in Section 3.3 to detect code clones, build clone genealogies, and identify

[1] https://ghtorrent.org/gcloud.html
[2] https://github.com/artem-solovev/gloc

deleted or renamed files for all the selected Java projects.

6.2.2 Extracting Pull Requests

The selected Java projects are all Git-based projects. Git provides multiple functions to extract the history of the projects. The history includes the renamed files, changed files, and changes made to each file. A pull request is a change(s) submitted by a developer for code review before the changed code can be merged with the main code base. Often a pull request is reviewed by a code reviewer and includes discussion among the stakeholders before a pull request is merged into the main code base. A single pull request has information about the developer who has submitted the pull request, associated commits in the pull request, the date when a pull request is submitted and merged, the title of the pull request, description of the pull request, comments in the pull request, and the status of the pull request. Figure 6.2 shows an example of a pull request in GitHub.

After downloading the repositories, we extract the pull request information using the GitHub REST API[3]. The API provides the following information related to a pull request.

- **Title.** The name of a pull request.
- **Description.** The details of the changes made in the pull requests, such as the bug information that might include what issues are resolved, how it is resolved and any other relevant details.
- **Commits.** One or more code changes (commits) associated with the pull requests.

[3] https://docs.github.com/en/rest/pulls

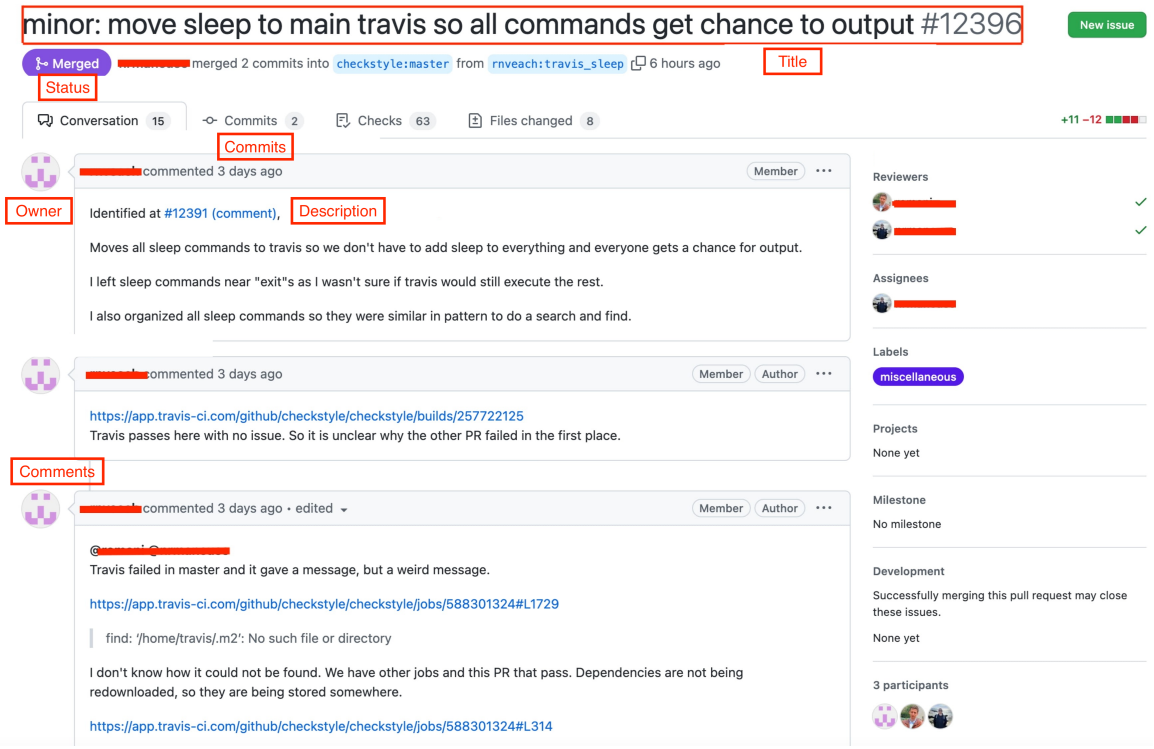


Figure 6.2: Annotated screenshot of a pull request in GitHub

- **Owner.** The developer who has submitted the pull request.
- **Comments.** The discussion between the developer, the code reviewer and other team members to review the code changes before the code changes are merged into the main code base.
- **Status.** The pull request can have three statuses: i.e., open, close and merged. The open status indicates that the pull request is under review. The close status indicates that the pull request is reviewed by not merged. The merged status indicates that the pull request is reviewed and merged to the main code base.

- **Bug Information.** If a bug is fixed by the pull request, the specific bug ID is provided.

6.2.3 Extracting Bug Information

To identify the code clone changes that have introduced a bug, we use SZZ algorithm [45]. We use the heuristics provided by Fischer *et al.* [58] to identify the bug-ID associated with the change and extract the bug-fixing code clones in the project. Then, we extract the issues that are fixed by a specific pull request as mentioned above. After identifying all the bug-fixing changes, we use the following command to extract the files changed in each pull request.

git log [pull-request-id] -n 1 –name-status

We use the following git command to identify the pull request when the same snippet is changed.

git blame -L startLineNumber,endLineNumber filePath

In this way, we can identify the code clones that are changed before the issue report is generated and we label them as bug-inducing code clone changes. We collect the pull requests and the bug information for every project in our study.

Herbold *et al.* [77] show that SZZ can identify the wrong labels for the bugs. Therefore, the limitations of the SZZ approach are applicable to our data as well. To assess the impact of the limitation in the SZZ algorithm, we perform a manual investigation of the precision of the SZZ algorithm. We collect a statistically significant sample of 96 commits using a confidence level of 95% and a confidence interval of 10% that are identified by the SZZ algorithm. We go through each commit one by one and identify whether the commit identified by the SZZ algorithm is associated with

a bug-fix commit and a bug-inducing commit. The first author of the chapter and a Ph.D. student who is an expert in code clones performed a manual investigation. We use Cohen's Kappa [92] to access the agreement between the evaluators. Cohen's Kappa is used to measure the agreement between the evaluators for categorical items (e.g., identifying whether bug-inducing and bug-fixing commits have related). The value of Cohen's kappa ranges from 0 to 1, one being the complete agreement between the evaluators. Our manual investigation shows that 84 out of 96 (i.e., 88%) of the commits are correctly associated with bug-fixing commits. The inter-rater agreement (Cohen's Kappa score) between the evaluators is 0.88 which represents a strong agreement.

6.2.4 Collecting Features

For each instance in a clone group genealogy, we extract multiple features as listed in Table 6.1.

The features are divided into five categories: product, process, genealogy, user metrics, and code change at a pull request level. Each feature captures a characteristic of clones and the pull request data.

Removing Correlated and Redundant Features. We remove the redundant features to avoid the possibility of having correlated features interfering in the interpretation of our models [59]. We use the Spearman rank correlation test with a cut-off value of 0.7 to identify the redundant features [60]. To construct the hierarchical overview of inter-feature relationships, we run the varclus function from the R package Hmisc [61]. The results indicate that: (1) clone age in days (*CAgeDays*) and clone age in commits (*CAgeCommits*) are correlated and (2) submitted pull requests

Table 6.1: The calculated features for the clone group genealogies

Metric	Description
	Product Metrics
CLOC	The number of cloned lines of code [1].
CPathDepth	The number of common folders within a project directory structure [1].
CCurSt	The current state of a clone group (consistent or inconsistent) [1].
FChgFreq	The average of the file changes experienced by a clone group.
CAgePulls	The clone age in terms of the number of pull requests.
CAgeDays	The clone age in terms of the number of days.
CSib	The number of siblings of a clone group at any specific pull request.
McCabe	The measure of McCabe cyclomatic complexity of the code clone change at the pull request level.
StrucMet	The numerical measure of the structuredness of a code clone in a change at the pull request level.
UnqMet	The total number of unique method calls in a code clone in a change at the pull request level.
	Process Metrics
EFltDens	The number of bug fix modifications to a clone group since the creation of the clone group divided by the total number of changes that modified the clone group [1].
TChurn	The sum of added and the changed lines of code in the history of a clone.
TPC	The total number of changes in the history of a clone [1].
NumOfBursts	The number of change bursts on a clone. A change burst is a sequence of consecutive changes with a maximum distance of one day between the changes [1].
SLBurst	The number of consecutive changes in the last change burst on a clone [1].
CFltRate	The number of bug-prone modifications to a clone group divided by the total number of changes that modified the clone group [1].
TotalCom	The total number of commits in a pull request.
	Genealogy Metrics
EConChg	The number of consistent changes experienced by a clone group [1].
EIncChg	The number of inconsistent changes experienced by a clone group [1].
EConStChg	The number of consistent changes of state within a clone group genealogy [1].
EIncStChg	The number of inconsistent changes of state within a clone group genealogy [1].
EFltConStChg	The number of re-synchronizing changes (i.e., *RESYNC*) that are a bug fix [1].
EFltIncSChg	The number of diverging changes (i.e., *DIV*) that are a bug fix [1].
EChgTimeInt	The time interval in days since the previous change to the clone group [1].
UUsers	The number of unique contributors in a clone genealogy.
	User Metrics
CommiterExp	The experience of the contributor (i.e., the number of previous changes submitted before a specific pull request.)
NFixPR	The number of pull requests submitted by a specific contributor.
NRejPR	The number of pull requests rejected for a specific contributor.
NAccPR	The number of pull requests accepted for a specific contributor.
Contributor	Core, if the number of changes by a specific contributor is higher than the average changes by contributors over the past months, otherwise, Casual.
CCurFile	The total number of changes to the current file by a specific contributor.
OChgRatio	The number of changes in the genealogy by a specific contributor.
	Code Change Metrics
ActSib	True, if the change is made to the most frequently changed sibling in a clone group at the pull request level, otherwise, false.
DecLines	The total number of declarative source code lines of code clone in a change at the pull request level.
ExcLines	The total number of executable source code lines of code clone in a change at the pull request level.
SynCon	The total number of syntactic constructs in the code clone in a change at the pull request level.
TotalFil	The total number of files changed in a pull request.

(*NFixPR*), accepted pull requests (*NAccPR*), and rejected pull requests (*NRejPR) are correlated*. Therefore, we retained *CAgeCommits* and *NFixPR*, and removed the rest of the correlated features from our model training step. We also perform the redundancy analysis after the completion of the correlation analysis using redun of rms R package. Redundant metrics are not able to aggregate values for the models and can be explained by other metrics. Our analysis shows that there is no redundant metric in our dataset.

6.3 Study Results

6.3.1 RQ6.1: How well can we predict the degree of code clone consistency in a pull request?

Motivation. Developers can make code changes during the maintenance phase of the project and then submit the code changes in the form of pull requests to merge the updated code into the code base. While performing the code changes, developers may not be aware that some siblings need to be co-changed. As a result, the clone group becomes inconsistent. Prior studies [104] [105] have shown that code clone inconsistency within a clone group can often lead to the risk of bug introduction and higher maintenance costs of clone siblings. Developers tend to keep clone groups consistent when it is possible. Existing studies [29] [27] [23] perform prediction of clone inconsistency at the commit level and contain more granular information, such as adding a comment and testing functionality on code changes. Such information at the commit level might not be useful for developers as the information might be related to transient code changes that are no longer needed to be merged into the code base at the pull request level. The code changes in a pull request can be more stable as developers would submit only clean and ready-to-merge code for review. We aim to predict the degree of inconsistency of a clone group as the output of our machine-learning model. We strive for assisting developers in taking proactive measures to identify the inconsistent clone groups before a pull request is submitted by a developer. It is desirable for developers to have consistent clone siblings for future clone maintenance. If a clone group is likely to experience multiple inconsistencies, it could potentially introduce bugs to the code and therefore can increase the effort for code clone maintenance. Using the prediction approach, developers can be notified

of early signs of changed code clones and understand their probability of becoming inconsistent and take proper action to prevent bugs from happening.

Approach. In this section, we describe the approach used to predict the degree of inconsistency of the code clones associated with pull requests in the following steps.

Step1: Labelling the Data. A clone group experiences many changes over its lifetime. A code change to the clone group can change the state of the clone group from consistent to inconsistent and vice versa. The states of the clone groups are identified using the clone detection tool (i.e., iClones). If the results of the clone detectors identify that all the siblings of the clone group are changed together, it indicates that the clone group is in a consistent state. If one or more siblings are not changed and are no longer in a cloning relationship with the rest of the siblings then the clone group is in an inconsistent state. There can be multiple state changes during the lifetime of the clone groups. For example, for one of the clone groups in our dataset for the gatk[4] project, there are 12 changes made to a clone group. The clone group started at commit baf741b[5]. The first two changes are consistent followed by three inconsistent changes before a single consistent change is performed. The rest of the changes to the clone group are inconsistent. Due to the existence of multiple state changes in a clone group, we consider each change individually at the pull request level in our study. In our dataset, there are a total of 23% clone groups with no inconsistency, 29% of the clone groups with a single inconsistency, and 48% of the clone groups with more than two inconsistencies. The distribution of the number of inconsistencies is shown in Figure 6.3. The majority of clone groups experience one or two inconsistent changes. Based on the distribution of the number

[4] https://github.com/broadinstitute/gatk
[5] https://github.com/broadinstitute/gatk/commit/baf741b

of inconsistent changes experienced by clone groups during the lifetime, we assign the following three labels to each of the code clone changes at the pull request level to create a labelled dataset.

- **No Inconsistency.** If a clone group experience no inconsistency during the lifetime, it is labelled as a clone group with no inconsistency. The clone groups with no inconsistency would be easy to maintain and have a lower chance of bug introduction.

- **Single Inconsistency.** If a clone group undergoes only one inconsistent change, it is labelled as a clone group with a single inconsistency. The clone groups that would have only a single inconsistency need to be maintained while other changes still achieve a consistent state.

- **Multiple Inconsistencies.** If a clone group encounters at least two or more inconsistent changes, it is labelled as a clone group with multiple consistencies. The clone groups with two or more inconsistencies can increase maintenance costs and introduce bugs that are challenging to resolve.

Step 2: Model Training and Evaluation. We use multi-class classification to train the machine learning models with the classification task of three classes (i.e., no inconsistency, single inconsistency, and multiple inconsistencies). From a modeling perspective, the classification requires a training dataset in which each data point is labelled with one of the classes showing the degree of inconsistencies and has the values of the features as listed in Table 6.1. Finally, we train and evaluate the classification models for predicting the inconsistency of the clone groups. The details of the machine learning model selection and evaluation are discussed as follows.

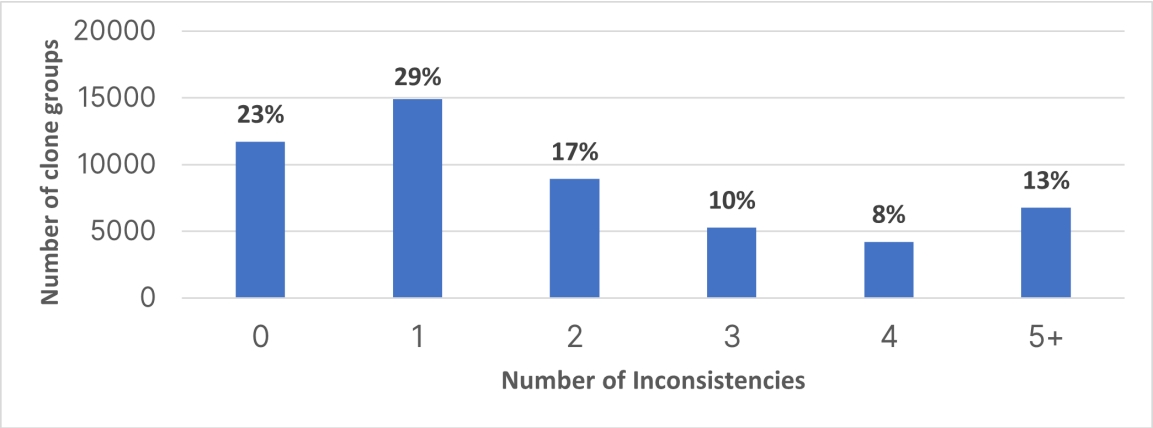


Figure 6.3: Distribution of the number of inconsistencies in a clone group in our dataset.

(a) Model Selection. We select the following commonly used multi-class classification machine learning models/frameworks.

- Naive Bayes is a supervised machine learning algorithm that uses the Bayes algorithm. Bayes algorithm works on the principle of finding the probability of the degree of inconsistency classes given the values of the features.

- Logistic Regression is a statistical analysis model used to predict the outcome of the previous observations of data. In our case, we use the values of the features to predict the degree of inconsistency.

- Random Forest is an ensemble learning method that constructs multiple decision trees to train for predicting the classes of the degree of inconsistency.

- XGBOOST is a scalable machine learning system for gradient tree boosting proposed by Chen and Guestrin [49]. XGBOOST combines the original model (i.e., gradient boosting) with weak base learning models in an iterative manner to generate a robust learning model. The residual in each iteration of the

boosting is used to improve the previous predictor (i.e., optimizing the loss function).

- LightGBM is a gradient boosting framework that uses tree-based algorithms [53]. The most distinctive feature of LightGBM among other tree-based algorithms is that it grows trees vertically. LightGBM is highly effective for large-scale data. LightGBM supports training on GPU, but it is sensitive to the size of datasets; there is a risk of overfitting if a relatively smaller dataset is used.

(b) Model Evaluation. For performing the model evaluation, we use the evaluation metrics mentioned in Section 3.2.3.

(c) Model Validation

Cross-validation is a resampling technique used to evaluate machine learning models on limited data samples. Essentially, there are multiple folds (known as k-folds) where the k value is the number of folds. We choose the value of k as 10. The data is randomly divided into ten equal groups where the first nine folds are used for training and validation, and the last fold is used for testing. This process is repeated ten times using different folds each time for testing. In each fold for training, data is divided into a training set (t1) and a validation set (v1). The model is trained on the data from the training set (t1) of the group and then validated within the validation set (v1) of the same group. The performance results of each fold are aggregated.

We use 10-fold cross-validation to evaluate the performance of the models. To avoid any data leakage issue [113], we create time-consistent folds and sort the data using pull request submission dates, to ensure that folds on which models are tested always contain data that is posterior to the data on which the models are trained.

Table 6.2: 10-fold cross-validation evaluation metrics results of three classification algorithms and two frameworks in terms of precision, recall, accuracy, AUC, and F1-score.

Evaluation Metrics	**Precision**	**Recall**	**Accuracy**	**AUC**	**F1-Score**
Logistic Regression	0.78	0.58	0.80	0.75	0.67
Naive Bayes	0.49	0.79	0.64	0.67	0.60
Random Forest	0.83	0.76	0.78	0.77	0.79
LightGBM	0.84	0.80	0.81	0.83	0.84
XGBOOST	**0.85**	**0.80**	**0.82**	**0.84**	**0.86**

Note that the nature of our dataset is time-sensitive which means each change in the clone group is dependent on the time of the change. In our dataset, we use the timestamp of the pull request to obtain the information when the change is made to the code clone. Using the timestamp information, we sort all the code changes in chronological order. The time ordering mechanism ensures that we do not predict past data by using future data to avoid data leakage.

Results. XGBOOST achieves the best evaluation metrics scores (i.e., F1-score of 0.86 and an AUC score of 0.84) for predicting the degree of inconsistencies of a clone group. Table 6.2 shows the results of the evaluation metrics scores for the machine learning models. The machine learning models are trained for the multi-class classification for the three classes (i.e., no inconsistency, single inconsistency, and multiple inconsistencies). XGBOOST achieves an F1-score of 0.86 and an AUC score of 0.84 which provides developers with confidence in the prediction. LightGBM and Random Forest also perform well in terms of all studied evaluation metrics. The prediction results of multi-class classification can be used to identify whether a clone group would experience an inconsistent state. The results of the multi-class classification can help developers allocate sufficient resources to manage

code clones based on their predicted degree of inconsistencies.

The churn of the code change (*TChurn*) and the number of prior inconsistent changes experienced by the clone group (*EIncChg*) are the most important features in explaining the model. Table 6.3 shows the top eight features with the highest importance scores. We use the gain importance type of XGBOOST that shows the average gain across all splits when a particular feature is used. The output of the feature importance is a score between 0 and 1 for all the features used to train the model. The output of 0 depicts that the feature is of the least importance and the output of 1 indicates that the feature is of the most importance. The feature importance score for *TChurn* is 0.62 and for *EIncChg* is 0.56. Developers can focus on these two features when reviewing the pull requests for code clone inconsistency analysis. The features can help the developers in determining the degree of inconsistencies that can be experienced by a code clone group and take actions to maintain and flag the code clone group. The complete scores for all the features are included in the appendix in Table 8.

Summary of RQ6.1

Developers can leverage our machine learning models to actively manage the potential inconsistent clones at the pull request level. XGBOOST is able to achieve an F1-score of 0.86 to predict multi-class inconsistency (i.e., no inconsistency, single inconsistency, multiple inconsistencies). The code churn and the number of previous inconsistent changes experienced by a clone group play a significant role in explaining the machine learning models.

Table 6.3: Top eight importance scores for XGBoost Model for the prediction of the degree of inconsistency, sorted by importance score descendingly.

Feature	Importance Score
TChurn	0.62
EIncChg	0.56
TotalCom	0.52
CAgeDays	0.44
EFltDens	0.39
experience	0.37
CPathDepth	0.34
ActSib	0.31

6.3.2 RQ6.2: How well can we predict the lifetime of a bug at a pull request level?

Motivation. Pull requests submitted by developers are prone to have bugs [111]. The inconsistent changes to code clones can be one of the reasons behind the bug introduced in a software project [104] [106]. The lifetime of a bug can be the duration from the time when the issue report of a bug is opened by a developer to the time when the bug is resolved. Some bugs are fixed quickly while others take time to be fixed. Developers strive for reducing the number of bugs that may remain in the system for a longer period (e.g., multiple releases). There can be multiple reasons for a long-lived bug, such as dependencies on other modules, low priority, unclear bug reports, and lack of bug-related knowledge [41] [39]. Prior studies [41] [39] [40] are unable to provide a precise prediction of the lifetime of bugs in clone groups rather predict a binary outcome (e.g., long-lived bugs or short-lived bugs). We intend to provide a more precise prediction of the lifetime of a bug in terms of the number of days or the number of releases that can be potentially induced in the clone groups before submitting a pull request. The information can help developers in prioritizing the clone groups that can be fixed first before merging the pull requests.

Approach. In this section, we present the steps in order to predict the lifetime of a bug in clone groups that are associated with pull requests as follows.

Step 1: Labelling the Data. In section 3.3, we present the steps to build the clone genealogies from the results of the clone detection tool. We treat each change in the clone genealogies as a data point in our dataset. We also provide details of extracting bug information in section 6.2. We curate our bug dataset from two different perspectives, first, the number of days, and second, the number of releases. To label the data for both models, we used the following approach.

- **Identifying the lifetime of a bug in terms of the number of days**. To create the dataset for the number of days a bug would remain in the system, we collect the bug reports of the subject systems and identify the bug reporting date and the date on which the bug report is closed. We count the number of days between the bug reporting date and the bug closing date and label the bug associated with a clone change with the calculated number of days. Finally, we have the days for all the bugs associated with the clone groups in the subject systems.

- **Identifying the lifetime of a bug in terms of the number of releases**. To calculate the number of releases that a bug has survived, we first identify all the releases for a particular project using the GitHub REST API [6]. The result of the API includes all the published releases for a particular project. Each release includes information about the date of the release. We compare the release dates with the dates of the bug reports and measure the total number of releases that happen between the creation date of a bug report and the closing

[6] https://docs.github.com/en/rest/repos/reposlist-repository-tags

date of the bug report. If there is no release available for the bug report, then we label the lifetime of the bug as 0, otherwise, we label the lifetime of the bug with the number of releases that the bug has experienced.

Step 2: Model Training and Evaluation. Regression analysis is a type of statistical method used to determine the strength of a relationship between a dependent variable (usually denoted as Y) and a number of independent variables. In contrast to only two possible outcomes in the classification problem, a regression problem can have multiple outcomes. e.g., predicting the lifetime of a bug in terms of the number of days.

(a) Model Selection. To train the machine learning models to predict the lifetime of a bug, we use a mixed-effect model [71], to take into account the context of the projects in our analysis of the lifetime of the bugs. As our projects are from different domains, we wish to understand whether the projects from different domains have significant behaviour towards the prediction. A mixed-effect model presents the significant features and consists of two types of features (i.e., explanatory features and context features). The explanatory features (i.e., process, product, genealogy, and user features in our dataset) are used to explain the dataset, while context features (i.e., project) are used to determine the effect of explanatory features. The mixed-effect model is able to show the relationship between the outcome (i.e., the lifetime of bugs in terms of the number of days or the number of releases) and the explanatory features while taking into consideration the context features (i.e., project).

Overall, we employ four regression algorithms and two frameworks. We use (1) the linear regression mixed-effect model as our baseline model as used in prior studies [66] [60] to determine the lifetime of bugs. We also use (2) ridge regression and (3) lasso

regression algorithms; which are commonly used to train regression models. Finally, we use (4) Random Forest, (5) the regressor function of XGBOOST, and (6) the regressor function of LightGBM. We use the implementation of scikit-learn [72] for the four algorithms and for the two frameworks (i.e., XGBOOST[7] and LightGBM[8]) by using their open source implementation available on GitHub.

(b) Model Evaluation. Multiple metrics have been introduced in the literature to evaluate the performance of regression models. Two of them are most commonly used by practitioners: (1) R-Squared (R^2) and (2) Root Mean Square Error (RMSE).

R-Squared (R^2): R-squared is a proportional improvement in the prediction of a regression model as compared to the mean model. The scale of R^2 ranges from zero to one, with zero indicating that the regression model is not able to improve than the mean model while one depicts that the regression model performs perfectly in terms of prediction. Equation 6.1 is used to calculate R^2.

$$R^2 = \frac{\text{Error from regression model}}{\text{Simple average model}} \tag{6.1}$$

Root Mean Square Error (RMSE): The RMSE is the square root of the variance of the residuals that indicates the absolute fit of the model. In simple words, it calculates how close are the predicted values and the actual values. Lower values of RMSE are desirable for regression models. Equation 6.2 is used to calculate RMSE.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(Predicted_i - Actual_i)^2}{N}} \tag{6.2}$$

Mean Absolute Error (MAE): Mean absolute error indicates the magnitude of the

[7] https://github.com/dmlc/xgboost
[8] https://github.com/microsoft/LightGBM

difference between the predicted observation and the actual observation. MAE takes the average of the absolute error from the group of predictions. Equation 6.3 is used to calculate MAE.

$$MAE = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n} \tag{6.3}$$

where y_i is the predicted value, x_i is the actual value and n is the total number of observations.

(c) Model Validation. We use a 10-fold cross-validation approach to evaluate the models, constructing time-consistent folds. We sort pull request data using pull request submission dates and ensure that folds on which models are tested always contained data that is posterior to the data on which the models were trained. We also ensure that validation sets always contain data that is posterior to the data on which the models are trained. These time-consistent folds help ensure that we are not predicting past data using future data. In our dataset, we use the timestamp of the pull request to make sure that all the code clone changes are sorted in chronological order. The chronological order ensures that we are not predicting past data using future data. From the results of the cross-validation approach, we can interpret whether the model is over-fitting. If the training RMSE/MAE and testing RMSE/MAE have significant differences, it indicates that the model is over-fitting.

(d) Impact of the degree of inconsistencies on the bug lifetime. We study whether the degree of inconsistencies (i.e., no inconsistency, single inconsistency, and multiple consistencies) have a different impact on the lifetime of the bugs. We test the following hypotheses.

$H0_1$: *There are no differences in the bug lifetimes in terms of the number of days for different degrees of inconsistencies in code clones.*

In this null hypothesis, we aim to examine if the bug lifetime in terms of the number of days has any relation with the degree of the inconsistencies presented in RQ1. To test $H0_1$, we collect three distributions including (1) the lifetime of a bug occurred in terms of the number of days for no inconsistency clone groups, (2) the lifetime of a bug occurred in terms of the number of days for single inconsistency clone groups, and (3) the lifetime of a bug occurred in terms of the number of days for multiple inconsistencies clone groups. To test the null hypothesis ($H0_1$), we apply the Kruskal-Wallis test [88] using the 5% confidence level (i.e., p-value < 0.05). Kruskul-Wallis is a rank-based parametric test to identify if there are statistically significant differences between two or more groups of an independent variable on the dependent variable. We measure the median and average lifetime of the bugs in terms of the number of days among the three distributions to identify if any distribution has a higher value if they are significant.

$H0_2$: *There are no differences in the bug lifetimes in terms of the number of releases for different degrees of inconsistencies in code clones.*

In this null hypothesis, we aim to examine if the lifetime of the bug in terms of the number of releases has any relation with the degree of the inconsistencies presented in RQ1. To test $H0_2$, we collect three distributions including (1) the lifetime of a bug occurred in terms of the number of releases for no inconsistency clone groups, (2) the lifetime of a bug occurred in terms of the number of releases for single inconsistency clone groups, and (3) the lifetime of a bug occurred in terms of the number of releases for multiple inconsistencies clone groups. To test the null hypothesis ($H0_2$), we apply the Kruskal-Wallis test [88] using the 5% confidence level (i.e., p-value < 0.05). We measure the median and average lifetime of the bugs in terms of the number of releases

Table 6.4: 10-fold cross-validation results of four regression algorithms and two frameworks, evaluation metrics R-squared, RMSE, MAE.

Type of Model	Evaluation Metrics	R-squared	Train RMSE	Test RMSE	Train MAE	Test MAE
Lifetime of the bugs in terms of number of days	Linear regression (mixed-effect)	0.78	23.4	24.5	42	44.2
	Ridge regression	0.79	21.2	22.6	37.2	38.9
	Lasso Regression	0.78	18.6	19.5	35.6	36.8
	Random Forest	0.86	9.5	10.3	18.6	20.3
	XGBOOST	0.86	5.2	5.8	13.8	15.1
	LightGBM	**0.87**	**4.7**	**4.95**	**11.1**	**12.3**
Lifetime of the bugs in terms of number of releases	Linear regression (mixed-effect)	0.77	24	25.7	44	42.1
	Ridge regression	0.76	22.4	23.9	38.6	39.5
	Lasso Regression	0.76	19.5	21.3	36.1	38.6
	Random Forest	0.82	10.5	11.7	19.3	21.8
	XGBOOST	0.84	5.9	**7.1**	14.8	**16.4**
	LightGBM	**0.84**	**5.8**	7.3	**14.5**	16.5

among the three distributions to identify if any distribution has a higher value if they are significant.

Results. LightGBM achieves the best evaluation metric scores (e.g., 0.87 R-squared value) when training the model for predicting the lifetime of a bug in terms of the number of days. Table 6.4 shows the results of the evaluation metrics of the regression algorithms for predicting the number of days that the bugs associated with code clones can survive. LightGBM achieves an R-squared value of 0.87, train RMSE of 4.7, test RMSE of 4.95, train MAE of 11.1 and test MAE of 12.3. Random Forest and XGBOOST also perform well than the linear regression (mixed-effect), ridge and lasso regression models as presented in Table 6.4. The results of the prediction can be used to identify the long-standing bugs in terms of the number of days.

LightGBM achieves the best evaluation metric scores when training the model for predicting the lifetime of a bug in terms of the number

of releases. Table 6.4 shows the results of the evaluation metrics of the regression algorithms for predicting the number of releases that a bug associated with code clones can survive. LightGBM achieves a R-squared value of 0.84, train RMSE of 5.8, test RMSE of 7.3, train MAE of 14.5 and test MAE of 16.5. Random Forest and XGBOOST also perform well than the linear regression (mixed-effect), ridge and lasso regression models as presented in Table 6.4. The results can be used to identify the duration of bug resolution in terms of the number of releases. Our results also indicate that for the number of days only the LightGBM model performs well among the studied machine learning algorithms while for the number of releases, LightGBM and XGBOOST are almost identical in terms of the evaluation metrics scores.

Clone groups with multiple inconsistencies are more susceptible to bugs with longer survival duration. Our results show that the average and median time for the number of survival days for clone groups with no inconsistency is 9.1 and 10 days respectively; for clone groups with single inconsistency is 11.4 and 12 days respectively; and for clone groups with multiple inconsistencies is 17.1 and 18 days respectively. The result of the Kruskal Wallis test states that we achieve a p-value of $6.5e^{-05}$(which is < 0.05) suggesting that the distributions of the lifetime of bugs in terms of the number of days for different degrees of inconsistencies are significantly different. Hence, we reject the null hypothesis ($H0_1$). For the number of releases that a bug can survive, the average and median for no inconsistency are 0.7 and 1 release respectively; for single inconsistency are 0.9 and 1 release respectively; and for multiple inconsistencies are 1.4 and 1 release. The result of the Kruskal Wallis test states that we achieve a p-value of $7.1e^{-04}$ (which is < 0.05) suggesting that the distributions of bug lifetime in releases for different degrees of inconsistencies are

Table 6.5: Top eight importance scores for LighGBM Model for the lifetime of bugs, sorted by importance score descendingly.

Lifetime of bugs in number of days		**Lifetime of bugs in number of releases**	
Feature	**Importance Score**	**Feature**	**Importance Score**
EFltDens	0.58	TChurn	0.52
EIncStChg	0.52	ActSib	0.49
age_pulls	0.47	cloc	0.46
TPC	0.72	EFltDens	0.43
EIncChg	0.39	CCurSt	0.41
cloc	0.95	EIncStChg	0.38
FCngFreq	0.32	FCngFreq	0.37
ActSib	0.3	EIncChg	0.35

significantly different. Hence, we reject the null hypothesis ($H0_2$). A clone group experiencing multiple inconsistencies tends to have bugs that live longer than a release. Such bugs in code clones could be experienced by the end users, can affect the user experience and impose more risk to the reliability of the product.

Bugs with a longer duration have more comments than bugs that are resolved faster. The average number of comments (i.e., 6.82) for the bugs that remain in the system for more than a year is higher than the average number of comments (i.e., 3.78) for the bugs that are fixed within a month. It indicates that the bugs which are getting more comments are likely to remain longer. Another interesting difference between the long (more than a year) and short-lived bugs (less than a month) is the number of participants including bug reporters, users posting comments, and the developers fixing the bugs. The average number of participants involved in long-standing bugs is 4.5 which is slightly higher than the average number of participants involved in short-lived bugs (2.8). This suggests that as more people get involved in the issue reports, it tends to take longer to fix a bug.

The number of prior bugs experienced by code clones (*EFltDens*) and

the number of prior inconsistent state changes experienced by clone groups (*EIncStChg*) are the most important features for predicting the lifetime of the bugs in terms of the number of days. For the lifetime in terms of the number of releases, the churn of code change (*TChurn*) and the clone sibling that is frequently changed (*ActSib*) are the important features for training the model. Table 6.5 shows the top eight features with the highest importance scores for the LightGBM models for predicting the lifetime of a bug in terms of both the number of days and the number of releases. We use the gain importance type of the LightGBM as mentioned in RQ1 which shows the average gain across all splits when a particular feature is used. Developers can examine the top most important features and take actions on the code clone groups that are expected to have a higher lifetime of the bugs. The complete results for the gain importance of the features are included in the appendix in Table 9.

Summary of RQ6.2

LightGBM achieves the R-squared value of 0.87 to predict the duration of bug resolution in terms of the number of days and 0.84 to predict the duration of bug resolution in terms of the number of releases. 21% of the bug reports remain open for more than a year in the studied software projects. The comparison among the different degrees of inconsistencies and bug lifetime shows that on average clone groups with multiple inconsistencies are more prone to have a longer lifetime of bugs than the clone groups with no or single inconsistency.

6.3.3 RQ6.3: How well can we predict whether a code clone change in a pull request should be propagated to the siblings?

Motivation. Clone groups can experience a change in one or more siblings and the changes do not propagate to the rest of the siblings in the clone groups. Failure to propagate inconsistent changes to clone siblings may potentially lead to bugs and can increase the maintenance costs [104] [29] [23]. Current approaches focus on the commit level information and may provide obsolete information (e.g., clones that are already fixed between different commits before a pull request is submitted). Therefore, the commit-level analysis might not be useful for understanding pull requests. Moreover, the decision for code clone propagation is not straightforward for developers. For example, a new feature can introduce a lot of code changes. Such changes might not need to be propagated as the developer wishes to experiment with the effect of the new feature before propagating the changes to other siblings. On the other hand, if a bug is fixed in a code clone change, the developer might wish to replicate it to all the siblings so that the bug can be fully resolved. In this research question, we assist developers at the pull request level by providing predictions on the propagation of the code changes in the clone groups. If clone propagation is required, the developers can fix the code clone propagation issues before submitting a pull request for review to avoid the rejection of the pull request. The predictions can greatly aid the developers in decreasing the risk of bug introduction and lowering maintenance costs by having fewer code clone groups that need propagation.

Approach. In this section, we provide an approach to predict whether the code change should be propagated to the siblings.

Step 1: Labelling the Data. In RQ6.1, we predict whether a code clone group

with a code change at a pull request level would become inconsistent. In RQ6.2, we predict bugs associated with code clones at the pull request level can survive from the perspective of the number of days and the number of releases. We use the results of RQ6.1 and RQ6.2 in predicting the propagation decision for the developers. To this extent, we add three new features to our set of features described in Section 6.2.4.

- **Degree of the inconsistency of the clone group (IncState).** The feature includes the prediction result from RQ6.1; i.e., no inconsistency, single inconsistency, and multiple inconsistencies.
- **The lifetime of the bug in terms of the number of days (DurDays).** A numerical value predicted in RQ6.2 relates to the number of days that a bug would remain in the system.
- **The lifetime of the bug in terms of the number of releases (DurReleases).** A numerical value predicted in RQ6.2 relates to the number of releases that a bug would remain in the system.

Dependant variable. Once we have all the features finalized, we aim to curate the dependent variable for the machine learning algorithms to predict. It is important to note that all the data is related to the code clone changes at the pull request level. We classify the dependent variable into two categories.

- **Propagated Category.** In a clone group, if a change happens to all the siblings, then the change is said to be propagated and the clone group remains in a consistent state. We label all such changes to clone groups as propagated.
- **Not Propagated Category.** In a clone group, if a change happens to one or more siblings but not all, such change is not propagated to the clone group and

the clone group remains in an inconsistent state. We label all such changes to clone groups as not propagated.

Step 2: Model Training and Evaluation.

We aim to predict if an inconsistent change needs to be propagated to other siblings or not. Therefore, the dependent variable for the prediction mode is binary and we apply the classification approaches to build the prediction models. We use Naive Bayes, Logistic Regression, and Random Forest algorithms to classify the clone group for the need of propagation. In addition, we also use the LightGBM and XGBOOST's classifier functions to classify the code clone groups for the need of propagation.

We evaluate the classification models trained for predicting the need of propagating the clone groups. We use precision, recall, F1 score, and AUC to evaluate the trained machine-learning models. The evaluation metrics used are described in RQ6.1. We used 10-fold cross-validation to evaluate the performance of the models. We create time-consistent folds. We sort the data using pull request submission dates and ensure that folds on which models are tested always contain data that is posterior to the data on which the models are trained. The nature of our dataset is time-sensitive which means each change in the clone group is dependent on the time of the change. The time consistency ensures that we are not predicting past data using future data.

Step 3: Impact of propagation on the bug lifetime. We aim to perform a statistical analysis to understand if a bug would survive longer in the clone groups that have inconsistent changes and the inconsistent changes are not propagated. To perform the analysis, we have two null hypotheses.

$H0_3$: *there are no differences in the lifetimes (measured in terms of the number of days) of a bug caused by the clone changes that are either propagated or not.*

Hypothesis $H0_3$ is two-sided and paired as we use two distributions to test hypothesis $H0_3$ including (1) the lifetime (measured in terms of the number of days) of a bug caused by the code clone changes that are not propagated; and (2) the lifetime (measured in terms of the number of days) of a bug caused by the code clone changes that are propagated. We calculate the lifetime of the bug as discussed in RQ6.2 and the propagation categories as presented in Step 1 of this RQ. To test the null hypothesis ($H0_3$), we apply the Kruskal-Wallis test [88] using the 5% confidence level (i.e., p-value < 0.05). We measure the average and median of the bug lifetime in terms of the number of days for the two distributions (i.e., propagated category and not propagated category).

$H0_4$: *there are no differences in the lifetimes (measured in terms of the number of releases) of a bug caused by the clone changes that are either propagated or not.*

Hypothesis $H0_4$ is two-sided and paired as we use two distributions to test hypothesis $H0_4$ including (1) the lifetime (measured in terms of the number of releases) of a bug caused by the code clone changes that are propagated; and (2) the lifetime (measured in terms of the number of releases) of a bug caused by the code clone changes that are not propagated. We calculate the lifetime of a bug as discussed in RQ6.2 and the propagation categories as presented in Step 1 of this RQ. To test the null hypothesis ($H0_4$), we apply the Kruskal-Wallis test [88] using the 5% confidence level (i.e., p-value < 0.05). We measure the average and median of bug lifetime in terms of the number of releases for the two distributions (i.e., propagated category and not propagated category).

Step 4: Identifying the significant features.

To predict the need for propagation in the clone groups, we use 37 independent variables to train the machine learning models. The description of the features extracted is mentioned in Section 6.2.4. The process of detecting code clones, building clone genealogies, extracting features, and running machine-learning models can be time-consuming. Developers might not have sufficient time to extract all 37 independent variables to build models due to pressure for the time to market of the software. We identify a subset of features that have a significant impact on the prediction of clone propagation. We train an explanatory model that is able to identify the significant features for clone propagation.

The nature of each of the studied projects might be different as they belong to different domains. Therefore, the behaviour of each of the projects is expected to be different. We use a mixed-effect model [71] to consider the context of each project separately. A number of prior studies [66] [60] [64] use mixed-effect models to identify the significant features. The goal of the mixed-effect model is to show the relationship between the explanatory features and the outcome (i.e., whether the change in the code clone would be propagated) while keeping the context of the projects in consideration.

To train the mixed-effect model, we use the R package named **lmer** [73] and use its **glmer** function. The ability of the model in predicting the outcome (i.e., whether the change should be propagated or not) is measured using the discriminative power of the model. The discriminative power of the model is calculated using the Area Under Curve (AUC). The values of the AUC range from 0 to 1 where a value of 0 indicates the worst performance, a value of 0.5 indicates the random guessing performance

Table 6.6: 10-fold cross-validation results of three classification algorithms and two frameworks, evaluation metrics precision, recall, accuracy, AUC, and F1-score.

Evaluation Metrics	Precision	Recall	Accuracy	AUC	F1-Score
Logistic Regression	0.72	0.58	0.80	0.73	0.67
Naive Bayes	0.68	0.79	0.64	0.67	0.60
LightGBM	0.81	0.78	0.80	0.81	0.84
XGBOOST	0.83	0.79	0.80	0.81	0.85
Random Forest	**0.85**	**0.83**	**0.82**	**0.84**	**0.88**

while a value of 1 indicates the best performance [48]. We use Wald statistic [74] to calculate the relative contribution (X2) that will evaluate the impact of explanatory variables. We use the R package named **Car** [75] which provides the calculation of Wald statistic using the ANOVA package. If a certain feature has a higher value of the Wald statistic's relative contribution (X2), it indicates that the feature has a higher impact on the performance of the model [60].

Results. Random Forest achieves the best F1-score (0.88) for predicting whether a clone change should be propagated to the siblings. Table 6.6 shows the 10-fold cross-validation results for the machine learning models used for the training. The evaluation metrics results show that Random Forest achieves the best results (0.88 F1-score). However, the results of lightGBM and XGBOOST are encouraging as well. The resulting models can be used to identify whether a change in a clone group should be propagated to the siblings or not.

Code clone groups with a need for propagation are more likely to have a longer bug lifetime than code clone groups with no need for propagation. We perform an in-depth analysis of the propagation prediction and the bug lifetime in terms of days and the number of releases. For the number of days, the average and median lifetime of a bug caused by clone changes in the no-propagation category

are 11.4 and 12 days, and the average and median lifetime of a bug caused by clone changes in the propagation category are 24.5 and 25 days, respectively. The result of the Kruskal Wallis test states that we achieve a p-value of $9.5e^{-05}$ (which is < 0.05), suggesting that the distributions of the bug lifetime measured in days between the propagation category and no-propagation category are significantly different. Hence, we reject the null hypothesis ($H0_3$).

For the number of releases, the average and median lifetime of a bug caused by clone changes in the no-propagation category are 0.9 and 1 release respectively, and the average and median lifetime of a bug caused by clone changes in the propagation category are 1.6 and 2 releases, respectively. The result of the Kruskal Wallis test states that we achieve a p-value of $2.8e^{-03}$ (which is < 0.05), suggesting that the distributions of bug lifetime measured in releases for propagation category and no-propagation category are significantly different. Hence, we reject the null hypothesis ($H0_4$).

Three features, that is, active siblings, multiple developers involved in changing the code clones, and the frequency of changes by a developer in the project, are significant in predicting the propagation of the code clone change. Table 3 shows the results of the mixed-effect model for the prediction of the code clone propagation when a change is made to the clone group. For active siblings (the clone siblings which are frequently changed in a clone group), we measure a Wald statistic score of 110,654 and coefficient value of $6.7e^{-03}$, which is the highest among all features, contributing to almost 34% significance in model training. The contributor level (Wald statistic value of 61,453, coefficient value of $3.4e^{-03}$, and 21% contribution) and the total content changed (Wald statistic value of 42,052,

coefficient value of $7.5e^{-02}$, and percentage contribution of 14%) are the second and third most important features. Our results can provide off-the-shelf warnings for code clone changes at the pull request level, and developers can focus more on these three features if looking for a propagation decision as they highly impact the model training.

58% of the code changes in the active siblings make the clone group inconsistent. From our analysis, we identify that when an active sibling is changed, it is likely that the clone group would become inconsistent. Our results indicate that active siblings can be used as an indicator of the clone group becoming inconsistent. Developers can flag the active siblings in the system which can help them in deciding about the propagation of the changes in clone groups.

Summary of RQ6.3

Random Forest achieves an F1-score of 0.88 to predict the necessity of propagation when not all siblings are changed within a clone group. The comparison of the lifetime of a bug caused by clone changes and propagation categories indicates that the clone groups that need propagation are likely to associate with bugs with longer lifetime. We train a mixed-effect model to show that active siblings, multiple developers changing the code clones, and the frequency of changes significantly affect the performance of the machine learning models.

6.4 Discussion

In this section, we describe the implications of our study for the developers at the pull request level by providing our results in the degree of consistency prediction,

Table 6.7: Results of the mixed-effect model for prediction of the code clone propagation, Sorted by χ^2 descendingly

Factor	Coef.	χ^2	Percentage	$Pr(<\chi^2)$	Sign.$^+$	Relationship
(Intercept)	-5.3e^{+00}	978		<2.2e^{-16}	***	↘
ActSib	6.7e^{-03}	110654	34.3	<2.2e^{-16}	***	↗
CCore	3.4e^{-03}	61453	21.2	<2.2e^{-16}	***	↗
TChurn	7.5e^{-02}	39052	12.6	<2.2e^{-16}	***	↗
IncState	3.4e^{-03}	17586	5.2	<2.2e^{-16}	***	↗
DurDays	1.9e^{-03}	15782	4.3	<2.2e^{-16}	***	↗
LvstDis	-6.4e^{-04}	9845	3.6	<2.2e^{-16}	***	↘
age_pulls	3.8e^{-02}	8431	3.2	<2.2e^{-16}	***	↗
UnqUsers	8.2e^{-03}	7952	3.1	<2.2e^{-16}	***	↗
CSib	-5.8e^{-02}	1553	2.7	<2.2e^{-16}	***	↘
TotalFil	3.9e^{-03}	1052	2.1	<2.2e^{-16}	***	↗
DurReleases	5.7e^{-04}	892	1.7	<2.2e^{-16}	***	↗
cloc	3.7e^{-04}	745	1.3	<2.2e^{-16}	***	↗
TotalCom	6.7e^{-03}	632	1.1	<2.2e^{-16}	***	↗
CAgeDays	8.4e^{-04}	514	0.9	<2.2e^{-16}	***	↗
McCabe	3.2e^{-03}	452	0.7	<2.2e^{-16}	***	↗
EFltDens	-4.6e^{-02}	423	0.7	<2.2e^{-16}	***	↗
Avg_CT	-6.1e^{-04}	395	0.6	<2.2e^{-16}	***	↘
SLBurst	2.4e^{-03}	366	0.6	<2.2e^{-16}	***	↗
experience	6.3e^{-03}	115	0.2	<2.2e^{-16}	***	↘
UChgRto	3.9e^{-02}	81	0.1	<2.2e^{-16}	***	↗
accepted	3.8e^{-05}	75	0.1	<2.2e^{-16}	***	↗
NumOfBursts	9.4e^{-02}	68	0.1	<2.2e^{-16}	***	↗
EConChg	7.3e^{-01}	55	0.09	<2.2e^{-16}	***	↗
TPC	-2.8e^{-03}	42	0.06	<2.2e^{-16}	***	↗
DecLines	5.4e^{-02}	40	0.06	<2.2e^{-16}	***	↗
FCngFreq	-4.6e^{-02}	39	0.05	<2.2e^{-16}	***	↘
ExcLines	9.7e^{-03}	33	0.04	<2.2e^{-16}	***	↗
EEvPattern	8.1e^{-02}	27	0.03	<2.2e^{-16}	***	↗
EIncStChg	5.7e^{-03}	21	0.02	<2.2e^{-16}	***	↗
EConStChg	-2.2e^{-02}	17	0.01	6.25e^{-11}	***	↗
CCurSt	5.1e^{-03}	15	0.01	4.81e^{-09}	***	↗
StrucMet	6.6e^{-02}	13	0.01	3.79e^{-08}	***	↗
NumOfBursts	6.8e^{-02}	11	0.00	3.85e^{-06}	***	↘
EChgTimeInt	-3.6e^{-04}	9	0.00	2.55e^{-05}	**	↘
SynCon	7.7e^{-04}	9	0.00	9.74e^{-04}	***	↗
UFileChg	6.7e^{-04}	8	0.00	8.54e^{-04}	***	↗
UnqMet	5.5e^{-04}	8	0.01	3.76e^{-04}	***	↗
rejected	-5.2e^{-03}	7	0.00	6.67e^{-03}	***	↗
CPathDepth	-9.3e^{-03}	6	0.00	5.24e^{-03}	***	↗
EChgTimeInt	-1.2e^{-05}	4	0.00	1.52e^{-03}	**	↘

$^+$Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

the duration of bug resolution prediction, and the clone propagation prediction. It is essential that a new code change should not introduce any new bugs or increase the maintenance cost of the software system. Our approach can help developers understand the potential survival duration of a bug associated with a code clone change. Such information allows developers to allocate the resources to fix the bugs if the bug may survive multiple releases. Lastly, developers can leverage our approach to decide if propagating code changes to all siblings is necessary. We describe the implications in more detail below.

Developers can leverage the detailed information from our approach regarding the degree of clone inconsistency prediction in the code clone changes. A developer makes the changes in the source code files and submits a pull request for the code to be merged into the code base. For example, pull request #10704 from one of the studied projects (i.e., checkstyle) contains two different code clone instances changed. There are a total of 20 files that are changed in the same pull request. It is time-consuming for the developers to identify the changed code clones and check whether all siblings are changed or not. Our approach identifies all code clones that are changed, any clone siblings that are not changed, and the clone group that can potentially experience an inconsistency, which can potentially improve the productivity of developers in managing pull requests.

Developers can be aware of the length of resolving bugs associated with the code clones to avoid long-lasting issues. An inconsistency in the clone group can lead to bugs. A bug can remain in the system for a longer period. We provide developers with the prediction of how long a bug that is related to code clones would survive in the project. We provide the prediction from two perspectives:

the number of days and the number of releases. It can greatly aid the developers in taking precautionary steps to make sure that the bugs that take longer to resolve may be addressed earlier. In particular, developers need to pay attention to the code clone bugs that might be existent across multiple releases and need to be fixed before the new release is published to avoid a negative impact on the user experience of using the released software.

Developers are suggested to be aware of and make a decision about whether a change in clone siblings can be propagated to the other siblings. We use machine learning algorithms to predict whether a change in the code clone group can be propagated to the other siblings. There are multiple scenarios while making the decision to propagate the changes. For example, if a new piece of code is added to the project, developers might wish to see its impact on the project and test it out before replicating it to the rest of the code clone siblings. However, if there is a bug fix occurred in the code clone change, the developer might wish to immediately propagate the changes to clone siblings in order to avoid the introduction of new bugs in the project. Our Random Forest model can be used by the developers to correctly identify which code clones should be propagated.

Developers can use our significant feature analysis to review a subset of features for a quick check on the changed code clone groups to assist them in deciding whether to propagate the code changes to other clone siblings. Our approach includes code clone detection, clone genealogy creation, feature calculation, and model setup that could consume time and might have associated costs. We provide a significant feature analysis that would allow the developers to have a quick peek into the subset of the features so they can decide on the propagation.

Table 6.8: A summary of previous studies which predict consistency maintenance.

Research Study	**No. of Projects**	**No. of clone instances**	**Number of ML Models**	**Analysis Level**	**Evaluation (F1-Score)**
Bettenburg *et al.* [26]	3	4,500	N/A	release	N/A
Wang *et al.* [27]	4	6,000	1	commit	0.72
Zhang *et al.* [28]	8	20,000	5	commit	0.77
Nguyen *et al.* [29]	5	5,000	1	commit	0.83
Our approach	**34**	**116,718**	**6**	**pull-request**	**0.86**

Our analysis shows active siblings, core contributors, and the size of the code change are the significant features. Developers can use the significant features in a way that if the active siblings are changed by a core contributor, it might be a good idea to propagate that change to the other siblings.

In a comparison of our work with the existing work, the size of the dataset used is low in previous studies. More specifically, the highest number of code clones used is 20,000 and the number of the projects studied in the existing work is fewer, i.e., the highest number of projects is eight. Most importantly, all of the prediction approaches are at the commit level which we have already seen are not sufficient at the pull-request level. Table 6.8 shows the detailed comparison between prior studies and our approach. Finally, the machine learning models and their evaluation scores are unable to achieve high performance. However, our approach is based on 34 Java projects, including a total of 116,718 clone instances, trained on six different machine learning models and achieves a better F1-score than existing approaches.

6.5 Threats to Validity

In this section, we discuss the threats to the validity of our approach.

Threats to conclusion validity deals with the relation between the treatment

and the outcome. In our case, the errors that occurred in the processing of the code clones are detailed in the conclusion validity. The clone detection tool determines the accuracy of the identified code clones. We use iClones as it shows higher accuracy for detecting code clones as evaluated by Savalajeko *et al.* [56]. The authors compare multiple code clone detection tools. We run the iClones with the same settings as recommended by Savalajeko *et al.* [56] in their comparison. We use the issue reports and pull requests from the GitHub API and all the data associated like creation date, closed date, comments, and associated commits are supposed to be accurate. There are limitations to the SZZ approach and we conduct a manual study among the first author and a Ph.D. student who is an expert in code clones. The inter-rater agreement (Cohen's Kappa score) between the evaluators is 0.88 which represents a strong agreement.

Threats to external validity concern the selection of projects and the analysis methods used in the chapter. We use explicit selection criteria to include large-scale open-source software projects from different domains. This helps reduce the issue of our results being biased towards a particular set of projects. The number of clones identified for each project is different among the selected projects. We intend to show the diversity of the selected projects to ensure that our results are not biased towards a specific domain.

We analyze all revisions of the projects starting from their creation date until September 2022. The code clones detected, issue reports and features calculated are accurate for this period only. A different number of clones can be detected, different clone genealogies can be built, and different values for the features can be identified if the projects from a different period are selected.

It is important to identify varying aliases used by developers in an open-source project as suggested by prior studies [78] [79], referred to as disambiguation of identity. The problem of disambiguation of identity (due to multiple aliases) is fixed in our study as follows. Previous approaches are valid for mailing lists and other similar datasets. We use the GitHub API to retrieve the GitHub account information of the committers. An email address is associated with each commit of the committer. We extract the email address of the commit and are able to verify that every committer has a different GitHub account. Prior studies also use a similar approach to solve the disambiguation problem [80]. We are unable to identify whether a developer has multiple GitHub accounts similar to existing approaches.

Threats to internal validity addresses the possibility of generalizing our results. GitHub is the most popular platform for open-source projects, and in addition, commits, pull requests, and issue reports are readily available, due to this reason we selected projects from GitHub. The iClones can achieve higher accuracy in detecting the code clones which we use it detects code clones from the selected projects. The projects from Java programming languages are popular in open-source and our selection criteria ensure that we select projects that are enriched with pull requests and issue reports. Other clone detection tools can be used to detect the clones in the projects to build on our results. Our study can be extended to software projects from other programming languages hosted on different platforms.

6.6 Summary

The existence of inconsistent code clone groups in the software system can lead to bugs and higher maintenance costs as stated by earlier studies [104] [105] [44] [23].

Therefore, a proactive decision needs to be taken by the developers at the pull request level before the code changes are merged into the code base. In the chapter, we present the machine learning approach that can predict whether a newly introduced change in a code clone group can be propagated to all siblings in the group. Our approach can help the pull request reviewers to examine the changes to code clones before merging the pull requests. Specifically, we have conducted the training of machine learning algorithms on the pull requests from 32 java projects comprising 117,638 pull requests. Our findings can be presented as follows.

- We train multi-class classification models to predict the consistency of clone groups. We predict whether a clone group would have no inconsistency, single inconsistency, or multiple inconsistencies during the lifetime of a code clone. LightGBM and XGBOOST can achieve an F1-score of 0.84 to classify the inconsistency of the clone groups.

- We train regression machine learning models to predict the survival time of a bug in the code clone group that would remain in the system in terms of the number of days and the number of releases. LightGBM achieves the best R-squared value of 0.87 for a lifetime in terms of the number of days and 0.84 for a lifetime in terms of the number of releases.

- We label the clone change data for the propagation decision, whether a newly introduce change should be propagated to all the siblings. We train classification machine learning algorithms using 37 features from code clones and pull request data. Random Forest can achieve the best F1-Score of 0.88 to predict the clone change propagation.

- Finally, we provide a significant feature analysis for the developers so they can quickly examine a subset of features at the pull request level. We train a mixed-effect machine learning model to train on the features using the studied projects as the context. Our results show that active sibling, contribution level, and size of a code change are the three most significant features that contribute to approximately 68% significance for training the models.

The future work can include adding software projects from different programming languages and different domains. Our approach can be made available as a tool for different integrated development environments (IDEs) for developers to use in their daily activities.

We attempt to provide all the details needed to replicate our study. [9]

[9] https://zenodo.org/record/7765625

Chapter 7

Conclusion

In this chapter, we summarize our work and provide future opportunities for our work. In this dissertation, we provide intelligent approaches for code clones that may be fault-prone, risky, harmful, inconsistent, and contain propagation issues. To achieve this, we address the problem from four different perspectives; (1) ranking code clones for bug proneness, (2) providing a clone propagation recommendation system for developers at the commit level, (3) predicting the late propagation in code clones and their bug-proneness, and (4)predicting the clone consistency maintenance for clone siblings at the pull request level.

7.1 Contributions

The main contributions of this thesis are as follows.

- **Propose a machine learning approach for automatically ranking bug-prone code clones (Chapter 3).** We identify features of code clones from the evolution history of the code clones to rank the code clones at the commit level. We use the learning-to-rank machine learning approach to rank the code

clones. We also use the classification approach to identify the clones that could be bugy in a commit and then rank them. Finally, we use regression approaches to rank the code clones based on the percentage of the code clone that remains buggy in their lifetime and finds the significant features as well.

Summary of Chapter 3

Our results show that the learning-to-rank approaches can rank the code clones with an evaluation score of 0.71. However, classification and regression approaches can achieve better evaluation scores. Changes by multiple developers and the age of the code clones show significance with the bug-proneness of the code clones.

- **A machine learning approach to estimate the effort for propagating code change in a clone group (Chapter 4).** We propose an approach that would allow developers to estimate efforts for propagating code clone changes in the clone siblings while a change has been made to at least one of the siblings. We predict the effort based on multiple metrics for propagating changes to clone siblings. We use NLP based machine learning approach to categorize the code change using commit text. We use the classification machine learning approach to predict the bug-proneness of the code change. Finally, we perform a user study among GitHub developer to identify the usefuleness of our approach.

Summary of Chapter 4

We provide the effort estimation model as well that can predict how much effort is needed to propagate the changes to siblings. Our results show that NLP approach is able to classify between perfective, corrective, and feature changes with the most common type being perfective. CodeBert is able to achieve AUC of 0.71 when predicting the bug-proneness in the changed code snippet. Lastly, the results of the survey shows that 79% of the participants agree that the proposed approach is useful to estimate the effort for propagating code clone changes.

- **Prediction approach for identifying the late propagation pattern and its harmfulness (Chapter 5).** In this work, we extend an existing study [1] by including a dataset of 10 large open-source software projects. We use state-of-the-art machine learning algorithms to predict whether a clone group would undergo a late propagation and if a late propagation happens, would it be harmful to the system or not.

Summary of Chapter 5

Our machine learning approaches achieve AUC of 0.87 while predicting whether a clone group would experience a late propagation while an AUC of 0.91 is measured for predicting the bug-proneness of the code clones experiencing the late propagations.

- **A machine learning approach to predict clone propagation at a pull request level (Chapter 6).** We provide a prediction mechanism for developers

to know in advance about the inconsistent clone groups. We also provide an approach to predict the long-standing bugs in the code clones concerning the number of days and number of releases. Finally, we present a machine learning approach that is able to predict whether a clone propagation should occur in the clone group.

Summary of Chapter 6

Our results show that we are able to predict the inconsistency of the clone groups with the AUC of 0.83 while we can identify the long-standing bugs and are able to describe the length in the number of days and number of releases with the r-squared value of 0.81. Finally, we are able to predict the clone propagation decision for developers with an AUC of 0.85.

7.2 Future Work

The results of this work assist the developers in maintaining the code clones that could reduce the maintenance cost and the number of bugs experienced by the software system due to code clones. The future work can build on the results accumulated in this dissertation and explore multiple perspectives in the code clone research.

Tagging active clone siblings. In Chapter 6, we identify a feature *active sibling*, which corresponds to the siblings that change more often with the clone groups. Future research can build a mechanism where the active siblings can be predicted and tagged. So, if any change is made in future, the developer would be notified. This would be a quick turnaround for the developer to have a quick look at if the changes are consistent due to the change in the active sibling.

Cross-language clone maintenance. Software projects with a multi-language development environment may contain clones that are expanded over two or three programming languages. Overlooking the clones in any particular languages can be harmful to software maintenance. Future work can propose an approach to analyze cross-language clones within the same software project. Maintainers of multi-lingual software projects can use the approach to better maintain the cross-language clones.

Refactoring suggestions for cloned code. Prior studies suggest that developers should improve the quality of the code clones rather than removing them. The code smells in the code clones can be fixed by applying appropriate refactoring approaches. Future work can propose an approach that can predict risky clones earlier in the life cycle of software projects that are related to the bad smells. Software maintainers can leverage such information to make the clones consistent and to maintain the quality of the code fragments.

A tool for IDE integrating all approaches. Developers use integrated development environments (IDEs) to develop software system. It would be helpful for developers if for every commit and pull requests, they are able to see the code clone related reports in the IDE. The approaches that we propose in this thesis can be integrated to provide the reports for each of the approaches.

Adding type-4 code clones in the analysis. In all of the proposed approaches, we have have used three types of code clones. Future research can focus on including the type-4 code clones as well. We will explore the feasibility of applying the proposed approaches in this thesis to type-4 code clones. An empirical analysis can be conducted to compare the results of type-4 code clones with the rest.

Bibliography

[1] L. Barbour, L. An, F. Khomh, Y. Zou, and S. Wang, "An investigation of the fault-proneness of clone evolutionary patterns," Software Quality Journal, vol. 26, no. 4, pp. 1187–1222, 2018.

[2] A. Kumar, R. Yadav, and K. Kumar, "A systematic review of semantic clone detection techniques in software systems," in IOP Conference Series: Materials Science and Engineering, vol. 1022, no. 1. IOP Publishing, 2021, p. 012074.

[3] L. Barbour, F. Khomh, and Y. Zou, "Late propagation in software clones," in 2011 27th IEEE International Conference on Software Maintenance (ICSM). IEEE, 2011, pp. 273–282.

[4] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in 2009 IEEE 31st International Conference on Software Engineering. IEEE, 2009, pp. 485–495.

[5] B. van Bladel and S. Demeyer, "A novel approach for detecting type-iv clones in test code," in 2019 IEEE 13th International Workshop on Software Clones (IWSC). IEEE, 2019, pp. 8–12.

[6] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in 29th International Conference on Software Engineering (ICSE'07). IEEE, 2007, pp. 96–105.

[7] M. Mondal, C. K. Roy, and K. A. Schneider, "Bug propagation through code cloning: An empirical study," in 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2017, pp. 227–237.

[8] N. Göde and R. Koschke, "Incremental clone detection," in 2009 13th european conference on software maintenance and reengineering. IEEE, 2009, pp. 219–228.

[9] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," IEEE Transactions on Software Engineering, vol. 28, no. 7, pp. 654–670, 2002.

[10] J. R. Cordy and C. K. Roy, "The nicad clone detector," in 2011 IEEE 19th International Conference on Program Comprehension. IEEE, 2011, pp. 219–220.

[11] N. Schwarz, M. Lungu, and R. Robbes, "On how often code is cloned across repositories," in Proceedings of the 34th International Conference on Software Engineering. IEEE Press, 2012, pp. 1289–1292.

[12] R. K. Saha, C. K. Roy, and K. A. Schneider, "An automatic framework for extracting and classifying near-miss clone genealogies," in 2011 27th IEEE International Conference on Software Maintenance (ICSM). IEEE, 2011, pp. 293–302.

[13] X. Yang, K. Tang, and X. Yao, "A learning-to-rank approach to software defect prediction," IEEE Transactions on Reliability, vol. 64, no. 1, pp. 234–246, 2014.

[14] C. K. Roy, "Detection and analysis of near-miss software clones," in 2009 IEEE International Conference on Software Maintenance. IEEE, 2009, pp. 447–450.

[15] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in 2015 IEEE international conference on software maintenance and evolution (ICSME). IEEE, 2015, pp. 131–140.

[16] S. Xie, F. Khomh, and Y. Zou, "An empirical study of the fault-proneness of clone mutation and clone migration," in 2013 10th Working Conference on Mining Software Repositories (MSR). IEEE, 2013, pp. 149–158.

[17] J. Li and M. D. Ernst, "Cbcd: Cloned buggy code detector," in 2012 34th International Conference on Software Engineering (ICSE). IEEE, 2012, pp. 310–320.

[18] R. K. Saha, C. K. Roy, K. A. Schneider, and D. E. Perry, "Understanding the evolution of type-3 clones: an exploratory study," in Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE Press, 2013, pp. 139–148.

[19] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim, "Reducing features to improve code change-based bug prediction," IEEE Transactions on Software Engineering, vol. 39, no. 4, pp. 552–569, 2012.

[20] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A developer centered bug prediction model," IEEE Transactions on Software Engineering, vol. 44, no. 1, pp. 5–24, 2017.

[21] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in Advances in Neural Information Processing Systems, 2019, pp. 10 197–10 207.

[22] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang et al., "Codebert: A pre-trained model for programming and natural languages," arXiv preprint arXiv:2002.08155, 2020.

[23] F. Zhang, S.-c. Khoo, and X. Su, "Predicting change consistency in a clone group," Journal of Systems and Software, vol. 134, pp. 105–119, 2017.

[24] P. Thongtanunam, W. Shang, and A. E. Hassan, "Will this clone be short-lived? towards a better understanding of the characteristics of short-lived clones," Empirical Software Engineering, vol. 24, no. 2, pp. 937–972, 2019.

[25] R. Garg and R. Tekchandani, "An approach to rank code clones for efficient clone management," in 2014 International Conference on Advances in Electronics Computers and Communications. IEEE, 2014, pp. 1–5.

[26] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An empirical study on inconsistent changes to code clones at the release level," Science of Computer Programming, vol. 77, no. 6, pp. 760–776, 2012.

[27] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei, "Predicting consistency-maintenance requirement of code clonesat copy-and-paste time," IEEE Transactions on Software Engineering, vol. 40, no. 8, pp. 773–794, 2014.

[28] F. Zhang and S.-c. Khoo, "An empirical study on clone consistency prediction based on machine learning," Information and Software Technology, vol. 136, p. 106573, 2021.

[29] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Clone management for evolving software," IEEE transactions on software engineering, vol. 38, no. 5, pp. 1008–1026, 2011.

[30] O. Fedotova, L. Teixeira, H. Alvelos et al., "Software effort estimation with multiple linear regression: Review and practical application." J. Inf. Sci. Eng., vol. 29, no. 5, pp. 925–945, 2013.

[31] F. Qi, X.-Y. Jing, X. Zhu, X. Xie, B. Xu, and S. Ying, "Software effort estimation based on open source projects: Case study of github," Information and Software Technology, vol. 92, pp. 145–157, 2017.

[32] M. F. Zibran and C. K. Roy, "Conflict-aware optimal scheduling of prioritised code clone refactoring," IET software, vol. 7, no. 3, pp. 167–186, 2013.

[33] E. Shihab, Y. Kamei, B. Adams, and A. E. Hassan, "Is lines of code a good measure of effort in effort-aware models?" Information and Software Technology, vol. 55, no. 11, pp. 1981–1993, 2013.

[34] A. Sabetta and M. Bezzi, "A practical approach to the automatic classification of security-relevant commits," in 2018 IEEE International conference on software maintenance and evolution (ICSME). IEEE, 2018, pp. 579–582.

[35] S. Levin and A. Yehudai, "Boosting automatic commit classification into maintenance activities by utilizing source code changes," in Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering, 2017, pp. 97–106.

[36] A. Mauczka, F. Brosch, C. Schanes, and T. Grechenig, "Dataset of developer-labeled commit messages," in 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. IEEE, 2015, pp. 490–493.

[37] G. E. dos Santos and E. Figueiredo, "Commit classification using natural language processing: Experiments over labeled datasets." in CIbSE, 2020, pp. 110–123.

[38] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," arXiv preprint arXiv:1607.01759, 2016.

[39] L. A. F. Gomes, R. da Silva Torres, and M. L. Côrtes, "On the prediction of long-lived bugs: An analysis and comparative study using floss projects," Information and Software Technology, vol. 132, p. 106508, 2021.

[40] M. Habayeb, S. S. Murtaza, A. Miranskyy, and A. B. Bener, "On the use of hidden markov model to predict the time to fix bugs," IEEE Transactions on Software Engineering, vol. 44, no. 12, pp. 1224–1244, 2017.

[41] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering, 2010, pp. 52–56.

[42] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007). IEEE, 2007, pp. 1–1.

[43] M. Fowler, Refactoring: improving the design of existing code. Addison-Wesley Professional, 2018.

[44] M. Mondal, C. K. Roy, and K. A. Schneider, "Does cloned code increase maintenance effort?" in 2017 IEEE 11th International Workshop on Software Clones (IWSC). IEEE, 2017, pp. 1–7.

[45] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" ACM sigsoft software engineering notes, vol. 30, no. 4, pp. 1–5, 2005.

[46] H. Li, "Learning to rank for information retrieval and natural language processing," Synthesis Lectures on Human Language Technologies, vol. 7, no. 3, pp. 1–121, 2014.

[47] C. Goutte and E. Gaussier, "A probabilistic interpretation of precision, recall and f-score, with implication for evaluation," in European conference on information retrieval. Springer, 2005, pp. 345–359.

[48] J. A. Hanley and B. J. McNeil, "The meaning and use of the area under a receiver operating characteristic (ROC) curve." Radiology, vol. 143, no. 1, pp. 29–36, 1982.

[49] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining, 2016, pp. 785–794.

[50] J. Walthers, "Learning to rank for cross-device identification," in 2015 IEEE International Conference on Data Mining Workshop (ICDMW). IEEE, 2015, pp. 1710–1712.

[51] S. S. Dhaliwal, A.-A. Nahid, and R. Abbas, "Effective intrusion detection system using xgboost," Information, vol. 9, no. 7, p. 149, 2018.

[52] B. Yang, Y. He, H. Liu, Y. Chen, and Z. Jin, "A lightweight fault localization approach based on xgboost," in 2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS). IEEE, 2020, pp. 168–179.

[53] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in Advances in neural information processing systems, 2017, pp. 3146–3154.

[54] C. Tang, N. Luktarhan, and Y. Zhao, "An efficient intrusion detection method based on lightgbm and autoencoder," Symmetry, vol. 12, no. 9, p. 1458, 2020.

[55] Q. Pan, W. Tang, and S. Yao, "The application of lightgbm in microsoft malware detection," in Journal of Physics: Conference Series, vol. 1684, no. 1. IOP Publishing, 2020, p. 012041.

[56] J. Svajlenko and C. K. Roy, "Evaluating modern clone detection tools," in 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, 2014, pp. 321–330.

[57] C. C.S., "whatthepatch - python's third party patch parsing library." Online, Accessed August 17th, 2020.

[58] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings. IEEE, 2003, pp. 23–32.

[59] M. Shepperd, D. Bowes, and T. Hall, "Researcher bias: The use of machine learning in software defect prediction," IEEE Transactions on Software Engineering, vol. 40, no. 6, pp. 603–616, 2014.

[60] S. Hassan, C. Tantithamthavorn, C.-P. Bezemer, and A. E. Hassan, "Studying the dialogue between users and developers of free apps in the google play store," Empirical Software Engineering, vol. 23, no. 3, pp. 1275–1312, 2018.

[61] F. E. H. Jr, "Harrell miscellaneous," https://cran.r-project.org/web/packages/Hmisc/Hmisc.pdf, (Last accessed: August 2019).

[62] J. Zhou and H. Zhang, "Learning to rank duplicate bug reports," in Proceedings of the 21st ACM international conference on Information and knowledge management, 2012, pp. 852–861.

[63] S. Wang, Y. Zou, J. Ng, and T. Ng, "Context-aware service input ranking by learning from historical information," IEEE Transactions on Services Computing, 2017.

[64] G. Zhao, D. A. da Costa, and Y. Zou, "Improving the pull requests review process using learning-to-rank algorithms," Empirical Software Engineering, vol. 24, no. 4, pp. 2140–2170, 2019.

[65] S. Wang, T.-H. Chen, and A. E. Hassan, "Understanding the factors for fast answers in technical q&a websites," Empirical Software Engineering, vol. 23, no. 3, pp. 1552–1593, 2018.

[66] O. Ehsan, S. Hassan, M. E. Mezouar, and Y. Zou, "An empirical study of developer discussions in the gitter platform," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 30, no. 1, pp. 1–39, 2020.

[67] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye, "Evolution patterns of open-source software systems and communities," in Proceedings of the international workshop on Principles of software evolution, 2002, pp. 76–85.

[68] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: patterns of cloning in software," Empirical Software Engineering, vol. 13, no. 6, pp. 645–692, 2008.

[69] J. Cohen, P. Cohen, S. G. West, and L. S. Aiken, Applied multiple regression/correlation analysis for the behavioral sciences. Routledge, 2013.

[70] S. Weisberg, Applied linear regression. John Wiley & Sons, 2005, vol. 528.

[71] T. A. Snijders, R. J. Bosker et al., "An introduction to basic and advanced multilevel modeling," Sage, London. WONG, GY, y MASON, WM (1985): The Hierarchical Logistic Regression. Model for Multilevel Analysis, Journal of the American Statistical Association, vol. 80, no. 5, pp. 13–524, 1999.

[72] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos,

D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.

[73] D. Bates, M. Maechler, B. Bolker, S. Walker, R. H. B. Christensen, H. Singmann, B. Dai, F. Scheipl, and G. Grothendieck, "Package 'lme4'."

[74] F. Lafontaine and K. J. White, "Obtaining any wald statistic you want," Economics Letters, vol. 21, no. 1, pp. 35–40, 1986.

[75] J. Fox, S. Weisberg, D. Adler, D. Bates, G. Baud-Bovy, S. Ellison, D. Firth, M. Friendly, G. Gorjanc, S. Graves et al., "Package 'car'," Vienna: R Foundation for Statistical Computing, 2012.

[76] K. Berg and O. Svensson, "Szz unleashed: Bug prediction on the jenkins core repository (open source implementations of bug prediction tools on commit level)," LU-CS-EX 2018-04, 2018.

[77] S. Herbold, A. Trautsch, F. Trautsch, and B. Ledel, "Issues with szz: An empirical assessment of the state of practice of defect prediction data collection," arXiv preprint arXiv:1911.08938, 2019.

[78] E. Kouters, B. Vasilescu, A. Serebrenik, and M. G. Van Den Brand, "Who's who in gnome: Using lsa to merge software repository identities," in 2012 28th IEEE International Conference on Software Maintenance (ICSM). IEEE, 2012, pp. 592–595.

[79] I. S. Wiese, J. T. da Silva, I. Steinmacher, C. Treude, and M. A. Gerosa, "Who is who in the mailing list? comparing six disambiguation heuristics to identify

multiple addresses of a participant," in 2016 IEEE international conference on software maintenance and evolution (ICSME). IEEE, 2016, pp. 345–355.

[80] G. Avelino, E. Constantinou, M. T. Valente, and A. Serebrenik, "On the abandonment and survival of open source projects: An empirical investigation," in 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, 2019, pp. 1–12.

[81] B. Hu, Y. Wu, X. Peng, C. Sha, X. Wang, B. Fu, and W. Zhao, "Predicting change propagation between code clone instances by graph-based deep learning," in Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, 2022, pp. 425–436.

[82] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, "Fasttext.zip: Compressing text classification models," arXiv preprint arXiv:1612.03651, 2016.

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

[84] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014, pp. 313–324.

[85] D. J. Sheskin, Handbook of parametric and nonparametric statistical procedures. crc Press, 2020.

[86] A. García-Floriano, C. López-Martín, C. Yáñez-Márquez, and A. Abran, "Support vector regression for predicting software enhancement effort," Information and Software Technology, vol. 97, pp. 99–109, 2018.

[87] S. Jha, R. Kumar, M. Abdel-Basset, I. Priyadarshini, R. Sharma, H. V. Long et al., "Deep learning approach for software maintainability metrics prediction," Ieee Access, vol. 7, pp. 61 840–61 855, 2019.

[88] D. J. Sheskin, Handbook of parametric and nonparametric statistical procedures. Chapman and Hall/CRC, 2003.

[89] C. Nord, Text analysis in translation: Theory, methodology, and didactic application of a model for translation-oriented text analysis. Rodopi, 2005, no. 94.

[90] S. S. Checker, "Sym spell checker," https://github.com/wolfgarbe/SymSpell, (Last accessed: August 2019).

[91] S. Robertson, "Understanding inverse document frequency: on theoretical arguments for idf," Journal of documentation, 2004.

[92] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen'sd indices the most appropriate choices," in annual meeting of the Southern Association for Institutional Research. Citeseer, 2006, pp. 1–51.

[93] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in 14th IEEE International Conference on Program Comprehension (ICPC'06). IEEE, 2006, pp. 35–45.

[94] P. E. McKnight and J. Najab, "Mann-whitney u test," The Corsini encyclopedia of psychology, pp. 1–1, 2010.

[95] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys," in annual meeting of the Florida Association of Institutional Research, vol. 177, 2006, p. 34.

[96] G. E. Box, "Some theorems on quadratic forms applied in the study of analysis of variance problems, i. effect of inequality of variance in the one-way classification," The annals of mathematical statistics, pp. 290–302, 1954.

[97] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 187–196.

[98] L. Aversano, L. Cerulo, and M. Di Penta, "How clones are maintained: An empirical study," in 11th European Conference on Software Maintenance and Reengineering, 2007, pp. 81 –90.

[99] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, "An empirical study on the maintenance of source code clones," Empirical Software Engineering, vol. 15, pp. 1–34, 2010.

[100] N. Göde, "Evolution of type-1 clones," in Proceedings of the 9th International Working Conference on Source Code Analysis and Manipulation. IEEE Computer Society, 2009, pp. 77–86.

[101] D. Sheskin, Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition). Chapman & All, 2007.

[102] Z. Chen, F. Jiang, Y. Cheng, X. Gu, W. Liu, and J. Peng, "Xgboost classifier for ddos attack detection and analysis in sdn-based cloud," in 2018 IEEE international conference on big data and smart computing (bigcomp). IEEE, 2018, pp. 251–256.

[103] M. Abidi, M. S. Rahman, M. Openja, and F. Khomh, "Are multi-language design smells fault-prone? an empirical study."

[104] K. Inoue, Y. Sasaki, P. Xia, and Y. Manabe, "Where does this code come from and where does it go?—integrated code history tracker for open source systems," in 2012 34th International Conference on Software Engineering (ICSE). IEEE, 2012, pp. 331–341.

[105] J. F. Islam, M. Mondal, and C. K. Roy, "A comparative study of software bugs in micro-clones and regular code clones," in 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2019, pp. 73–83.

[106] M. Mondal, C. K. Roy, and K. A. Schneider, "A comparative study on the bug-proneness of different types of code clones," in 2015 IEEE International conference on software maintenance and evolution (ICSME). IEEE, 2015, pp. 91–100.

[107] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, 2005, pp. 187–196.

[108] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl, "Assessing the benefits of incorporating function clone detection in a development process," in 1997 Proceedings International Conference on Software Maintenance. IEEE, 1997, pp. 314–321.

[109] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," in Proceedings of the 2015 10th joint meeting on foundations of software engineering, 2015, pp. 805–816.

[110] Y. Jiang, B. Adams, and D. M. German, "Will my patch make it? and how fast? case study on the linux kernel," in 2013 10th Working Conference on Mining Software Repositories (MSR). IEEE, 2013, pp. 101–110.

[111] Z.-X. Li, Y. Yu, G. Yin, T. Wang, and H.-M. Wang, "What are they talking about? analyzing code reviews in pull-based development model," Journal of Computer Science and Technology, vol. 32, no. 6, pp. 1060–1075, 2017.

[112] P. Pooput and P. Muenchaisri, "Finding impact factors for rejection of pull requests on github," in Proceedings of the 2018 VII International Conference on Network, Communication and Computing, 2018, pp. 70–76.

[113] A. Tagra, H. Zhang, G. K. Rajbahadur, and A. E. Hassan, "Revisiting reopened bugs in open source software systems," Empirical Software Engineering, vol. 27, no. 4, p. 92, 2022.

Appendix A

Supporting Data

This section presents the appendix of the thesis and include the detailed tables from Chapter 3, Chapter 4, and Chapter 6

Table 1: Chapter 3: Details of the selected JAVA projects in our study

Project Name	Commits	Issues	SLOC	% of files	Genealogies
Java projects					
Anki-Android	10,647	18,079	402.2k	91.70	3,303
che	8,733	10,474	475.5k	73.80	1,056
checkstyle	9,454	12,108	457.4k	97.80	7,705
druid	10,496	8,878	1.2m	94.50	61,718
elasticsearch	53,815	69,010	3.2m	99.80	4,544
framework	18,969	6,879	867.9k	95.50	11,961
gatk	4,173	7,364	2.2m	93.70	22,651
graylog2-server	16,934	7,498	720.3k	73.90	5,782
grpc-java	4,327	17,698	283.1k	98.30	69,248
jabref	15,271	6,991	1.3m	92.70	4,394
k	15,997	1,704	243.3k	83.50	6,026
k-9	9,579	7,130	177.6k	75.80	2,560
mage	31,307	5,007	1.9m	99.90	15,520
minecraftForge	7,451	5,967	136.1k	99.30	659
molgenis	23,001	6,918	359.8k	86.70	14,891
muikku	16,970	5,156	318.4k	50.20	23,836
nd4j	7,021	10,636	467.0k	99.80	45,413
neo4j	68,916	12,676	837.6k	77.50	60,857
netty	9,910	1,587	476.2k	98.60	13,750
openhab	9,687	9,271	968.5k	99.50	2,678
osmand	64,012	10,091	939.3k	95.70	7,688
pinpoint	11,290	6,451	635.4k	89.30	49,191
presto	17,837	4,987	1.4m	98.80	189
product-apim	7,383	6,451	445.6k	91.60	12,871
realm-java	8,318	6,918	199.9k	83.80	13,540
reddeer	1,550	4,171	136.4k	93.60	20
rxjava	5,762	8,374	474.9k	99.90	8,866
smarthome	5,162	7,099	514.0k	93.80	1,348
spring-boot	27,850	25,295	625.5k	98.80	7,841
terasology	10,405	3,043	321.2k	97.80	56,837
wildfly-camel	1,662	6,216	141.3k	99.20	741
xchange	10,434	2,094	655.6k	100.00	3,545
xp	22,615	3,750	423.0k	95.10	68
zaproxy	7,450	9,271	516.4k	72.00	330

Table 2: chapter 3: Details of the selected C projects in our thesis

Project Name	Commits	Issues	SLOC	% of files	Genealogies
C projects					
betaflight	16,458	6,467	3.2m	94.50	4,658
cleanflight	16,589	4,554	3m	94.50	2,587
collectd	11,650	6,832	251k	79.50	1,586
fontForge	19,314	3,834	2.8m	97.90	33,848
freeRDP	14,657	8,444	561k	83.00	11,398
inav	10,339	15,916	2.8m	95.60	4,826
johnTheRipper	15,785	11,411	1.1m	75.90	13,874
libgit2	13,602	3,630	413k	98.20	2,657
lxc	9,748	8,464	138k	86.30	9,356
micropython	11,902	4,587	594k	87.00	3,274
mpv	48,889	5,761	235k	88.30	1,952
netdata	11,823	17,907	603k	73.80	23,874
ompi	31,167	10,463	857k	79.90	6,874
radare2	25,047	10,566	1.3m	93.30	8,947
redis	9,867	6,754	303k	82.60	814
riot	33,594	18,392	2.5m	88.30	567
systemd	48,352	3,116	1.5m	88.30	40,785
zfs	6,459	8,375	808k	75.90	328

Table 3: Chapter 3: Results of the mixed-effect model for Model$_{all}$, Sorted by χ^2 descendingly

Factor	Coef.	χ^2	Percentage	$Pr(<\chi^2)$	Sign.$^+$	Relationship
(Intercept)	-4.653e^{+00}	1238	0.27	<2.2e^{-16}	***	↘
UnqUsers	1.617e^{-01}	300315	64.53	<2.2e^{-16}	***	↗
age_commits	4.907e^{-01}	117350	25.22	<2.2e^{-16}	***	↗
FCngFreq	-2.470e^{-03}	20924	4.50	<2.2e^{-16}	***	↘
UChgRto	2.668e^{-03}	5917	1.27	<2.2e^{-16}	***	↗
age_days	2.823e^{-04}	4352	0.94	<2.2e^{-16}	***	↗
LvstDis	-7.189e^{-03}	3481	0.75	<2.2e^{-16}	***	↘
CCore	6.768e^{-02}	2295	0.49	<2.2e^{-16}	***	↗
Avg_CT	-8.360e^{-04}	2096	0.45	<2.2e^{-16}	***	↘
sib_cnt	-1.076e^{-02}	1553	0.33	<2.2e^{-16}	***	↘
EFltDens	-5.211e^{-01}	1316	0.28	<2.2e^{-16}	***	↘
SLBurst	5.835e^{-02}	1059	0.23	<2.2e^{-16}	***	↗
experience	3.002e^{-05}	820	0.18	<2.2e^{-16}	***	↗
accepted	2.345e^{-04}	709	0.15	<2.2e^{-16}	***	↗
EConChg	1.594e^{-01}	291	0.06	<2.2e^{-16}	***	↗
rejected	-5.577e^{-04}	273	0.06	<2.2e^{-16}	***	↘
TChurn	6.956e^{-05}	250	0.05	<2.2e^{-16}	***	↗
EIncStChg	9.339e^{-02}	244	0.05	<2.2e^{-16}	***	↗
TPC	-8.662e^{-02}	210	0.05	<2.2e^{-16}	***	↘
EConStChg	-6.570e^{-02}	203	0.04	<2.2e^{-16}	***	↘
CCurSt	5.299e^{-02}	119	0.03	<2.2e^{-16}	***	↗
cloc	9.776e^{-05}	105	0.02	<2.2e^{-16}	***	↗
UFileChg	2.491e^{-04}	88	0.02	<2.2e^{-16}	***	↗
CPathDepth	-4.163e^{-03}	66	0.01	3.734e^{-16}	***	↘
EEvPattern	1.088e^{-02}	45	0.01	1.937e^{-11}	***	↗
NumOfBursts	6.726e^{-03}	33	0.01	8.970e^{-09}	***	↗
EChgTimeInt	-1.025e^{-05}	7	0.00	0.006643	**	↘

$^+$Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Table 4: Chapter 3: Results of the mixed-effect model for Model_{mature}, Sorted by χ^2 descendingly

Factor	Coef.	χ^2	Percentage	$Pr(<\chi^2)$	Sign.$^+$	Relationship
(Intercept)	-6.772e^{+00}	324		<2.2e^{-16}	***	↘
age_commits	5.230e^{-01}	11639	52.24	<2.2e^{-16}	***	↗
UnqUsers	1.228e^{-01}	2152	9.66	<2.2e^{-16}	***	↗
UFileChg	-1.668e^{-03}	1318	5.92	<2.2e^{-16}	***	↘
FCngFreq	-1.185e^{-03}	1248	5.60	<2.2e^{-16}	***	↘
age_days	5.290e^{-04}	1119	5.02	<2.2e^{-16}	***	↗
CPathDepth	4.664e^{-02}	987	4.43	<2.2e^{-16}	***	↗
LvstDis	1.182e^{-02}	969	4.35	<2.2e^{-16}	***	↗
CCore	1.679e^{-01}	718	3.22	<2.2e^{-16}	***	↗
EFltDens	-9.565e^{-01}	345	1.55	<2.2e^{-16}	***	↘
NumOfBursts	-8.230e^{-02}	331	1.49	<2.2e^{-16}	***	↘
SLBurst	1.091e^{-01}	324	1.45	<2.2e^{-16}	***	↗
TChurn	-1.598e^{-03}	155	0.70	<2.2e^{-16}	***	↘
EIncStChg	2.624e^{-01}	128	0.57	<2.2e^{-16}	***	↗
cloc	-7.454e^{-04}	117	0.53	<2.2e^{-16}	***	↘
TPC	-2.227e^{-01}	92	0.41	<2.2e^{-16}	***	↘
EConChg	2.528e^{-01}	48	0.22	2.564e^{-12}	***	↗
UChgRto	7.602e^{-04}	42	0.19	7.937e^{-11}	***	↗
sib_cnt	-4.957e^{-03}	41	0.18	1.279e^{-10}	***	↘
experience	1.944e^{-05}	33	0.15	7.821e^{-09}	***	↗
EEvPattern	-3.791e^{-02}	32	0.14	9.858e^{-09}	***	↘
CCurSt	1.097e^{-01}	31	0.14	2.399e^{-08}	***	↗
rejected	3.149e^{-04}	24	0.11	5.928e^{-07}	***	↗
accepted	7.361e^{-05}	21	0.09	4.022e^{-06}	***	↗
Avg_CT	2.603e^{-04}	17	0.08	2.806e^{-05}	***	↗
EConStChg	7.308e^{-02}	15	0.07	6.575e^{-05}	***	↗
EChgTimeInt	-4.249e^{-05}	9	0.04	0.001636	**	↘

$^+$Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Table 5: Chapter 3: Results of the mixed-effect model for Model$_{early}$, Sorted by χ^2 descendingly

Factor	Coef.	χ^2	Percentage	$Pr(<\chi^2)$	Sign.$^+$	Relationship
(Intercept)	-4.400e^{+00}	103	0.13	<2.2e^{-16}	***	↘
UnqUsers	1.638e^{-01}	40938	52.42	<2.2e^{-16}	***	↗
age_commits	4.420e^{-01}	26567	34.02	<2.2e^{-16}	***	↗
CPathDepth	1.082e^{-01}	3762	4.82	<2.2e^{-16}	***	↗
cloc	5.170e^{-03}	3736	4.78	<2.2e^{-16}	***	↗
SLBurst	-1.163e^{-01}	430	0.55	<2.2e^{-16}	***	↘
UChgRto	1.463e^{-03}	404	0.52	<2.2e^{-16}	***	↗
experience	1.075e^{-04}	290	0.37	<2.2e^{-16}	***	↗
LvstDis	6.167e^{-03}	282	0.36	<2.2e^{-16}	***	↗
NumOfBursts	-8.708e^{-02}	262	0.34	<2.2e^{-16}	***	↘
Avg_CT	-5.338e^{-04}	198	0.25	<2.2e^{-16}	***	↘
CCore	6.091e^{-02}	186	0.24	<2.2e^{-16}	***	↗
FCngFreq	-1.450e^{-03}	138	0.18	<2.2e^{-16}	***	↘
EConStChg	2.030e^{-01}	129	0.17	<2.2e^{-16}	***	↗
EFltDens	4.681e^{-01}	127	0.16	<2.2e^{-16}	***	↗
EChgTimeInt	4.041e^{-04}	125	0.16	<2.2e^{-16}	***	↗
age_days	9.855e^{-05}	123	0.16	<2.2e^{-16}	***	↗
sib_cnt	5.250e^{-03}	81	0.10	<2.2e^{-16}	***	↗
EConChg	-3.142e^{-01}	77	0.10	<2.2e^{-16}	***	↘
TChurn	-2.313e^{-04}	64	0.08	8.367e^{-16}	***	↘
TPC	1.335e^{-01}	38	0.05	7.029e^{-10}	***	↗
CCurSt	-7.345e^{-02}	13	0.02	0.000211	***	↘
EIncStChg	-7.046e^{-02}	10	0.01	0.001138	**	↘
rejected	-1.914e^{-03}	7	0.01	0.005693	**	↘
EEvPattern	1.643e^{-02}	6	0.01	0.013103	*	↗
accepted	1.771e^{-04}	2	0.00	0.088527	.	↗
UFileChg	-5.907e^{-04}	0	-	0.562663		↘

$^+$Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Table 6: Chapter 4: Description of the questions in our survey.

Question	**Type/Options**
1. Do you use GitHub?	Yes/No
2. What is the highest computer science degree you have completed?	Student College diploma Bachelor's Masters Professional doctorate self-taught
3. What is your overall software development experience?	less than year 1-3 3-5 5-10 10+
4. Do you often use code clone detection tools?	yes/no
5. Do you perceive the type of code changes differently, for example, corrective (bug-fixes), perfective (refactoring changes), new features etc?	likert scale *strongly agree (1)* *strongly disagree (5)*
6. What do you think of the usefulness of providing information regarding the likelihood that changes made to cloned code may cause bugs before issue reports are submitted?	likert scale *strongly agree (1)* *strongly disagree (5)*
7. In your opinion, how are the complexity/ churn/size metrics better than LOC to determine effort estimation? (a) complexity metrics are better than LOC (b) churn metrics are better than LOC (c) Multiple size metrics are better than LOC	likert scale *strongly agree (1)* *strongly disagree (5)*
8. Do you think the information provided can help in deciding the propagation decision of the code clones?	yes/no
9. Do you have any suggestions to improve the approach to recommend the propagation of code clone changes?	open-ended
10. How comprehensive is the information provided for every code change for making a decision on propagation?	NPS Score (0-10) *not comprehensive (0)* *extremely comprehensive (10)*

Table 7: Chapter 6: Details of the selected JAVA projects in our study.

Project Name	Commits	Pull Requests	Issues	SLOC	% of files	Genealogies
Java projects						
Anki-Android	14,332	6,854	5,897	413.4k	91.70	1,458
che	9,132	7,263	12,368	483.2k	76.75	957
checkstyle	11,428	7,845	13,856	479.9k	98.1	4,325
druid	10,496	8,816	9,227	1.2m	94.1	7,451
elasticsearch	62,819	56,781	69,010	3.2m	99.80	2,817
framework	17,272	10,267	11,124	886.3k	83.8	3,687
gatk	4,351	8,476	7,576	2.2m	93.70	5,857
graylog2-server	16,934	10,785	11,383	720.3k	73.90	2,183
grpc-java	5,028	8,257	9,284	283.1k	98.30	7,197
jabref	16,482	4,659	4,257	1.3m	91.60	2,267
k	18,969	2,453	3,879	886.3k	83.8	3,492
k-9	10,627	3,159	3,223	177.6k	99.80	1,359
mage	36,850	4,207	4,583	1.9m	99.90	6,288
minecraftForge	7,954	4,238	4,895	136.1k	97.5	459
molgenis	23,151	2,453	5,358	359.8k	86.70	3,894
muikku	20,351	2,286	3,267	318.4k	50.20	6,334
nd4j	7,537	3,586	6,397	467.0k	99.80	5,289
netty	10,589	6,746	5,268	476.2k	98.60	2,185
openhab	9,687	3,182	2,890	968.5k	99.50	1,038
pinpoint	12,138	5,672	4,268	635.4k	89.30	5,298
presto	19,264	11,938	4,235	1.4m	98.80	118
product-apim	8,982	4,284	6,028	445.6k	91.60	3,171
realm-java	8,566	3,158	3,098	199.9k	83.80	3,082
reddeer	1,580	1,423	1,096	136.4k	93.60	18
rxjava	5,923	3,567	3,033	474.9k	99.90	2,334
smarthome	5,162	4,361	3,129	514.0k	93.80	759
spring-boot	36,128	5,268	27,358	625.5k	98.80	3,098
terasology	11,109	2,804	1,442	321.2k	97.80	2,850
wildfly-camel	1,677	1,377	1,986	141.3k	99.20	358
xchange	11,595	3,294	1,394	655.6k	100.00	1,296
xp	23,186	6,257	2,667	423.0k	95.10	45
zaproxy	8,288	3,458	4,168	516.4k	72.00	117

Table 8: Chapter 6: The importance scores for XGBoost Model for the prediction of degree of inconsistency, sorted by importance score descendingly.

Feature	**Importance Score**
TChurn	0.62
EIncChg	0.56
TotalCom	0.52
CAgeDays	0.44
EFltDens	0.39
experience	0.37
CPathDepth	0.34
ActSib	0.31
TPC	0.28
rejected	0.23
ExcLines	0.21
LvstDis	0.2
EConStChg	0.17
EEvPattern	0.15
CCurSt	0.14
accepted	0.12
Avg_CT	0.11
UnqMet	0.1
CCore	0.1
NumOfBursts	0.1
McCabe	0.1
CSib	0.1
EConChg	0.1
EChgTimeInt	0.1
EIncStChg	0.1
SynCon	0.1
cloc	0.1
DecLines	0.1
age_pulls	0.1
UnqUsers	0.1
EChgTimeInt	0.1
StrucMet	0.1
UChgRto	0.1
UFileChg	0.1
SLBurst	0.1
FCngFreq	0.1
TotalFil	0.1

Table 9: Chapter 6: The importance scores for LighGBM Model for the lifetime of bugs, sorted by importance score descendingly.

Lifetime of bugs in number of days		**Lifetime of bugs in number of releases**	
Feature	**Importance Score**	**Feature**	**Importance Score**
EFltDens	0.58	TChurn	0.52
EIncStChg	0.52	ActSib	0.49
age_pulls	0.47	cloc	0.46
TPC	0.72	EFltDens	0.43
EIncChg	0.39	CCurSt	0.41
cloc	0.95	EIncStChg	0.38
FCngFreq	0.32	FCngFreq	0.37
ActSib	0.3	EIncChg	0.35
TChurn	0.27	SLBurst	0.33
EConChg	0.25	EChgTimeInt	0.29
UnqUsers	0.21	TPC	0.25
EChgTimeInt	0.19	experience	0.23
CSib	0.17	UFileChg	0.2
NumOfBursts	0.13	rejected	0.17
Avg_CT	0.12	EChgTimeInt	0.15
EConStChg	0.11	LvstDis	0.12
ExcLines	0.1	EConChg	0.1
CPathDepth	0.1	CCore	0.1
UChgRto	0.1	CAgeDays	0.1
EEvPattern	0.1	accepted	0.1
UFileChg	0.1	ExcLines	0.1
TotalFil	0.1	DecLines	0.1
TotalCom	0.1	McCabe	0.1
CCore	0.1	TotalFil	0.1
CAgeDays	0.1	UnqMet	0.1
experience	0.1	NumOfBursts	0.1
accepted	0.1	UnqUsers	0.1
rejected	0.1	Avg_CT	0.1
SLBurst	0.1	age_pulls	0.1
McCabe	0.1	CSib	0.1
CCurSt	0.1	StrucMet	0.1
SynCon	0.1	TotalCom	0.1
StrucMet	0.1	EConStChg	0.1
EChgTimeInt	0.1	CPathDepth	0.1
UnqMet	0.1	UChgRto	0.1
LvstDis	0.1	EEvPattern	0.1
DecLines	0.1	SynCon	0.1