Aspect of Code Cloning Towards Software Bug and Imminent Maintenance: A Perspective on Open-source and Industrial Mobile Applications.

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Md Shamimur Rahman

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Or

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9
Canada

Abstract

As a part of the digital era of microtechnology, mobile application (app) development is evolving with lightning speed to enrich our lives and bring new challenges and risks. In particular, software bugs and failures cost trillions of dollars every year, including fatalities such as a software bug in a self-driving car that resulted in a pedestrian fatality in March 2018 and the recent Boeing-737 Max tragedies that resulted in hundreds of deaths. Software clones (duplicated fragments of code) are also found to be one of the crucial factors for having bugs or failures in software systems. There have been many significant studies on software clones and their relationships to software bugs for desktop-based applications. Unfortunately, while mobile apps have become an integral part of today's era, there is a marked lack of such studies for mobile apps. In order to explore this important aspect, in this thesis, first, we studied the characteristics of software bugs in the context of mobile apps, which might not be prevalent for desktop-based apps such as energy-related (battery drain while using apps) and compatibility-related (different behaviors of same app in different devices) bugs/issues. Using Support Vector Machine (SVM), we classified about 3K mobile app bug reports of different open-source development sites into four categories: crash, energy, functionality and security bug. We then manually examined a subset of those bugs and found that over 50% of the bug-fixing code-changes occurred in clone code. There have been a number of studies with desktop-based software systems that clearly show the harmful impacts of code clones and their relationships to software bugs. Given that there is a marked lack of such studies for mobile apps, in our second study, we examined 11 open-source and industrial mobile apps written in two different languages (Java and Swift) and noticed that clone code is more bug-prone than non-clone code and that industrial mobile apps have a higher code clone ratio than open-source mobile apps. Furthermore, we correlated our study outcomes with those of existing desktop-based studies and surveyed 23 mobile app developers to validate our findings. Along with validating our findings from the survey, we noticed that around 95% of the developers usually copy/paste (code cloning) code fragments from the popular Crowd-sourcing platform, Stack Overflow (SO) to their projects and that over 75% of such developers experience bugs after such activities (the code cloning from SO). Existing studies with desktop-based systems also showed that while SO is one of the most popular online platforms for code reuse (and code cloning), SO code fragments are usually toxic in terms of software maintenance perspective. Thus, in the third study of this thesis, we studied the consequences of code cloning from SO in different open-source and industrial mobile apps. We observed that closed-source industrial apps even reused more SO code fragments than open-source mobile apps and that SO code fragments were more change-prone (such as bug) than non-SO code fragments. We also experienced that SO code fragments were related to more bugs in industrial projects than open-source ones. Our studies show how we could efficiently and effectively manage clone related software bugs for mobile apps by utilizing the positive sides of code cloning while overcoming (or at least minimizing) the negative consequences of clone fragments.

Acknowledgements

At first, I like to praise Almighty God, the most gracious and most merciful, who gave me the ability to carrying out this work. Next, I would like to express my sincerest appreciation to my supervisor Dr. Chanchal K. Roy for his continuous support, guidance, motivation, and remarkable endurance during this thesis work. Without his support, this work would have been unthinkable.

I would like to thank Dr. Ralph Deters, Dr. Khan A Wahid and Dr. Manishankar Mondal for their willingness to take part in the advisement and evaluation of my thesis work. I would also like to thank them for their valuable time. I am also thankful to Prof. Kevin Schneider for his help and suggestions during the hospitalization of Dr. Roy.

I would also wish to express my gratefulness to my industrial mentor, Chad Jones, CEO of Push Interactions Inc. for extended discussions, valuable suggestions, passionate participation, and input, which have contributed greatly to the improvement of the thesis.

I express my heartiest gratitude to my father Md Hafizar Rahman and my mother Mst Bilkis Banu who are the architects of my life. Their endless sacrifice, unconditional love, and constant good wishes have made me reach this stage of my life. I am also thankful to my elder sister Mst Shoheli Akter Sathi and sister-in-law Mostakim Billah for their support and inspiration in my bad times.

I would like to convey my greatest respect and heartiest gratitude to my beloved friend Abdul Awal and Khairul Alam for helping me to see the new light of life and start anew.

Thanks to all of the members of the Software Research (SR) and Interactive Software Engineering (iSE) Labs with whom I have had the opportunity to grow as a researcher. In particular, I would like to thank Muhammad Mainul Hossain, Tonny Kar, Farouq Al. Omari, Kawser Wazed Nafi, Saikat Mondol, Amit Kumar Mondal, Debasish Chakroborti, Hamid Khodabandehloo, Md Nadim, Avijit Bhattacharjee, Naz Zarreen Oishie, Daniel Abediny, Sristy Sumana Nath, C M Khaled Saifullah, Shamima Yeasmin, Judith Islam, Zonayed Ahmed, Shamse Tasnim Cynthia, Rayhan Islam Shuvo and Saumendu Roy.

Most importantly, I am grateful to Natural Sciences and Engineering Research Council of Canada (NSERC) and University of Saskatchewan for supporting my study through NSERC Industry Engage and Discovery grants, and Graduate Teaching Fellowships (USask).

I would like to thank all of my friends and other staff members of the Department of Computer Science who have helped me to reach this stage. In particular, I would like to thank Raouf Ajami, Greg Oster, Maurine Powell, Cary Bernath, Smit Choksi, Sophie Findlay, Shakiba Jalal, Heather Webb, and James Ko.

I dedicate this thesis to my beloved son, ***Md Mukthadir Rahman Samit***. His priceless heavenly innocence and unforgettable smiles give me strength and inspiration to be a great father and an honest human being.

Contents

List of Tables

List of Figures

List of Abbreviations

AI	Artificial Intelligence
ALOC	Average Lines of Code
AST	Abstract Syntax Tree
AUROC	Area Under ROC Curve
DT	Decision Tree
FPR	False Positive Rate
GNB	Gaussian Naive Bayes
GPS	Global Positioning System
GPU	Graphics Processing Unit
KNN	K-Nearest Neighbor
LOC	Line of Code
LSI	Latent Semantic Indexing
NB	Naive Bayes
NR	Number of Revision
ML	Machine Learning
MWW	Mann-Whitney-Wilcoxon
PDG	Program Dependency Graph
RAE	Recursive Auto Encoders
RF	Random Forest
ROC	Receiver Operating Characteristic
SO	Stack Overflow

1 Introduction

This chapter provides the motivation and a short introduction to the thesis. Research problems are discussed in Section 1.1 along with thesis motivation. Section 1.2 presents our contributions to address the research inadequacy. This thesis prepares the possible publications for future submission, which are shown in Section 1.3. Finally, Section 1.4 exhibits an outline of the remaining chapters of this thesis.

1.1 Motivation and Research Problem

In recent years, mobile applications (apps) provide consumer-oriented solutions that convey highly designated functionalities and customized experiences with the assistance of artificial intelligence. The popularity of mobile apps is rising day by day with advanced features such as social integration, secure payment, cross-platform coverage, augmented reality integration, managing positioning information and so on. These not only enrich our lives but also bring new risks and challenges. In particular, software bugs and crashes cost trillions of dollars every year, including fatalities such as a software bug in a self-driving car that resulted in a pedestrian fatality[1] in March 2018 and the recent Boeing-737 Max tragedies[2] that resulted in hundreds of deaths. Additionally, several catastrophic incidents in software-control medical arrangements, Marine aircraft and missile systems that caused severe injuries and deaths during the last decades [31, 193, 198]. In 2017, 314 commercial organizations and over 3.7 billion people were affected by 606 software bugs which cost around $1.7 trillion[3]. In addition, during 2009-2018, a software bug caused scheduling errors for mammography tests of elderly British women. As a result, half a million women missed their tests, leading to hundreds of premature deaths[4]. Furthermore, in 1985, four Canadian cancer patients lost their lives, and two more were having severe injuries because of fatal software bugs in the Therac-25 radiation therapy system[5], which happened after successfully performing 20,000 irradiations on the region's cancer patients. All these catastrophic incidents and unfortunate tragedies illustrate the severe consequences of software bugs and malfunctions. So who is/are really responsible for these software-induced catastrophes is a great question to ask. The answer might significantly indicate the faults in development strategies, the inadequacy to report potential severe software bugs during testing phases and the lack of diligence in triaging newly incoming bug reports that limit immediate actions and allocate resources. Although there have been numerous active researches to prevent software inconsistencies and deadly consequences, further research is warranted more than ever regarding

[1] https://bbc.in/3B44YRH
[2] https://abcn.ws/3jl21WV
[3] https://tek.io/2FBNl2i
[4] https://bit.ly/2E1fYap
[5] https://bit.ly/2KU9IR2

the exploration of the implementation procedures (source code level), potential bugs' characteristics, and probable categories to illustrate the severity.

At the stage of development and further the app's lifetime, developers, maintainers and even end-users report a huge amount of app inconsistencies and incompatibility issues in their development sites. Reported issues might be software bugs, enhancement requests or compatibility suggestions. Manual analysis of bug reports requires both time and resources, which can delay the decision-making where immediate actions are needed. Like modern desktop-based systems, mobile apps usually follow Event-Driven Architecture [130], bugs from one event might be harmful to another event if there are any dependencies. So, immediate decision and resource allocation are demanded by developers and maintainers in bug management. To decide and allocate resources, developers need to know the type, characteristics and severity of the occurred bug. Hence, it is a must to classify bugs (i.e., bug reports) into certain groups. Initially, we focused on the characteristics of software bugs in the context of mobile apps, which might not be prevalent for desktop-based apps. For example, we found energy-related bugs/issues (battery drain while using apps), which is considered one of the key concerns of user satisfaction, compatibility constraints (different behaviours of the same app in different devices), and other functional and security issues (e.g., GPS, device configurations, authentication, and secure network) that arise only upon user interactions. Therefore, it is promising to know the bug category so that the developers and maintainers can respond and allocate resources on a priority basis. Therefore, we plan to classify the mobile application bug reports into four groups i.e., security bug, energy bug, functionality bug and app crash bug [207]. The complexities will arise when end users report these type of bugs in different formats and styles. Different users report apps' disharmony differently, so we need an automated classifier to deal with. Several existing studies classified bug reports mostly in two/three categories for example, corrective (defect fixing) and perfective (major maintenance) [32, 150], security and non-security [68, 72, 111, 205], important, not important and request for enhancement [28, 78]. All of the related works investigated both bug and non-bug issues of several desktop-based systems. To the best of our knowledge, no existing studies classified only bug reports into further categories or according to mobile applications' specific defects.

All software systems are often profoundly intricate parts of modern technology and innovation that require regular updates and maintenance support, which are tedious and time-consuming and should be anticipated well ahead of time. It is a common belief that software cost is a one-time investment brought about when the software is being developed/purchased. Unexpectedly, industry specialists estimate that more than 90% of all expenses are regular maintenance costs that most companies do not consider in the first place [22]. Therefore, software changes are unavoidable during maintenance and apps' evolution, but they might create risks and inconsistencies if the changes are completed without proper awareness. While investigating the incorporated code-change at the time of bug-fixing, we found that majority of the code-change are happened in clone code than non-clone code. In addition, there are number of studies revealed that code clones have been supposed to be liable for introducing additional change requirements and also software bugs.

Frequent copy/paste or code cloning is a commonly accepted practice during software development and further maintenance. Repetitive standard software features, technology and solutions constraints, code understand-ability, limited development time-frame and outer business races are potential reasons for doing code cloning [99, 167]. Whatever the reasons behind code cloning, it is still a controversial issue among software research communities with both positive and negative impacts of cloning. Several studies [35, 74, 83, 95, 99, 107, 108, 109] showed positive impacts of code cloning, while there are other studies [40, 70, 93, 95, 115, 121, 122, 136, 137, 183] that prospected harmful results of cloning. In addition, there are numerous studies [36, 40, 75, 88, 89, 138, 144] that investigated the relationship of cloned code and bugs, but none of the existing studies have ever investigated the actual impacts of code clones in mobile apps and not for industrial mobile apps at all. Moreover, developers frequently copy/reuse external code elements from different open-source software systems and crowd-sourced sites [24, 117]. Therefore, those external source code might have impacts at the stages of software development that also need to be investigated.

Crowd-sourced developers sites such as Stack Overflow (SO) have emerged much popularity regarding discussion of programming problems, implementation constraints, impactful solutions from domain experts, and enormous resources of current development trends and practices. To make high-quality software in a timely and cost-efficient manner, source code reuse, cloning the common functionalities, and designs are widely accepted fundamental approaches [118]. Several studies [42, 161] showed that SO is very popular among mobile developers where developers seek assistance frequently about implementation problems and they reuse SO code snippets to mitigate their issues in a systematic manner [161, 169]. A prior work [24] investigated 22 open-source Android applications that reused SO code snippets. They examined SO code ratio in the mobile codebase, possible reasons for code reusing, who mainly reuse source code, and its implication after reusing. However, their study is limited in investigating the actual behaviors of reused SO code snippets in mobile apps development. For example, their study claimed mobile apps experienced more bug-fixing commits after the reuse than before. Nevertheless, it is not sure that the reused code were responsible for those bugs and that they did not consider the change-proneness of the SO code snippets throughout all the revisions. A plethora of studies [58, 156, 178] presented the adverse effects of SO code snippets such as code smell, toxic code fragments, source of technical debt and so on. So it is promising to investigate the actual consequences of SO code snippets in the context of open-source mobile apps as well as commercial projects where external code reused is nearly prohibited due to code security, attribution, and licensing issues.

1.2 Research Contributions

Focusing on the above research limitations on mobile apps' bug classification and impact of code cloning, we performed the following studies. The following subsections briefly describe each of these studies.

1.2.1 Study 1: Exploring Classification of Software Bug Reports Towards Mobile Applications

First, we focused on the characteristics of software bugs in the context of mobile apps, which might not be prevalent for desktop-based apps. In most cases, apps' inconsistencies (i.e., bugs) arise upon interactions between app and users depending on specific times, positions and device varieties. Therefore, it is promising to investigate the characteristics/categories of mobile bugs so that the developers and maintainers can respond and allocate resources accordingly. Hence, we plan to classify the mobile app bug reports into four groups, i.e., security, energy, functionality, and app crash.

Second, we collected around 3K bug reports of several Android and iOS apps assembled from numerous open-source software developer sites (i.e., GitHub, Google Code, Trac and F-droid). Since the collected bug reports were not labelled, we prepared a distinctive feature set of each type of bug report and then manually labelled them (bug reports) into four groups.

Third, we prepared a machine learning-based classifier which helps to categorize newly incoming bug reports. Therefore, our distinctive feature set is fed into several classification algorithms (Gaussian Naive Bayes, K-Nearest Neighbor, Decision Tree, Support Vector Machine and Random Forest) to find the best classification model. As a result, the Support Vector Machine classifier performs the finest among the distinctive algorithms with a promising f1 score (91%) with the compiled dataset.

After the classification, we wanted to explore the possible code-change in a particular software codebase while fixing a bug. So, we investigated randomly selected 50 bug reports of Java projects from our collected 3K bug reports dataset. As a result, we noticed 82% of bug-fixing commits, Java code experienced code-change while the rest 18% of code-changes appeared in HTML, XML and other non-Java files. Furthermore, from the Java code-changes, we marked 56% of code-changes are happened in clone code (exact or similar code fragments throughout the codebase). Moreover, there have been a number of studies with desktop-based software systems that clearly show the harmful impacts of code clones and their relationships to software bugs. However, given that there is a marked lack of such studies for mobile apps, the second study explored several mobile applications to find the consequences of clone and non-clone code throughout the app lifetime regarding bug-proneness and change-proneness. This next study also shows the diversity between mobile and non-mobile (i.e., desktop) apps in terms of code maintenance while fixing software bugs.

1.2.2 Study 2: Analysis of Code Cloning in Open Source and Industrial Software Development Stages: A Perspective of Mobile Applications

To mitigate the study gap regarding the impacts of code clones in mobile apps development, we analyzed 11 open-source and industrial mobile apps written in Java (Android) and Swift (iOS). We collected these industrial projects from a famous local software company in Canada and open-source projects from GitHub. This experiment perceived that clone code is more bug-prone than non-clone code. Furthermore, more

clone code fragments are present in industrial apps than open-source apps, i.e., cloning is more frequent in industrial apps. Additionally, we compared our experimental results with non-mobile (i.e., desktop) apps and noticed that mobile apps need less maintenance in clone code than non-mobile. Overall, we investigated and scrutinized the bug-proneness and change-proneness of clone code and later compared open-source vs industrial and mobile vs non-mobile apps.

In the next part of this study, we surveyed 23 developers from 17 different companies across Canada and Bangladesh concerning code clones usability, possible reasons to clone code, further effects in code maintenance and so on. The survey strengths our study outcomes as industry people experienced similar thoughts about clone code, software bugs, and later maintenance in their app development stages. Besides cloning code from one file to another, developers also copy/clone code from several crowd-sourced sites, in specific SO. Around 95% of developers usually reused code from SO during the implementation stages of apps, and later, more than 75% of developers experienced software bugs because of the reused SO code snippets. Lack of required resources, less experience of developers and availability of experts suggestions by asking questions all are potential reasons for reusing SO code snippets. Whatever the reasons, developers noticed software bugs in later revisions, and these bugs required extra maintenance. Since a single bug could be responsible for any catastrophic incident, it is promising to understand the actual consequences of SO code elements in mobile apps in the context of software bugs and code maintenance. However, since several studies showed that SO contains toxic code fragments involving outdated code and licence violation issues, no study has been conducted on how SO code elements behave adversely in real systems. Therefore, we conveyed our next study to explore the impacts of SO code elements in the context of open-source and industrial mobile apps.

1.2.3 Study 3: Cloning and Consequences of Stack Overflow Source Code: A Study on Open-source and Industrial Mobile Applications

In order to understand the impact of code cloning or reusing from SO, we examined SO code snippets reused in open-source and industrial apps in the context of code ratio, change-proneness, bug-proneness and analyzed the properties of the buggy and non-buggy answer code snippets. Therefore, we extracted more than two million SO answer code snippets to find the reused code fragments in the apps' codebase. Our analysis exhibits:

- The proportion of reused SO code is comparatively higher in industrial mobile apps than open-source.
- Open-source projects mostly reuse SO code to enhance existing features, whereas industrial projects reuse it to add new features into the application.
- SO code fragments are significantly more change-prone than non-SO code.
- SO code snippets are responsible for bug occurrence in later revisions, which is comparatively higher in industrial projects than open-source ones.

Our experimental results can assist the SO, research, and mobile developer communities to strengthen usability and concerns to facilitate code-quality improvement and minimize software bugs due to SO code. Additionally, while investigating the buggy and non-buggy SO code snippets, both user reputation and answer score seem inadequate to authenticate the code quality when reusing code fragments from SO. Possible supports might consolidate from domain experts at the time of answer submission regarding code quality. However, manual code quality analysis is strenuous and time-consuming, referring to an automatic code review system integrated with SO.

1.3 Related Prepared Publications

The list of publications is prepared from this thesis work for submission in different conferences and journals with collaborators.

- Md Shamimur Rahman, Abdul Awal, Chanchal K. Roy. Exploring Classification of Software Bug Reports Towards Mo-bile Applications. *Journal of Systems and Software (JSS)*, 25 pages (under review).
- Md Shamimur Rahman, Chanchal K. Roy, Kevin Schneider, James R. Cordy, Chad Jones. Analysis of Code Cloning in Open Source and Industrial Software Development Stages: A Perspective of Mobile Applications. *Journal of Systems and Software (JSS)*, 36 pages (under review)
- Md Shamimur Rahman, Chanchal K. Roy, Chad Jones. Cloning and Consequences of Stack Overflow Source Code: A Study on Open-source and Industrial Mobile Applications. *The International Conference on Mining Software Repositories (MSR)*, Double column 12 pages (to be submitted)

1.4 Outline of the Thesis

There are six chapters in total, including the **Introduction**. In this thesis, we conducted three individual but interconnected studies regarding software bugs and code cloning from the perspective of mobile apps. Hereabouts, we outline the thesis as follows:

- Chapter 2 presents several background concepts and terminologies, which include code clones, types of clones, the impact of code clones, different approaches for clone code detection, supervised and unsupervised machine learning algorithms and several statistical testing methods.
- Chapter 3 discusses our first study on classifying mobile bug reports using machine learning algorithms. Here, we show how manual labelling of bug reports is done and then evaluate the classifier with different matrices.
- Chapter 4 depicts the overall clone analyses on several open-source and industrial mobile apps. Here, we reveal that clone code is more bug-prone than non-clone and make comparison between mobile vs non-mobile and open-source vs industrial apps.

- Chapter 5 illustrates the study result of SO code snippets reused in open-source and industrial mobile applications. In this study, we experienced that SO code are more change-prone than non-SO code elements and therefore, need special review before cloning SO code.
- Chapter 6 concludes the thesis with the direction of significant future works.

2 Fundamentals and Terminologies

In this chapter, we would like to provide a brief idea of our background studies, fundamentals and different technical terminologies for making the thesis more understandable. Section 2.1 refers the idea of code cloning, different types of code clones with appropriate examples and impact of code clones in software communities. Section 2.2 describes the techniques and approaches of detecting code clones from software codebase. A number of machine learning algorithms that we used in our thesis study are discussed in Section 2.4. In Section 2.5, we reference some methods of statistical significance testing and finally Section 2.6 concludes this chapter.

2.1 Code Clone

In a software code-base system, code clone happens if multiple code fragments are exactly or nearly similar to each other. During the evolution of software system, frequent copy/paste activities which are performed by the developers causes code clones. A pair of two code snippets that are the same or similar to each other is called a code clone pair or simply clone pair. A group of similar clone pair form a clone class hence called clone cluster. There might be several reasons for code clones such as developers' behaviour like laziness and tendency to repeat common solutions, technology limitations [43, 152] (e.g., lack of reusing mechanism in programming languages), code evolvability and code understandability [135]. Cloning code might be done from same software system or other software systems (it is called cross clone) and different open source development site (i.e., SO, GitHub). Whatever the sources and possible reasons of code cloning, it is a matter of great concern at the stage of software evolution and maintenance. We briefly describe the clone types, detection techniques and impacts of code cloning in the following subsections.

2.1.1 Types of Code Clones

Based on the existing studies and literatures, there are mainly four types of code clones. These are defined as follows:

Type 1 Clone: If two clone fragments are exactly similar or identical to each other where the only difference is comments and/or code formation (ex., blank space, newline etc). An example of three Type 1 clone fragments is shown in Figure 2.1[1]

[1]https://harvest.usask.ca/handle/10388/ETD-2013-01-911

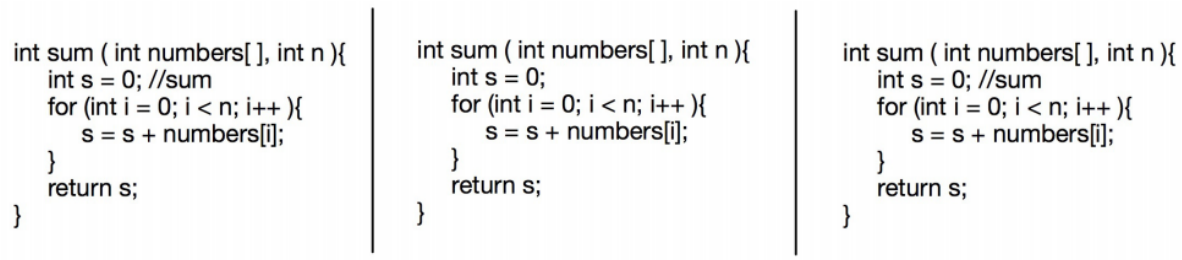


Figure 2.1: Example of Type 1 clone code (adapted from [134]).

Type 2 Clone: Type 2 clone fragments are syntactically similar but changed in identifier names and data types. Figure 2.2[1] depicts Type 2 clone fragments where code fragments are different in function and identifier naming.

Type 3 Clone: Type 3 clones are created because of additions and deletions of lines in code fragments. If one or multiple lines of code is/are added or deleted from Type 1 and Type 2 clones, Type 3 clones are formed. In Figure 2.3[1], these code fragments are considered as Type 3 clone.

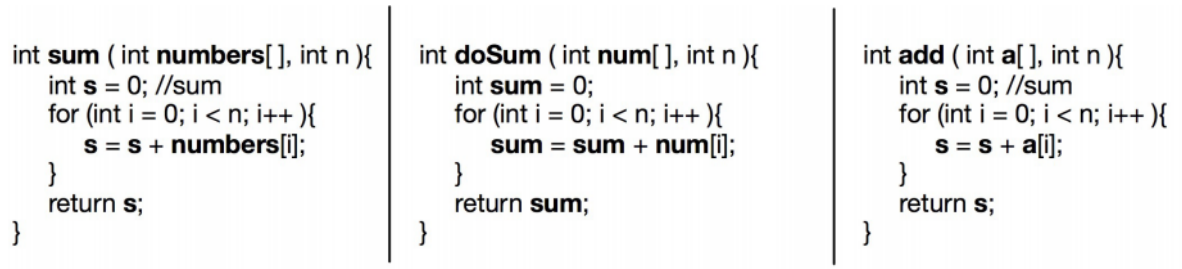


Figure 2.2: Example of Type 2 clone code (adapted from [134]).

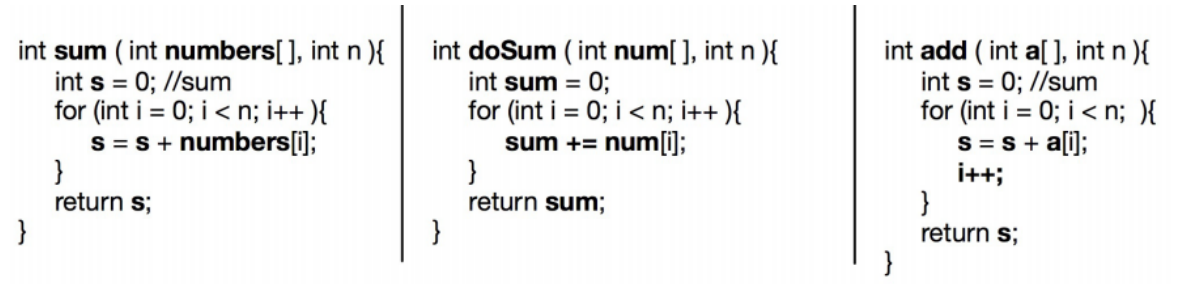


Figure 2.3: Example of Type 3 clone code (adapted from [134]).

Type 4 Clone: In general, Type 4 clone fragments perform identical task but with different coding convention. It denotes that Type 4 clones are semantically similar but not in syntax (i.e., one problem statement and it can be solved by different implementations). Figure 2.4[1] shows two different method (using loop and recursion) to solve same problem.

2.1.2 Impacts of Code Clones

Whatever the motives and sources of clones, software developers often copy and paste code in their project codebase both in open-source and industrial sectors. Existing researches show that the percentage of code

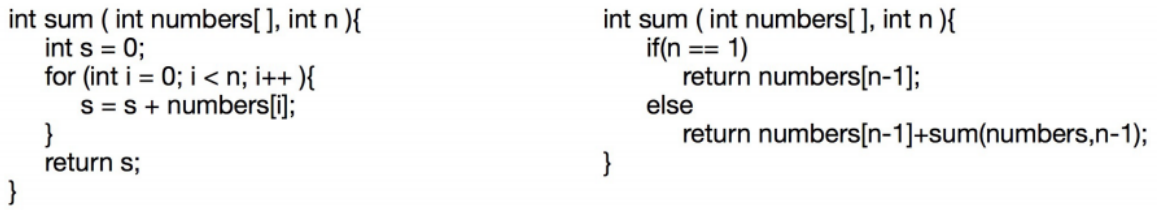


Figure 2.4: Example of Type 4 clone code (adapted from [134]).

clones may vary from 5% to 23% in both development environment (open-source and industrial) [44, 92, 98, 110, 116, 128] where another study [60] shows that it can be up to 50% of the entire software codebase.

A plethora of studies and experiments were conducted on exploring the consequences of code cloning in software development and maintenance. While several studies [35, 74, 83, 95, 99, 107, 108, 109] showed the positive effects of code cloning, there are other studies [40, 70, 93, 95, 115, 121, 122, 136, 137, 183] that prospected negative results of cloning. In addition, there are numerous studies [36, 40, 75, 88, 89, 138, 144] that investigated the relationship of cloned code and bugs. In this thesis, we explore the relevance of code clone and software bug in the perspective of mobile apps. If a code fragment contains bug and then it is copied in other files or other projects without knowing the bug, it must increase the maintenance cost during the system evolvement. On the other hand, if one clone fragment is modified, the same modification need to be done to other clone fragments of same clone class to secure consistency. If the modifications are not propagated properly because of developer unwariness or other possible reasons, software system might collapse and require huge cost to recover it.

In this thesis study, we also analyse code clone in multiple mobile apps and discover the level of required post maintenance and bug-proneness of clone code through system development lifetime. We got negative consequences of code cloning and also clone code are more bug-prone than non-clone code. We will discuss the study in details in Chapter 4.

2.2 Clone Detection Techniques

To study the consequences of clone and process of software maintenance regarding software bugs occurred by clone or reused code, we need to detect clone fragments from software system first so that we can handle them properly. There are several existing clone detection tools including those of cross-language clone detectors [29, 148] and also continuing research to make it more appropriate. The existing tools follow different methodology to detect clone snippets. Here, we discuss these methodologies in the following subsections.

2.2.1 Textual-based Approach

In this way of code clone detection, the source code of software system is considered as a set of characters and independent of programming principles and methodology. Several researches [48, 94, 113, 125, 127, 163, 196,

204] follow this approach and these tools are also worked even where source code is not able to compile. Some of those studies [94, 113, 125, 127, 204] even do not apply any process of source code transformation prior to contrasting two separate code snippets. As a result, these tools do not give satisfactory result when same syntactical structure is addressed differently in various places. To identify significant level concept clones, Marcus and Maletic [127] applied latent semantic indexing (LSI) procedure to source text. Nonetheless, they considered just comments and identifiers ignoring the whole source code. On the contrary, NiCad [163] is a hybrid clone detection tool that use tree-based structural analysis based on lightweight parsing and text based comparison to implement pretty-printing, code normalization, source code transformation and filtering. That properties of NiCad take out the downsides of the past textual approaches. In addition, a cloud based code clone analysis tool named Clone Swarm [39] internally uses NiCad to help the outside community of software clones. Besides, NiCad+ [62] provides the faster version of previous NiCad [163] which performs better in large scale clone detection without effecting its recall and precision. Another variant of NiCad is SimCad [197] that uses a similarity preserving data hashing technique, Simhash for scalable detection of near-miss clones. CloneWorks [187] also evolves from NiCad which is designed to detect both exact and near-miss clones from large repositories even using standard hardware. CloneWorks uses our fast and scalable partitioned partial indexes approach, which can handle any input size on an average workstation using input partitioning.

2.2.2 Lexical-based Approach

According to lexical approach, source code is considered as a set of tokens. This approach mainly suitable for code fragments who have small difference such as identifier renaming. Dup [37], CCFinder][97], iClones [81], SourcererCC [175] are some examples of token-based clone detection approach. Similar approach LVMapper [206] converts the token into sequence and apply alignment approach of bioinformatics to find two similar sequence with more difference. This method inspired by the idea of the seedand-extend method in bioinformatics. SAGA [114] is another efficient and large scale clone detection technique designed with sophisticated GPU optimization where source code is converted into suffix-array after tokenization. For detecting large-gap clone, a token based CCAligner [199] tool design a novel e-mismatch index and asymmetric similarity coefficient for similarity measurement.

2.2.3 Tree-based Approach

In tree-based approaches [44, 45, 50, 61, 92, 104, 105, 112, 114, 191], first source code is converted into parse tree or abstract syntax tree (AST) and then detect clones of similar sub-trees using different matching algorithm of tree data structure. It needs to traverse trees while making comparison that causes higher time complexity than text or toked based approach. Another study CLCDSA [148] uses syntactical features and API documentations as AST and then apply deep learning techniques to find cross language code clones.

2.2.4 Graph-based Approach

In graph-based clone detection approaches [63, 80, 82, 109, 216], a program dependency graph (PDG) is generated from software source code. These methods are aimed for very simple modifications for example reordering of lines. Source code syntactically need to be correct and these approaches has programming language dependencies. Like tree-based approaches, it has also high time complexity.

2.2.5 Metrics-based Approach

The metrics-based clone detection tools [53, 67, 128, 177] use several metrics such as names, layouts, expression and control flow of functions. Poster [185] uses a simple Jaccard-based clone similarity metric which make the tool fast, scalable and user guided clone detection. Metrics-based detection approaches have also been used to find clone in webpage or web documents [46, 57].

2.2.6 Learning-based Approach

Several clone detection method [84, 91, 100, 148, 201, 210, 211] use different machine learning and deep learning techniques for example Support Vector Machine, Neural Network, Weighted Recursive Autoencoders (RAE). Authors [30, 173, 179, 180] also use learning algorithm, information retrieval and software metrics to find semantic clones (i.e., Type 4) where most tools work well up to Type 3.

The concept of clone detection is not restricted to software source code. It has also influenced on clone detection in binary executable [170], Bytecode in Java [101, 102, 176], assembly language instructions [54, 55], software requirement specification [59, 119] and many more fields.

2.3 Code Stability

The concept of *Stability* [83, 108, 109, 123] introduces to evaluate the change-proneness of a software system. In general, the stability of a software codebase refers to the amount of maintenance needed to amplify code elements during the software evolution. The code with low stability needs more maintenance and is change-prone than the code with high stability. Existing researches introduced this term to quantify the stability between clone and non-clone code. There are several matrices to measure the stability in different viewpoints, which are summarised in two broad categories.

2.3.1 Stability in respect to code-changes

According to studies [69, 83, 108, 123], stability in terms of code-changes between clone and non-clone code is calculated in two ways:

1. Determine the rate of lines of code added, modified, and deleted from codebase throughout all the revisions of a software system. The higher code-change rate indicates lower stability (or high instability) [69, 108, 123].

2. Determine the modification frequency, i.e., how many times or operations added, modified, and deleted code from the codebase. The higher the modification frequency of a code region, the less stable it is [83].

2.3.2 Stability in respect of code-age

The approaches [109, 136] calculate the average changed dates of code of a software system. The older the average changed date, the more stable it is. The following example shows how to calculate average changed date [136].

If we consider a codebase of multiple lines of code and five lines of that codebase are changed at different times, creating separate five commits or revision dates 5-Jan-2018, 8-Jan-2018, 15-Jan-2018, 22-Jan-2018, and 29-Jan-2018, respectively. Then, the average date is calculated by the average day differences from the older commit date to all other dates, i.e., the day differences between 5-Jan-2018 and 8-Jan-2018, 5-Jan-2018, and 15-Jan-2018, and so on. So, the average day difference is (3+10+17+24)/4 = 13.5, and thus the average date is 13.5 days later to 5-Jan-2018, which is 19-Jan-2018.

In this thesis work, we use two approaches for code clone detection: NiCad [163] and CCFinderX [96]. Using NiCad, we detect clone code from several open-source and industrial mobile apps of two programming languages (Java and Swift) to find the consequences of clone code in terms of software maintenance and bug-fixing. NiCad can detect three types of clone (i.e. Type 1, Type 2 and Type 3) separately. It is much more easy to install and use NiCad as well as detect iOS (Swift) apps' clone code without further setup requirements. For another work, we need to find cross clones between SO answers' code fragments and changed code fragments of each revision of mobile apps. We can not use NiCad this situation because NiCad works as a full parser and the source code must compilable where SO code fragments might have syntax errors or not complete code. So, we use CCFinderX which is a widely used token-based clone detector and can work even source code contains programming errors.

2.4 Machine Learning Algorithms

Machine learning [10] is a concept that focuses on the systematic study of algorithms and statistical models to get the best accuracy to intimate the way human beings learn. It takes control of greater part of today's Computer Science and Artificial Intelligence (AI) and emerging component of data science. The learning is classified into three main domains i.e., Supervised, Unsupervised and Semi-supervised and they handle both labeled and unlabeled data to train their learning process. Supervised learning trains its algorithm by using labeled data and perform cross validation to classify data or predict accurate outcome with several trial and error (also called Reinforcement learning). Unlabeled dataset are handle by Unsupervised learning when algorithm discover hidden patterns to categorise them. Where semi-supervised learning has less labeled data which help to extract the feature from unlabeled data. Here, we include a brief discussion of some supervised and unsupervised algorithms used in this thesis study.

2.4.1 Naïve Bayes Classifier (NB)

Naïve Bayes is a simple probabilistic classifier but powerful statistical technique that works on Bayes Theorem [8, 159] with an assumption of independence among predictors. It does consider the conditional independence property exists among the features in a class. It helps to simplify the calculation, and that's why it is called Naïve. From Bayes theorem we know that:

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} \tag{2.1}$$

Here, X is the evidence variable (represents the features) and Y is the target variable. From Bayes theorem, we can calculate the probability of Y, given that X is already known. X is defined as: $X = x_1, x_2, ..., x_n$

Here, $X = x_1, x_2, ..., x_n$ represents the features of a training instance. If we substitute the value of X into (2.1) and expand the equation using product rule, we get:

$$P(y|x_1, x_2, ..., x_n) = \frac{P(x_1|y)P(x_2|y)...P(x_n|y)}{P(x_1)P(x_1)...P(x_n)} \tag{2.2}$$

Now, we can calculate the values for each of numerator by looking at the dataset using maximum likelihood approach and substitute them into (2.2). The denominator is fixed for all entries in dataset. Hence, the denominator can be removed as considering it constant, and we get the following equation:

$$P(y|x_1, x_2, ..., x_n) \propto P(y) \prod_{i=n}^{n} P(x_i|y) \tag{2.3}$$

Equation (2.3) will give us probability values for all classes. Given the predictors, now we can find the class label using the following equation:

$$y = argmax_y P(y) \prod_{i=n}^{n} P(x_i|y) \tag{2.4}$$

Gaussian Naive Bayes (GNB)

(GNB) [8] is a case of Naive Bayes supports real-valued attributes with an absolute assumption of having a Gaussian or normal distribution given the class label i.e., dealing with continuous data. For example, suppose that i^{th} attribute is continuous and its mean and variance are represented by $\mu_{y,i}$ and $\sigma^2_{y,i}$, respectively, given the class label y. Hence, the probability of observing the value x_i in i^{th} attribute given the class label y, is computed by (2.5), that is also called as normal distribution.

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma^2_{y,i}}} exp^{(-\frac{(x_i-\mu_{y,i})^2}{2\sigma^2_{y,i}})} \tag{2.5}$$

2.4.2 K-Nearest Neighbors Classifier (KNN)

K-Nearest Neighbors (KNN) [212, 213] is a simple, non-parametric scalable algorithms in machine learning where, K denotes the number of nearest neighbors. Suppose x is an instance for which we will predict the class label. First, we find the k closest instances to x. After that, we count total numbers for each separate class from closest instances. Then we apply majority voting technique to find the class with the highest vote and it will be the final prediction of the model. Figure 2.5[2] illustrates this concept. For finding closest similar points, we normally calculate the distance between instances using distance measure techniques such as Euclidean distance, Hamming distance, Manhattan distance and Minkowski distance.

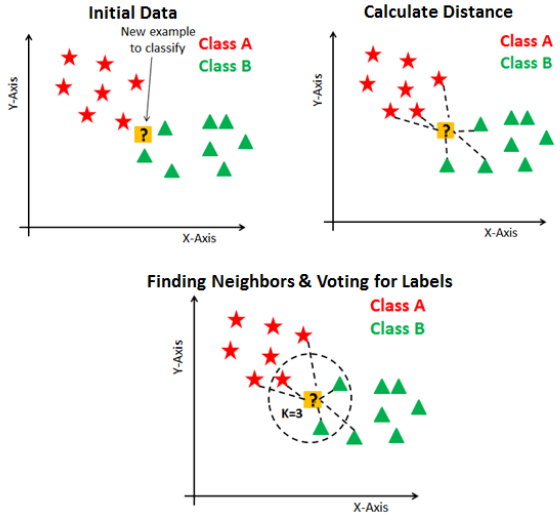


Figure 2.5: K-Nearest Neighbors approach on classification

The value of K is considered as the main factor for classification. Research [212] has shown that no fixed value of K is perfect for all types of datasets. We must carefully select the value of K, so that the accuracy will be maximum. There are several ways to select the appropriate value of K. A small value of K means that noise will have a higher influence on the result and a large value make it computationally expensive. Cross-validation [90] is one of the technique where divide the entire dataset into training set and test set, then apply KNN for different k value into training set and cross validate it with test set and observe the performance. Another approach called Elbow method [7] is a heuristic used in determining the number of clusters in a data set. According to this method, we need plot a graph with distortion and value of K starting from 1 to $\sqrt{n}$, where n is the total count of the data points and take the elbow value for the optimal number of clusters. In Figure 2.6[3], clusters number is either 2 or 3.

[2]https://www.datacamp.com/community/tutorials/k-nearest-neighbor-classification-scikit-learn
[3]https://pythonprogramminglanguage.com/kmeans-elbow-method/

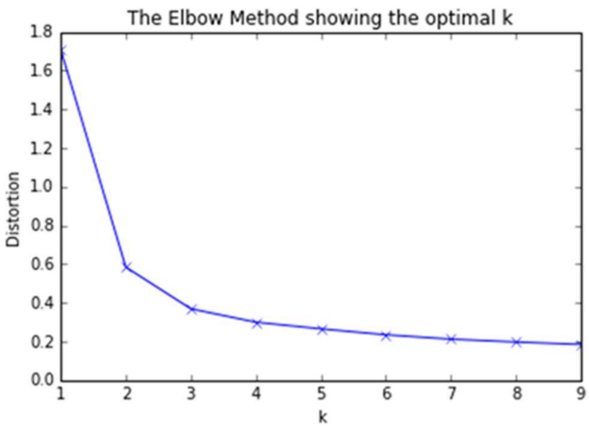


Figure 2.6: Elbow method plotting for optimal value of K.

2.4.3 Decision Tree (DT) Classifier

Decision Tree [171] consists of two types of nodes: internal node and leaf node. Internal node represents the feature or attribute and leaf node denotes the decision of classification. The edge between nodes represents the decision logic. DT algorithm works as follows:

1. Select the best feature based on some feature selection techniques to split the instances
2. Make that feature as a node and break the dataset into smaller sets
3. Repeat process 1 and 2 recursively until all the instances belong to the same feature value or all the attributes are used for splitting the dataset or all the instances are used to make a decision.

It is common to use the concept of entropy in Information Theory [52] to find out the best feature. Entropy denotes the impurity in a group of instances. We need to calculate the information gain to select the best feature. For this, we measure the entropy before the split and after the split. By the difference of previous two values, we calculate information gain and it is the decreased value in entropy. We calculate information gain for all the features accordingly and the feature with the most information gain score will be selected to split the dataset. Sometimes Gain Ratio and Gini index are also used to select the best feature.

2.4.4 Random Forest (RF) Classifier

Random Forest [117] classifier creates multiple individual decision trees using ensemble learning methods and merges them together to get a better and stable prediction. From individual trees in random forest, we get a class prediction and majority voting technique is used to find the the ultimate prediction of the model. Figure 2.7[4] illustrates this concept. This classification algorithm uses two key concepts which are as follows.

1. Make trees using random sampling of training instances and

[4]https://medium.com/@williamkoehrsen/random-forest-simple-explanation-377895a60d2d

2. Randomly selected feature's subset is used for splitting nodes.

At the root of each tree, it first splits the data points in such a way that the class differences in each part of data becomes as small as possible. It repeats this process until the depth of tree reaches to it is maximum length of the leaf nodes contain data points of a single class. Here is the procedure[5] of Random Forest classifier:

1. Create multiple decision trees using random sampling of training data.
2. Predict the class label for the test features using each randomly created decision tree
3. From predicted class labels count the votes for each class
4. The class with the majority votes will be considered as the final prediction of the model.

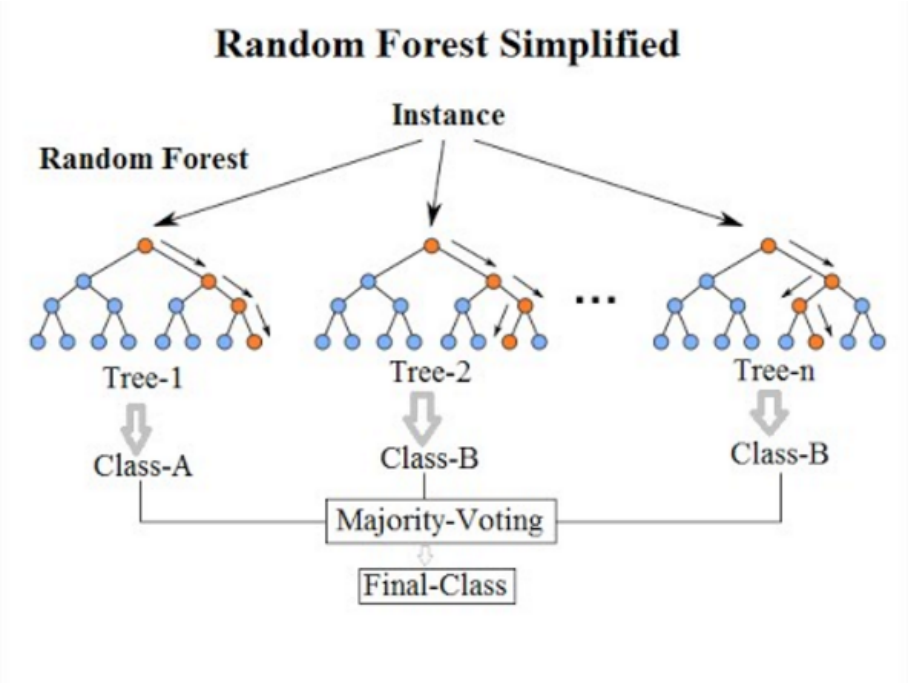


Figure 2.7: Random Forest Classifier

2.4.5 Support Vector Machine (SVM)

Support Vector Machine classifier [184] takes data points as input and gives output a line (or hyper-plane) that separates classes if possible. There are many possible hyper-plane could be drawn (in Figure 2.8[6]) to classify data points into two separate classes. During each step of the classification process, SVM tries to find a hyper-plane that has maximum margin, i.e., the distance between data points of two classes will be

[5]https://medium.com/@Synced/how-random-forest-algorithm-works-in-machine-learning-3c0fe15b6674
[6]https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47

maximum. Hyper-planes are considered as decision boundaries that help to classify data points. The data points which goes through the margin line of each class are known as support vectors showing in Figure 2.9[6], and the original classification line lies between the middle of both margins.

For multi-class classification problem, SVM will classify data points into $class-1$ and other $n-1$ classes will be in a single class. After that, it will take those $n-1$ classes and find the second class from it while keeping $n-2$ classes in the same class. This process will continue until SVM completes the classification of all the data points into a class. The controlling parameter C tries to keep balance between smooth decision boundary and correctly classification of training examples. Large value of C will classify more data points correctly and vice versa. The controlling parameter *Gamma* denotes how a single data point (considered as outlier) affects on the classification accuracy. High value of *Gamma* ignores data points that are considered as outlier in decision boundary and vice versa.

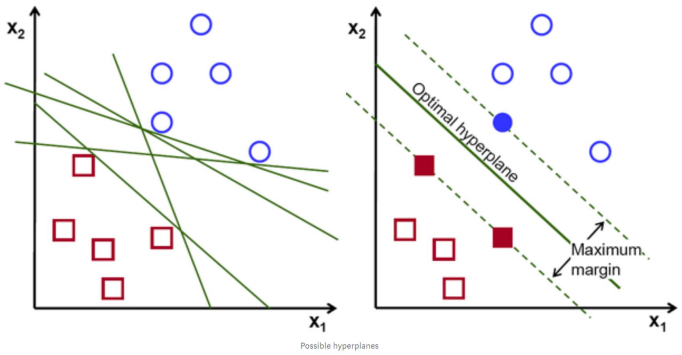


Figure 2.8: Support Vector Machine Classifier

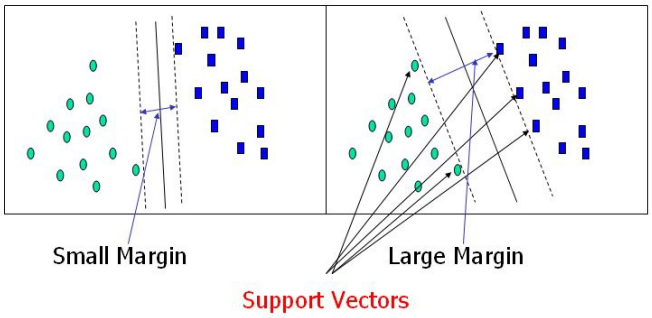


Figure 2.9: Support Vectors in SVM

2.4.6 K-means clustering Algorithm

Clustering is the way of splitting data points into many groups where similar data points belong to the same group and data points of one group is dissimilar to the data points of other groups. K-means clustering [9, 76] is one of the most common and simplest unsupervised algorithm in machine learning. It is normally used when we have data without predefined classes or categories, and we want to classify data points into some clusters where K denotes the number of clusters. K-means is an iterative algorithm that tries to keep

the inter-group data points as similar as possible while also making the groups as dissimilar as possible. It places data points into a cluster in such a way that the sum of the squared distance between data points and the centroid of the cluster will be minimum. Here, centroid is that point which holds the arithmetic mean of all the data points of a cluster.

Given a set of observation $(X_1, X_2, ..., X_n)$ where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into $k(\geq n)$ sets $S = \{S_1, S_2, ..., S_k\}$ so as to minimize the within-cluster sum of squares (i.e., variance). The following (2.6) finds the objectives of k-means where μ_i is the mean points in S_i.

$$\underset{S}{\operatorname{argmin}} \sum_{i=1}^{k} \sum_{x \epsilon S_i} ||x - \mu_i||^2 = \underset{S}{\operatorname{argmin}} \sum_{i=1}^{k} |S_i| Var S_i \tag{2.6}$$

Like KNN (in Section 2.4.2), too choose the right value of K mostly Elbow method is used. From the sum of square distance between data points and their assigned cluster's centroids we get a curve. We pick K at the point where sum of square distance starts to flatten out and forming an elbow. The Figure 2.6 illustrates this concept.

2.5 Statistical Significance Tests

We use several non-parametric statistical tests to validate our study findings. The tests are briefly discussed in the following subsections.

2.5.1 Mann Whitney U Test

Mann Whitney U test (also called Wilcoxon Rank Sum Test or Mann Whitney Wilcoxon Test) is a popular non-parametric test to compare results between two independent samples and also can tell whether the samples are likely to derive from the same distribution [11]. It compares two group of data (either ordinal or continuous) that allow statistical inference without making the assumptions that the sample has been taken from a particular distribution (i.e., normal).

Initially, the data points of each group are sorted into ascending order and then each data point must get a rank value. The smallest data points of both group receive a rank value of 1.0, the second smallest are ranked with 2.0 and this continues until all data points are properly ranked. If there are multiple data points of same value are assigned the median rank of the entire identically sized group. For example, if the smallest data value appear twice, then both of them receive the rank of 1.5. It implies that the rank of 1.0 and 2.0 has been utilized and that the following highest data point has a rank value of 3.0. Again, if the least data point value seems three times, then each value is ranked of 2.0 and the next greatest data has been ranked as 4.0 and so on. Finally the groups are being compared by the sum of all ranks of each group. According to Mann-Whitney, a test statistic U is calculated by this following equation 2.7.

$$U = n_1 n_2 + \frac{n_2(n_2+1)}{2} - \sum_{i=n_1+1}^{n_2} R_i \tag{2.7}$$

Where n_1 and n_2 are sample size of two groups and R_i is the rank of each sample data points. We also measure the Critical U value with a specific level of significance and compare with statistics U to reject/accept null hypothesis. The significance level (a.k.a. alpha or α) is a strength measurement factor of a particular evidence that surely available in data samples before reject/accept the null hypothesis [21]. If we reject the null hypothesis , it concludes that the impact is statically significant. So, α value is the likelihood of rejecting the null hypothesis when it is valid or true. For instance, we consider 0.05 as significance level which indicates a 5% risk of inferring that a difference exists where there is no real difference. Before the rejection of null hypothesis in lower significance levels, we require strong grounded proof. In this thesis, we use 5% level of significance (i.e., $\alpha = 0.05$) and find the Critical U value from statistical table [6] where the sample size of two groups are the row and column value of corresponding table.

2.5.2 Willcoxon Signed Rank Test

Like Mann Whiteney U test, Willcoxon Signed Rank test is a non-parametric test which is designed to find difference between two treatments or conditions where there is co-relation between the samples [19]. It can differentiate conditions for same size of samples (Where Mann Whitney U test also works different sized data) and tells how different two sets are from one another to establish statistical significance between the two distributions. We also use 5% level of significance and find the Critical U value from statistical table [6].

2.5.3 p-value

A p-value is the probability which would obtain the impact observed in samples, if the null hypothesis is valid for the populations [18]. p-values are determined based on sample data points and the assumption of the validity of null hypothesis. Lower p-values show more prominent proof against the null hypothesis. Using significance levels at the time of hypothesis testing, we also calculate p-value to compare with selected significance level. If the p-value is less than the significance level then we can reject the null hypothesis and summarised that the samples are statistically significant.

2.5.4 One-tailed and Two-tailed Test

In terms of test statistic, one-tailed and two-tailed test are another effective ways of testing statistical significance of a parameter construed from a sample [15]. If the test statistic is symmetrically distributed, we can pick one of three alternatives hypotheses. Two of these direct to one-tailed tests and one relates to a two-tailed test. Nonetheless, the p-value introduced is (almost always) for a two-tailed test [16]. If we use 5% significance level (i.e., $\alpha = 0.05$), a one-tailed test distributes all of α in the one direction of interest to test the statistical significance where a two-tailed test allots half of α in one direction and rest half of α

in other direction. So, in two-tailed test, we are testing the possibility of the relationship in both direction where one-tailed test completely disregards the possibility of one direction.

2.5.5 Cohen's kappa Coefficient (k)

Cohen's kappa coefficient (k) [4] is a statistical measurement used to evaluate the agreement (also called inter-rater reliability) between two raters. Simply, it provides a quantitative measure of reliability for two raters' agreement precisely on the same thing. The value of k varies -1 to $+1$. A score of $k = 0$ implies a random agreement between raters, whereas $k = 1$ means perfect or complete agreement. Moreover, a score of $k = -1$ or less than 0 denotes less agreement than random chance. The meaning of k is formulated by the following equation 2.8.

$$k = \frac{p_0 - p_e}{1 - p_e} = 1 - \frac{1 - p_0}{1 - p_e} \tag{2.8}$$

Where p_0 = the relative observed agreement amount raters and p_e = the hypothetical probability of chance agreement. Cohen's kappa coefficient is also used in classification tasks and imbalanced class problems [5].

In this thesis study, we use Mann Whitney U test for testing statistical significance between two group of paired data (sample size are same) and Willcoxon Signed Rank test for different sample size groups. Both cases, we use 5% level of significance and two-tailed test and hence, we calculate p-value and statistic U and find the Critical U according to significance level value from statistical chart [6]. If the calculated p-value is less than 0.05 and statistics U is less then Critical U, then we can say that the data groups are not being taken from same distribution and they are statistically significant.

2.6 Conclusion

In this chapter, we introduce several fundamental concepts and terminologies that would help to understand and follow the rest of the thesis. We mentioned different types of code clones, impact of clone in software maintenance and evolution and several strategies of code clone detection approaches. We also describe a couple of machine learning algorithms (i.e., Naive Bayes, Decision tree and so on) that we use in our thesis study. Finally, we discuss some statistical significance tests to verify our study findings. From the next chapter, we will represent each of our study and analysis of this thesis step by step.

3 Exploring Classification of Software Bug Reports Towards Mobile Applications

Software bug reports are significantly used in a plethora of research areas such as bug classification, bug prediction, bug triaging and so on. However, unlike traditional desktop-based software systems, there are very a few studies (to the best of our knowledge) about the classification of bug reports and analysis of bug severity from the perspective of mobile apps. This chapter explores several machine learning algorithms to build and evaluate classifiers capable of classifying newly incoming mobile applications' bug reports into four classes (i.e., crash bug, energy bug, functionality bug, and security bug). Appropriate classification result assists developers and maintainers to quickly identify and understand the newly occurred bugs and hence, take necessary steps or allocate resources to fix them. To build classifiers, we collect 2,700 bug reports from diverse subject systems of Android and iOS apps and label these bug reports by hand into predefined classes. We prepare a distinctive feature set that is then fed into a number of classification algorithms (Gaussian Naïve Bayes, K-Nearest Neighbors, Decision Tree, Support Vector Machine, and Random Forest) to find the best classification model. Among the distinctive algorithms, the Support Vector Machine classifier performs the best with a promising f1 score (91%) and a low error rate (4.7%) with the compiled data set.

The rest of the chapter is organized as follows. Section 3.1 has introductory discussion and motivation. Several related works associated with this field are presented in Section 3.2. Section 3.3 contains the data description and dataset preparation. Section 3.4 talks about experimental results and analysis that we found. Section 3.5 presents the discussion of experimental findings and comparison with other existing works. Section 3.6 reports the possible threads to validity and finally, Section 3.7 concludes this chapter and proposes probable future works.

3.1 Introduction

At the end of 2020, there are more than 3.9 billion people use smartphones to explore different apps [13]. So, each day developers and maintainers are reported a humongous amount of apps' inconsistencies and incompatibility issues in their development sites. These issues carry bug reports, enhancement requests, compatibility suggestions, etc. Manual analysis of bug reports requires both time and resources, which can delay the decision-making about a bug where immediate actions are needed. As mobile apps usually follow Event-Driven Architecture [130], the bug in one event might affect other events as well if there are

dependencies among them. So, appropriate decisions and actions are demanded by developers and maintainers in the context of bug management. To help developers categorize mobile apps' bug reports based on specific domains, we investigate Android and iOS applications' (mobile apps) bug reports to build a prominent classifier that helps identify and understand the bug severity.

User interaction with mobile applications plays a significant role in helping developers and maintainers know app behaviors and development faults. As it is popular to use Event-Driven Mechanism [130] for building mobile apps, each operation is treated as an event, and several events are connected based on their dependencies. So, to minimize the inconsistencies, developers need necessary synchronization among these unique events that are immensely occurred when users start using apps. In the evolution of mobile apps, clients or users report the inconsistencies of a particular app in the development sites (for open-source apps) such as BugZilla[1] , GitHub[2], Google Code[3], Trac[4], and F-droid[5]. So, the complexities are found to categorize the reported bugs or enhancement requests because of using different formats and styles of bug reports. While finding bug reports from different open-source sites, we experienced several issues regarding formats and styles with multiple questions answering mechanisms, but users hardly maintained the report structures to respond to appropriate bugs or any contradiction. As different users report apps' disharmony in different ways, so we need an automated classifier that helps developers and maintainers to understand the bug types and take appropriate actions to resolve.

Unlike desktop-based software systems, mobile applications have several types of bug such as security bug, concurrency bug, compatibility bug, energy bug, functionality bug and app crash bugs. [207], which might not be prevalent for desktop-based apps. For example, we found energy-related bugs/issues (battery drain while using apps), which is considered one of the key concerns of user satisfaction, compatibility constraints (different behaviors of the same app in different devices), and other functional and security issues (e.g., GPS, device configurations, authentication, and secure network) that arise only upon user interactions. Developers usually have information on these types of bugs in the form of bug reports. A bug report has three main sections: bug ID or issue ID, bug title, and bug description. We analyze the bug title and description after some preliminary processing to classify them into predefined classes. Sometimes, difficulties arise when there is inadequate information of a bug report, such as having only a bug title with no description or a very brief description. To complete this task, we collect around 3K bug reports from several open source development sites such as GitHub, Google Code, F-droid. All the bug reports can be accessed in this link.

From bug reports, we choose some meaningful words as features that are appropriate for the detection of each class label, and we prepare a feature set using the bag-of-word approach. We discuss a brief description of each class type and, for better understanding, also show an example of each type of bug report in Table 3.1.

[1]https://www.bugzilla.org/
[2]https://github.com/
[3]https://opensource.google/
[4]https://trac.edgewall.org/
[5]https://f-droid.org/

Table 3.1: Sample bug report properties and predefined class labels of classifier.

Project Name	Issue ID	Title	Description	Class Label
Wordpress	6668	IndexOutOfBounds Exception in ImageSettingsDialogFragment	Open image settings Tap the Width field.Input 10.Press confirm on the soft keyboard.The width field should display "10px" Tap on the field again and delete the "10px". Notice the crash. Fatal Exception: java.lang.IndexOutOfBoundsException: setSpan (2 ... 2) ends beyond length 1.	Crash
K9Mail	6894	K9mail is eating lots of battery power	K9mail is a powerful mail client, but it is too much power hungry! I test it on several smartphones, k9mail is consuming around 10% (the screen is consuming 14% on my Xperia M4 for the same period) of my battery for a 24 hours period... In the same time, Yahoo Mail app is consuming 2% and Gmail less than 1%.	Energy
Osmdroid	690	Secondary tile overlay does not always display tiles	I'm using osmdroid 5.6.5 in my Android application. I'm using a custom base layer, and I add a second layer above it: Everything is working fine, but when scrolling the map, sometimes there are some tiles not show properly. When scrolling a little bit further, they are loaded. Is this a known bug	Functionality
Connectbot	76	ECDSA 521 bit keys auth always fails	Trying to authenticate the client with a 521 bit sized ECDSA key signature always fails on the client side (it fails to format the signature to the ssh format, and doesn't send anything).	Security

Crash Bug: There are a couple of reasons for app crashes like out of indexing, i.e., such as memory management, error condition and exception handling, network management and so on. However, users' feedback are most important in this situation because they find inconsistencies while interacting with the apps.

Energy Bug: In the development stages, developers must think about the energy efficiency or power consumption which is one of the most concerning parts of users' satisfaction while using an app. Estimating energy consumption for android apps can make the developers aware of the energy cost of each code module in their apps.

Functionality Bug: It mainly means functional inconsistency between the expected result and the actual result. It is the most common type of bug that occurs more frequently until the final evolution of apps. It also includes fragmentation issues (apps are required to work perfectly on different operating system versions and devices), the absence of necessary synchronization among distinctive events, and different behaviors in different systems of the same apps.

Security Bug: Many confidential and private information (for example, contacts, accounts info and so on) is available on mobile device, creating security complexities. In addition, there are several security issues like permission over-privileged, permission re-delegation, network attacks, ICC attacks, app clones and many more. Any inconsistency of these issues is considered a security bug.

As the collected bug reports are unlabeled, first we apply K-means algorithm to group them into four classes. However, because of features overlapping among bug reports, the clustering result is not much satisfactory. In consequence, we label bug reports manually though it is a strenuous and time-consuming task. We group the collected bug reports into security, crash, functionality and energy bug. Concurrency and compatibility bug are also considered as part of functionality bug. Then we apply a number of supervised classification approaches. For training and validation purposes and to lower the validation error rate [73], we split our dataset as 70% and 30% for training and validation, respectively. The accuracy result of the classifiers is acceptable, ranging from (88%-91%) except for Naïve Bayes, where the Support Vector Machine (SVM) classifier has the highest accuracy.

3.2 Related Work

In software development, bug reports play a vital role in several tasks like the prediction of future bugs [77, 181], triaging reported bugs [147, 192] and categorization of bugs into some specific types [194]. Therefore, it is crucial to have excellent and clear descriptive bug reports for perfect alignment. There are a number of studies related to the classification of bug reports, but a few that addressed mobile app bug reports, and no studies had class labels more than three. Nevertheless, we consider four categories of bug reports that are significantly relevant for mobile apps in the classification task in this work.

Antoniol et al. [32] proposed an automated classification approach. Their techniques asserted that only textual content of bug reports is enough for classifying bugs and non-bugs reports, i.e., their classification task had two class labels. They also addressed misclassifying bug reports, which creates severe complexities on the bug management system. Herzing et al. [79] extended this research and said only textual content is inadequate and unreliable because sometimes prediction result is biased. However, they [32, 79] worked with the issues which were mixed with bugs and non-bugs but we consider only bug reports which are labeled by the developers and maintainers.

Otoom et al. [150] constructed a classification model that could detect bug reports into two categories, i.e., corrective (defect fixing) and perfective (major maintenance). This method was almost similar to Antoniol et al. [32] because they also tried to find which issues are bugs that are corrective and which are non-bug that is perfective. However, the difference was that they tried to help maintainers quickly understand major non-bug issues so that they (maintainers) would allocate resources to solve them (major non-bug issues) immediately. They investigated three different open source desktop projects and used manual labeling to categorize bug reports. They prepared a feature set and fed it into different classification algorithms (Support

Vector Machine, Naïve Bayes and Random Trees) and achieved high average accuracy (93.1%) with the SVM classification algorithm. Our work considers only bug reports to be categorized into different classes so developers or maintainers can easily take necessary actions based on bug types (crash, energy, functionality, or security).

Katerina et al. [72] built an automated classifier of software bug reports related to security and non-security, using both supervised and unsupervised approaches. The limitation is that their dataset had a smaller number of security-related bug reports. That is why their dataset did not provide as good results as expected (only 25% of the data provides as good results as training on 90% of the data).

Bo Zhou et al. [214] focused specifically on the difference in bugs and bug-fixing processes between desktop-based software and smartphone application. They analyzed 88 open source projects on desktop, android, and iOS. Their main task was how developers and maintainers behave with the bug-fixing process of multiple platforms, understanding the nature of bugs, similarities, and differences between desktop-based software and smartphone application.

Gegick et al. [68] proposed an automated classifier that trained by a statistical model using text mining on wrongly hand-labeled security bug reports. The authors used the SAS text mining tool to prepare feature vectors, and their classifier found several security bug reports that were manually identified as non-security bug reports. They investigated four large Cisco projects and identified that around 77% of security bug reports were miss-classified. However, their classifier experienced a very high false-positive rate varying from 27% to a maximum of 96%. Like this work, [68], another miss-classification research conducted by Wright et al. [205] that identified the non-vulnerabilities bugs in the MySQL database, which were labeled as vulnerabilities bug reports.

There are several works that researchers do to prioritize desktop software issues (bug and non-bug) reports. Herraiz et al. [78] analyzed bug reports of Eclipse and recommended a more straightforward format for bug reports by reducing severity levels to three levels as important, not important, and request for enhancement based on the time taken to close the bug. They also found priority-level clusters based on the same consideration and recommended three classes i.e., high, medium, low. Ahsan et al. [28] proposed an automatic developer prediction system for incoming bug reports. They applied feature selection and feature reduction on the Mozilla bug report. They found the best result in using latent semantic and SVM among all the other machine learning methods they used. They got an accuracy of 44.4% for their classification task. This research told maintainers just priority of issues but did not categorize in problem domains (for example, this issue is a concern to the security of mobile apps).

Lamkanfi et al. [111] implemented text mining algorithms on bug reports of Eclipse, Mozilla, and GNOME to automatically classify them in different severity levels. They got an average precision of 65% in Eclipse and Mozilla and 75% in GNOME, an average recall of 75% in Eclipse and Mozilla, and 85% in GNOME.

All of the related works investigated both bug and non-bug issues of several desktop-based systems. However, no such study classified only bug reports into further potential categories that would help predict

appropriate resources and take immediate actions efficiently. Therefore, we analyzed mobile apps bug reports which might not be prevalent for desktop-based apps, such as energy and compatibility related bugs/issues. Moreover, we classified them into significant classes/categories by specific features presented in the bug report, which assists to know the bug domains, reasons, probable consequences and that developers take immediate actions accordingly.

3.3 Data Description and Dataset Preparation

To tag each bug report that we collect, we apply K-Means clustering approach but get an unsatisfactory result. Hence, based on meaningful words that help to distinguish bug reports, we label manually and finally organize a complete dataset with two thousand seven hundred rows and five columns (four features and one class label). The dataset characterization and preparation steps are described in detail in the following subsections.

3.3.1 Dataset Collection and Pre-processing

We collect bug-related issues from open source developer sites (Google Code, GitHub, and F-droid) and build our own dataset. All the information in a bug report is not necessary for the classification task. Therefore, we must pre-process bug reports to remove punctuations, stop words, numbers, and other special symbols. For this purpose, the bug reports are tokenized into a set of tokens. Tokenization means large texts are split into words. For example, if we tokenize the sentence: *'A large text is divided into a set of tokens'*, we get [*'a,' 'large,' 'text,' 'is,' 'divided,' 'into,' 'set,' 'of,' 'tokens'*]. Before tokenization, punctuations and numbers are removed, and also capital letters are replaced with small ones.

In the next step, we apply the technique stemming. It converts each word to its root form because our feature set for each class contains only root form to minimize the number of feature words. For example, "connect" is the root form of "connection," "connecting," "connected," etc. We also remove the tokens whose word length is less than three. We apply stop-word removal as the final pre-processing step. Stop words in any language are a collection of widely used words. For instance, "the," "is," and "to" are qualified as stop words in English. After stemming and removing stop words, we get the final tokenized version of each bug report.

3.3.2 Feature Selection and Preparation of Feature Set

We scrutinize only bug titles and bug descriptions for the classification task. We consider the attribute "title" that plays the most important role in classifying bug reports. Usually, developers and users set the summary of a bug report as the title, and the maintainers easily get the gist of a particular bug only read the title. From the definition of class label and detail study, we prepare a meaningful set of words to get the prediction of class labels which are shown in Table 3.2. We assume these keywords can distinguish bug reports and

Table 3.2: Significant words for each class label

Class Label	**Word set**
Crash	'indexoutofbound', 'arrayindexoutofboundsexception', 'arithmeticexception', 'crash', 'jsonexception', 'runtimeexception', 'nullpointerexception', 'activitynotfoundexception', 'classcastexception', 'fatalexception', 'eventbusexception', 'stringindexoutofboundsexception', 'numberformatexception', 'memoryexception', 'outofmemory', 'illegalstateexception'
Energy	'consume', 'energy', 'battery', 'power', 'utilize', 'drain'
Functionality	'support', 'display', 'should', 'miss', 'wrong', 'differ', 'invalid', 'correct', 'cutoff', 'work', 'incorrect', 'match', 'delete', 'instead', 'show', 'able', 'unavailable', 'allow', 'access', 'upgrade', 'ignore', 'fail', 'add', 'stop', 'visible', 'active', 'infinite', 'connect', 'lose', 'use', 'handle', 'break', 'hide', 'disappear', 'appear', 'change', 'load', 'expect', 'create', 'disable', 'inconsistent'
Security	'security', 'permission', 'sslexception', 'httpclient', 'phish', 'proxy', 'guaranty', 'clone', 'protect', 'private', 'public', 'signin', 'signup', 'verify', 'password', 'authenticate', 'login', 'credential', 'autocomplete', 'admin', 'leak', 'bypass', 'lock', 'ssh', 'key', 'warn', 'telnet', 'torrent', 'vpn', 'firewall', 'certify', 'cipher', 'socket'

categorize them into different class labels, and words from bug titles are given more priority than description. For example, bug report containing keywords such as *authentication*, *connection* and *SSH* will be classified as security bug report. Table 3.3 shows an example of a bug report and highlighted keywords that help to indicate the class label.

We consider the whole document, i.e., an individual bug report is our instance, and the features are words in that bug report. After pre-processing, we get the tokenized version of that bug report with every token is in root form and then find the words that we have enlisted for class label prediction and treat the presence or absence of each word as a feature. We give special attention to the title of a bug report. If a keyword appears in the title, we increase the token count ten times greater than the keyword that appears in the description of a bug report (as we consider a bug title helps more on classification task, we tried several weights ranging from 2 to 15 and inspect classifiers' performance and for weight = 10, we find significant performance result than other weights). Finally, we sum up all the token values to get a single score of each feature set, i.e., a single report has four different scores (crash score, energy score, functionality score and security score) to form our 2700×4 dataset.

3.3.3 Clustering and Manual Approach for Class Labeling

Before applying K-Means, we manually set the class label of each bug report to justify the accuracy of the clustering approach to label distinguishing mobile bug reports. We classify 2,700 bug reports into four

Table 3.3: The presence of word features in a bug report.

Project Name	Issue ID	Title	Description	Class Label
Zxing	748	Can't **connect** to Tiny **SSH** Server	"Can't **connect** to Tiny **SSH** Server On Android Side: Error Key exchange was not finished, **connection** is closed. Cannot negotiate, proposals do not match. On Server Side: tiny **SSH**: C9jP74r2: info: **connection** from 192.168.0.19:48492 main tiny **SSH** d.c:106 tiny **SSH** d: C9jP74r2: fatal: unable to receive kex-message (protocol error) main tiny **SSH** d.c:148 "	Security

predefined class labels and get the manual classification result which is shown in Table 3.4. After dealing with K-Means clustering, we see that each cluster has a mixture of different class labels. That means we can not distinguish any cluster into a fixed class labels. Figure 3.1 shows the actual class labels with colors and cluster elements with symbols. There are overlaps among the class labels, such as crash bug reports available in all clusters, and other clusters behave similarly. Crash bug reports have more influence on cluster 1, which is 947, but there are also 74 functionality bug reports and four security bug reports in that cluster. The condition of cluster 3 is worsened than others. All the contents of clusters are showing in Table 3.5 for a more precise understanding.

As the result of K-Means has unusual overlapping, we stick to manual labeling. Two graduate students independently and separately investigated the collected bug reports and finally classified them into four groups (Crash, Energy, Functionality and Security bug). We performed Cohens' kappa coefficient [4] to evaluate the level of agreement between the two graduate students regarding the classification task. Cohens' kappa coefficient (k) is a popular statistical measurement used to estimate the inter-rater reliability of agreement between two raters, ranging between -1.0 and +1.0 where a negative value indicates poorer than chance agreement and positive means better than chance agreement (details discussed in Section 2.5.5). Therefore, we determined the coefficient k that shows the level of agreement between the two students to be +0.978, and it is an excellent reliable agreement to justify the categorization [64].

After manual labeling, We apply five supervised classification algorithms (Naïve Bayes, K-Nearest Neighbour, Decision Tree, Support Vector Machine, and Random Forest) on our dataset and analyze the result based on classifier evaluation metrics. In spite of the manual labeling of each bug report, there are pretty much overlapping among the class labels (except crash bug class which is almost separable with others), which is shown in Figure 3.2. This is because the description of the bug report contains lots of meaningful words that might predict multiple class labels at a time, which might affect the classification task.

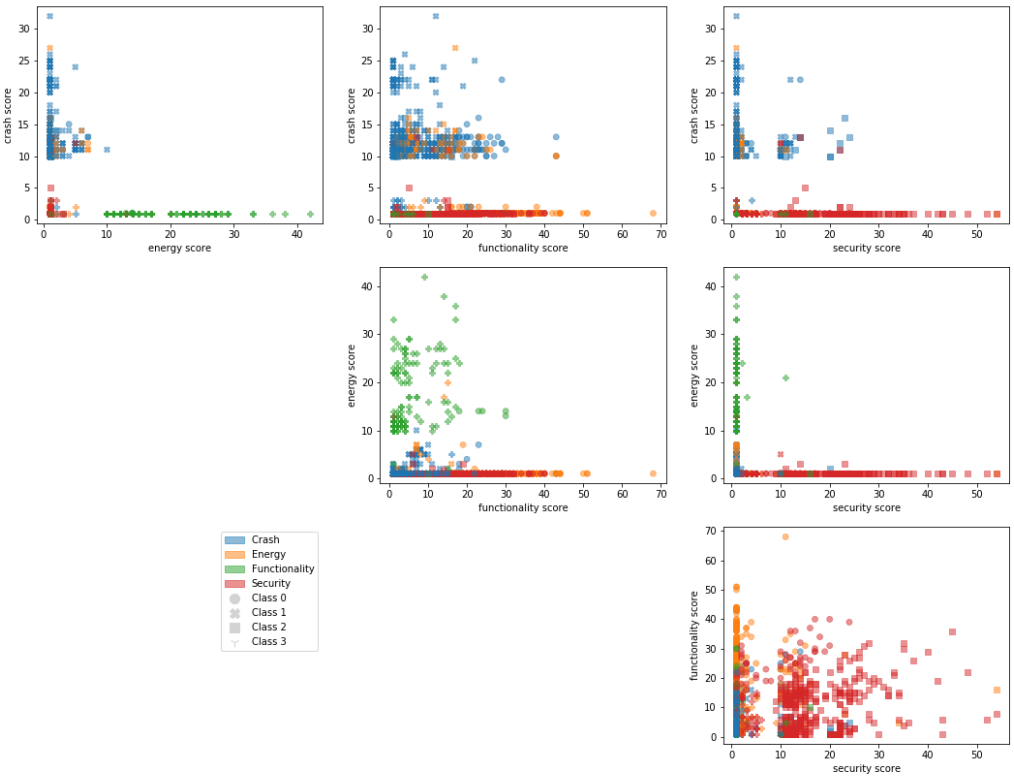


Figure 3.1: Visualization of Clustering result using K-Means algorithm.

3.4 Experimental result and analysis

To find the best classifier to categorize bug reports, we apply five supervised approaches, i.e., Gaussian Naïve Bayes (GNB), K-Nearest Neighbors (KNN), Decision Tree (DT), Support Vector Machine (SVM), and Random Forest (RF) and evaluate their results with several evaluation metrics. Finally, we discuss our findings and observations in the following subsections.

3.4.1 Performance Metrics

For evaluating the performance of each classifier, we consider mainly f1 score and error rate. Besides, we also discuss the classifier's precision, recall, average accuracy, Receiver Operating Characteristics (ROC) curve, and area under ROC curve [154]. In this subsection, We briefly discuss calculation procedures for all the metrics.

To know about these metrics, we must know about the confusion matrix, which is shown in Figure 3.3 and different terminologies are calculated by it are described below:

Table 3.4: Bug Reports of different categories

Class label	Manual Counting
Crash	1072
Energy	131
Functionality	1000
Security	497
Total	2700

Table 3.5: Results of clustering approach for class labeling.

Clusters	Actual Class Labels			
	Crash	Energy	Functionality	Security
Cluster 0	63	5	342	58
Cluster 1	947	0	74	4
Cluster 2	15	2	37	363
Cluster 3	47	124	547	72

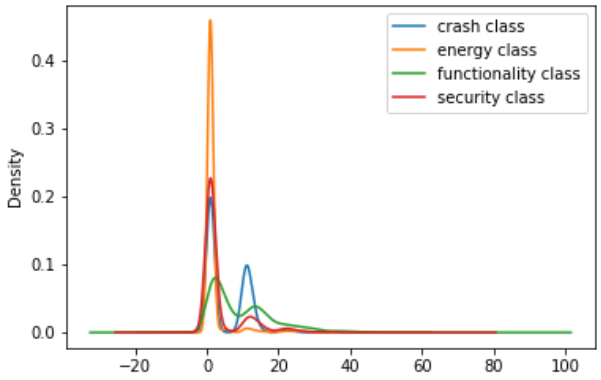


Figure 3.2: Density plot of each class label on manually classifying dataset.

		Predicted	
		Negative	**Positive**
Actual	**Negative**	True Negative	False Positive
	Positive	False Negative	True Positive

Figure 3.3: Confusion Matrix

- **True Positive (T_P)**: It defines the correctly predicted instances by the classifier. That means the actual class was Crash, and the classifier predicted as Crash.
- **False Positive (F_P)**: It defines the incorrectly predicted instances by the classifier. That means the actual class was non Crash, and the classifier predicted as Crash.
- **False Negative (F_N)**: It defines the incorrectly predicted instances by the classifier. That means the actual class was Crash, and the classifier predicted as Non-Crash.
- **True Negative (T_N)**: It defines the correctly predicted instances by the classifier. That means the actual class was non Crash, and the classifier predicted as non Crash.

Accuracy: Accuracy is defined as the ratio of the total number of examples correctly classified by the classifier to the total number of examples in the testing dataset.

$$Accuracy = \frac{T_P + T_N}{T_P + F_P + T_N + F_N}$$

Error rate: Error rate is defined as the ratio of the total number of examples incorrectly classified by the classifier to the total number of examples in the testing dataset.

$$Error\ rate = \frac{F_P + F_N}{T_P + F_P + T_N + F_N}$$

Precision: It is defined as the ratio of the total number of T_P correctly classified by the classifier to the total number of predicted T_P. Thus, it can answer the question 'What proportion of positive identifications was actually correct?'

$$Precision = \frac{T_P}{T_P + F_P}$$

Recall: It is defined as the ratio of the total number of T_P correctly classified by the classifier to the total number of T_P in the testing dataset. Recall attempts to answer the following question 'What proportion of actual positives was identified correctly?'

$$Recall = \frac{T_P}{T_P + F_N}$$

f1-Score: F-1 score is useful for the imbalanced dataset, which means when the class distribution is uneven. It is the harmonic mean of precision and recall and can be defined as follows:

$$f1\ Score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Receiver Operating Characteristic (ROC) curve: ROC curve represents how the relationship between precision and recall varies as we change the threshold for detecting positive instances in our dataset. ROC curve plots false positive rate (probability of false prediction) on the x-axis and true positive rate (recall) on the y-axis.

$$False\ Positive\ Rate = \frac{F_P}{F_P + T_N}$$

Finally, we also use another metric, Area Under the Curve (AUROC), to quantify a models' ROC curve. The value of AUROC falls between 0 and 1, and the higher number means better performance of a classification model.

3.4.2 Algorithms parameter tuning

Parameter tuning of any supervised classification algorithm is an essential task to seek the best performance of that classifier, and it mainly depends on data characterization and presentation. We use 70% data as training and 30% data for testing purposes from the total dataset. First, we use the Naïve Bayes approach for the classification. Though there are different types of Naïve Bayes, we choose GNB [159] because of our continuous features set, and the features are almost normally distributed as we see in Figure 3.2. Second, K-Nearest Neighbor is used in categorizing the bug reports, and choosing the value of K is a significant job. To search the optimal value of K, we run a process of different K values (starting from 3 and increasing by 2) and inspect the classifiers' accuracy (f1 score) and get the highest accuracy (f1 score) when $K = 7$.

Third, in DT [171] classifier, we use entropy (information gain) as a criterion because of imbalanced continuous attributes of the dataset and minimize misclassification though it is slower than the Gini impurity due to log calculation. Gini impurity measures how often a randomly chosen bug report from the dataset would be incorrectly labeled, and it is calculated by subtracting the sum of the squared probabilities of each class from one. Also, entropy gives good results for multi-label classification rather than Gini for binary classification. We also tune the parameter max depth as 5. Fourth, for SVM classifier [184], we take radial basis kernel function because of multi-class classification with other parameters set to default.

Moreover, finally, for RF Classifier [151], we set 400 trees as estimators to the forest. We scrutinize several number of estimators and get the best accuracy (f1 score) for 400.

3.4.3 Classifier Results Analysis and Evaluation

The results of each classifier are manifested in Figure 3.4 and showed both separate class labels comparison and overall accuracy comparison among the five supervised classification algorithms (GNB, KNN, DT, SVM, and RF). We present a comparative analysis among classifiers in each class label regarding precision, recall, f1 score, and accuracy.

All the classifiers perform equally for the crash bug label because the number of crash reports and features of that class are rich; the GNB experiences the highest f1 score for the crash class label, which 95% and all other classifiers have a similar score of 94%. Precision, recall, and accuracy are nearly equal to each classifier. All the investigations on the crash bug label are shown in Figure 3.4(a). For the context of energy bug, we have the utmost f1 score vary from 93% to 96% except GNB classifier and SVM holds the highest one. We get a lower f1 score for the GNB classifier because of the imbalanced dataset problem. Our dataset has fewer energy bug reports compare to other bug types. As GNB does not perform well for the imbalance dataset, the precision (13%) and f1 score (23%) are comparatively lower than the other classifier for this class label.

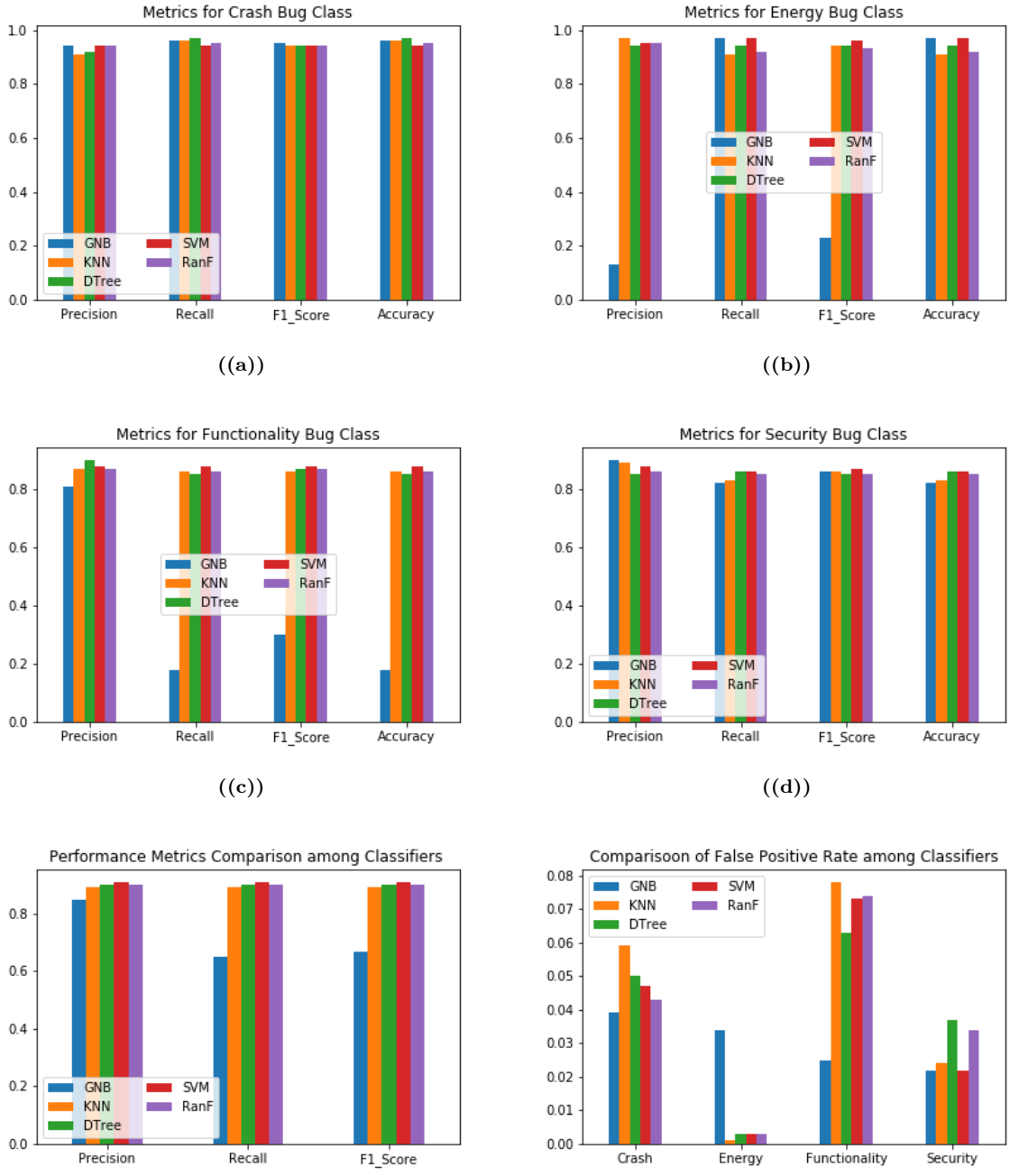


Figure 3.4: Precision, Recall, f1 Score, Accuracy and false positive rate comparison among classification algorithms for (a) Crash; (b) Energy; (c) Functionality; (d) Security; class labels. Overall comparison among classifiers (e) Micro Average f1 score; (f) False Positive Rate.

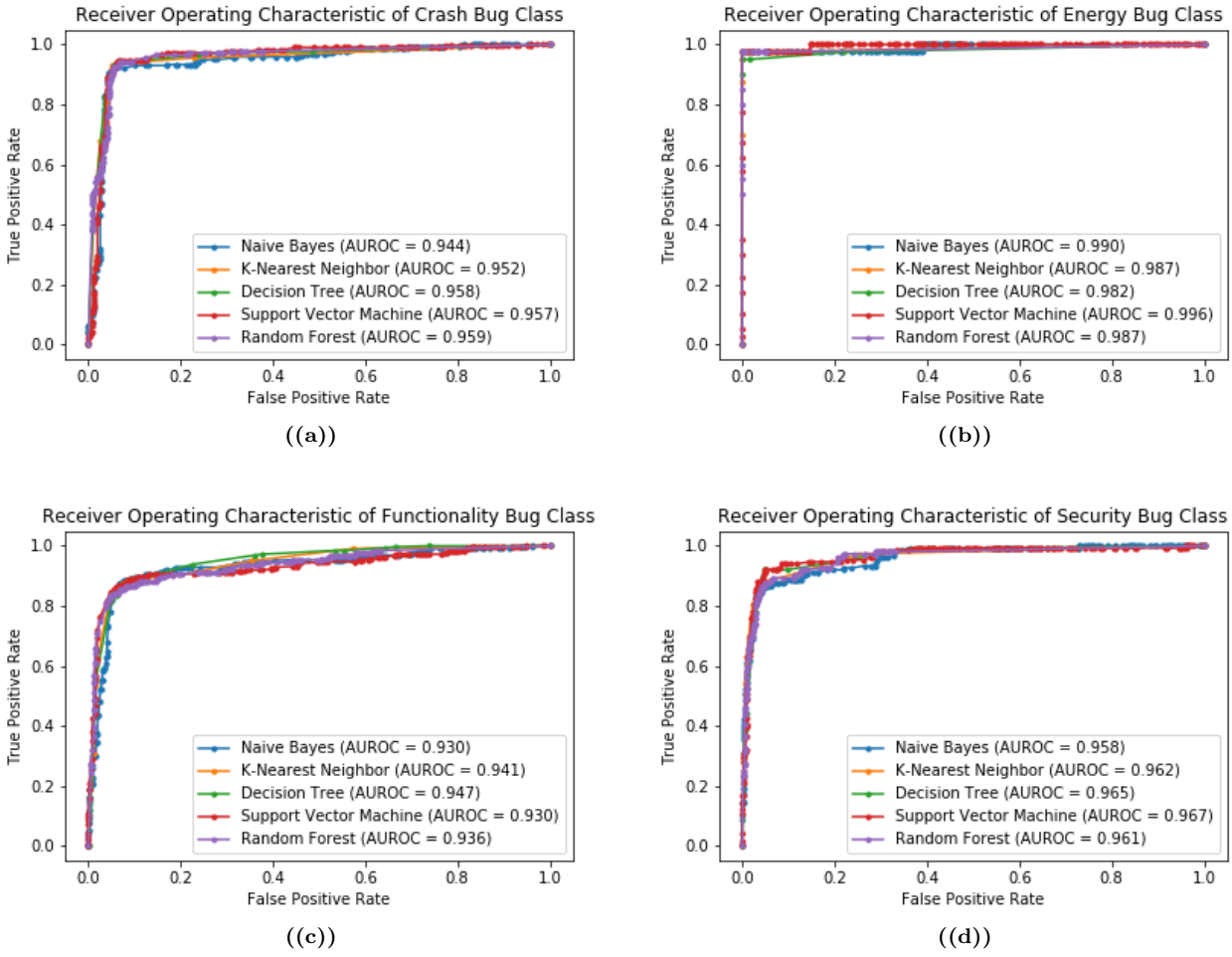


Figure 3.5: Representation of ROC curve among classifiers of each class labels (a) Crash; (b) Energy; (c) Functionality; (d) Security.

In addition, unlike GNB, we get 95-97% precision scores for KNN, DT, SVM, and RF. On the other hand, the main reason for getting that much accuracy for other classifiers is crash class features (words). The energy bug reports are represented by some specific keywords that are hardly available in other types of bug reports. Figure 3.4(b) depicts the analysis of five supervised algorithms for the energy class label.

If we observe Figure 3.4(c), we can see the highest precision (90%) at DT, highest recall (88%), and highest accuracy (88%) at SVM. Like energy class labels, GNB accuracy is relatively low because of low recall scores (18%). From the context of the f1 score, SVM performs best (88%) for the functionality class bugs. Finally, security class label observations are presented in Figure 3.4(d). Although we select some unique words to identify security bug reports than other class labels, there are many misclassifications due to the extensive presence of other class label keywords. This issue is also true for other class label detection. In the security class label, SVM has the most f1 score (87%). Here GNB performs well of highest precision (90%) and good f1 score (86%). Other classifier results are quite similar to each other.

We also plot the overall accuracy of each classifier in terms of precision, recall, and f1 score in Figure 3.4(e) of each class label performance. As our dataset is not balanced, so we consider the average micro score to detect the best classifier. All true positives, false positives, false negatives, and true negatives are considered in the micro average calculation. SVM has the finest score in terms of f1 score, which is 91%. We also investigate the False Positive Rate (FPR) of each classifier to detect each class label. Figure 3.4(f) shows all the class label FPR (a non-crash bug report is detected as a crash class) in different classifiers. As we can see in the energy class label, GNB has the highest rate, as expected as a low f1 score. All the other class labels have rates that vary from 2% to 7.5% maximum along with classifiers. To conclude this subsection, we can say that the support vector machine performs best as a bug reports classifier for our manually labeled dataset.

3.4.4 Analysis of ROC curve and Area Under ROC curve

The Receiver Operating Characteristics(ROC) curve determines the capability of classification algorithms to separate data among classes, and Area Under ROC (AUROC) narrates the measurement of that separability. Higher AUROC means the high number of positive values are labeled as positive and negative values are negative. In Figure 3.5, we show each class label ROC with different classification algorithms and calculated AUROC.

In Figure 3.5(a), we can see the ROC curve and area under the ROC curve of crash class along with different classifiers, which are varied from 94% to 96%. The RF classifier has the highest area coverage, although DT and SVM scores are quite similar.

It is obvious that energy bug reports classification gets maximum area because of its distinctive features, almost 100% for SVM and 99% for other classifiers. All classifiers' ROC curve for energy class label is constituted in Figure 3.5(b). All the classifiers perform well for functionality and security class label and cover from 93% to 97% area under ROC curve, respectively. These are depicted in Figure 3.5(c) and Figure 3.5(d). It is notable that GNB dominates 93% to 99% area under the ROC curve for all the class labels classification, but the classifier accuracy is poor. This is because GNB can separate each bug report in certain class labels but not in appropriate criterion. For example, it covers all the energy bug reports(AUROC = 99%), but most of them are miss-classified, i.e., marked energy bug as other class labels (crash, functionality, or security).

Finally, we also calculate the overall separability measurement and error rate of each classifier which is represented in Table 3.6. Again, the measurement of AUROC of each classifier is quite similar to each other varies from 94% to 96%, and SVM experiences low error rate (4.7%) than others.

3.5 Discussion

Almost all of the classification approaches used traditional desktop-based software bug reports to categorize and prioritize bug severity. Here in this work, we investigate several open-source mobile applications' bug

Table 3.6: Overall evaluation among classifiers in terms of AUROC and Error rate.

Classifier	AUROC	ErrorRate
GNB	0.943	0.173
KNN	0.951	0.053
DT	0.955	0.051
SVM	0.951	0.047
RF	0.952	0.051

reports to classify certain criteria that help developers and maintainers make appropriate decisions to tackle an incoming bug.

After an intensive analysis of 2700 manually labeled bug reports along with five supervised classification algorithms (GNB, KNN, DT, SVM, and RF), we got our best result from the SVM classifier in terms of f1 score and error rate (we could not consider AUROC because except GNB every classifiers' AUROC measurement is too close to each other). The overall accuracy result (91%) of the SVM classifier is very promising in terms of mobile application bug reports and classifying them into four defined categories, and also, the SVM has a lower error rate(4.7%) among the other classifiers.

Similar types of tasks which were done by Otoom et al. [150] got 93.1% average accuracy for SVM classifier. They also used manually labeled bug reports of desktop-based subject systems and had only two class labels (corrective and perfective), i.e., bug and non-bug. However, the challenges of our work are manual classification of bug reports of mobile apps where developers and end-users report their inconsistencies in different formats and representations, labeling these reports into four predefined categories, and finding distinctive features for each class. In addition, no other existing research worked with futher classification of bug-related issues.

Another work is done by Katerina et al. [72] to detect security and non-security bug reports for the NASA dataset. They mentioned that they experienced lower accuracy because of the unavailability of security bug reports in their dataset. They used three types of feature vectors: Binary Bag-of-Words Frequency (BF), Term Frequency (TF), and Term Frequency-Inverse Document Frequency (TF-IDF), but finally, their result was poor (only 25% provides good result while training on 90% of the data). In our approach, we manually analyze each bug report and enlist meaningful keywords as features and prepare a feature vector of each bug report to detect the presence of word with predefined weight (bug title word features get more weight than bug description feature word) to feed machine learning algorithm and get a dominant result of 91% accuracy. They also applied both supervised and unsupervised approaches but got comparatively good results in supervised learning like our approach.

In Gegick et al. [68], the authors said that their classifier had a very high false-positive rate ranging from 27% to 96%. However, in our case, the rate of false-positive vary from 0.5% to 7.5% among different classification approaches. Moreover, they only worked with security-related manual labeled bug reports, but

here we consider four types of bug reports that are important for mobile applications' bug report classification.

Although all the classifiers (except GNB) provide a good accuracy score for each bug report, the accuracy might be increased if the dataset is balanced. We cannot find enough security and energy bug reports where functionality and crash bug reports are more frequent in developer sites. So, it is possible to bias the low number of bug reports by the high number, which is the main possible reason for Gaussian Naïve Bayes to produce poor f1 score results.

3.6 Threats to Validity

We randomly collect 2,700 bug reports which are labelled as bugs by the corresponding app developers from diverse Android and iOS apps of several open-source development sites by hand. We do not cover all the bug reports of any specific app, and also, the reported issues were different in format at each development site. However, we consider only the textual features (plaintext in bug titles and descriptions) for categorisation that might not be affected by distinguished bug reports in formats and designs.

The classification task is performed based on our selected features of each class label. In specific, we manually select the feature words to prepare feature vectors. These words might not be optimal or possibly can be an overhead for the classification task, and we do not consider any synonyms of the selected feature words. However, our selected features of each bug category can predict the expected bug label with an accuracy of more than 95%. Moreover, two graduate students did the manual categorization of bug reports and labelled them by considering our selected features. However, the categorization may be biased and inclined to human error. Therefore, we calculated Cohens' kappa coefficient to evaluate their mutual agreement and got an excellent inter-rater agreement with the value of +0.978.

In our classification task, we tuned several parameters of distinguished machine learning algorithms. The parameters tuning might not be appropriate when the dataset has a lower or higher number of bug reports than we have, though it provides acceptable accuracy for our analysis. Finally, we found a limited number of energy and security bug reports in the development sites that might yield a poor predictive performance, which arose from the data imbalance problem. However, we chose such unique keywords to identify the above class labels, which sustains nearly 100% accuracy.

3.7 Conclusion

In this paper, we applied the approach based on utilizing machine learning algorithms to classify the bug reports. We collected 2,700 mobile apps (Android and iOS) bug reports from different open-source developer sites and manually labeled them into four predefined class labels. We also tried clustering to classify bug reports into four clusters, but due to excessive overlapping among class types, we did not get a satisfactory result, so we went for manual labeling. We split our dataset into 70% training and 30% testing, respectively. Then we apply five supervised classification algorithms (GNB, KNN, DT, SVM, and RF) to build a model for

automatic classification of mobile bug reports. Despite the imbalanced dataset, we got a promising accuracy score for distinguishing mobile bug reports into four classes when all the existing research worked only for two class labels. All the classifiers (except GNB) provided good accuracy scores varying from 88% to 91%, and we got the best result with SVM classifier with a high f1 score (91%) and low error rate (4.7%). Moreover, the classifiers in our approach provided very low false-positive rates (up to 7.5%) where some existing works [68, 205] experienced a very high false-positive rate (up to 96%).

Although we got good classification results for our own manually built dataset, some limitations need to be addressed. One of them is that our dataset is not balanced. Though the manual collection of bug reports from distinct open-source sites was strenuous and time-consuming, we explored more than a hundred android and iOS apps to find security and energy bug reports but experienced inadequate numbers. For example, crash and functionality bug reports were nearly balanced, but the number of energy bug reports was too low. We will collect more bug reports to solve this issue and explore more mobile applications present in open-source development sites.

After the classification, we wanted to explore the possible code-change in a particular software codebase while fixing a bug. So, we investigated a randomly selected subset of entire bug reports of Java projects and noticed that more than 50% of code-change are happened in clone code (exact or similar code fragments throughout the codebase). Moreover, existing studies with desktop-based software systems clearly show the harmful impacts of code clones and their relationships to software bugs. However, given a marked lack of such studies for mobile apps, the next chapter will explore several mobile apps to find the consequences of code clones throughout the app lifetime regarding later software maintenance.

4 Analysis of Code Cloning in Open Source and Industrial Software Development Stages: A Perspective of Mobile Applications

To mitigate the study gap regarding the impacts of code clones in mobile apps development and their relationship with software bugs that are not common in desktop-based apps (as discussed in the previous chapter). Therefore, this chapter focuses on the bug-proneness of clone and non-clone code in eleven mobile apps: five open-source and six from the industry. Our analysis of thousands of revisions of eleven diverse Android and iOS apps shows that the rate of source code-change in bug-fix commits is significantly higher in clone code than non-clone code. We also observe that bug-fixing commits changed clone code at a higher rate in open-source mobile apps than in industrial mobile apps. In addition, the ratio of code-change between clone and non-clone code differs in open-source and industry apps. According to our study, clone code is more bug-prone than non-clone code, and thus clone code should be more carefully treated during app development and maintenance. This study also performs a comparative analysis between mobile and non-mobile apps through the percentage of bug-fixing commits that changed clone and non-clone code and states that mobile apps have a lower code-change rate than non-mobile apps. Finally, we conduct an online survey with industrial developers to determine their experiences with clones and bugs.

The rest of the chapter is organized as follows. Section 4.1 discusses the motivation of this work and research questions. Section 4.2 depicts several studies related to this work. The overall methodology is discussed in Section 4.3. In Section 4.4, key findings and answer of the research questions are presented. Section 4.5 reports developers' survey and detail discussion. Threats to the validity of this study are presented at Section 4.6 and finally, concludes this chapter with Section 4.7.

4.1 Introduction

It is estimated that there are more than 2.56 million Android apps in Google Play Store and around 1.85 million iOS apps in Apple's App Store [14], and the number of mobile apps for smartphones and tablets is increasing day by day. Such a large number of apps enrich our life, as well as bring new challenges and risks. Similar to traditional software, mobile apps may be buggy when running on different systems and devices. Clone research has considered the bug-proneness of clone and non-clone code in desktop-based software system. On the other hand, mobile app research [33, 203, 208, 209], has mostly focused on detecting

bugs in apps, improving app performance, and protecting data privacy in smartphones from malicious apps, with numerous approaches and techniques proposed and prototyped [207].

Our focus is on the bug-proneness of clone and non-clone code in mobile apps. In our study, we analyze the effects of clone code for thousands of revisions of eleven subject systems both from the open-source software development site (GitHub) and from a mobile app development company to address this gap in the research. Two or more blocks of code or code snippets in a software system are considered to be clone when they are exactly or nearly similar to each other [162, 166]. During software development and maintenance, developers frequently copy-and-paste code that creates code clones. A plethora of studies and experiments were conducted on exploring the consequences of code cloning in software development and maintenance. While several studies [35, 74, 83, 95, 99, 107, 108, 109, 136] showed the positive effects of code cloning, there are other studies [40, 70, 93, 95, 115, 121, 122, 136, 137, 141, 183] that prospected negative results of cloning. Several software issues such as high instability [136], late propagation [40], and unintentional inconsistencies [70] are created because of code cloning. In addition, there are numerous studies [36, 40, 75, 88, 89, 138, 144] that investigated the relationship of cloned code and bugs, but none of the existing studies have ever investigated the actual impacts of code clones in mobile apps and not for industrial project at all.

Although today's desktop and mobile apps use similar designs, architectures, implementations, coding approaches, and testing strategies, there are unique characteristics of mobile applications and their development [66]. Flora et al. [66] conducted a survey involving software companies, app development teams members, mobile experts, researchers, and relevant stakeholders to identify the key characteristics and other things make mobile apps different from traditional software. The author mentioned several requirements such as potential interaction with other apps, integration with device sensors, input mechanisms, responsiveness, native and hybrid (mobile web) apps, storage limitations and security issues [103], variability and inconsistency of user interfaces, power consumption and unit testing, localization and multiple networking protocols and short length session activities [65, 202]. In addition, because of the many mobile devices, developers need to create versions of the same application for each type of devices [56, 71]. According to a survey [66], software bug, such as concurrency bug, security bug, compatibility bug, app crash bug in mobile apps usually occur in session management and when users start interacting with the apps. As there are a couple of potential studies related to software bugs in traditional software system due to code cloning, it is equally important to investigate the effect of code cloning in mobile software code bases and corresponding development environments.

Increased awareness of bug-proneness in code clones for Android and iOS apps could lead to app users experiencing fewer bug-related issues due to code cloning. In this work, our goal is to investigate several mobile apps' commits to observe the effects of clone code and software bugs, and build a comparative analysis of code-change rate between clone and non-clone code during software evolution and maintenance. We also investigate the code cloning ratio of open-source and industrial applications and find correlative

characteristics of clone code change while fixing bugs. In addition, we make comparative analysis between industrial and open source, mobile and non-mobile (results are taken directly from [87]) apps. Therefore, we consider bug-fixing commits reported by developers from thousands of commits in five open-source apps and six industrial apps (from a mobile app development company we refer to as ABCD Software) written in two different programming languages (Java and Swift).

To determine the impact of bug-fixing and non-bug-fixing change in maintenance throughout all revisions of each app, first, we detect code clones from each of the revisions of a mobile app using the NiCad [51] clone detector, scrutinize the history of app evolution for these clone code and investigate in what proportion they contain bugs. To find the bug-fixing commits, we review the commit messages to find some specific words related to bug and bug-fixing according to Mockus and Votta [132] method. After detecting the bug-fixing commits, we analyze those commits' clone code to find whether they are responsible for the bugs. We also investigate clone code of all the commits to determine code-change throughout the development stages and further maintenance. Omitting the change of the clone code, we are left with the change in the non-clone code. Then we calculate the rate of change per thousand lines of the codebase. We conclude that the rate of clone code change is significantly higher than the rate of non-clone code change. We also analyze the use of code cloning in open source and industrial mobile apps and find that the ratio of code cloning in industrial apps is higher than open-source apps, and we also determine the code change rate between open-source and industrial apps while fixing bugs. We validate our findings using the Wilcoxon Signed Rank test [20] for three types of clones with non-clone code and the Mann-Whitney U test [12] for comparing our findings for open source versus industrial. In this study, We investigate five research questions listed in Table 4.1 which are described details in Section 3.4.

Table 4.1: Research Questions

Sl.	**Questions**
RQ 1	What is the rate of change in clone and non-clone code in bug-fixing commits?
RQ 2	What are the impacts and characteristics of clone code and non-clone code throughout the evolution of non-bug-fixing commits?
RQ 3	Are clones more prevalent in open source or industrial mobile applications?
RQ 4	Is the change-proneness of code clones in the open-source mobile apps similar to that of the code clones in the industrial mobile apps?
RQ 5	Is there a difference in terms of clone and non-clone code change in bug-fixing commits between mobile and non-mobile apps?

4.2 Related Work

There are several existing analyses and research on code cloning and bug severity measurement but most of them used non-mobile apps as subject system.

There have been a number of studies that conducted the relationships between bugs and software clones [136] including comparative studies in cloned code and non-cloned code [87], on the relationships between stability and bug-proneness of code clones [157], propagation of bugs through code cloning [143], identifying code clones having high possibilities of containing bugs [142], investigating context adaptation bugs in copied fragments [144], replication of bugs in clones and micro-clones [89], and even harmfulness and intensity of late-propagation of clones [140].

Islam et al. [87] conducted a comparative study on bug proneness in clone code and non-clone code by analyzing several projects' commit logs from SVN repositories [23]. According to their inspection, the percentage of changed files due to bug-fix commits is significantly higher in clone code compared with non-clone code, and they limited their research to traditional desktop-based software, and they did not consider the lines of code change rate in clone code and non-clone code. A similar study is conducted by Mondal et al. [138], but the authors did not consider the bug-proneness of non-clone code in their study.

Azeem et al. [36] experimented with the effect of code cloning in mobile apps. The authors used open-source iOS game apps and tried to find the coupling relationship among the five main classes, which are developed by third parties. But authors did not analyze the bug-proneness of clone code throughout all revisions of the apps' lifetime. They also did not consider the developers' level code change measurement to mitigate bugs. Our study examines bug-fix and non bug-fix commits that changed clone and non-clone code throughout industrial and open-source mobile apps.

Several studies investigated bug-proneness of code clones like Li and Ernst [115] did an empirical study on clones' bug-proneness by scrutinizing four software systems and developed a tool called CBCD based on their findings. To find inconsistencies in copy-paste activities, Li et al. [116] built a CP-Miner tool. Steidl and Gode [183] investigated clone code related to machine learning and found fixed bug in near-miss clones that were incomplete and might affect other features.

Gode and Koschke [70] examined the occurrences of inconsistencies created unintentionally to the code clones. They tested three mature desktop software systems and experienced that around 14.8% of all change to the code clones were inconsistent but happened unintentionally. Shajnani et al. [174] accomplished a comparative study between clone and non-clone code for bugs' distinguish patterns. However, they did not present any comparison or relation between clone and non-clone code.

Focusing on different subject systems (i.e., mobile apps) than existing works (i.e., desktop apps), this study aims to find the impacts of clones regarding software bugs and maintenance from the perspective of industrial and open-source mobile apps. Moreover, there is no such study that incorporates industrial/close-sourced projects in clone study. In addition, To strengthen our study results, we intended to survey industrial people

(mobile apps developers) regarding their thoughts and experiences concerning code clones, software bugs and later maintenance. Accordingly, this study provides potential implications for a better understanding of code clones in software bugs in both development environments.

4.3 Experimental Methodology

We performed our experiment on the apps listed in Table 4.2. These apps have variations in domains, sizes, number of revisions, and weather they were previously used in other studies [24]. An important factor for selecting these apps is to consider a broad range of domains such as travel, music video, and shopping. As well, we selected the most popular and satisfactory apps of the ABCD software company, which are also in diverse domains with differing sizes and revisions.

All of these mobile apps are downloaded from online GitHub[1] repositories (for open-source projects) and BitBucket[2] which is the development site of ABCD Software company (for industrial projects). In Table 4.2, each apps is listed with its domain of application, the programming language its written in, and the total number of revisions (NR) along with the average number of lines of code (ALOC) per revision. We calculate ALOC by counting all the revisions lines of code and then measure the mean to determine the ALOC. We ignore comment lines and blank newlines to compute the actual lines of code. We perform the following five steps for our study. All the steps of our study will be construed in details in subsequent subsections.

4.3.1 Clone Detection Technique

We use NiCad [51] for detecting clone code since it can detect all major types (Type 1, Type 2, and Type 3) of clones with high precision and recall [146, 164, 165]. Another important reason for choosing NiCad is its ability to detect clone fragments in Swift language directly. Using NiCad, we detect clone code blocks with 30% dissimilarity threshold and blind renaming of identifiers. We use this configuration of the NiCad clone detector shown in Table 4.3 to get the higher precision and recall rate used by Roy et al. [163].

4.3.2 Bug-proneness Detection Technique

For all the mobile apps (Android and iOS) of two different languages, first, we extract developers commit messages by applying *git log* command [47]. Developers usually include the purpose of a corresponding commit operation in a commit message in the form of natural text. We automatically infer the commit messages using the heuristic approach proposed by Mockus and Votta [132] to identify those commits which are occurred to fix bugs or solve bugs-related issues. If one or more clone fragments are deleted or modified in a particular bug-fix commit, then it is an implication that the modification of those clone fragment(s) was necessary for fixing the corresponding bug and called these code fragments are responsible for that bug

[1] https://github.com/
[2] https://bitbucket.org/product

Table 4.2: Subject Systems

SL	Project Name	Type	Lang.	Domain	ALOC	NR
1	SIGA	Industrial	Java	Entertainment	3211	519
2	TRUSTED	Industrial	Java	Online directory	32,317	182
3	AskAvenue	Industrial	Swift	Live Chat for Real Estate	39,994	498
4	Bolt Mobile	Industrial	Swift	Service Provider	17,684	267
5	My Shire	Industrial	Swift	Business	40,426	276
6	Reperformance	Industrial	Swift	Workout and Fitness Test	48,134	1167
7	Ankidroid	Open source	Java	Education	49,185	9298
8	Frostwire	Open source	Java	Media and Video	243,848	6027
9	TramHunter	Open source	Java	Travel	10,783	288
10	Andlytics Track	Open source	Java	Shopping	38,274	1524
11	OpenDocument Reader	Open source	Java	Reader	3778	612

Table 4.3: NiCad settings for different kind of clones.

Clone Types	Identifier Renaming	Dissimilarity Threshold
Type 1	none	0%
Type 2	blindrename	0%
Type 3	blindrename	30%

occurrence. This way, we investigate the commit messages to detect the bug-fixing commits and identify the code snippets associated with the bug-fix, such as the number of change operations, number of lines of code change and so on. We also determine the number of non-clone code change, both for bug-fixing and non-bug-fixing commits parallel with different types of clone code. The main idea of [132] to find bug-fixing commits is to spot five specific keywords (i.e., *'bug'*, *'fix'*, *'fixup'*,*'error'*,*'crash'*, and *'fail'*) in the commit messages. The authors of [165] evident that this way of detecting covers almost all the bug-fixing commits. Other studies, such as Islam et al. [87] and Barbour et al. [40] reused this idea previously to detect bug-fix commits.

4.3.3 Code Change Detection Steps

We only consider code change by a commit to further maintain clone code and non-clone code throughout all revisions. To detect the changes between every two consecutive revisions, we use the output of UNIX *diff* command. And then, compare change output, NiCad results, revision number, programming file names, and start-line and end-line of clone code fragment to spot the clone code change throughout the revision when fixing bugs. We consider all the changes (deletions, additions, and modifications) in all commits and find the change that occur in clone fragments and categorize them in terms of clone types. We also scrutinize the line of code change in clone code and non-clone code for non-bug-fix commits.

4.3.4 Statistical Significance Testing

We verify our analysis results using two types of non-parametric statistical methods i.e., Wilcoxon Signed Rank Test [20] and Mann Whitney U test (also called Wilcoxon Signed Rank Sum test) [12]. Non-parametric methods allow statistical inference without making the assumptions that the sample has been taken from a particular distribution (i.e., normal). For matched pairs data, we justify our calculated outcomes using the Wilcoxon Signed Rank Test (when we try to find statistical significance among different types of clone and non-clone code for all industrial and open-source projects). Furthermore, we employ the Mann Whitney U test for independent samples data to encounter the significance between industrial and open-source mobile apps. Besides, we consider 5% level of significance and two-tailed test (because of testing statistical significance in both directions and calculating the p-value) and observe the critical U value from the statistical chart [6].

4.3.5 Developers Survey

To relate our findings with the experience of mobile app developers, we conduct an online survey with 23 developers to perceive their thoughts/experiences about clones and bugs. The developers are from ABCD company and sixteen different renowned software companies across Canada and Bangladesh with 1-10 years of development knowledge on mobile apps and also working as an open-source developer in distinguished teams. We received many potential comments and feedback from each developer. We present the survey outcomes in Section 4.5 in detail.

4.4 Experimental Results and Analysis

In this section, we elaborately justify our experimental findings and relate them to find the answers to our mentioned research questions.

4.4.1 Answering the first research question (RQ 1)

RQ 1: ***What is the rate of change in clone and non-clone code in bug-fixing commits?***

Motivation: It is important to investigate the rate of change of different operations (deletion, addition and modification) and lines of code are updated in bug-fixing commits and to make comparison among all types of clone code and non-clone code. If software code-base files experience more change on the number of operations and lines of code that indicates more files are affected due to bug and it is obvious to give much attention dealing with code that are responsible for those occurrence. By knowing of the change information throughout the evolution of apps, we can emphasize on which type of code (clone or non-clone) influence the system more and require more discretion at the stage of app development.

Methodology: To answer this research question, we analyze the commit messages to find the bug-fixing commits using the approach discussed in [132]. There are around 7% to 23% commits are occurred as bug-fixing commits in our selected Android and iOS apps. We detect all the change both in clone code and non-clone code of those bug-fixing commits and figure out this research question's answer.

After analyzing clone code fragments and results from UNIX *diff* command, we measure the changes that are happened to fix bugs. If a single line or multiple lines of code are deleted or added or modified to solve a bug, we call it as one change occurred. We determine the rate of change of each type of clone by the ratio of total clones change count while fixing bugs and total clones in all revisions. There are several terms that are used to calculate change rate are represented as short form for easy understanding which are discussed below and used as columns header of Table 4.4.

LOCCB: Total number of lines of code of each type of clone in bug-fixing commits. We calculate this term by summing up clone code that are available in all bug-fixing commits.

LOCNCB: Total number of lines of non-clone code in bug-fixing commits. We measure its value as same way as LOCCB (add all lines of non-clone code available in all bug-fixing commits).

NCC: Total number of changes (delete, add and modify) in different types of clone code while fixing bugs in bug-fixing commits.

NCNC: Total number of changes (delete, add and modify) in non-clone code while fixing bugs in bug-fixing commits.

CRTL: To find out the rate of changes for thousand lines of code (we use thousand lines for visualizing the calculated value in a better way)

Using NiCad [51] results of each subject system, we calculate LOCCB and LOCNCB for individual clone types and non-clone code. After analysing clone code fragments and results from UNIX *diff* command, we

measure the changes that are happened to fix bugs. We compute in what extent each type of clone code are changed. For better presentation, We determine all the rate of changes per thousand lines of code. The change rate (CRTL) is calculated for clone and non-clone code by equation 4.1 and equation 4.2 respectively.

$$CRTL(CloneCode) = \frac{NCC * 1000}{LOCCB} \tag{4.1}$$

$$CRTL(NonCloneCode) = \frac{NCNC * 1000}{LOCNCB} \tag{4.2}$$

We observe that most of the projects experience a high change rate in clone code (for different clone types) than non-clone code. To verify the significance of change rates, we apply the Wilcoxon Signed Rank statistical test [20] with 5% level of significance and two-tailed hypothesis between each type of clone and non-clone. After performing the test, we could not find significant results (see in Table 4.5) (p-value >0.05 and U Statistics >U critical), i.e., there is no meaningful distribution differences among different clone code and non-clone code (in a word, we can say they follow similar distribution). All the measured values and calculated rate of changes are presented in Table 4.4. Therefore, we do another experiment to answer this research question that considers lines of code change among clone code and non-clone code. Previously, we consider only operations, not the consequences of those operations. For example, if hundreds of new code lines are added to fix a bug, we count only one for previous consideration. Now we reflect on the complete lines of code change of all bug-fixing commits among clone and non-clone code. Along with previous measurement LOCCB and LOCNCB, we introduce four new terms.

LCC: Lines of code which are changed in clone code while fixing bugs.

LCRTL: Lines of clone code-change rate per thousand lines of code.

LCNC: Lines of code change in non-clone code in all bug-fixing commits and

LNRTL: Lines of non-clone code-change rate per thousand lines of code.

we determine LCRTL and LNRTL using following equation 4.3 and 4.4 for several types of clone code and non-clone code per thousand lines of code (for better visibility) respectively.

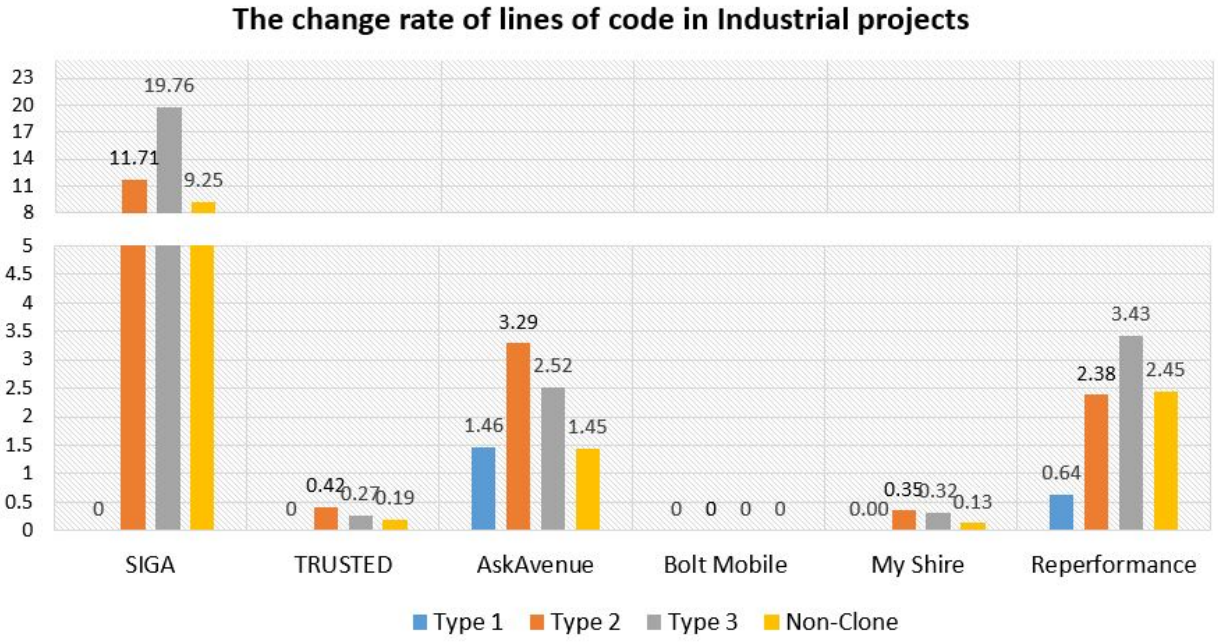
$$LCRTL = \frac{LCC * 1000}{LOCCB} \tag{4.3}$$

$$LNRTL = \frac{LCNC * 1000}{LOCNCB} \tag{4.4}$$

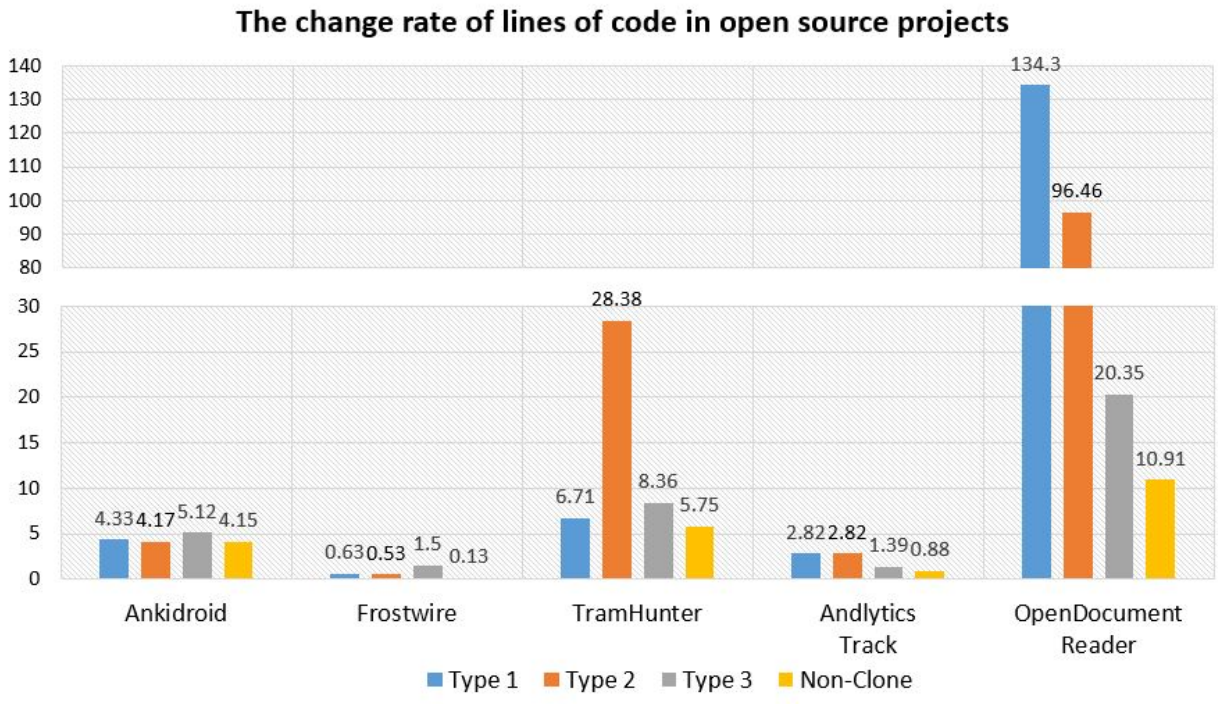
Table 4.6 shows all the calculated values of code change in three types of clones and non-clone codes. We also present the lines of code change rates (LCRTL & LNRTL) in Figure 4.1, where we can observe that the

Table 4.4: Number of changes occur in different types of clone and non-clone codes while fixing bugs.

Project Name	Bug-fixing Commits	Type 1			Type 2			Type 3			Non-Clone		
		LOCCB	NCC	CRTL	LOCCB	NCC	CRTL	LOCCB	NCC	CRTL	LOCNCB	NCNC	CRTL
SIGA	70	312	0	**0.0**	2050	2	**0.98**	14830	40	**2.70**	207578	883	**4.25**
TRUSTED	30	21478	0	**0.0**	32994	2	**0.06**	145940	13	**0.09**	769098	97	**0.13**
AskAvenue	102	119298	28	**0.23**	233173	131	**0.56**	636319	282	**0.44**	3090598	1480	**0.48**
Bolt Mobile	19	3303	0	**0.0**	20598	0	**0.0**	52512	0	**0.0**	259583	0	**0.0**
My Shire	35	7348	0	**0.0**	61753	137	**2.22**	283347	332	**1.17**	658202	144	**0.22**
Reperformance	142	258221	57	**0.22**	580037	362	**0.62**	2898801	2793	**0.96**	3097969	2497	**0.81**
Ankidroid	2122	1351714	1050	**0.78**	1347470	1073	**0.80**	3465226	3940	**1.14**	98210404	83638	**0.85**
Frostwire	787	2626219	319	**0.12**	2839496	364	**0.13**	6012680	1236	**0.21**	180429981	18634	**0.10**
TramHunter	54	12960	36	**2.78**	3348	30	**8.96**	16038	32	**2.0**	549990	1113	**2.02**
Andlytics Track	178	61447	36	**0.59**	160766	67	**0.42**	983178	290	**0.29**	5607381	2299	**0.41**
OpenDocument Reader	108	1452	15	**10.33**	2540	17	**6.69**	28454	83	**2.92**	375578	1246	**3.32**



((a))



((b))

Figure 4.1: The rate of lines change per thousand lines of code occurred in bug-fixing commits on different types of clone and non-clone code for (a) Industrial project;(b) Open-source projects.

Table 4.5: Wilcoxon Signed Rank test to determine change rate significance.

Clone Types	**p-value**	U_{stat}
Type 1	0.646	23
Type 2	0.509	21
Type 3	0.880	26

line changing rate in clone code is higher than in non-clone code. To verify the significance, we again apply the Wilcoxon Signed Rank test for each type of clone with non-clone code based on the rate of lines change per thousand lines of code. To apply the test, we take pairwise values of Type1 (i.e., 0, 0, 1.46, 0, 0, 0.64, 4.33, 0.63, 6.71, 2.82, 134.3) and Non-clone (i.e., 9.25, 0.19, 1.45, 0, 0.13, 2.45, 4.15, 0.13, 5.75, 0.88, 10.91) and same as for Type2 and Type3 with Non-clone code (from Figure 4.1). Table 4.7 shows the corresponding p-value and U statistics where the two-tailed level of significance is 5% and U critical is 8 [6]. If we observe the content of Table 4.7, there is a significant difference in the distribution of Type 2 and Type 3 clone code with non-clone code. The p-value and U statistics of Type 2 and Non-clones are 0.009 and 2, which are much lower than 0.05 (5% level of significance) and 8 (U-critical) respectively, where Type 3 and non-clones hold lesser value than Type 2, which are 0.005 and 0 accordingly. However, there is no significant difference between the Type 1 clone and non-clone code. We can prove by this statistical experiment that Type 2 and Type 3 clone code are more responsible for occurring bugs than non-clone code.

Answer to RQ 1: According to our experimental analysis and results, the rate of code-change is higher in clone code (for Type 2 and Type 3) than in non-clone code during bug-fixes. So, according to our dataset, clone code are more accountable for creating bugs and needed more attention in development stages.

4.4.2 Answering the Second research question (RQ 2)

RQ 2: ***What are the impacts and characteristics of clone code and non-clone code throughout the evolution of non-bug-fixing commits?***

Motivation: As we already have the answer of RQ 1 and hence, we know the code-change in clone and non-clone code in bug-fixing commits, we are still not aware of the consequences of clone code and non-clone code in non-bug-fixing commits. It is essential to know the frequency of change occurrence throughout all the commits, both in bug-fix commits and non-bug fix commits. This research question helps us to gain insights into managing clone and non-clone code throughout all revisions in the area of mobile app development.

Methodology: To answer this research question, first, we detect the non-bug-fixing commits by simply separate bug-fixing commits from total commits. We follow the same procedure as answering the first research

Table 4.6: Line of codes change occur in different types of clone and non-clone codes while fixing bugs.

Project	Bug-fixing Commits	Type 1			Type 2			Type 3			Non-Clone		
		LOCCB	LCC	LCRTL	LOCCB	LCC	LCRTL	LOCCB	LCC	LCRTL	LOCNCB	LCNC	LNRTL
SIGA	70	312	0	**0.0**	2050	24	**11.71**	14830	293	**19.76**	207578	1920	**9.25**
TRUSTED	30	21478	0	**0.0**	32994	14	**0.42**	145940	40	**0.27**	769098	142	**0.19**
AskAvenue	102	119298	174	**1.46**	233173	766	**3.29**	636319	1603	**2.52**	3090598	4490	**1.45**
Bolt Mobile	19	3303	0	**0.0**	20598	0	**0.0**	52512	0	0.0	259583	0	**0.0**
My Shire	35	7348	0	**0.0**	61753	228	**0.35**	283347	959	**0.32**	658202	821	**0.13**
Reperformance	142	258221	164	**0.64**	580037	1378	**2.38**	2898801	9940	**3.43**	3097969	7597	**2.45**
Ankidroid	2122	1351714	5848	**4.33**	1347470	5615	**4.17**	3465226	17756	**5.12**	98210404	407882	**4.15**
Frostwire	787	2626219	1642	**0.63**	2839496	1500	**0.53**	6012680	9021	**1.5**	180429981	22818	**0.13**
TramHunter	54	12960	87	**6.71**	3348	95	**28.38**	16038	134	**8.36**	549990	3164	**5.75**
Andlytics Track	178	61447	173	**2.82**	160766	454	**2.82**	983178	1369	**1.39**	5607381	4907	**0.88**
OpenDocument Reader	108	1452	195	**134.30**	2540	245	**96.46**	28454	579	**20.35**	375578	4099	**10.91**

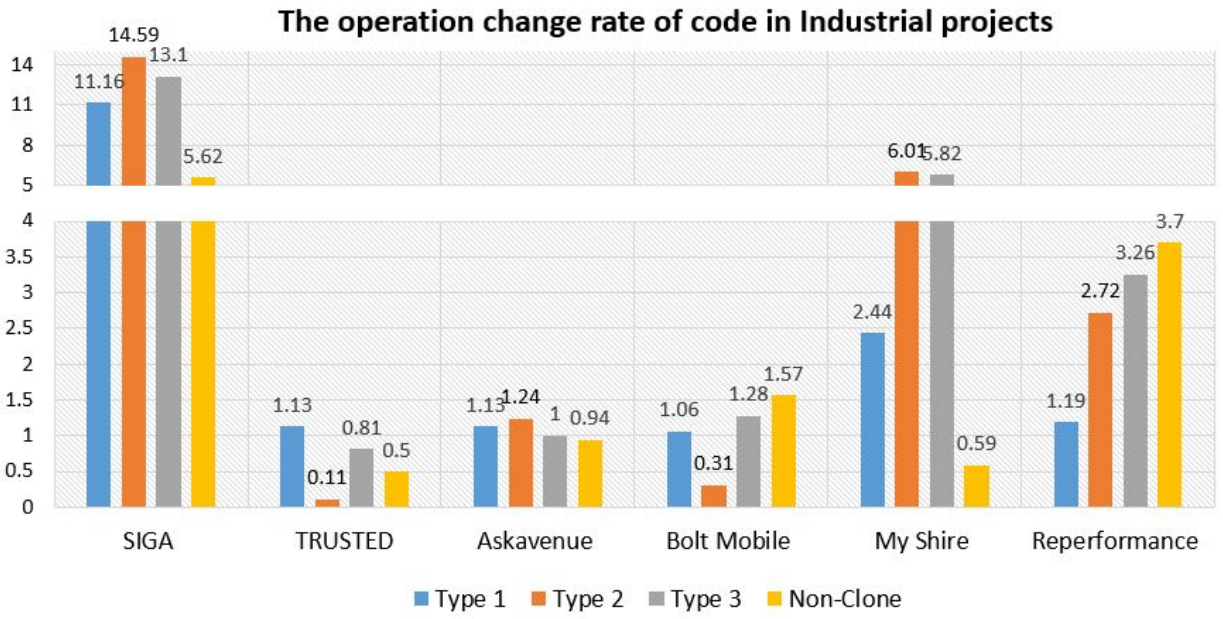
Table 4.7: Wilcoxon Signed Rank test to determine Line of code change rate significance.

Clone Types	**p-value**	U_{stat}
Type 1	0.575	22
Type 2	0.009	2
Type 3	0.005	0

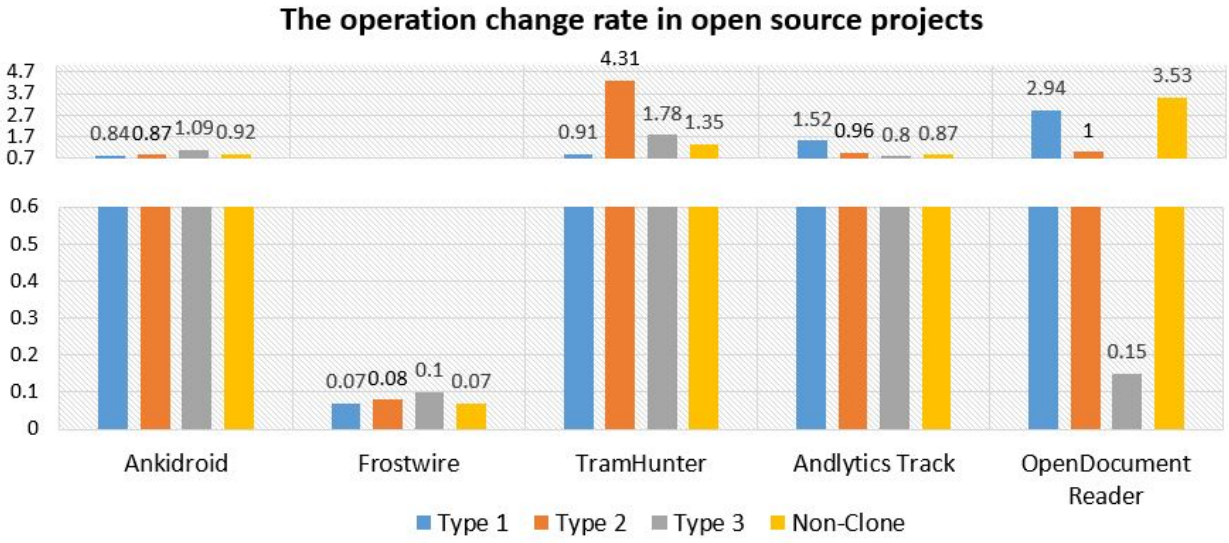
question. With the help of NiCad results, calculate the lines of three types of clone code and discard from total lines of code to measure the non-clone code present in non-bug-fixing commits. By comparing *diff* and NiCad results, we get the number of change operations and the lines of code change for each type of clone and non-clone. Then we determine the rate of change and the rate of lines of code change per thousand lines of code.

Figure 4.2 and Figure 4.3 depict the rate of change operations (deletions, additions, and modifications) and rate of lines change in three types of clone code and non-clone code in non-bug-fixing commits. We perform the statistical test (Wilcoxon Signed Rank) to know significant difference among different clone types and non-clone code. In both cases (change rate and lines of code change rate), we do not get significant results that means the p-value is greater than 0.05. So, in that sense, maintenance of three types of clone code and non-clone code in non-bug commits follow similar distribution. However, if we closely look at the Figure 4.2, four of the six industrial projects experience higher change rate in clone code than non-clone code. This is also true for open source projects. Only one project *'OpenDocument Reader'* has higher non-clone code changing rate out of five projects. We can also notice similar patterns in lines of code changing rates, which are shown in Figure 4.3. All the industrial projects have higher lines of code change in different types of clones than non-clone code, but for different projects, one of the three clones takes the lead. That is the main reason not to get a significant result from the Wilcoxon Signed Rank test. We can also see much higher line of code-change in clone code of several open-source projects.

Answer to RQ 2: Although we do not get statistical significance between each of three clone types and non-clone code, we deeply investigate each of the eleven projects and observe that most cases of different project type of clones (Type 1, Type 2, or Type 3) need more maintenance and prolongation than non-clone code. Even for multiple projects, all three types of clones have higher changing rates than non-clone. To recapitulate, in mobile app development both in open source and industrial, we also need to handle and maintain clone code in non-bugfixing commits carefully.

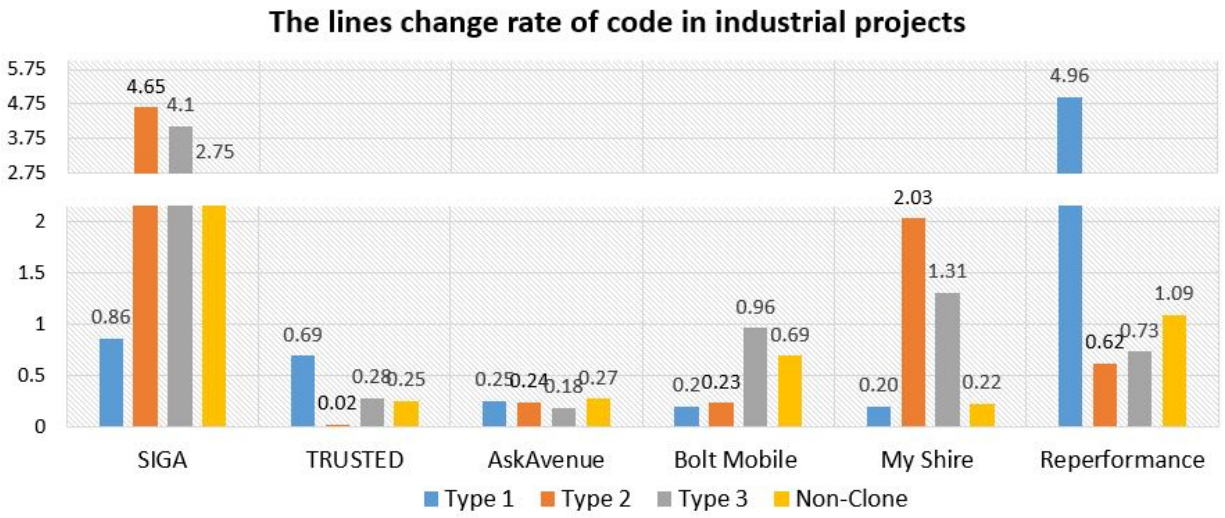


((a))

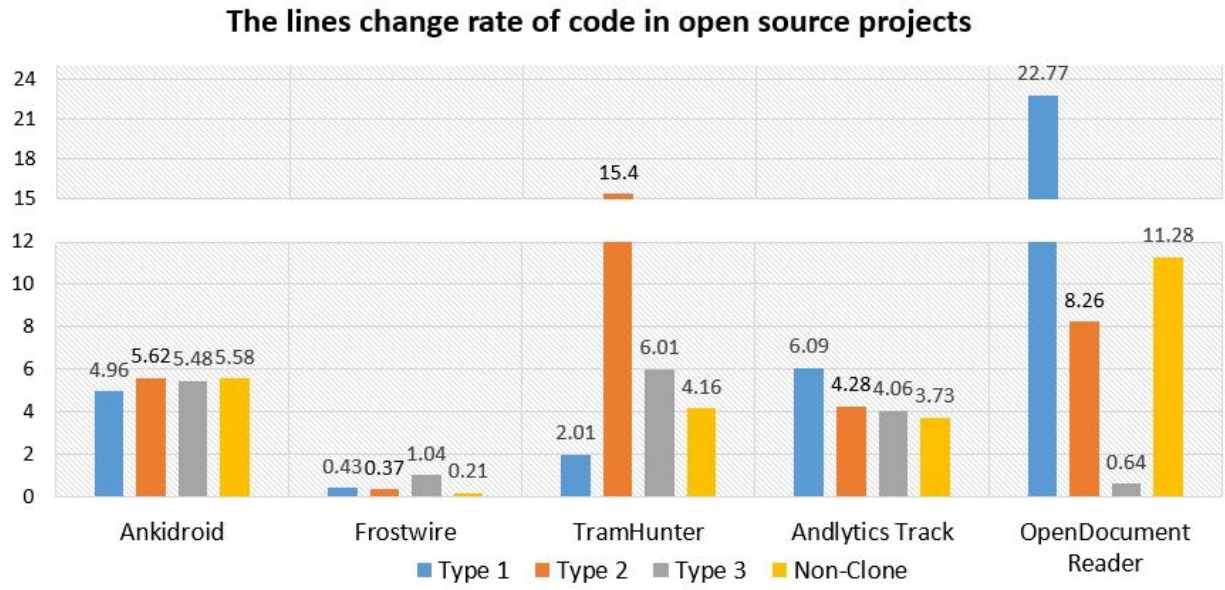


((b))

Figure 4.2: The rate of change per thousand line of code occurred in non-bug-fixing commits on different types of clone and non-clone code. (a) Industrial project;(b) Open Source projects.



((a))



((b))

Figure 4.3: The rate of lines of code-change per thousand lines of code occurred in non-bug-fixing commits on different types of clone and non-clone code (a) Industrial project;(b) Open-source projects.

4.4.3 Answering the third research question (RQ 3)

RQ 3: ***Are clones more prevalent in open source or industrial mobile applications?***

Motivation: We already know about the complications of clone code throughout bug and non-bug-fixing commits. By this research question, we want to know how frequently open-source and industrial projects use clone code from diverse sources. We measure the use of clone code and non-clone code of each project and make a comparison between the use of clone code in open source and industrial context. If we find any significant difference in the usability of clone code in two different development environments (open source and industrial), then we can carefully mitigate the threads of clone code from bug occurring and frequent maintenance of that specific development environment according to the corresponding clones' consequences.

Methodology: To measure the percentages of clone and non-clone code of each project, we simply sum up all the revisions clone code getting from the NiCad result and calculate the mean value divided by the total number of commits. We determine the average line of code (ALOC) earlier, so we get the mean lines of code of non-clone type. After that, we make percentages of those values for better representation. Figure 4.4 shows the percentage ratio of clone code and non-clone code of industrial and open-source projects, respectively where x-axis denotes the subject systems as presented in the Table 4.2.

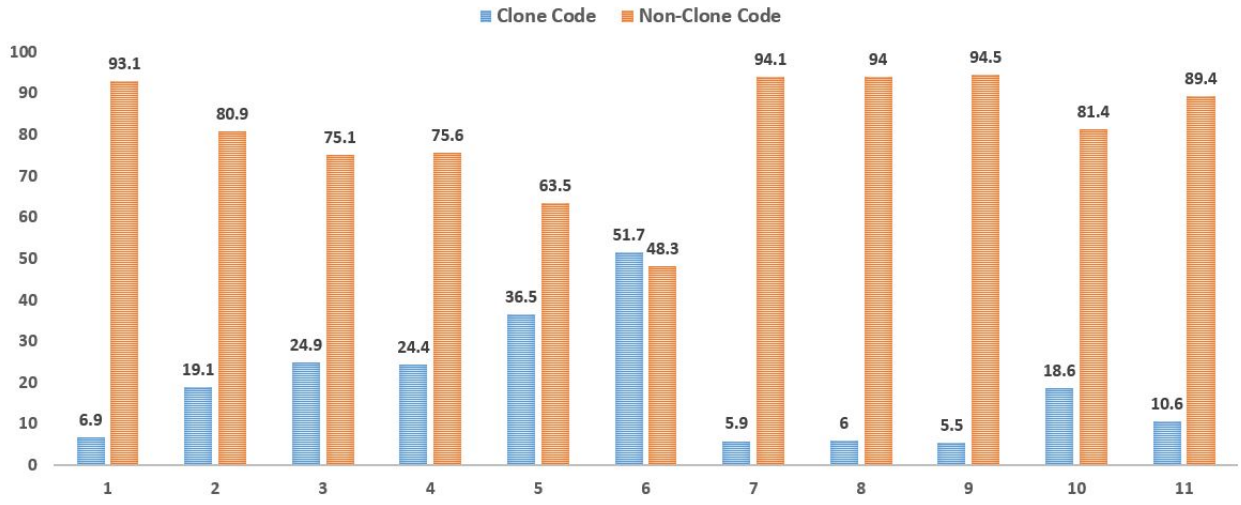


Figure 4.4: Ratio of clone and non-clone code in industrial and open source projects.

In order to know the significant difference, we perform the Mann Whitney U test [12]. We can not use the Wilcoxon Signed Rank test [20] for this case because it is only applicable for equal sample size and paired data where the Mann Whitney U test also works for different independent sample size. As we have six industrial and five open source projects, we apply this test [12] with 5% level of significance where U critical is 3. After completing the test, we get $p\text{-}value = 0.023$ and $U_{stat} = 2$ which are less than 0.05 and 3 respectively. Therefore, we find a significant difference in using clone code higher in industrial projects than in open-source.

Answer to RQ 3: After investigating the ratio of each projects' clone code for two different development environments, we discover a higher distribution rate of using different types of clone in industrial projects than in open-source mobile apps.

After knowing the answer of RQ 3, we wanted to know how these two different environments (industrial and open-source) behave with software bug having higher and lower percentages of clone code. We will answer this question in our RQ 4.

4.4.4 Answering the fourth research question (RQ 4)

RQ 4: ***Is the change-proneness of code clones in the open-source mobile apps similar to that of the code clones in the industrial mobile apps?***

Motivation: From RQ 3, we come to know that industrial mobile apps have more clone fragments (sometimes more significant than non-clone, for example, subject system six *"Reperformance"* in Figure. 4.4 carries more than 50% clones in code-base) than open-source projects. So, we are interested in knowing bug-proneness in both types of apps and the extent to which each environment needs more maintenance to tackle occurring bugs from others.

Table 4.8: Mann Whitney U test to determine significant difference in industrial and open-source apps.

Clone Types	**Change Rate**		**Line Change Rate**	
	p value	U_{stat}	p value	U_{stat}
Type 1	0.023	2	0.023	2
Type 2	0.412	10	0.121	6
Type 3	0.522	11	0.234	8

Methodology: To find the answer, we perform the Mann Whitney U test [12] (5% two-tailed level of significance and U critical is 3) between each type of clone code of industrial and open-source projects. First, we apply the rate of operations change and rate of lines of code change of Type 1 clone of industrial and Type 1 clone of open-source projects using the obtained values from RQ 1. As industrial applications carry more clone fragments, it is expected to get a higher changing rate in that environment, but the statistical testing [12] proves that open-source projects' Type 1 clone code are more bug-prone and require more maintenance than Type 1 clone of industrial projects. We do the same for Type 2 and Type 3 clone and do not find any significant differences. All the calculated testing value is presented in Table 4.8.

We perform another experiment to know how often bug-fix commits change clone and non-clone code in both (open-source and industrial) development environments. To answer, we identify the total number

of bug-fixing commits and total number of all commits where clone and non-clone code changed. Then, we calculate the percentage rate of these change to make comparison between industrial and open-source mobile apps. This experiment also aids to answer of RQ 5. We follow the following procedures to calculate all the associated measurements. We assume several short form to make the experiment understandable. CC: Total number of commits that made change to clone code; BCC: Total number of bug-fix commits that made change to clone code; PBCC: Percentage of the bug-fix commits that made change to clone code. We consider all types of clone code separately to calculate PBCC; CNC: Total number of commits that made change to non-clone code; BCNC: Total number of bug-fix commits that made change to non-clone code; PBCNC: Percentage of the bug-fix commits that made change to non-clone code. All the terms represent columns in Table 4.9. We use the following equations to calculate PBCC and PBCNC.

$$PBCC = \frac{BCC * 100}{CC} \tag{4.5}$$

$$PBCNC = \frac{BCNC * 100}{CNC} \tag{4.6}$$

We also compute overall PBCC (for every clone type) for industrial and open-source projects separately by the ratio of the sum of all BCC and CC, then make percentage and so for PBCNC (non-clone). The overall percentage of bug-fixing commits that changed clone and non-clone in industrial and open-source project are depicted in Figure 4.5. As we discover earlier that industrial projects have more clone code than open source but open source apps require more maintenance than industrial mobile apps, we get similar result from this experiment. If we observe Figure 4.5, the percentage of bug-fixing commits which were responsible for changing different types of clone and non-clone code in open-source projects is higher than in industrial projects. Additionally, clone code (specially Type 1) has more changing rate than non-clone code for the open-source mobile apps. On other hand, the bug-fixing rate of non-clone for industrial projects is sightly higher than clone code, but these clone types (Type 2 and Type 3) are prior responsible for more maintenance than non-clone code (as per RQ 1).

Answer to RQ 4: Although industrial mobile app projects have a higher clone code ratio, they require low bug-fixing maintenance than open-source projects, especially for Type 1 clones. Furthermore, open-source projects have more erroneous clone fragments than industrial projects in terms of bug-fixing commits. So, for open-source mobile app development, cloning should be treated more carefully.

4.4.5 Answering the fifth research question (RQ 5)

RQ 5: ***Is there a difference in terms of clone and non-clone code change in bug-fixing commits between mobile and non-mobile apps?***

Motivation: In our analysis and experiments, we investigated the characteristics of mobile apps (industrial and open-source) in the field of clone and non-clone study. We are also interested in investigating the relative similarity or dissimilarity between mobile and non-mobile apps (desktop based) considering bug-fixing

Table 4.9: Percentage of bug-fix commits that changed clone and non-clone fragments.

Projects	Type 1			Type 2			Type 3			Non-Clone		
	CC	BCC	PBCC	CC	BCC	PBCC	CC	BCC	PBCC	CNC	BCNC	PBCNC
SIGA	2	0	0.00%	8	1	12.50%	44	7	15.91%	284	48	16.90%
TRUSTED	1	0	0.00%	3	1	33.33%	30	4	13.33%	80	13	16.25%
AskAvenue	40	6	10.00%	116	18	15.52%	173	31	17.92%	316	55	17.41%
Bolt Mobile	1	0	0.00%	14	0	0.00%	14	0	0.00%	15	0	0.00%
My Shire	8	0	0.00%	99	11	11.11%	131	11	8.40%	171	17	9.94%
Reperformance	73	8	10.96%	250	27	10.80%	470	54	11.50%	709	76	10.72%
Ankidroid	783	171	21.84%	733	160	21.83%	1266	255	20.14%	5081	1133	22.30%
Frostwire	195	40	20.51%	203	29	14.30%	343	56	16.33%	1959	280	14.29%
TramHunter	14	6	42.86%	20	7	35.00%	26	5	19.23%	118	21	17.80%
Andlytics Track	116	8	7.14%	168	13	7.74%	333	30	9.01%	666	57	8.56%
OpenDocument Reader	12	6	50.00%	20	6	30.00%	120	23	19.17%	319	88	27.59%

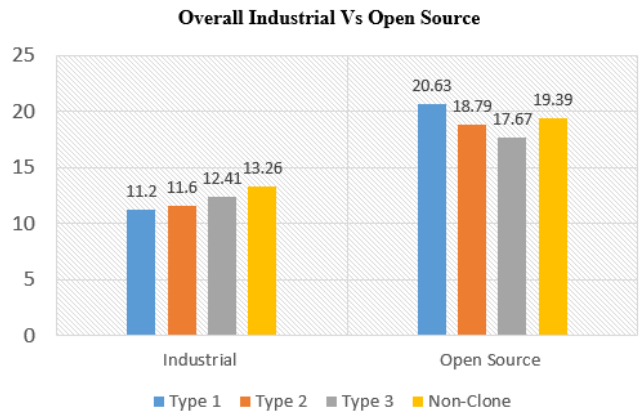


Figure 4.5: Percentage of bug-fix commits that have changed clone and non-clone fragments of Industrial and Open-source mobile apps.

commits to clone and non-clone code. With this research question, we can determine the level of effectiveness of clone and non-clone code in the creation of software bugs in two different users centric environments (mobile and non-mobile).

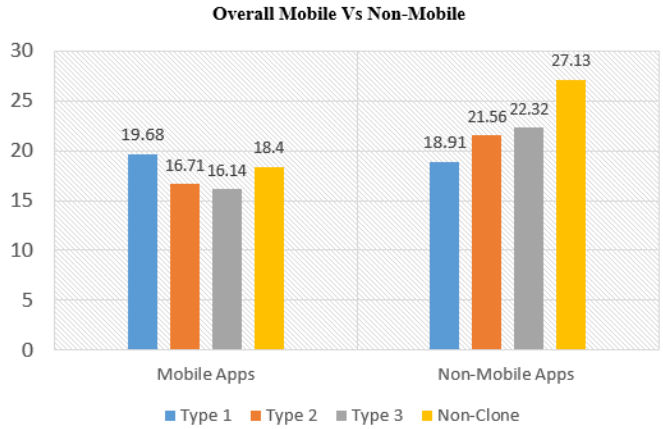


Figure 4.6: Percentage of bug-fix commits that changed clone and non-clone fragments of Mobile and Non-mobile apps.

Methodology: Islam et al. [87] conducted a similar study by analyzing seven desktop systems written in C and Java and summarised that clone code is more bug prone than non-clone code. They considered the number of files changed due to fix bugs and the number of bug fixes that changed clone vs non-clone code. However, In our study, we take into account the number of operations (additions, deletions and modifications) and number of lines of code change. To compare with Islams et al. [87] outcomes, we calculate the percentage of bug-fix commits that changed clone and non-clone code according to the authors' [87] methodology, which we already discussed when answering RQ 4. The final calculated results of our eleven mobile subject systems are presented in Table 4.9. We take the associate outputs from [87] directly to make our comparison. The overall percentages of bug-fix commits that changed clone and non-clone code of mobile and non-mobile apps

is shown in Figure 4.6. Like the method [87], we calculate the overall percentage of bug-fixing commits by using equation 4.5 and 4.6, where overall BCC and CC represent the sum of all projects' BCC and CC respectively and then make percentage. We do the same for non-clone code to calculate the overall percentage rate.

In Figure 4.6, we notice that the percentage of bug-fix commits that changed Type 2 and Type 3 clone code and non-clone code is comparatively higher in non-mobile apps than in mobile apps, which tells us that mobile apps have less erroneous code fragments. But for mobile apps change to Type 1 clone code, is sightly higher (19.68%) than non-mobile apps (18.91%). This is mainly for open source mobile apps. If we closely observe the values for Type 1 clone code in Table 4.9, the industrial percentage rates (PBCC) are much lower (most are 0.00%) than open-source projects.

Answer to RQ 5: Though the environments, number of revisions, diversity of users are quite different for mobile and non-mobile apps, it is equally important to treat different types of clone code carefully. According to our analysis, clone and non-clone code in mobile apps has a lower bug-fix change than non-mobile apps, but the difference is not negligible. For user satisfaction and lower software maintenance costs, we should be careful when cloning code in both mobile and non-mobile (desktop) apps.

4.5 Developers Survey and Discussion

We conducted an online survey with 23 developers from 17 companies who are also linked with various open-source teams. Table 4.10 shows the survey questions. We have eight closes and one open questions and summarize each responses. Two of the questions have four possible answers i.e. "Often" (3 or more times per week), "Sometimes" (1-2 times per week), "Rarely" (1-2 times per bi-week) and "Never". Figure 4.7 depicts the results of the survey.

The majority of the developers have two to four years of development experience. However, five developers from ABCD software company have been working as mobile apps developers for more than six years (one developer has ten years of experience). Categories of developer experience are shown in Figure 4.7(a) (the result of survey question 1). In the next question, we ask developers about their frequency of code cloning when developing mobile apps, and most indicated that they sometimes clones code (see Figure 4.7(b)). We asked reasons why they sometimes cloned code and most mentioned reasons like industrial project structures and core implementation similarities, predefined patterns, and unwillingness to start from scratch. However, some of them (only three) barely and never clone code because of personal choices and company restrictions. Almost all developers (95.2% in Figure 4.7(c)) clone/copy code from development Q&A sites like GitHub, and Stack-Overflow. AbdalKareem et al. [24] revealed that after reusing code from Stack-Overflow, mobile apps experience more bug-fixing commits than earlier. The statement from AbdalKareem et al. [24] is also justified by the developers' answer to survey question 4. More than three-fourths of developers (displayed in Figure 4.7(d)) experience bugs after cloning code into their project (Cloning from either the same project,

Table 4.10: Survey Questions

Sl.	**Questions**
1	How long (i.e. year) you are working as a Mobile apps developer?
2	How frequently do you clone code in mobile application development (from one file to another or one project to another project)?
3	Do industry people clone code from open source repositories such as GitHub, Stack Overflow?
4	Did you experience any software bugs generated by cloning code at the time of mobile application development?
5	How frequently do you experience bugs in apps' lifetime because of code cloning?
6	Does a higher number of files get changed because of fixing bugs in clones compared to fixing bugs in non-clone code?
7	Do you perceive any difference between the oftenness of making clones in industrial and open-source projects?
8	Do you think that code clones have more impacts (number of change, change effort, the required time to manage) on the software system than non-clone code during system evolution?
9	If you experience bugs because of code clone, then could you please define the steps you would like to take to minimize software bugs generated from code clone?

another project, or a development Q&A site), and this is also true for bug occurrences within the evolution (or lifetime) of software (answser to question 5). 47.6% of developers experienced a lower number of bugs, while more than 52% of developers struggle with the bugs that occurred in clone code throughout the apps' lifetime (depicted in Figure 4.7(e)). Moreover, this result also validates the findings of RQ 1, i.e., clone code are more responsible for software bug than non-clone code.

Like the previous survey questions, question 6 also supports the RQ1 findings. Around 57% of developers agree with the statement that says while fixing bugs; clone code need more change and maintenance than non-clone code (shown in Figure 4.7(f)). Nevertheless, the rest of the developers says non-clone code need more attention and care because, in a software codebase, the amount of non-clone code is much higher than clone code (see Figure. 4.4).

As the participating developers of this survey are from industry and connected with numerous open-source development teams, we also wanted to know the extent to which clone code is used in industry versus open source. Development of industrial mobile apps are tailored for a particular business case and driven by the client's requirements, whereas open-source allows for contributions from multiple parties. Open-source mobile apps are developed on a generalized basis so that a large number of parties can take advantage of it and also there is no time constraint from the users' side, but Industrial apps are developed for specific services/tasks with predefined timing and budget constraints so that they can focus on their service more closely and make those apps more efficient. So, more clone code in industrial projects causes difficult situations for project delivery and further maintenance. In question 7, we ask developers whether they perceive any difference between the frequencies of making clones in industry versus open-source projects. One-third of developers state that industry projects have more clone fragments than open-source, while other 23.8% claim the opposite. In answering RQ3, we identified more clone code in industrial projects than open-source projects and find statistical significance where most developers characterize industry and open-source apps in the same way regarding the presence of clone code (values are shown in Figure 4.7(g)).

In Question 8, developers' answers especially from ABCD company greatly reflect the outcomes of RQ 1, RQ 2, and RQ 4 because we analyzed their developed apps and the majority of developers (around 70% of responses from ABCD company matched the study outcomes) insist that clone code need more attention and care than non-clone code, both bug-fixing and non-bug-fixing commits. According to 71.4% of developers (in Figure 4.7(h)), the time, effort, and change required for clone code is much higher than non-clone code during apps' evolution. In the last question, we ask developers to provide some potential approaches to mitigate software bugs because of cloning. We received multiple promising suggestions; for example, one developer from ABCD says, *"If you are cloning code without understanding why, you're going to run into bugs. It's not the act of cloning code that is the problem."* That means proper understanding is important before cloning code. Some insist on reusing code as much as possible with proper functioning feasibility and modularising because the issue of code clones can never be fully remedied. From Bangladesh, one of the developers suggests, *"Following coding principles (SOLID). Also, the use of design patterns can help mitigate*

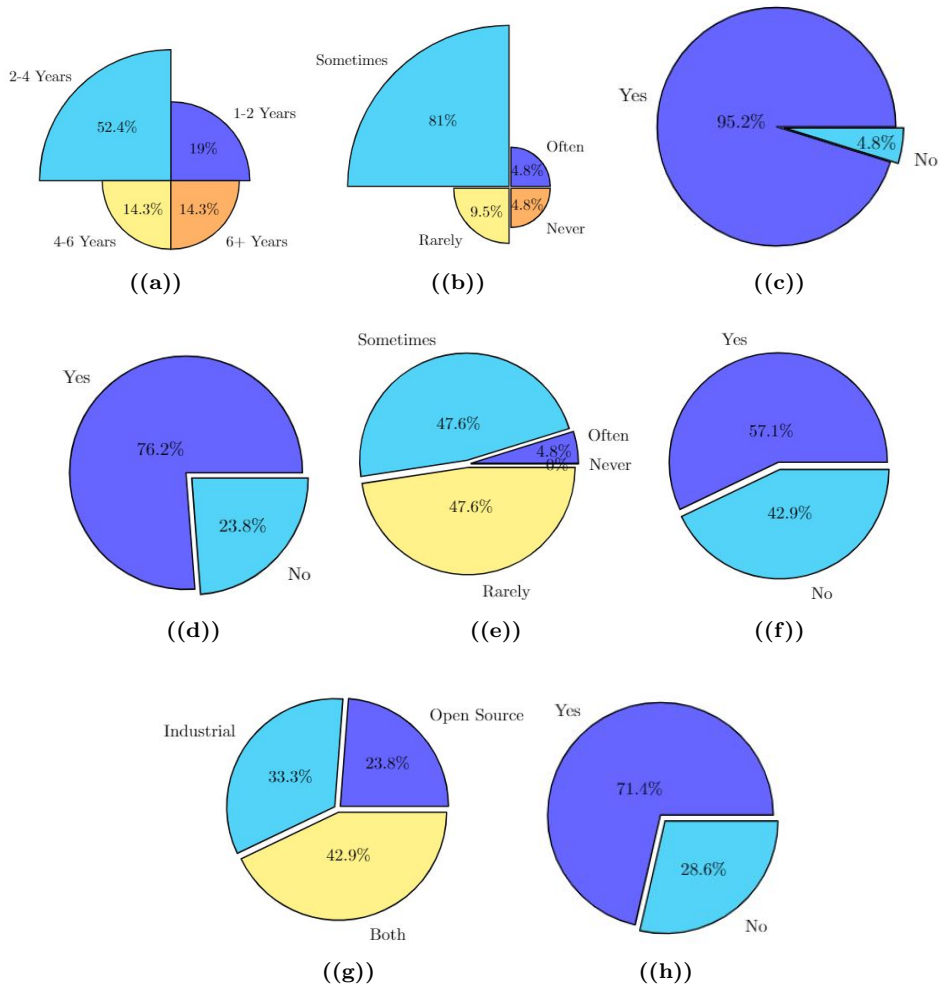


Figure 4.7: The summarised results of the developers' survey. (a) Years of experience; (b) Frequency of code cloning; (c) Code cloning from Q&A Site; (d) Occurrence of bugs after cloning; (e) Bug occurrence in apps' lifetime ; (f) Agreement of clone code change more than non-clone code while fixing bugs; (g) Agreement of clone fragment ratio in industry and open source; (h) Agreement on more change effort on clone code than non-clone code.

code cloning. Code cloning is the starting point when code starts to smell." Other suggestions like *"Follow design pattern to write code to prevent code clone. For instance, Singleton Pattern. Also, it's necessary to write OOP based code, so that minimize software bugs."*. Some of suggestions denote tool support for automated testing and follow a solid plan of action during development.

To recapitulate, our study and survey result, we can say that clone code are liable to create software bug related issues, and it needs more consideration at the time of cloning and perpetuation for whole apps' lifetime. So, it must be treated accordingly than non-clone code both for industry and open-source projects.

4.6 Threads to Validity

For detecting different types of clone fragments, we used NiCad [51] clone detector, which may suffer from confounding configuration issues [163] and might return different results in different settings. However, the NiCad settings that we used are standard and shown to be given high precision and recall [164, 165, 186, 188].

Our research is also involved with the detection of bug-fix commits by analyzing commit messages. We followed similar detection technique, which was proposed by Mocus and Votta [132]. By the technique of Mockus and Votta [132], it can be possible to select a non-bug-fix commit as a bug-fix commit erroneously. However, Barbour et al. [41] proved that the probability of success is 87% for desktop subject system, which is acceptable. To determine the accuracy of detecting bug-fixing commits in mobile apps, we manually analyzed all the commit messages of each industrial and open-source projects and found 93-100% accuracy scores. We got higher accuracy for industrial mobile apps (for example, three projects have 100% accuracy) than open-source mobile apps. Besides, open-source projects are developed by different developer groups which have different coding conventions, quality assurance and group sizes, where the selected industrial projects are developed by same development teams of ABCD software company.

4.7 Conclusion

In this paper, we conducted an in-depth comparative study of several Android and iOS apps' bug and change proneness in both clone and non-clone code. We also considered industrial projects written in Swift and Java programming languages and investigated bug-proneness because of code cloning. Additionally, we collaborated with mobile apps' developers to perceive their opinions on code cloning and software bug. Our study analyzed all major types of clones (Type 1, Type 2, and Type 3) and non-clone code to determine their consequences in bug occurrence and maintenance throughout the project evolution. We investigated thousands of revisions of eleven open-source and industrial mobile projects and analyzed commit messages to detect bug-fix commits determining change rates in clone and non-clone. According to our investigation, clone code (especially Type 2 and Type 3) are more bug-prone than non-clone code. Furthermore, we observed that code clone occurs more frequently in industrial projects but experiences a lower amount of change in Type 1 clone code than in open-source projects. We found that mobile apps are less bug prone than non-mobile apps

while code cloning. We believe that our exploratory analysis of bug-fix commits and clone and non-clone code characteristics are valuable for understanding clone management in open source and industrial mobile software maintenance and make understandable to treat mobile and non-mobile (desktop) apps equally.

Moreover, the survey of 23 developers from 16 different companies across Canada and Bangladesh strengths our study outcomes concerning code clones usability and effects in code maintenance. Besides cloning code from one file to another, developers frequently copy/clone code from SO. In the survey, we received experiences of more than 75% of developers who witnessed software bugs because of the cloned/reused code from SO. Therefore, the next chapter will convey the impacts of SO code elements in the software codebase of open-source and industrial mobile apps.

5 Cloning and Consequences of Stack Overflow source code: a study on open-source and industrial mobile applications

As discussed in Chapter 4, developers reused SO code elements into their projects and experienced bugs in later revisions. So, further investigations are warranted regarding the impacts of SO code snippets in mobile apps. In particular, software bugs and code maintenance that cost trillions of dollars every year, including fatalities such as a software bug in a self-driving car that resulted in a pedestrian fatality in March 2018 and the recent Boeing-737 Max tragedies that resulted in hundreds of deaths. As a result, a number of studies were conducted about usability and how developers acquire insights from SO regarding implementation problems in open-source mobile apps. However, to the best of our knowledge, no existing work covered the actual impacts of SO source code reused in mobile apps in the context of change-proneness, bug-occurrence, and at the time of bug-fixing. Additionally, no study incorporated industrial mobile apps to study the impacts of SO code snippets. In this study, we conducted an exploratory study in order to investigate the change-proneness and bug-proneness of reused SO code fragments considering ten open-source and industrial mobile apps. Following our selected dataset, our analysis exhibits, 1) the proportion of reused SO code is comparatively higher in industrial mobile apps than open-source; 2) open-source projects mostly reuse SO code to enhance existing features where industrial projects reuse for adding new features into the app; 3) SO code fragments are significantly more change-prone than non-SO code; 4) SO code snippets are responsible for bug occurrence in later revisions, and that is comparatively higher in industrial projects than open-source. Our experimental results can assist the SO, research, and mobile developer communities to strengthen usability and concerns to facilitate code-quality improvement and minimize software bugs due to SO code.

The rest of the chapter is organized as follows. Section 5.1 discusses the background of this work and research questions. Section 5.2 shows the motivating example behind this work. Section 5.3 depicts several studies related to this work. The overall methodology is discussed in Section 5.4. In Section 5.5, key findings and answer of the research questions are presented. Threats to the validity of this study are presented at Section 5.6 and finally, concludes this chapter with Section 5.7.

5.1 Introduction

Stack Overflow (SO) is the progressively essential and most widespread crowd-sourced forums for developers and practitioners due to the lack of conventional and official learning resources [153, 160, 195], where the developers search for solutions by asking questions and share potential knowledge by answering questions. On the other hand, customer satisfaction, software quality, efficient timing, and cost-effectiveness are primary concerns during software development, especially for mobile apps. Therefore, software developers usually reuse similar and new source code fragments from different sources such as open-source sites (e.g., GitHub) [85], external libraries [118], developers forums and questions answering sites [161, 169] (e.g., SO) in a consistent manner. Although, the usage of external source code and development constraints are considerably different in open-source and industrial projects in terms of security, quality of support, and various policies [17], source code cloning is a widely accepted practise in both environments [118, 158].

Several existing studies [58, 145, 156, 178, 182] investigated that SO code snippets might be toxic, error-prone and sometimes code are irreproducible to its actual usage. On the contrary, studies [42, 149, 161] showed insights into the usability of SO code in software development. Furthermore, a plethora of research considered open-source mobile apps and SO code snippets that introduced possible domains where SO code can be reused, such as small development teams with time constraints [24, 190], novice or less experienced developers [24, 133] and when required resources are limited [133]. A similar prior study [24] illustrated how much, why, who, and when SO code snippets are reused in open-source mobile apps. According to Abdalkareem et al. [24], the amount of reused SO code varies to different apps, enhancement of existing code is the potential reason for reusing, more experienced people in smaller teams/apps and less experienced people in large team/apps mostly reuse SO code. Moreover, mid-age and older apps reused SO code snippets later in the lifetime. They also claimed that the open-source projects that reused the SO code snippets experienced more bug-fixing commits than before the reuse, which might affect the app's quality. However, they did not examine the actual consequences of the SO code elements regarding how they behaved upon the rest of the commits, change-proneness, and bug-proneness. In addition, none of the studies illustrated industrial mobile apps towards reusing SO code where the same group of people are involved with the majority of mobile projects of a company. If one of the team members has benefited from using SO code in their developed project, then it is possible to have SO code in all projects developed by the same team. If we are aware of further consequences of reused SO code, either positive or negative, we would be able to revise development strategies accordingly to the companys' benefits.

Unlike open-source projects, commercial companies do not really want to use or have limited access to the external source code, such as third-party libraries and available crowd-sourced sites (e.g., SO, Quora) [158]. Instead, most monetary software industries use their built-ins to prevent code leaking, fraudulent code behaviour, attributions, and license violations [158]. We surveyed 23 mobile app developers from 17 different commercial companies in order to understand the usages and later consequences of SO code snippets. We

noticed that around 95% of the developers usually copy/paste (code cloning) code fragments from SO to their projects and that over 75% of such developers experience bugs after such activities (the code reusing from SO). Moreover, software bugs and failures cost trillions of dollars every year, including severe catastrophic incidents such as pedestrian fatality in March 2018 by an autonomous car, Boeing-737 Max tragedies, Therac-25 radiation therapy system and other software-control arrangements that resulted in hundreds of deaths. To leverage the study gap regarding industrial mobile apps and actual consequences of SO code snippets, we studied ten open-source and industrial mobile apps containing thousands of revisions along with more than two million SO answer posts that contain Java code snippets to get insights into the usage diversity, change-proneness, and bug-proneness of SO code in the context of mobile apps. We formulate four research questions (RQs) for the study, which are shown in Table 5.1. In RQ1, we noticed that the amount of reused SO code varies among selected mobile apps and that reused proportion is significantly higher in industrial projects than open-source. Enhancing existing code and integrating new features are the most potential reasons for SO code reusing, discussed while answering RQ2. From the analysis of RQ3, we experienced that SO code fragments encountered more code-change during app maintenance than non-SO code. Although SO code snippets are reused in bug-fixing commits, they (reused SO code) are also a source of software bugs in later revisions (examined in RQ4). Our experimental results can benefit both SO and mobile developer communities to get insights into code reusability, maintenance and software bugs in the context of mobile apps development.

Table 5.1: Research questions of this study.

Serial	**Research Question**
RQ1	What is the amount of SO code snippets reused in mobile apps? Is there any quantitative difference between open-source and industrial mobile apps regarding the reuse of SO code? If so, is it significant?
RQ2	What are the possible reasons for reusing SO code? Do industry developers reuse code for similar purposes as open-source developers?
RQ3	How stable are the reused code snippets? What is the rate of code-change of SO code snippets throughout the app evolvement? Is it significantly higher or lower than non-SO code change?
RQ4	Does SO contain buggy code snippets that reused in mobile software codebase?

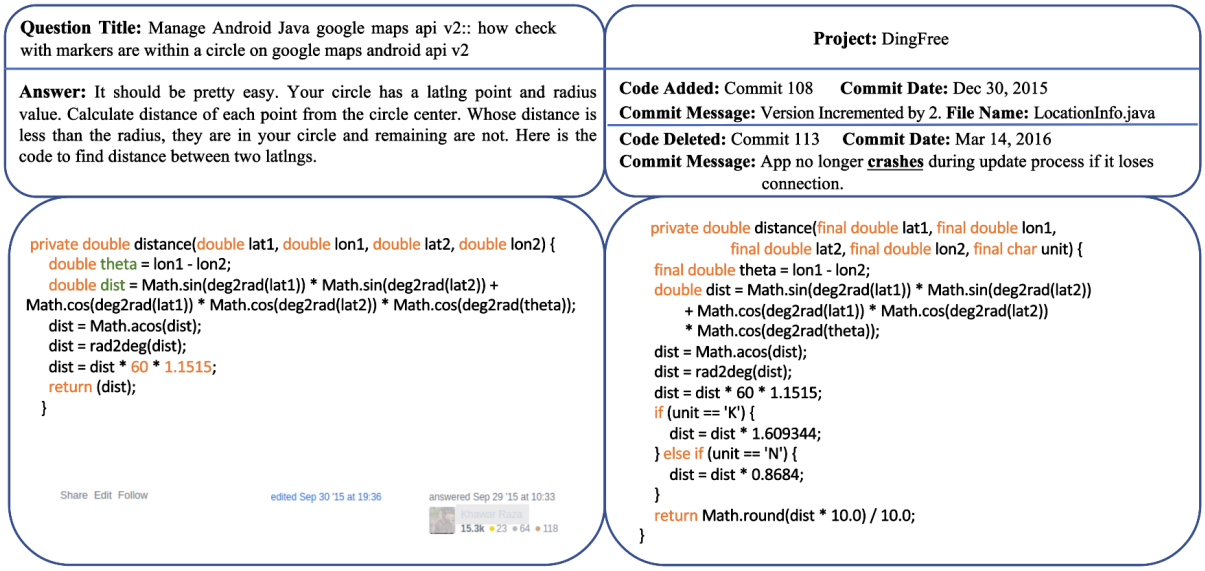


Figure 5.1: An example of buggy SO code snippets reused in an industrial project.

5.2 Motivating Example

Existing studies [24, 42, 161] revealed that mobile developers seek assistance regarding implementation problems during app development by asking questions in SO, and even for open-source mobile apps, developers reused SO code snippets for different purposes. However, although the studies are performed with open-source projects, there is no existing knowledge concerning reusing SO code snippets in industrial mobile apps where external code cloning is almost prohibited or limited, especially from crowd-sourced sites [158]. Additionally, the question is raised regarding the bug-proneness of resued SO code in mobile apps. Therefore, we present the motivation of our research, where SO code snippets have been reused in mobile apps of the ABCD (blinded for reviewing purpose) software company and later induced bugs. Figure 5.1 portrays an example where an answer post[1] with a code fragment that was reused in a project named "DingFree" of ABCD company of its 108^{th} revision and later in 113^{th} revision, the reused SO code snippets is deleted to fix a crash bug.

The answer post[1] illustrates a solution to the functional issue that maintained a specific location within a circle. Using the timestamps (answer posted date and project commit date), we can determine that the SO code snippet was cloned or reused in the industrial project after it was available on SO. However, in the 112^{th} revision, the reused code-block occurred a system crash when it lost internet connectivity (as per commit message of 113^{th} revision). As SO code elements induce software bugs (as of motivating example), the detailed analysis of SO code snippets is warranted significantly regarding bug occurrence and further maintenance.

[1] https://stackoverflow.com/questions/32840709/

5.3 Related Work

Abdalkareem et al. [24] conducted an exploratory study on several open-source Android apps to understand the usage of code elements from SO. They mainly investigated the *how much, why, when and who* questions related to cloning code from SO. Additionally, they mentioned that the Java files with SO code snippets experienced more bug-fixing changes than before the reuse of SO code. However, they did not analyze whether the reused SO code snippets were indeed responsible for bugs or not. Moreover, their study was limited to SO code fragments that have a length of 30 or more lines, whereas the lengths of SO code were below 20 lines in most cases in the entire dataset. Our study analyzed the impact of SO code elements reused in open-source and industrial mobile apps in the context of bug-proneness and change-proneness in later revisions. We selected all the Java code snippets (except traditional loop or if-else statement) from SO answer posts and investigated to what extend SO code elements could be responsible for software bugs and later maintenance. As a result, developers and practitioners from mobile industries would understand the implications of reusing SO code elements into their apps.

Lotter et al. [120] investigated code reuse between SO and open-source software (OSS) from SourceForge and GitHub. Their research aimed to find the code clone rate 1) within SO posts, 2) between SO and OSS and 3) between one OSS to another OSS. Their quantitative analysis showed around 1% of SO code reuse within popular Java projects, and code reuse from one OSS to another revealed up to 77.2%. They also talked about the appropriate use of code reuse and awareness of attribution requirements. Similar kind of study [126] examined key trends and inspect the co-change patterns of code fragments on SO and GitHub projects.

Nikolaidis et al. [58] presented a relationship between the presence of reusing code taken from SO and technical debt (TD) of a target system and found that TD density is significantly lower and also majority of reused code maintained the design and code rule violations, which confirmed the high efficiency and code quality of SO code snippets. Where another study [155, 156] says, SO contains around 20% outdated code which might create TD in future software maintenance and 66%-69% developers never check license requirements at the time of reusing SO code snippets.

A preliminary study [27] investigated the behaviour of SO code snippets on software cohesion and reveal that in 70% of copied cases, SO code reduces the cohesion of recipient classes and, therefore, lowers the code quality. Their analysis indicated that developers should be more concerned when reusing SO code into their projects' code-base. Another research [129] evaluated the code quality of SO answers' code fragments in terms of reliability and programming practices.

SO is a profoundly popular learning and knowledge-sharing online crowd-sourced forum where users ask for help on various domains. Among them, web and mobile apps related topics asked and discussed more and more by the developers [42]. Rosen et al. [161] showed that mobile developers usually ask about almost all issues related to apps development, where mobile apps configuration and user interface got the highest

number of questions to answer. According to the evidence of [25, 161], it says that mobile app developers get knowledge or code snippets and use them in their projects if needed, both on open-source and industrial platforms.

Several studies have been conducted on mobile app development regarding code reuse. For example, Ruiz et al. [168] observed that in several Android apps, around 23% of Java classes inherit from one of the base classes API and 27% from domain-specific classes. In addition, they examined 217 mobile apps and concluded that another app completely reuses these apps. Other studies [131, 189] showed that mobile apps intensely rely on third-party libraries and APIs.

Apart from all existing researches on open-source mobile apps and code reuse from SO, here in this study, we explore several industrial mobile apps written on Java along with open-source and investigate the SO code usage and stability in further maintenance. Furthermore, we also inquire about the bug-proneness and reasons for SO code reusing both for open-source and industrial mobile software perspectives. However, to the best of our knowledge, no existing studies covered industrial projects. Furthermore, to what extent the actual implications of SO code elements differ from open-source mobile apps regarding usability, software bugs and later code maintenance was not investigated earlier.

5.4 Study Design and Dataset Preparation

This study intends to perform an empirical analysis on the reuse of SO code in open-source and industrial mobile apps. In order to understand the additional maintenance effect of reused SO code, we extract all code fragments from SO answers related to Java and select five open-source and five industrial mobile apps for exploratory analysis. Figure 5.2 depicts the overall approach of this study. The following subsections discuss the dataset preparation and experimental design of our study.

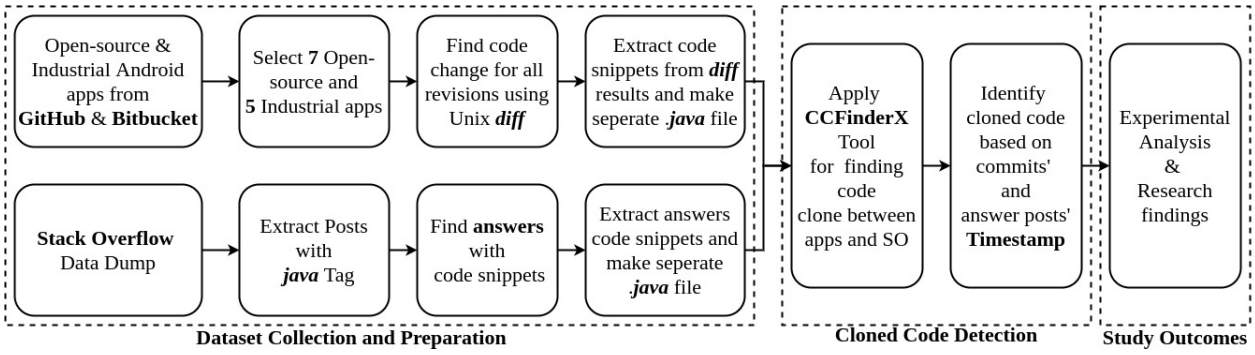


Figure 5.2: Overall methodology of our study.

5.4.1 Mobile Apps Dataset Preparation

To find reuse code from SO in mobile apps, we require source code from the mobile apps' codebase. For this study, we select ten open-source and industrial mobile projects. These apps have variations in domains,

sizes, number of revisions, and whether another study [24] used them as a dataset. An important factor for selecting these apps is considering a broad range of application domains such as travel, music and video, shopping and so on. We also choose the most popular and satisfactory apps of the ABCD software company in diverse domains with differing sizes and revisions. All of these mobile apps are downloaded from online GitHub[2] repositories (for open source projects) and BitBucket[3] which is the development site of ABCD Software company (for industrial projects). Table 5.2 shows an overview of mobile apps data preparation steps.

Table 5.2: Preparation of Mobile dataset from codebase.

Step	# Quantity
Total number of open-source mobile apps	5
Total number of industrial mobile apps	5
Total number of revisions (both in open-source and industrial)	18, 528
Total number of revisions where Java code change	11, 405
Total number of '.java'file created using *diff* result	1, 18, 474
Average lines of code per file (excluding outliers)	12.61
and Median (excluding outliers)	8.00

We extracted all the revisions (aka. commits) of each mobile apps from the respective repositories and got 18, 528 revisions of the entire 10 projects. Developers usually add, modify or delete code segments in each revision to solve particular issues or features, and it might be the high time of reusing code from open-source development sites such as SO. Numerous operations (additions, deletions, and modifications) are performed during the development stages and eventually form a final revision. So, it is not an efficient idea to study code reuse from SO only in the last revision and therefore, we consider each revision to the time of this study conducted. Next, we make *.java* files (in original filename) by integrating all the code differences between to consecutive revisions identified by UNIX command ***diff***. We separate the latest revisions' changes as new (newly added or modified code) and earlier revisions' changes as old (previous code that was deleted or modified at later revision). For example, if there is a file named *SampleFile.java* in revision n and n+1, then the newly added/modified code in revision n+1 and the deleted code from revision n will be formed *SampleFile_new.java* and *SampleFile_old.java*, respectively.

We experienced code-change in 11, 405 revisions that constructed 1, 18, 474 Java files that will be used in the clone detection stage along with SO code snippets. We excluded all the code snippets containing only variable renaming, adding extra import statements or identifiers to reduce the dataset being compared effectively. We also calculated the average lines of code and got 12.61 LOC per file while excluding the outliers.

[2]https://github.com/
[3]https://bitbucket.org/product

5.4.2 SO Dataset Preparation

TO build the SO code snippets dataset, we download SO posts from Stack-Exchange Data Explorer [1] via SQL query. The site [1] aims to make queries from SO original database covering every data and metadata of SO posts like authors, question body, answers, votes, creation date and so on. But the limitation is that it can return only fifty thousand rows at a time while performing a particular query. Therefore, we apply the same query mentioned with post tag *'java'* again and again and get the result from the beginning to August 04, 2021. In that time, SO contained $53,603,853$ posts and $4,073,112$ posts with *'java'* tag containing $6,495,603$ answers. A similar study by Abdalkareem et al. [24] further filtered with *'Android'* tag to reduce the dataset, while another study Cheon et al. [49] illustrated that Android and Java apps often share the same codebase because of having unique programming language. As a result, we stick to consider all the code fragments tagged with *'java'*.

Table 5.3: Preparation of SO dataset from answer posts.

Step	# Quantity
All posts in the SO dataset(up-to August 04, 2021)	$53,603,853$
All posts tagged with 'java'	$4,073,112$
Number of answers present in 'java'tag	$6,495,603$
Number of answers contain code snippets	$2,169,249$
Number of code snippets contain Java code	$2,159,056$
Code snippets with $>$ 20 lines	$2,17,064$
Code snippets with $>= 10 and <= 20$ lines	$3,76,730$
Code snippets with $<$ 10 lines	$1,575,450$
Average lines of code per snippets (excluding outliers)	9.07

In the next step, we identify the relevant answers which have code snippets. We find $2,169,249$ answers have code snippets containing Java and non-Java (ex. JavaScript, XML, and stack traces) code. To separate the Java code, first, we extract the code snippets with the help of HTML tag '$<code><$ $/code>$', which surrounds the code elements. And then, we make individual *.java* files for each code snippet. Next, to eliminate the non-Java code elements, we apply regular expressions to get only code snippets that secure proper Java syntax. After eliminating non-Java code, we also delete the statements that started with 'import', 'Scanner', and blank lines, and finally get $2,159,056$ Java files ready for future experiments.

We determine the number of lines of each code snippet and find the minimum length of one line and the maximum length of $1,458$ lines with an average of 9.07 lines of code per file. A prior research [149] suggested that the excellence of quality of code depends on the number of votes it holds. Accordingly, Abdalkareem

et al. [24] focused only on code snippets containing 30 lines of code which have high average votes than the code snippets of less than 30 lines. In our case, we consider all the code snippets received from SO answers. Because, when preparing the mobile apps dataset, we found code changes (additions, modifications or deletions) between two consecutive revisions, mostly 5 to 15 lines of code, and some were even less than five lines of code. Therefore, we wanted to find every possible code reuse from SO answers that have lower code lines. Besides, at the end of comparing SO and mobile apps code snippets, we check manually to remove the lower lines of reused code fragments (if any) that might contain common and straightforward code, such as if-else and loop statements. Table 5.3 summaries how we prepared this final number of SO code snippets.

5.4.3 Detection of Reused Code from SO

After preparing the SO code snippets and mobile apps code-change throughout the app lifetime dataset, it is time to find the SO code fragments reused in both open-source and industrial mobile apps. Therefore, we apply CCFinderX [96], a clone detection tool that follows the token-based approach to identify exact (i.e. Type 1) and near-miss (i.e. Type 2) clones, which implies that the two source code fragments have a similar syntax except for variety in identifiers and literals. We restricted our experiment to Type 1 and Type 2 clones to reduce the number of potential false positives associated with the Type 3 clone, created by adding and deleting code lines in Type1 and Type 2 clones. Moreover, we also did not consider Type 4 or semantic clone (same results but different implementation) for this study.

Since CCFinderX detects clones using token matching, we must set the number of tokens it should consider while detecting similar code. This study wanted to know the detailed use of SO code, even for smaller code snippets. So we set the token match count 50, which is the lowest and default configuration of CCFinderX. There are several reasons behind the selection of CCFinderX. 1) it can work on code snippets having syntactical errors, while other clone detectors need compilable code snippets as input. As almost all code snippets from SO and mobile apps are not complete and have syntax issues, CCFinderX is best suited for these kinds of datasets; 2) it detects exact and near-miss clones, which is the main goal of investigating our research questions; 3) it provides great efficiency in CPU and memory usage. The prerequisite to run CCFinderX is a 32 bit operating system (we use Windows 7 and it does not work in Windows 8 or above) and python version 2.6 (not work on python 2.7 or above). Initially, we faced difficulties related to memory issues when executing all datasets at a time because it has more than two million Java files. Later, we split the SO dataset into five halves and ran them multiple times separately with mobile apps dataset and got smooth clone result with acceptable running time; 4) it returns multiple clones (if any) of varied sizes in single code fragment and such cases, we considered the largest matched clone.

So far, we discuss the first part of clone code detection shown in Figure 5.2. When analyzing CCFinderX outputs after a sequence of processing to get exact line numbers of clones, we saw a large number of code clones between apps vs SO, apps vs apps, SO vs SO, and even a single code snippet has multiple identical smaller code blocks. An existing research [38] showed that SO itself is a source of duplicate code segments.

Therefore, ignoring other clones, we selected only the clone pairs between SO and mobile apps Java files. Still, we are not aware whether the mobile apps' developers reused those code fragments from SO or not. To identify the actual reused code from SO, we followed the idea of Abdalkareem et al. [24] and assembled the answers' post date (not questions' post date) and mobile apps commit date of each clone pair. If the commit date is later than the date of the answer post in SO, we consider these code elements reused in the development stages of the corresponding mobile apps. All the qualitative and quantitative analyses regarding reuse clone pairs, lines of code newly added, modified or deleted are described in Section 5.5.

5.4.4 Statistical Significance Analysis

We verify some of our experimental results using two statistical methods. First, Cohens' kappa coefficient (k) [4], which is a measurement to evaluate inter-rater agreements between two raters. For example, one of our research questions (RQ2) needs classification agreements between two graduate students on the same resources. The coefficient score helps to assess their mutual agreements and acceptability of the classification. Cohens' kappa coefficient (k) value varies between -1 to +1, and the more positive value, the more efficient the agreement is. Second, Mann Whitney U test (also called Wilcoxon Signed Rank Sum test) [12] is a non-parametric statistical method that allows statistical inference without making the assumptions that the sample has been taken from a particular distribution (i.e., normal). We employ the Mann Whitney U test for independent samples data to encounter the significance between open-source and industrial, SO and non-SO code snippets of mobile apps. Besides, we consider 5% level of significance and two-tailed test (because of testing statistical significance in both directions and calculating the p-value) and observe the critical U value from the statistical chart [6].

Table 5.4: Statistics of CCFinderX output.

Item	# Quantity
Total number of clone pairs bet^n SO & mobile apps	41,644
Total number of code snippets originally reused	1,302
Total number of unique posts reused	176
Total number of revisions used SO code	327
Total number of Java files contained SO code	210
Total lines of code from unique posts reused	2,958

5.5 Experimental Results and Analysis

Before digging into our research questions, we wanted to perform a preliminary analysis on CCFinderX output to quantify how many clone pairs are detected as clones between SO and mobile apps' code snippets. Initially, many clones were detected between SO code snippets, between SO and mobile apps' code snippets, and between mobile apps' code snippets. But, for this study, we extracted only the clone pairs that detected between SO code elements and added code fragments of mobile apps of each revision. As a result, we detected 41, 644 clone pairs where SO code snippets matched with the same code snippets from mobile apps having different lengths of clone size. Ignoring the duplicates and getting the largest clone size, we got 1, 302 SO code snippets used originally in the mobile app's codebase according to timestamp (SO answers' post date prior to the commit date). We also experienced multiple reuses of one single code snippet of SO that dig up 1, 302 individual SO answers snippets with 2, 958 lines of Java code. Throughout our ten mobile apps, SO code is used in 327 commits and 210 Java file contain the reused SO code snippets. Table 5.4 and Table 5.5 represent the statistics of CCFinderX output and our subject systems with its type, domains, total number of revisions and average lines of code, respectively. In this section, we elaborately justify our experimental findings and relate them to find the answers to our aforementioned research questions.

Table 5.5: Subject systems and percentage of reused code from SO.

Projects	Type	Domain	Revisions	ALOC	Code Reuse(%)
Siga	Industrial	Entertainment	519	3211	4.61
Bolt Mobile	Industrial	Service Provider	170	2165	4.20
Its Mylife	Industrial	Life Style	230	1921	1.92
Trusted	Industrial	Online Directory	182	32317	0.66
Ding Free	Industrial	Finance	240	11443	1.42
AnagramSolver	Open	Word	50	45728	0.03
Andlytics Track	Open	Shopping	1524	38274	0.65
Ankidroid	Open	Education	9298	49185	1.80
Frostwire	Open	Media & Video	6027	243848	0.40
Tramhunter	Open	Travel	288	10783	0.63
				Average	1.68
				Median	1.04

5.5.1 Answering the first research question (RQ1)

RQ1: ***What is the amount of SO code snippets reused in mobile apps? Is there any quantitative difference between open-source and industrial mobile apps regarding the reuse of SO code? If so, is it significant?***

Motivation: Code reuse is a frequent trend among developers, especially from open-source development sites like SO. Although several studies [58, 156, 178] revealed that SO might have toxic code elements which increase technical debts or even produce software bugs, developers use code from SO regularly. A prior study [24] showed that open-source mobile developers reuse code fragments in their apps but did not study such insights in the context of industrial mobile apps. In closed-source development environments, external code reuse is limited or almost impossible to protect source code from leaking [158]. A survey of 23 industrial developers from 17 different companies presented that more than 95% of developers frequently clone or reuse code from crowd-source development sites like GitHub and SO. Although industries give less access to external source code, it is promising to know the amount of SO code used in closed-source mobile apps and find out the difference between open-source and industrial mobile apps in terms of SO code usages. If there is some difference, then we need to know they are statistically significant or not.

Methodology: To find the answer, we calculated the percentage of reused SO code present in each app with their average lines of code. Previous research Abdalkareem et al. [24] also considered the same open-source mobile apps as we did. But our dataset preparation in both SO and mobile apps was utterly different than Abdalkareem et al. [24]. In particular, we selected all the SO code snippets to feed into the clone detector (CCFinderX) where they used only code fragments that contain 30 lines of code or more. It creates a huge gap where the majority of SO code snippets are less than 20 lines (See statistics in Table 5.3). Therefore, it is needed to recalculate the percentage score of reused code from SO in open-source projects, and which would give us the actual scenario of SO code reuse in open-source projects.

Table 5.5 (column 6) shows the percentage score of cloned code reused by each mobile app. For example, among the industrial apps, "Siga" has the highest amount (4.61%) of code elements came from SO answers' posts and also had the lead among all projects, and "Trusted" has the lowest number (0.66%) of SO code snippets. In open-source projects, "Ankidroid" and "AnagramSolver" possessed the highest (1.80%) and lowest score (0.03%) respectively where "AnagramSolver" had the lowest amount of SO code elements in all projects. We also measured the average amount of SO code reused for an app is 1.63%, and the median is 1.04%.

If we closely analyze the percentage of code reuse (in Table 5.5 column 6) in mobile apps, it seems industry developers use SO code more than open-source developers. To verify the difference and know the statistical significance, we apply the Mann Whitney U test [12] with a 5% level of significance and two-tailed test. With five industrial and fine open-source projects, we got critical value of U at $\alpha = 0.05$ is 2 [6]. From the test calculator [12], we obtained U statistics is 2, which is equal to U critical and p-value is 0.03662, which is less than 0.05 (at 5% level of significance). Hence, the difference is statically significant that says, in our dataset,

industrial mobile apps have more reused SO code elements than open-source mobile apps.

Answer to RQ1: Although studies noted that SO has code smells and commercial companies have limited or no access to external source code, developers from open-source and industries reuse code snippets from SO in their apps' development stage. Moreover, we found closed-source projects have higher ratio in reused SO code than open-source, and their distribution is statistically significant.

5.5.2 Answering the second research question (RQ2)

RQ2: ***What are the possible reasons for reusing SO code? Are industry developers reuse code for similar purposes as open-source developers?***

Motivation: As we experienced in the first research question, open-source and industry developers use SO code in their codebase, and closed-source mobile apps have more SO clone code than open-source. Now the question is, what are the possible reasons for reusing SO code snippets. Is the reuse pattern different in open-source and closed-source mobile apps? By answering this research question, developers from both environments can better understand the purpose and usage of SO code snippets in their apps development stages. Additionally, it is promising to know how SO helps industry people where commercial organizations have limited access to external source code, especially open-access sites. Overall, this RQ reveals some insight into the usability of SO Java code snippets in Android app development.

Table 5.6: Purposes of SO code reuse in mobile apps.

Categories	% of Commits		
	Open Source	Industrial	Overall
Enhancing existing code	**28.67%**	36.36%	**29.45%**
Adding new features	19.11%	**42.42%**	21.47%
Refactoring	10.58%	0%	9.51%
API usage	4.10%	0%	3.68%
Fixing bugs	19.45%	9.09%	18.41%
Test	6.48%	6.06%	6.44%
Other	11.60%	6.06%	11.04%

Methodology: To answer this aforementioned research question, we followed similar qualitative analyses as as of Abdalkareem et al. [24]. But the difference is that at the time of finding reuse purposes of SO code snippets, prior study [24] examined only commit messages. However, we considered both commit messages and SO code snippets reused in specific revisions and manually classified them in seven predefined categories as of Abdalkareem et al.[24]. Their experiment [24] contained only 135 commits that cloned SO code in the 22 selected open-source mobile apps because they restricted their analysis to only SO code snippets with 30

lines of code or more, which prevented the actual reuse of code snippets in mobile apps. We considered all code snippets from SO and ensured that all the clone pairs truly reused identified by the CCFinderX clone detector in mobile apps development both in the open-source and industrial sectors. As a result, we spotted a total of 327 commit messages (294 for five open-source projects and 33 for five industrial projects) and 209 Java files that carried SO code snippets of our ten mobile apps.

To categorize the selected commit messages, two graduate students independently and separately investigated the commit messages associated with SO code snippets used in the corresponding commits and finally classified them into seven groups (i.e., enhance existing code, adding new features, refactoring, API usage, fixing bugs, test and other). Each graduate student completed their groups by reading and analyzing commit messages and code snippets and sometimes learning from comments in original SO answers' posts. After completing the classification, they shared their findings and decided the group of commits messages that arose difficulties to classify. Finally, they met on an agreement and categorized 327 commits into seven groups. Table 5.6 depicts the percentage of commits reused SO code snippets of each category of possible reused purposes. The columns of Table 5.6 represent seven categories, percentage of commits only for open-source mobile apps, percentage of commits only for industrial mobile apps and overall percentage of commits for both open-source and industrial mobile apps respectively.

Before discussing the result of categorization, we performed Cohens' kappa coefficient [4] to evaluate the level of agreement between the two graduate students. Therefore, we determined the coefficient k that shows the level of agreement between the two students to be +0.803, and it is an excellent reliable agreement to justify the categorization [64].

In Table 5.6, we can see that around 51% commits of our selected mobile apps use SO code for enhancing or modifying existing code and for the purpose of adding new features. Although the number of investigated commits varies between open-source 294 commits (1.71% of all commits) and closed-source 33 commits (2.46% of all commits) projects, industry projects have more reused code from SO than open-source projects (discussed and visualized in Section 5.5.1 and Table 5.5). So, it is promising to know the individual score of each category both for open-source and industrial mobile apps. Hence, we calculated the score and found that open-source developers mostly reuse SO code to enhance their existing code (28.67%), where industry people mostly employ SO code snippets for adding new features (42.42%) in their projects. Additionally, there was no usage of SO code in industrial projects regarding refactoring and API. Commercial organizations usually avoid open-source API or third-party libraries because of security concerns, probability of bug occurrence and having licenses violations issues [26]. However, even most software industries develop their own secure libraries and use them in their projects. According to the selected industrial projects, reuse API from SO might have harmful effects or license violation issues that prevent API insertion in commercial mobile apps. Refactoring is the way to improve the internal structure of functions or codebases without changing the actual behaviours. Unfortunately, our selected industrial projects did not use SO code snippets regarding refactoring (0%), but open-source did (10.58%). Therefore, industry people do not regularly change already developed

properties or features because commercial projects typically receive proper instructions and specifications from targeted customers or clients directly and are developed by the same team (all the industrial projects in our dataset are developed by the same team of ABCD software company). Additionally, refactoring is labour-intensive, costly, and an error-proneness process that might increase maintenance work [106]. On the other hand, many collaborators or developers shared open-source projects from different parts of the world (projects "Ankidroid" has more than 110 collaborators) and designed and kept open for all. According to our investigation, open-source developers often need to change design patterns while keeping the goal fixed, and SO code snippets significantly associate with overcoming these issues.

Our analysis also reveals that SO is a great resource of potential code snippets that help bug-fixing in open-source (19.45%) and industrial (9.09%) mobile apps. Although the percentage is low for industrial projects, developers from both environments always seek assistance when needed from SO experts, and commercial companies also encourage to get help by posting questions and share knowledge by answering questions [3]. Besides, SO code snippets also help in testing or reviewing code quality in both development environments. Finally, category "Other" contains the commit messages that were not descriptive enough to classify and also, the two raters did not become capable of making a precise classification by examined actual reused SO code. Commit messages like "tags", "dump version 2.0", "cleaning" are classified as "others". However, this was a small percentage of the spotted commits and whatever the reasons, developers reuse SO code snippets significantly in their mobile apps.

Answer to RQ 2: Based on RQ2 findings, the SO community will have an idea about what purposes developers from open-source and industrial mobile apps use SO code snippets to support more effectively by providing error-free and bug-proneness-free Java code. Open-source developers mainly reuse SO code to enhance existing code, where industry developers reuse for appending new features to their mobile apps. SO has also contributed to refactoring and API usage in open-source projects where commercial companies avoid these because of maintenance costing and security issues. Moreover, developers from both platforms get help regarding bug-fixing from SO answers. Finally, the researchers and the developers' community will know how SO resources are used in mobile app development. If the SO community can develop techniques to manage fraudulent API, code smells and other bug-proneness matters, it might be an excellent resource for the mobile developers' community.

5.5.3 Answering the third research question (RQ3)

RQ3: ***How stable are the reused code snippets? What is the rate of change of SO code snippets throughout the app evolvement? Is it significantly higher or lower than non-SO code change?***

Motivation: From the above research questions, we learned how much SO code resued and for what purposes in mobile app development stages. Although SO is one of the most popular crowd-sourced sites, prior studies [58, 156, 178] revealed that SO code might be toxic and, therefore, might need further maintenance after the reuse, which is not cost-effective. Thus, it might be promising to know the behaviours of reused SO code in our selected open-source and industrial mobile apps. Moreover, developers can reuse SO code

more effectively if they know the actual performances of SO code in real systems. By this research question, we would get the insights into what extend SO code needs additional maintenance work and find out the modification differences with non-SO code throughout the evolution process of mobile apps.

Methodology: At the time of preparing the mobile dataset, we identified the code fragments, which are modified and deleted in the subsequent commits by UNIX *diff* command. We extracted the code elements and named them as "_old.java" in corresponding apps' revisions. Like each commits' newly added code elements, we also applied CCFinderX to find the SO clone fragments, which are modified and even deleted in a particular commit. From the CCFinderX output, we separated the clone pairs where SO code snippets matched with "_old.java" files (such as *ConnectionHelper_old.java*). Then, we manually examined each new and old file to confirm whether the clone block was changed in the new revisions. For example, suppose one clone is paired between SO code snippet and the *ConnectionHelper_old.java* file in commit n (the reused SO code was reused in one of the commits between 1 to $n-1$). The commit n also has the new file as *ConnectionHelper_new.java*, which is created by the modification of *ConnectionHelper_old.java* for the next revision $n+1$. In the next step, we identified the SO code snippet from the *ConnectionHelper_old.java* (the old file also contains code elements other than SO code which are modified or deleted from the current revision) and compared it with the *ConnectionHelper_new.java* files' code elements. If the identified SO code snippet was changed or missing, we concluded that SO code experienced code-changes and therefore, noted the number of lines of code which were changed. After checking all the matched SO code snippets manually, we counted the lines of code that were modified or deleted and calculated the rate of code-change by the following equation 5.1. We also determine the code-change rate of non-SO code utilizing the output of *diff* command and average lines of code of each project via equation 5.2. Table 5.7 shows all the calculated values of SO and non-SO code. To determine the stability of SO and non-SO code, we used the concept of [69, 108, 123] considering lines of code changed throughout all revisions. There are other methods of measuring stability regarding modification frequency and code age which are not applicable for this situation because there is a full possibility of having non-SO code in each revision but SO code are not. In equations 5.1 and 5.2, CRSO and CRNSO stand for the percentage of code-change rate in SO and non-SO code respectively.

$$CRSO = \frac{LOCs\ changed\ in\ SO\ Code\ *\ 100}{LOCs\ reused\ from\ SO} \tag{5.1}$$

$$CRNSO = \frac{LOCs\ changed\ in\ non\text{-}SO\ Code\ *\ 100}{ALOCs\ of\ non\text{-}SO\ \ *\ \ total\ commits} \tag{5.2}$$

In Table 5.7, the rates of lines of code changed in SO code is much higher than in non-SO code. For example, project "Its Mylife" has a maximum code-change rate of 70.27% (26 lines changed out of total 37 lines of code reused from SO), and two mobile apps experienced no change in reused code snippets from SO. Thus, the overall change rate is 21.26% (629 lines modified or deleted out of 2958 lines of code in total), and 25.57% SO answer posts (45 out of 176 posts) are modified or deleted after being reused in mobile

Table 5.7: Percentage of code change rate between reused SO and non-SO code.

Projects	% rate of SO code change	% rate of non-SO code change
Siga	25.00%	0.66%
Boltmobile	8.79%	1.10%
Its Mylife	70.27%	1.86%
Trusted	0.00%	0.05%
Ding Free	11.04%	0.98%
AnagramSolver	0.00%	0.06%
Andlytics Track	16.14%	0.12%
Ankidroid	25.59%	0.57%
Frostwire	23.09%	0.02%
Tramhunter	36.76%	0.42%

apps' codebase. Although more than 74% SO code snippets are consistent throughout the app evolution, the code-change rate of SO code is still much higher than non-SO code and hence, SO code is less stable than non-SO. To find the statistical significance between the code-change rate of SO and non-SO code, we apply the Mann Whitney U test [11] with a 5% level of significance and two-tailed test. From the statistical chart [6], we got the U-critical = 23 for the 5% level of significance. From the Mann Whitney U test, we calculate U-statistic = 20, which is less than U-critical and p-value = 0.02575, which is less than 0.05. Hence, the SO and non-SO code-change rate distribution is different and statically significant, proving that SO code are more change-prone than non-SO code.

Answer to RQ3: According to our analysis, 21.26% lines of code and 25.57% of reused SO posts in the selected ten mobile apps experienced further maintenance. The change rates are much higher in SO code than in non-SO code, which is statistically significant. Although the majority of the SO code is unchanged, the change-proneness is not to be disregarded. Therefore, developers should be careful when reusing code snippets from SO. Now, the concern is what proportion of code-change was performed at the time of bug-fixing. Thus, we can get insights into the bug-proneness of SO code reused in mobile apps. Although developers from both open-source and industrial apps reused SO code elements to fix bugs (according to RQ2), it is equally important to explore whether SO code induce software bugs or not. We discuss this concern in our next research question.

5.5.4 Answering the fourth research question (RQ4)

RQ4: ***Does SO contain buggy code snippets that reused in mobile software codebase?***

Motivation: In the previous research question, we acknowledged that SO code needs more maintenance than non-SO code. As a result, questions might arise about why the developers updated the code. Was it

for regular code changes or bug-fix changes? Abdalkareem et al. [24] claimed that the files that reused SO code experienced more bug-fixing changes after the reuse. In specific, the number of bug-fix commits before reusing SO code is lower than the number of bug-fix commits after the reuse. But they did not investigate whether the SO code snippets were responsible for the bugs after the reuse or not. Also, we learned from RQ2 that 18.41% of commits reused SO code snippets to fix software bugs. So, it is necessary to analyze the behaviours of SO code fragments either reliable for bug-fixing or responsible for occurring bugs. Thus, developers can act accordingly to the beneficial or adverse effects of SO code elements while using in project codebase.

Table 5.8: Characteristics of SO code for bug-fixing and bug generation.

Projects	**#BC**	**#BC added SO code**	**#BC modified SO code**
Siga	70	2	2
Boltmobile	17	1	0
Its Mylife	26	0	1
Trusted	30	0	0
Ding Free	42	0	1
AnagramSolver	5	0	1
Andlytics Track	140	3	1
Ankidroid	1598	40	2
Frostwire	713	13	5
Tramhunter	11	1	0
Percentage of total bug-fix commits:		**2.26%**	**0.49%**
Percentage of total number of posts:		**36.81%**	**7.98%**

Methodology: To find the answer to RQ4, first, we identified the bug-fixing commits (BC) after SO code were reused for the first time following the idea of Mockus et al. [132] to spot six specific keywords (i.e., "bug", "fix", "fixup", "error", "crash", and "fail") in the commit messages. However, Barbour et al. [41] proved that the probability of success using the idea of Mockus et al. [132] is 87% for desktop subject systems, which is acceptable. We also manually scrutinized all the commit messages of each industrial and open-source project and found 93-100% accuracy scores. Second, we determined the number of bug-fixing commits that reused SO code from the investigation result of RQ2 (SO code reuse for bug-fixing) and the number of bug-fixing commits that modified or deleted the previously used SO code fragments to fix bugs from the analyses of RQ3. If one or more code fragments are deleted or modified in a particular bug-fix commit, then it is an implication that the modifications of those code fragment(s) were necessary for fixing the corresponding bug. We noted that these code fragments are responsible for that bug occurrence. It is

also true for newly added code fragments in bug-fixing commits, noting that the code fragments are used to solve the occurred bugs. The number of total bug-fixing commits, the number of bug-fixing commits reused SO code, and the number of bug-fixing commits that modified or deleted SO code snippets of each project are shown in Table 5.8.

From Table 5.8, we see that about 2.26% of overall bug-fixing commits used SO code (obviously along with non-SO code if needed) to solve software bugs that occurred in the codebase. Open-source projects reused more SO code snippets (2.31% of total open-source bug-fixing commits) to solve software inconsistencies than industrial projects (1.62% of total industrial bug-fixing commits). On the other hand, only 0.49% of overall bug-fixing commits modified or deleted SO code snippets to fix the confronted bugs. Nevertheless, the SO code snippets occurred more bugs in industrial projects (2.16% of industrial bug-fixing commits) than open-source (0.36% of open-source bug-fixing commits) which turned modification or deletion of SO code. On the contrary, 36.81% of SO answers' posts were used to fix bugs, and only 7.98% of SO answers created bugs in later revisions (the total unique SO answers' post reused in our mobile dataset is 176. Details are in Table 5.4). Although a limited portion (7.98%) of the reused SO posts created software bugs, it is not negligible toward the quality of mobile apps. Moreover, a single bug could be responsible for catastrophic incidents that could cost massive economic disruption[4], including fatalities that result in death or severe injuries such as pedestrian fatality by an autonomous car[5], Boeing-737 Max tragedies[6], Therac-25[7], scheduling errors for mammography tests of elderly British women, leading to hundreds of premature deaths[8] and so on. Therefore, although SO is one of the most popular crowd-sourced sites and developers from distinguishing platforms get benefits from reusing SO code elements, developers should carefully reevaluate SO code when incorporating it into the software codebase. Because reused SO code fragments induced bugs in our selected mobile apps and hence, SO is a possible source of buggy code along with other code smells such as outdated code, attributions and licence violations [156]. A potential code review technique is warranted to strengthen SO code elements that might increase code quality and minimize app inconsistencies.

In general, the answer score and the answerers' reputation score reveal the quality of an answer for a particular question. The higher the answer score of an answer post, the more efficient it is. The answer score depends on the number of upvotes and downvotes it gets from the users of the SO community. A regular user can see only the difference between upvotes and downvotes as an answer score. Additionally, the upvotes and downvotes score affects the reputation score of a user. The higher the reputation of a user can be more trusted and received new privileges than low-reputation users [2]. In this experiment, we intensely scrutinized the answer scores and users' reputation scores of the SO answer posts resued for bug-fixing and generated bugs in our selected open-source and industrial mobile apps. According to our analyses, we could not find any correlation among users' reputations, answer scores, buggy, and non-buggy SO code snippets. For example,

[4]https://tek.io/2FBNl2i
[5]https://bbc.in/3B44YRH
[6]https://abcn.ws/3jl21WV
[7]https://bit.ly/2KU9IR2
[8]https://bit.ly/2E1fYap

two answerers of the two buggy-SO code snippets have reputation scores 19,806 and 425, respectively. On the other hand, users with reputations 875 and 21,737 posted non-buggy-SO code snippets of asked questions. Although [2, 124] says that a user with a high reputation score has diverse subjects expertise, the answers given by the user might be negatively valued by other SO users. For example, a user with a reputation of more than 7K has given an answer with a score of -1. Likewise, we noticed buggy and non-buggy answers with +5 and -1 answer scores. So, both user reputation and answer score seem inadequate to authenticate the code quality when reusing code fragments from SO. Possible supports might consolidate from domain experts at the time of answer submission regarding code quality. However, manual code quality analysis is strenuous and time-consuming, referring to an automatic code review system integrated with SO.

Answer to RQ 4: SO code are frequently reused in mobile apps to assist bug-fixing commits, especially for open-source apps. On the other hand, the industrial projects reused less SO code for bug-fixing but experienced higher bugs occurred by the SO answer posts than opensource projects. The percentage of SO posts that generated bugs is limited but can not be ignored as software bugs can cost even life beyond the limit.

5.6 Threats to Validity

To detect the reused code snippets from SO, we analyzed the output of CCFinderX [96], which may suffer from confounding configuration choice problem [200] that might have influenced our study outcomes for different settings of the tools. The setting that we used for CCFinderX for this experiment are considered standard and with these settings, CCFinderX has been shown to detect Type 1 and 2 clones with high accuracy [96]. However, there might have been false positives. To help mitigate this issue, we manually examined all the clone pairs identified as reused code snippets and ensured all the pairs were indeed clones to each other.

Like Abdalkareem et al. [24], we considered commits' and SO answer posts' timestamp to identify the reused code fragments (SO answer posts' date is prior to the commit date). So, it might be possible that the detected code snippets were not originally reused rather, the performed commits were delayed by the developers. In RQ2, we determined why developers reuse code from SO by categorizing commit messages manually. However, the categorization may be biased and inclined to human error. Two of our graduate students (previously worked as mobile apps developer) manually investigated the commits messages and reused code fragments to alleviate this threat. Then they gave their opinions regarding each commit classification and came to the point of agreement. We also applied Cohens' kappa coefficient [4] to evaluate their mutual agreement and got an excellent inter-rater agreement with the value of +0.803.

Finally, we examined industrial mobile apps collected only from the ABCD software company. Additionally, we have also analyzed open-source projects developed by different-sized distinguish teams, where industrial apps mostly share the same development team.

5.7 Conclusion

Code cloning or reusing from one source to another is a frequent phenomenon at the stages of the development of any software system. In this study, we conducted experiments regarding the usage and implication of SO code snippets in ten open-source and industrial mobile apps. As a result, we noticed that the ratio of reused SO code snippets varies for different mobile apps. Furthermore, industrial mobile apps carried more SO code elements than open-source apps, and the difference is statically significant. The overall purpose of using SO code fragments is to enhance existing code elements but specifically, closed-source projects used SO code to add new features into mobile apps. Additionally, the resued SO code snippets are more change-proneness and need potentially high maintenance than non-SO code throughout the evolution of the app lifetime. Moreover, developers from the mobile industry reused SO code snippets to fix software bugs, but also experienced software inconsistencies because of the reuse code fragments. Our study outcomes indeed provide some relevant insight into the reuse of SO code in mobile apps development. Our findings inform the SO community to what extent the SO code snippets are reused in mobile apps and then acknowledges the consequences of buggy SO code by making efficient code fragments in their answers. Our study further implies that, the users of more expertise should participate in answers evaluation and improvement regarding the code quality so that mobile developers can have benefits to the secure use of SO code snippets with proper attributions and license requirements. Automatic investigation of code level properties (code review) that prevent further bug occurrence and notify developers about the code smell before reusing in mobile apps might be an extension of our work in the future, as well as integrate more industrial and open-source projects along with other crowd-sourced sites.

6 Conclusion

6.1 Concluding Summary

The development of mobile apps has become to be one of the fastest-growing areas of software communities. With the growth of mobile apps usage and associated intelligent technologies, the maintenance of mobile apps is inevitable regarding enhancing performances, applying advanced development, reducing error, and overall steady customer satisfaction. To help mobile app developers regarding software development and later maintenance, we conduct three studies in the context of bug classification, bug-proneness and stability of source code during app evolution stages, comparative analysis between open-source and industrial mobile apps, source code reuse from Stack Overflow (SO) and the consequences in bug-fixing and bug creation and finally significant premise to choose better code snippets for SO. We believe our conducted studies contribute significant insights into the mobile development environment both in open-source and commercial projects.

Our first study collected 2,700 mobile apps bug reports collected from different open-source developer sites (e.g., GitHub, F-droid, Trac, and Google Code). Then, we applied several supervised and unsupervised machine learning algorithms to classify the collected Android and iOS mobile bug reports. However, almost all the existing similar studies used traditional desktop-based software bug reports to categorize a maximum of three classes where we classified mobile bug reports into four classes (i.e., Crash, Energy, Functionality, and Security) to understand the bug severity in more specific contexts which help take appropriate decision and allocate related resources in urgent basis. Therefore, we first attempted to apply a clustering algorithm (i.e., K-means) to categorize the bug reports into four classes. However, due to excessive overlapping among class types, we did go for manual labeling. Finally, after so many manual hours of two graduate students, we labeled each bug reports based on specific features. We also prepared a feature set of each type of bug that helps to classify the newly occurred bug. We fed the features into five supervised algorithms and got the most acceptable result for the Support Vector Machine (SVM) algorithm with an f1 score of 91%. Apart from some shortcomings such as data imbalance (explored more than a hundred Android and iOS applications to find security and energy bug reports but experienced inadequate numbers), our analysis helps mobile developers and maintainers to take appropriate actions and resources clearly (because bugs are reported in developer sites are sometimes messy and not properly formatted) and get insight into which bug types of mobile application are frequently proclaimed so that developers give more attention in this regard.

In the second study, we did an in-depth comparative analysis of eleven open-source and industrial Android and iOS mobile apps bugs and change- proneness in clone and non-clone code. In addition, our study analyzed

all significant types of clones (Type 1, Type 2, and Type 3) and non-clone code to determine their consequences in bug occurrence and maintenance throughout the project evolution. According to our investigation, clone code (especially Type 2 and Type 3) are more bug-prone than non-clone code. Furthermore, we observed that code clone occurs more frequently in industrial projects but experiences a lower change in Type 1 clone code than in open-source projects. We also conducted a comparative study between mobile and non-mobile applications and found that mobile apps are less bug-prone than non-mobile apps while code cloning. By this research, mobile developers can get insights into the negative effect of code cloning that cloning or reusing similar code from one file to another or one project to another might have the meaningful possibility of occurring bugs into their apps and need more maintenance. In addition, the study is also promising to understand clone management in open-source and industrial mobile software maintenance and make it understandable to treat mobile and non-mobile (desktop) apps equally.

The third study showed a practical use of source code of crowd-sourced developer sites, specifically Stack Overflow (SO), in the context of mobile applications. We investigated ten open-source and closed-source mobile apps written in Java with the immense resources of SO questions and answers. We selected only the answers which contained only Java code snippets. For the mobile dataset, we extracted all the added, modified, and deleted source code from the thousands of revisions of each mobile apps. According to our experimental results, industrial mobile apps contained more SO code elements than open-source apps. In order to understand the statistical significance, we applied the Mann-Whitney U test and got a significant result. In industrial projects, developers reuse SO code mostly for adding new features when developers from open-source mostly do for enhancing existing features. We also experimented with the behaviour of SO code elements throughout the app evolution and found that SO code snippets are more change-prone than non-SO code elements. SO code snippets also help fix potential bugs but later make inconsistencies in the apps, especially for closed-source projects. We did an intensive analysis on buggy and non-buggy SO code properties to get insights about choosing bug-free code fragments. We tried to correlate users' reputation scores, answer scores, buggy and non-buggy code snippets but failed because a user with a high reputation score shared buggy code snippets in SO and vice versa. Likewise, we noticed buggy and non-buggy answers with high and low answer scores. So, both user reputation and answer score seem inadequate to authenticate the code quality when reusing code fragments from SO. Possible supports might consolidate from domain experts at the time of answer submission regarding code quality and automatic code review systems integrated with SO. With the outcomes of our study, the SO community knows to what extent the SO code snippets are reused in mobile apps and then acknowledges the consequences of buggy SO code by making efficient code fragments in their answers. As a result, SO experts would explore more potential techniques and tools to make code snippets more reliable for reusing in mobile apps development.

For study 2 (Chapter 4) and 3 (chapter 5), we answered a total of nine research questions (five for study 2 and four for study 3). Table 6.1 shows a short summary of each research questions.

Table 6.1: A short summary of all research question of this thesis study.

Serial	Question	Summary Answer
RQ1	What is the rate of change in clone and non-clone code in bug-fixing commits?	Type 2 and Type 3 clones are more bug-prone than non-clone code, statically significant.
RQ2	What are the impacts and characteristics of clone code and non-clone code throughout the evolution of non-bug-fixing commits?	Clone code has higher maintenance and prolongation (for most projects) than non-clone code but not statically significant.
RQ3	Are clones more prevalent in open source or industrial mobile applications?	Industrial projects have higher clone ratios than open-source statically significant.
RQ4	Is the change-proneness of code clones in the open-source mobile apps similar to that of the code clones in the industrial mobile apps?	Open-source projects have more erroneous clone fragments (Type 1) than industrial projects,statically significant.
RQ5	Is there a difference in terms of clone and non-clone code change in bug-fixing commits between mobile and non-mobile apps?	Mobile apps have a lower bug-fix change than non-mobile apps
RQ6	What is the amount of SO code snippets reused in mobile apps? Is there any quantitative difference between open-source and industrial mobile apps regarding the reuse of SO code? If so, is it significant?	Code ratio varies, average LOC 1.68%, median 1.04%. Higher SO code in industrial projects than open-source, statically significant.
RQ7	What are the possible reasons for reusing SO code? Do industry developers reuse code for similar purposes as open-source developers?	Mostly for enhancing existing code and adding new features. No refactoring and API usage in industrial projects.
RQ8	How stable are the reused code snippets? What is the rate of change of SO code snippets throughout the app evolvement? Is it significantly higher or lower than non-SO code change?	SO code are less stable than non-SO code i.e., high change-prone, statically significant.
RQ9	Does SO contain buggy code snippets that reused in mobile software codebase?	SO contains buggy code. Domain expertises' participations and automatic code review tools are warranted.

6.2 Future Work

In this thesis, we focus most on the source code copied from different sources and pasted in mobile app codebase and their possible implication regarding maintenance, software bugs, and bug reports classification. We plan to extend our study along with building tool support to the following concerns in the future.

Bug Report Classification Framework and Recommendation for Solutions Based on Historical Events:

In our first study (Chapter 3), we classified bug reports based on supervised algorithms. We plan to extend this study by collecting more bug reports and generating algorithms to get significant features automatically. Then, we try to prepare a framework that will be attached with app development sites (where developers and users report app inconsistencies) so that the reported bug is classified accordingly. Additionally, based on historical data, i.e., what kind of actions and resources were allocated in similar situations, the framework can suggest possible actions to mitigate the reported inconsistencies. Thus, it could save both time and cost.

Toxic Code Detection at the time of Cloning:

Clone code experienced more software bugs than non-clone code (according to study 2 in Chapter 4). So, we plan to build a plugin that will be installed with the developers' IDE. When the developers try to clone code fragments from one file to another, one project to another, or from any crowd-sourced sites such as SO, they get a warning based on the features of previously cloned code that created software inconsistencies after the reuse. This plugin will help mobile developers be more careful when cloning code from internal or external sources and working as a history-driven application. Existing IDE-based clone detection approaches [215] could be adapted first to detect mobile app clones in real-time and then to have features like detecting toxic cloned fragments at the time of cloning. Along the same line, it would be interesting to see what sort of changes [34], in particular what types of changes that developers make in copy/pasted code are more likely to introduce bugs and then warn the developers in real-time.

It would be also interesting to see which copied fragments have more possibilities of replicating bugs [86] and for what types of clones [138] at the time of reuse and take necessary actions in advanced. By understanding the evolution of mobile app clones as of traditional software [172] could provide insights in the above studies proposed. This could also help find the mobile app clones that are more important candidates for refactoring as of SPCP-Miner [139].

Automatic Code Completion and Cross Language Answers Code Snippets:

In study 3 (Chapter 5), we showed mobile developers reused SO code snippets in their developed apps, and most of them are deleted or modified in a later revision. So, we plan to design a code completion generator that will automatically refactor the code snippets, which has the possibility of occurring bugs. That means

changing the code element without altering its functionality. Additionally, we experienced fewer resources for other mobile development areas such as iOS than Android apps. Therefore, a developer from iOS (e.g., Swift, Objective C) might be faced similar problems as developers from Android (i.e., Java). As a result, developers need the same code snippets in their targeted languages. We also plan to integrate the functionality with the code generator for better community services.

References

[1] Stack-exchange data explorer viewing query. accessed on: August 2021. *Online Available:* https://data.stackexchange.com/stackoverflow/query/new.

[2] How does "reputation" work? accessed on: September 2021. *Online Available:* https://meta.stackexchange.com/questions/7237/how-does-reputation-work.

[3] Should i avoid telling others that i ask questions on stack overflow? accessed on: July 2021. *Online Available:* https://workplace.stackexchange.com/questions/97200/should-i-avoid-telling-others-that-i-ask-questions-on-stack-overflow.

[4] Cohen's kappa. accessed on: September 2021. *Online Available:* https://en.wikipedia.org/wiki/Cohen%27s_kappa, .

[5] Cohen's kappa, understanding cohen's kappa coefficient. accessed on: September 2021. *Online Available:* https://towardsdatascience.com/cohens-kappa-9786ceceab58, .

[6] Critical value chart for wilcoxon signed rank test. *Online Available:* https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470776124.app1.

[7] Elbow method (clustering), accessed on: June 2021. *Online Available:* https://https://en.wikipedia.org/wiki/Elbow_method_(clustering).

[8] Wikipedia. naive bayes classifier, accessed on: June 2021. *Online Available:* https://en.wikipedia.org/wiki/Naive_Bayes_classifier.

[9] Wikipedia. k-means clustering, accessed on: June 2021. *Online Available:* https://en.wikipedia.org/wiki/K-means_clustering.

[10] Machine learning. accessed on: June 2021. *Online Available:* https://www.ibm.com/cloud/learn/machine-learning.

[11] Mann whitney u test (wilcoxon rank sum test), accessed on: July2 021. *Online Available:* https://sphweb.bumc.bu.edu/otlt/mph-modules/bs/bs704_nonparametric/bs704_nonparametric4.html.

[12] Mann-whitney u test calculator. *Online Available:* https://www.socscistatistics.com/tests/mannwhitney/.

[13] Mobile phone users statistics. *Online Available:* https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/.

[14] Number of mobile apps statistic till 2020. *Online Available:* https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[15] One- and two-tailed tests accessed on: July 2021. *Online Available:* https://en.wikipedia.org/wiki/One-_and_two-tailed_tests, .

[16] Faq: What are the differences between one-tailed and two-tailed tests? accessed on: July 2021. *Online Available:* https://stats.idre.ucla.edu/other/mult-pkg/faq/general/faq-what-are-the-differences-between-one-tailed-and-two-tailed-tests/, .

[17] Comparison of open-source and closed-source software, accessed on: June 2021. *Online Available:* https://en.wikipedia.org/wiki/Comparison_of_open-source_and_closed-source_software.

[18] P-value accessed on: July 2021. *Online Available:* https://statisticsbyjim.com/glossary/p-value/.

[19] Wilcoxon signed-rank test accessed on: July 2021. *Online Available:* https://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test.

[20] Wilcoxon signed rank test calculator. *Online Available:* https://www.socscistatistics.com/tests/signedranks/, .

[21] Significance level, accessed on: July 2021. *Online Available:* https://statisticsbyjim.com/glossary/significance-level/, .

[22] Software maintenance cost accessed on: August 2021. *Online Available:* https://galorath.com/software-maintenance-costs/.

[23] Svn repository. *Online Available:* http://sourceforge.net/.

[24] Rabe Abdalkareem, Emad Shihab, and Juergen Rilling. On code reuse from stackoverflow: An exploratory study on android apps. *Information and Software Technology*, 88:148–158, 2017.

[25] Rabe Abdalkareem, Emad Shihab, and Juergen Rilling. What do developers use the crowd for? a study using stack overflow. *IEEE Software*, 34(2):53–60, 2017.

[26] Md Ahasanuzzaman, Muhammad Asaduzzaman, Chanchal K Roy, and Kevin A Schneider. Caps: a supervised technique for classifying stack overflow posts concerning api issues. *Empirical Software Engineering*, 25(2):1493–1532, 2020.

[27] Mashal Ahmad and Mel O Cinnéide. Impact of stack overflow code snippets on software cohesion: a preliminary study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 250–254. IEEE, 2019.

[28] Syed Nadeem Ahsan, Javed Ferzund, and Franz Wotawa. Automatic software bug triage system (bts) based on latent semantic indexing and support vector machine. In *2009 Fourth International Conference on Software Engineering Advances*, pages 216–221. IEEE, 2009.

[29] Farouq Al-Omari, Iman Keivanloo, Chanchal K Roy, and Juergen Rilling. Detecting clones across microsoft. net programming languages. In *2012 19th Working Conference on Reverse Engineering*, pages 405–414. IEEE, 2012.

[30] Farouq Al-Omari, Chanchal K Roy, and Tonghao Chen. Semanticclonebench: A semantic code clone benchmark using crowd-source knowledge. In *2020 IEEE 14th International Workshop on Software Clones (IWSC)*, pages 57–63. IEEE, 2020.

[31] Homa Alemzadeh, Ravishankar K Iyer, Zbigniew Kalbarczyk, and Jai Raman. Analysis of safety-critical computer failures in medical devices. *IEEE Security & Privacy*, 11(4):14–26, 2013.

[32] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement? a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, pages 304–318, 2008.

[33] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[34] Muhammad Asaduzzaman, Michael C Bullock, Chanchal K Roy, and Kevin A Schneider. Bug introducing changes: A case study with android. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 116–119. IEEE, 2012.

[35] Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. How clones are maintained: An empirical study. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 81–90. IEEE, 2007.

[36] Muhammad Zubair Azeem, Azhar Imran, Faheem Akhtar, Ahsan Wajahat, Jahanzaib Latif, and Suhail Ahmed Memon. Effects of code cloning in mobile applications. In *2020 3rd International Conference on Computing, Mathematics and Engineering Technologies (iCoMET)*, pages 1–6. IEEE, 2020.

[37] Brenda S Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of 2nd Working Conference on Reverse Engineering*, pages 86–95. IEEE, 1995.

[38] Sebastian Baltes and Christoph Treude. Code duplication on stack overflow. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 13–16. IEEE, 2020.

[39] Venkat Bandi, Chanchal K Roy, and Carl Gutwin. Clone swarm: A cloud based code-clone analysis tool. In *2020 IEEE 14th International Workshop on Software Clones (IWSC)*, pages 52–56. IEEE, 2020.

[40] Liliane Barbour, Foutse Khomh, and Ying Zou. Late propagation in software clones. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 273–282. IEEE, 2011.

[41] Liliane Barbour, Foutse Khomh, and Ying Zou. An empirical study of faults in late propagation clone genealogies. *Journal of Software: Evolution and Process*, 25(11):1139–1165, 2013.

[42] Anton Barua, Stephen W Thomas, and Ahmed E Hassan. What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering*, 19(3):619–654, 2014.

[43] Hamid Abdul Basit, Damith C Rajapakse, and Stan Jarzabek. Beyond templates: a study of clones in the stl and some general implications. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 451–459. IEEE, 2005.

[44] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998.

[45] Lutz Büch and Artur Andrzejak. Learning-based recursive aggregation of abstract syntax trees for code clone detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 95–104. IEEE, 2019.

[46] Fabio Calefato, Filippo Lanubile, and Teresa Mallardo. Function clone detection in web applications: a semiautomated approach. *J. Web Eng.*, 3(1):3–21, 2004.

[47] Scott Chacon and Ben Straub. Git commands. In *Pro Git*, pages 407–419. Springer, 2014.

[48] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, 2002.

[49] Yoonsik Cheon, Carlos V Chavez, and Ubaldo Castro. Code reuse between java and android applications. 2019.

[50] Michel Chilowicz, Etienne Duris, and Gilles Roussel. Syntax tree fingerprinting for source code similarity detection. In *2009 IEEE 17th international conference on program comprehension*, pages 243–247. IEEE, 2009.

[51] James R Cordy and Chanchal K Roy. The nicad clone detector. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 219–220. IEEE, 2011.

[52] TM Cover, JA Thomas, and John Wiley. Sons.,". *Elements of information theory," New York: Wiley*, 1991.

[53] Neil Davey, Paul Barson, Simon Field, Ray Frank, and D Tansley. The development of a software clone detector. *International Journal of Applied Software Technology*, 1995.

[54] Ian J Davis and Michael W Godfrey. Clone detection by exploiting assembler. In *Proceedings of the 4th International Workshop on Software Clones*, pages 77–78, 2010.

[55] Ian J Davis and Michael W Godfrey. From whence it came: Detecting source code clones by analyzing assembler. In *2010 17th Working Conference on Reverse Engineering*, pages 242–246. IEEE, 2010.

[56] Josh Dehlinger and Jeremy Dixon. Mobile application software engineering: Challenges and research directions. In *Proceedings of the Workshopon Mobile Software Engineering*, pages 29–32. Springer, 2011.

[57] Giuseppe A Di Lucca, Massimiliano Di Penta, and Anna Rita Fasolino. An approach to identify duplicated web pages. In *Proceedings 26th Annual International Computer Software and Applications*, pages 481–486. IEEE, 2002.

[58] Georgios Digkas, Nikolaos Nikolaidis, Apostolos Ampatzoglou, and Alexander Chatzigeorgiou. Reusing code from stackoverflow: the effect on technical debt. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 87–91. IEEE, 2019.

[59] Christoph Domann, Elmar Juergens, and Jonathan Streit. The curse of copy&paste—cloning in requirements specifications. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 443–446. IEEE, 2009.

[60] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 109–118. IEEE, 1999.

[61] Raimar Falke, Pierre Frenzel, and Rainer Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Software Engineering*, 13(6):601–643, 2008.

[62] Chenhui Feng, Tao Wang, Jinze Liu, Yang Zhang, Kele Xu, and Yijie Wang. Nicad+: Speeding the detecting process of nicad. In *2020 IEEE International Conference on Service Oriented Systems Engineering (SOSE)*, pages 103–110. IEEE, 2020.

[63] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[64] Joseph L Fleiss, Bruce Levin, Myunghee Cho Paik, et al. The measurement of interrater agreement. *Statistical methods for rates and proportions*, 2(212-236):22–23, 1981.

[65] Harleen K Flora, Xiaofeng Wang, and Swati V Chande. An investigation on the characteristics of mobile applications: A survey study. *IJ Information Technology and Computer Science*, 11:21–27, 2014.

[66] Harleen K Flora, Xiaofeng Wang, and V Swati. Chande,"an investigation of characteristics of mobile applications: A survey study". *International Journal of Information Technology and Computer Science (IJITCS). Communicated*, 2014.

[67] Yoshihiko Fukushima, Raula Kula, Shinji Kawaguchi, Kyohei Fushida, Masataka Nagura, and Hajimu Iida. Code clone graph metrics for detecting diffused code clones. In *2009 16th Asia-Pacific Software Engineering Conference*, pages 373–380. IEEE, 2009.

[68] Michael Gegick, Pete Rotella, and Tao Xie. Identifying security bug reports via text mining: An industrial case study. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 11–20. IEEE, 2010.

[69] Nils Göde and Jan Harder. Clone stability. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 65–74. IEEE, 2011.

[70] Nils Göde and Rainer Koschke. Frequency and risks of changes to clones. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 311–320, 2011.

[71] Aaron J Gordon. Concepts for mobile programming. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pages 58–63, 2013.

[72] Katerina Goseva-Popstojanova and Jacob Tyo. Identification of security related bug reports via text mining using supervised and unsupervised classification. In *2018 IEEE International conference on software quality, reliability and security (QRS)*, pages 344–355. IEEE, 2018.

[73] Isabelle Guyon et al. A scaling law for the validation-set training-set size ratio. *AT&T Bell Laboratories*, 1(11), 1997.

[74] N. Göde and J. Harder. Clone stability. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 65–74, 2011. URL `https://ieeexplore.ieee.org/document/5741247`.

[75] Muhammad Hammad, Hamid Abdul Basit, Stan Jarzabek, and Rainer Koschke. A systematic mapping study of clone visualization. *Computer Science Review*, 37:100266, 2020.

[76] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, 28(1):100–108, 1979.

[77] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Bug prediction based on fine-grained module histories. In *2012 34th international conference on software engineering (ICSE)*, pages 200–210. IEEE, 2012.

[78] Israel Herraiz, Daniel M German, Jesus M Gonzalez-Barahona, and Gregorio Robles. Towards a simplification of the bug report form in eclipse. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 145–148, 2008.

[79] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *2013 35th international conference on software engineering (ICSE)*, pages 392–401. IEEE, 2013.

[80] Yoshiki Higo and Shinji Kusumoto. Code clone detection on specialized pdgs with heuristics. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 75–84. IEEE, 2011.

[81] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Refactoring support based on code clone analysis. In *International Conference on Product Focused Software Process Improvement*, pages 220–233. Springer, 2004.

[82] Yoshiki Higo, Ueda Yasushi, Minoru Nishino, and Shinji Kusumoto. Incremental code clone detection: A pdg-based approach. In *2011 18th Working Conference on Reverse Engineering*, pages 3–12. IEEE, 2011.

[83] Keisuke Hotta, Yukiko Sano, Yoshiki Higo, and Shinji Kusumoto. Is duplicate code more frequently modified than non-duplicate code in software evolution? an empirical study on open source software. In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, pages 73–82, 2010.

[84] Wei Hua, Yulei Sui, Yao Wan, Guangzhong Liu, and Guandong Xu. Fcca: Hybrid code representation for functional clone detection using attention networks. *IEEE Transactions on Reliability*, 70(1):304–318, 2020.

[85] Katsuro Inoue, Yusuke Sasaki, Pei Xia, and Yuki 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)*, pages 331–341. IEEE, 2012.

[86] Judith F Islam, Manishankar Mondal, and Chanchal K Roy. Bug replication in code clones: An empirical study. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 68–78. IEEE, 2016.

[87] Judith F Islam, Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. A comparative study of software bugs in clone and non-clone code. In *SEKE*, pages 436–443, 2017.

[88] Judith F Islam, Manishankar Mondal, and Chanchal 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)*, pages 73–83. IEEE, 2019.

[89] Judith F Islam, Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. Comparing bug replication in regular and micro code clones. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 81–92. IEEE, 2019.

[90] Mohammed J Islam, QM Jonathan Wu, Majid Ahmadi, and Maher A Sid-Ahmed. Investigating the performance of naive-bayes classifiers and k-nearest neighbor classifiers. In *2007 International Conference on Convergence Information Technology (ICCIT 2007)*, pages 1541–1546. IEEE, 2007.

[91] Shruti Jadon. Code clones detection using machine learning technique: Support vector machine. In *2016 International Conference on Computing, Communication and Automation (ICCCA)*, pages 399–303. IEEE, 2016.

[92] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105. IEEE, 2007.

[93] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-based detection of clone-related bugs. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 55–64, 2007.

[94] J Howard Johnson. Substring matching for clone detection and change tracking. In *ICSM*, volume 94, pages 120–126, 1994.

[95] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *2009 IEEE 31st International Conference on Software Engineering*, pages 485–495. IEEE, 2009.

[96] Toshihiro Kamiya. Ccfinderx: An interactive code clone analysis environment. In *Code Clone Analysis*, pages 31–44. Springer, 2021.

[97] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7): 654–670, 2002.

[98] Cory J Kapser and Michael W Godfrey. Supporting the analysis of clones in software systems. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):61–82, 2006.

[99] Cory J Kapser and Michael W Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

[100] Amandeep Kaur, Sandeep Sharma, and Munish Saini. Code clone detection using machine learning techniques: A systematic literature review. *International Journal of Open Source Software and Processes (IJOSSP)*, 11(2):49–75, 2020.

[101] Iman Keivanloo, Chanchai K Roy, and Juergen Rilling. Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning. In *2012 6th International Workshop on Software Clones (IWSC)*, pages 36–42. IEEE, 2012.

[102] Iman Keivanloo, Chanchal K Roy, and Juergen Rilling. Sebyte: A semantic clone detection tool for intermediate languages. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 247–249. IEEE, 2012.

[103] Viktor Kolokolov, Paul Baumann, Silvia Santini, Stefan T Ruehl, and Stephan AW Verclas. Flexible development of variable software features for mobile business applications. In *Proceedings of the 17th International Software Product Line Conference co-located workshops*, pages 67–73, 2013.

[104] Rainer Koschke. Large-scale inter-system clone detection using suffix trees. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 309–318. IEEE, 2012.

[105] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *2006 13th Working Conference on Reverse Engineering*, pages 253–262. IEEE, 2006.

[106] Mark Kramer and Philip H Newcomb. Legacy system modernization of the engineering operational sequencing system (eoss). In *Information Systems Transformation*, pages 249–281. Elsevier, 2010.

[107] Jens Krinke. A study of consistent and inconsistent changes to code clones. In *14th working conference on reverse engineering (WCRE 2007)*, pages 170–178. IEEE, 2007.

[108] Jens Krinke. Is cloned code more stable than non-cloned code? In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 57–66. IEEE, 2008.

[109] Jens Krinke. Is cloned code older than non-cloned code? In *Proceedings of the 5th International Workshop on Software Clones*, pages 28–33, 2011.

[110] Bruno Laguë, Daniel Proulx, Jean Mayrand, Ettore M Merlo, and John Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *1997 Proceedings International Conference on Software Maintenance*, pages 314–321. IEEE, 1997.

[111] Ahmed Lamkanfi, Serge Demeyer, Emanuel Giger, and Bart Goethals. Predicting the severity of a reported bug. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 1–10. IEEE, 2010.

[112] Hyo-Sub Lee and Kyung-Goo Doh. Tree-pattern-based duplicate code detection. In *Proceedings of the ACM first international workshop on Data-intensive software management and mining*, pages 7–12, 2009.

[113] Seunghak Lee and Iryoung Jeong. Sdd: high performance code clone detection system for large scale source code. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 140–141, 2005.

[114] Guanhua Li, Yijian Wu, Chanchal K Roy, Jun Sun, Xin Peng, Nanjie Zhan, Bin Hu, and Jingyi Ma. Saga: efficient and large-scale detection of near-miss clones with gpu acceleration. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 272–283. IEEE, 2020.

[115] Jingyue Li and Michael D Ernst. Cbcd: Cloned buggy code detector. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 310–320. IEEE, 2012.

[116] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *OSdi*, volume 4, pages 289–302, 2004.

[117] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[118] Wayne C Lim. Effects of reuse on quality, productivity, and economics. *IEEE software*, 11(5):23–30, 1994.

[119] Hui Liu, Zhiyi Ma, Lu Zhang, and Weizhong Shao. Detecting duplications in sequence diagrams based on suffix trees. In *2006 13th Asia Pacific Software Engineering Conference (APSEC'06)*, pages 269–276. IEEE, 2006.

[120] Adriaan Lotter, Sherlock A Licorish, Bastin Tony Roy Savarimuthu, and Sarah Meldrum. Code reuse in stack overflow and popular open source java projects. In *2018 25th Australasian Software Engineering Conference (ASWEC)*, pages 141–150. IEEE, 2018.

[121] Angela Lozano and Michel Wermelinger. Assessing the effect of clones on changeability. In *2008 IEEE International Conference on Software Maintenance*, pages 227–236. IEEE, 2008.

[122] Angela Lozano and Michel Wermelinger. Tracking clones' imprint. In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, page 65–72, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605589800. doi: 10.1145/1808901.1808910. URL https://doi.org/10.1145/1808901.1808910.

[123] Angela Lozano and Michel Wermelinger. Tracking clones' imprint. In *Proceedings of the 4th International Workshop on Software Clones*, pages 65–72, 2010.

[124] Laura MacLeod. Reputation on stack exchange: Tag, you're it! In *2014 28th international conference on advanced information networking and applications workshops*, pages 670–674. IEEE, 2014.

[125] Udi Manber et al. Finding similar files in a large file system. In *Usenix winter*, volume 94, pages 1–10, 1994.

[126] Saraj Singh Manes and Olga Baysal. Studying the change histories of stack overflow and github snippets. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 283–294. IEEE, 2021.

[127] Andrian Marcus and Jonathan I Maletic. Identification of high-level concept clones in source code. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 107–114. IEEE, 2001.

[128] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *icsm*, volume 96, page 244, 1996.

[129] Sarah Meldrum, Sherlock A Licorish, Caitlin A Owen, and Bastin Tony Roy Savarimuthu. Understanding stack overflow code quality: A recommendation of caution. *Science of Computer Programming*, 199: 102516, 2020.

[130] Brenda M Michelson. Event-driven architecture overview. *Patricia Seybold Group*, 2(12):10–1571, 2006.

[131] Roberto Minelli and Michele Lanza. Software analytics for mobile applications–insights & lessons learned. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 144–153. IEEE, 2013.

[132] Audris Mockus and Lawrence G Votta. Identifying reasons for software changes using historic databases. In *icsm*, pages 120–130, 2000.

[133] Israel J Mojica, Bram Adams, Meiyappan Nagappan, Steffen Dienst, Thorsten Berger, and Ahmed E Hassan. A large-scale empirical study on software reuse in mobile apps. *IEEE software*, 31(2):78–86, 2013.

[134] Manishankar Mondal. *On the stability of software clones: A genealogy-based empirical study.* PhD thesis, University of Saskatchewan, 2013.

[135] Manishankar Mondal, Md Saidur Rahman, Ripon K Saha, Chanchal K Roy, Jens Krinke, and Kevin A Schneider. An empirical study of the impacts of clones in software maintenance. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 242–245. IEEE, 2011.

[136] Manishankar Mondal, Chanchal K Roy, Md Saidur Rahman, Ripon K Saha, Jens Krinke, and Kevin A Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1227–1234, 2012.

[137] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. An empirical study on clone stability. *ACM SIGAPP Applied Computing Review*, 12(3):20–36, 2012.

[138] Manishankar Mondal, Chanchal K Roy, and Kevin 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)*, pages 91–100. IEEE, 2015.

[139] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. Spcp-miner: A tool for mining code clones that are important for refactoring or tracking. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 484–488. IEEE, 2015.

[140] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. A comparative study on the intensity and harmfulness of late propagation in near-miss code clones. *Software Quality Journal*, 24(4):883–915, 2016.

[141] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. Bug propagation through code cloning: An empirical study. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 227–237. IEEE, 2017.

[142] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. Identifying code clones having high possibilities of containing bugs. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 99–109. IEEE, 2017.

[143] Manishankar Mondal, Banani Roy, Chanchal K Roy, and Kevin A Schneider. An empirical study on bug propagation through code cloning. *Journal of Systems and Software*, 158:110407, 2019.

[144] Manishankar Mondal, Banani Roy, Chanchal K Roy, and Kevin A Schneider. Investigating context adaptation bugs in code clones. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 157–168. IEEE, 2019.

[145] Saikat Mondal, Mohammad Masudur Rahman, and Chanchal K Roy. Can issues reported at stack overflow questions be reproduced? an exploratory study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 479–489. IEEE, 2019.

[146] Golam Mostaeen, Jeffrey Svajlenko, Banani Roy, Chanchal K Roy, and Kevin A Schneider. On the use of machine learning techniques towards the design of cloud based automatic code clone validation tools. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 155–164. IEEE, 2018.

[147] G Murphy and Davor Cubranic. Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, pages 1–6. Citeseer, 2004.

[148] Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K Roy, and Kevin A Schneider. Clcdsa: cross language code clone detection using syntactical features and api documentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1026–1037. IEEE, 2019.

[149] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What makes a good code example?: A study of programming q&a in stackoverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 25–34. IEEE, 2012.

[150] Ahmed Fawzi Otoom, Sara Al-jdaeh, and Maen Hammad. Automated classification of software bug reports. In *Proceedings of the 9th International Conference on Information Communication and Management*, pages 17–21, 2019.

[151] Mahesh Pal. Random forest classifier for remote sensing classification. *International journal of remote sensing*, 26(1):217–222, 2005.

[152] J-F Patenaude, Ettore Merlo, Michel Dagenais, and Bruno Laguë. Extending software quality assessment techniques to java systems. In *Proceedings Seventh International Workshop on Program Comprehension*, pages 49–56. IEEE, 1999.

[153] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Turning the ide into a self-confident programming assistant. 2015.

[154] David Powers. Ailab. evaluation: From precision, recall and f-measure to roc, informedness, markedness & correlation. *J. Mach. Learn. Technol*, 2(22293981):01, 2011.

[155] Chaiyong Ragkhitwetsagul, Jens Krinke, and Rocco Oliveto. Awareness and experience of developers to outdated and license-violating code on stack overflow: An online survey. *arXiv preprint arXiv:1806.08149*, 2018.

[156] Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixao, Giuseppe Bianco, and Rocco Oliveto. Toxic code snippets on stack overflow. *IEEE Transactions on Software Engineering*, 2019.

[157] Md Saidur Rahman and Chanchal K Roy. On the relationships between stability and bug-proneness of code clones: An empirical study. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 131–140. IEEE, 2017.

[158] Wahidur Rahman, Yisen Xu, Fan Pu, Jifeng Xuan, Xiangyang Jia, Michail Basios, Leslie Kanthan, Lingbo Li, Fan Wu, and Baowen Xu. Clone detection on large scala codebases. In *2020 IEEE 14th International Workshop on Software Clones (IWSC)*, pages 38–44. IEEE, 2020.

[159] Irina Rish et al. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46, 2001.

[160] Martin P Robillard and Robert DeLine. A field study of api learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[161] Christoffer Rosen and Emad Shihab. What are mobile developers asking about? a large scale study using stack overflow. *Empirical Software Engineering*, 21(3):1192–1223, 2016.

[162] Chanchal K Roy. Detection and analysis of near-miss software clones. In *2009 IEEE International Conference on Software Maintenance*, pages 447–450. IEEE, 2009.

[163] Chanchal K Roy and James R Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th iEEE international conference on program comprehension*, pages 172–181. IEEE, 2008.

[164] Chanchal K Roy and James R Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 157–166. IEEE, 2009.

[165] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7): 470–495, 2009.

[166] Chanchal K Roy, Minhaz F Zibran, and Rainer Koschke. The vision of software clone management: Past, present, and future (keynote paper). In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 18–33. IEEE, 2014.

[167] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[168] Israel J Mojica Ruiz, Meiyappan Nagappan, Bram Adams, and Ahmed E Hassan. Understanding reuse in the android market. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 113–122. IEEE, 2012.

[169] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. How developers search for code: a case study. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 191–201, 2015.

[170] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 117–128, 2009.

[171] S Rasoul Safavian and David Landgrebe. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics*, 21(3):660–674, 1991.

[172] Ripon K Saha, Chanchal K Roy, Kevin A Schneider, and Dewayne E Perry. Understanding the evolution of type-3 clones: an exploratory study. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 139–148. IEEE, 2013.

[173] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 354–365, 2018.

[174] Hitesh Sajnani, Vaibhav Saini, and Cristina V Lopes. A comparative study of bug patterns in java cloned and non-cloned code. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 21–30. IEEE, 2014.

[175] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1157–1168, 2016.

[176] Antonella Santone. Clone detection through process algebras and java bytecode. In *Proceedings of the 5th International Workshop on Software Clones*, pages 73–74, 2011.

[177] Doaa M Shawky and Ahmed F Ali. An approach for assessing similarity metrics used in metric-based clone detection techniques. In *2010 3rd International Conference on Computer Science and Information Technology*, volume 1, pages 580–584. IEEE, 2010.

[178] Sergei Shcherban, Peng Liang, Amjed Tahir, and Xueying Li. Automatic identification of code smell discussions on stack overflow: A preliminary investigation. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6, 2020.

[179] Abdullah Sheneamer and Jugal Kalita. Semantic clone detection using machine learning. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1024–1028. IEEE, 2016.

[180] Abdullah Sheneamer, Hanan Hazazi, Swarup Roy, and Jugal Kalita. Schemes for labeling semantic code clones using machine learning. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 981–985. IEEE, 2017.

[181] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. High-impact defects: a study of breakage and surprise defects. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 300–310, 2011.

[182] Rodrigo FG Silva, Klérisson Paixão, and Marcelo de Almeida Maia. Duplicate question detection in stack overflow: A reproducibility study. In *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*, pages 572–581. IEEE, 2018.

[183] Daniela Steidl and Nils Göde. Feature-based detection of bugs in clones. In *2013 7th International Workshop on Software Clones (IWSC)*, pages 76–82. IEEE, 2013.

[184] Johan AK Suykens and Joos Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.

[185] Jeffrey Svajlenko and Chanchai K Roy. Fast, scalable and user-guided clone detection. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 352–353, 2018.

[186] Jeffrey Svajlenko and Chanchal K Roy. Evaluating modern clone detection tools. In *2014 IEEE international conference on software maintenance and evolution*, pages 321–330. IEEE, 2014.

[187] Jeffrey Svajlenko and Chanchal K Roy. Fast and flexible large-scale clone detection with cloneworks. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 27–30. IEEE, 2017.

[188] Jeffrey Svajlenko, Chanchal K Roy, and James R Cordy. A mutation analysis based benchmarking framework for clone detectors. In *2013 7th International Workshop on Software Clones (IWSC)*, pages 8–9. IEEE, 2013.

[189] Mark D Syer, Bram Adams, Ying Zou, and Ahmed E Hassan. Exploring the development of micro-apps: A case study on the blackberry and android platforms. In *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, pages 55–64. IEEE, 2011.

[190] Mark D Syer, Meiyappan Nagappan, Ahmed E Hassan, and Bram Adams. Revisiting prior empirical findings for mobile apps: An empirical case study on the 15 most popular open-source android apps. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, pages 283–297, 2013.

[191] Robert Tairas and Jeff Gray. Phoenix-based clone detection using suffix trees. In *Proceedings of the 44th annual Southeast regional conference*, pages 679–684, 2006.

[192] Ahmed Tamrawi, Tung Thanh Nguyen, Jafar Al-Kofahi, and Tien N Nguyen. Fuzzy set-based automatic bug triaging (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, pages 884–887, 2011.

[193] M Thomas and H Thimbleby. Computer bugs in hospitals: An unnoticed killer. 2018.

[194] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 271–280. IEEE, 2012.

[195] Gias Uddin and Martin P Robillard. How api documentation fails. *Ieee software*, 32(4):68–75, 2015.

[196] Md Sharif Uddin, Chanchal K Roy, Kevin A Schneider, and Abram Hindle. On the effectiveness of simhash for detecting near-miss clones in large scale software systems. In *2011 18th Working Conference on Reverse Engineering*, pages 13–22. IEEE, 2011.

[197] Md Sharif Uddin, Chanchal K Roy, and Kevin A Schneider. Simcad: An extensible and faster clone detection tool for large scale software systems. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 236–238. IEEE, 2013.

[198] Dolores R Wallace and D Richard Kuhn. Failure modes in medical device software: an analysis of 15 years of recall data. *International Journal of Reliability, Quality and Safety Engineering*, 8(04): 351–371, 2001.

[199] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K Roy. Ccaligner: a token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1066–1077, 2018.

[200] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. Searching for better configurations: a rigorous approach to clone evaluation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 455–465, 2013.

[201] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271. IEEE, 2020.

[202] Anthony I Wasserman. Software engineering issues for mobile application development. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 397–400, 2010.

[203] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1329–1341, 2014.

[204] Richard Wettel and Radu Marinescu. Archeology of code duplication: Recovering duplication chains from small duplication fragments. In *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*, pages 8–pp. IEEE, 2005.

[205] Jason L Wright, Jason W Larsen, and Miles McQueen. Estimating software vulnerabilities: A case study based on the misclassification of bugs in mysql server. In *2013 International Conference on Availability, Reliability and Security*, pages 72–81. IEEE, 2013.

[206] Ming Wu, Pengcheng Wang, Kangqi Yin, Haoyu Cheng, Yun Xu, and Chanchal K Roy. Lvmapper: A large-variance clone detector using sequencing alignment approach. *IEEE access*, 8:27986–27997, 2020.

[207] Tianyong Wu, Xi Deng, Jun Yan, and Jian Zhang. Analyses for specific defects in android applications: a survey. *Frontiers of Computer Science*, pages 1–18, 2019.

[208] Wei Xu, Fangfang Zhang, and Sencun Zhu. Permlyzer: Analyzing permission usage in android applications. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 400–410. IEEE, 2013.

[209] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 89–99. IEEE, 2015.

[210] Jie Zeng, Kerong Ben, Xiaowei Li, and Xian Zhang. Fast code clone detection based on weighted recursive autoencoders. *IEEE Access*, 7:125062–125078, 2019.

[211] Lingxi Zhang, Zhiyong Feng, Wei Ren, and Hong Luo. Siamese-based bilstm network for scratch source code similarity measuring. In *2020 International Wireless Communications and Mobile Computing (IWCMC)*, pages 1800–1805. IEEE, 2020.

[212] Shichao Zhang, Xuelong Li, Ming Zong, Xiaofeng Zhu, and Debo Cheng. Learning k for knn classification. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(3):1–19, 2017.

[213] Shichao Zhang, Xuelong Li, Ming Zong, Xiaofeng Zhu, and Ruili Wang. Efficient knn classification with different numbers of nearest neighbors. *IEEE transactions on neural networks and learning systems*, 29(5):1774–1785, 2017.

[214] Bo Zhou, Iulian Neamtiu, and Rajiv Gupta. A cross-platform analysis of bugs and bug-fixing in open source projects: Desktop vs. android vs. ios. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–10, 2015.

[215] Minhaz F Zibran and Chanchal K Roy. Ide-based real-time focused search for near-miss clones. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1235–1242, 2012.

[216] Yue Zou, Bihuan Ban, Yinxing Xue, and Yun Xu. Ccgraph: a pdg-based code clone detector with approximate graph matching. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 931–942. IEEE, 2020.