Machine Learning Driven Guidance for Software Maintenance: Enhancing Code Management and Features

by

Maram Assi

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctoral of Philosophy

Queen's University
Kingston, Ontario, Canada
September 2024

Abstract

Software systems undergo continuous maintenance to fix defects, enhance features, and improve code quality to ensure that systems stay reliable and relevant in a competitive market. Due to the escalating growth of the code base and the increasing complexity of software systems, it is essential to adopt automated approaches that support and drive software maintenance process. Despite existing research efforts using machine learning (ML) techniques to improve software maintenance, developers continue to encounter challenges in managing code changes and clone evolution in code maintenance and enhancing features.

To assist developers in maintaining software effectively, in this thesis, we introduce a set of approaches that leverage software artifacts and ML to assist developers with two key areas of software maintenance: managing code and enhancing features. Specifically, we conduct four studies: (1) improving the accuracy of predicting change impact for defect resolution by utilizing common characteristics of issue reports; (2) analyzing the dynamics and impact of code clones in deep learning frameworks to ensure long-term code quality and maintainability; (3) performing automatic, comprehensive competitor feature analysis, and (4) generating suggestions for feature improvements based on competitor user review analysis. The aim of this thesis is to empower developers with novel methodologies for streamlined and effective software maintenance, thereby fostering a high-quality maintained codebase and competitive, sustainable software systems. The approaches proposed in the thesis can be adapted to software projects where mobile user reviews, source code, and issue reports are available to demonstrate the applicability of the research contributions.

Acknowledgments

To my family, I owe an immeasurable debt of gratitude. First and foremost, my deepest gratitude extends to my brother, Amer—a father, brother, and role model. Thank you for supporting my education from school till now. To my mother, who believed in the power of education and sacrificed all her youth to provide my siblings and me with the best opportunities—your love and dedication have been my foundation. Dana, my sister, my best friend, and supporter, thank you for always being there for me.

I would like to extend my thanks to my academic mentors. To my supervisor, Dr. Ying Zou, thank you for teaching me perseverance, to always strive for the best, and never settle for anything average. To my collaborators, especially Dr. Safwat Hassan, your practical approach to research has been eye-opening. Thank you for your flexibility, availability, and unwavering support. I look up to you immensely. Drs. Hassanein and Zulkernine, you made my time at Queen's University unique, and it was my pleasure working alongside leaders like you.

My journey to pursuing a PhD was inspired by exceptional mentors from my alma mater. I am deeply grateful to my Master's supervisor, Dr. Ramzi Haraty, and the exemplary professor, Dr. Samer Habre, whose guidance and encouragement have been instrumental in shaping my academic path.

To my friends, your support and belief in me have been beyond helpful. Thank you to my Lebanese friends in Kingston—you made my PhD journey enjoyable. Lama, George and Karim, your support has been a lifeline. My roommates, Marah and Alaa, sharing this journey with you has been a pleasure. Thank you for being considerate and understanding and for grounding me during hard times.

To my dear best friend, Mona, who is more than a sister to me—thank you for always

being there, especially at the beginning of my PhD journey. I admire your ambition, and you always inspire me to reach new heights.

I would like to dedicate this thesis to my home country, Lebanon, with the hope that one day, it will be known for its science and culture.

Finally, I am deeply thankful to the Government of Canada for providing me with the Vanier Scholarship. I will be forever honoured by this support.

Statement of Originality

With this statement I, Maram Assi, hereby declare that the described research in this thesis is my own work. Any publications, ideas, and inventions from others have been properly referenced.

The publications related to this thesis are listed as follows:

- **FeatCompare: Feature comparison for competing mobile apps leveraging user reviews (Chapter 5).** Maram Assi, Safwat Hassan, Yuan Tian, and Ying Zou. 2021. Empirical Softw. Engg. 26, 5 (Sep 2021).

- **Predicting the Change Impact of Resolving Defects by Leveraging the Topics of Issue Reports in Open Source Software Systems (Chapter 3).** Maram Assi, Safwat Hassan, Stefanos Georgio, and Ying Zou. 2023. P ACM Trans. Softw. Eng. Methodol. 32, 6, Article 141 (November 2023), 34 pages.

I am the primary author of all the above publications. The research papers are coauthored with Dr. Ying Zou, Dr. Yuan Tian, Dr. Safwat Hassan, Dr. Stefanos Georgio. Dr. Ying Zou supervised all of the related research. The co-authors participated in meetings and provided me with feedback and suggestions to improve my work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software systems have become an integral part of our daily lives, underpinning aspects from communication to business operations and personal productivity. The success and longevity of these systems hinge on effective and continuous maintenance. According to the IEEE [2], software maintenance is 'the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment'. This ongoing maintenance process ensures that systems remain reliable and relevant in a competitive market, which is essential for user satisfaction and business success.

In this thesis, we focus on two main areas of software maintenance: code maintenance and feature enhancement for mobile applications. Code maintenance, including defect fixing and managing code clones, leads to enhanced software quality and facilitates future development. Feature enhancement for mobile applications is essential for meeting evolving user needs and maintaining a competitive edge in the market.

Defect fixing is an inevitable aspect of code maintenance, and it involves identifying, analyzing, and resolving defects in the software. Bug fixing is often a time-consuming and resource-intensive task, consuming a considerable portion of the project budget. For instance, it is estimated that 113 billion dollars are budgeted yearly in the United States to identify and fix software defects [1]. Consequently, developers need to efficiently identify, prioritize, and resolve bugs while minimizing disruptions to existing functionalities. Therefore, maintaining code quality over time is crucial for long-term stability and reliability.

Another significant challenge in software maintenance is the management of code clone evolution. Code clones are identical or similar code fragments found in multiple locations within the codebase. These clones can arise from copy-paste programming practices or the need for similar functionality in different parts of the system. While code clones can offer some initial development speed, they pose significant maintenance challenges [152, 167]. For instance, if a bug is fixed in one instance of a clone, it needs to be applied to all other clones to ensure complete resolution. Similarly, modifying functionality requires changes across all cloned segments.

Software maintenance also involves enhancing existing features and introducing new functionalities to adapt to evolving user needs and preferences [47]. Given the competitive nature of the software market, developers must be acutely aware of user needs, satisfy user requirements, and contend with similar software products (i.e., competing software). Staying ahead of the competition is essential [146, 274]. For instance, user complaints can lead to user churn, where users switch to competitors' products [74]. To meet user expectations and outperform competitors, regular updates and enhancements to the features are necessary.

To achieve effective software maintenance, developers leverage historical data from various software artifacts. These artifacts, such as user reviews, issue reports, and source code, provide a rich record of past experiences. By analyzing these artifacts, developers can identify areas for improvement and anticipate potential issues. However, the complexity and volume of these artifacts necessitate advanced analytical methods such as Machine Learning (ML) to automate the analysis of software artifacts and provide valuable insights and predictions that support developers in making informed decisions quickly and accurately. The following subsections provide background on the role of software artifacts and the application of ML in enhancing maintenance efforts.

1.1 Background

1.1.1 Mining Issue Reports

Issue reports are the direct line of communication between users and developers. Issue reports are employed to submit change requests, report software bugs, or inquire about the functionality of the software product [33]. These reports typically include information such as the title, description of the issue, steps to reproduce it, its priority level, and assignee. Figure 1.1 shows a sample issue report from the Apache software repository [1]. Issue reports serve as a critical source of information that empowers developers to conduct effective maintenance. For instance, issue reports help developers understand reported issues, facilitating the identification of software defects [145], prioritizing maintenance tasks [6] and allocating resources effectively [125]. Common issue tracking systems such as Bugzilla[2] and Jira[3] play a vital role in managing these reports by providing platforms for organizing, tracking, and resolving bugs or feature requests, thereby streamlining the software maintenance process.

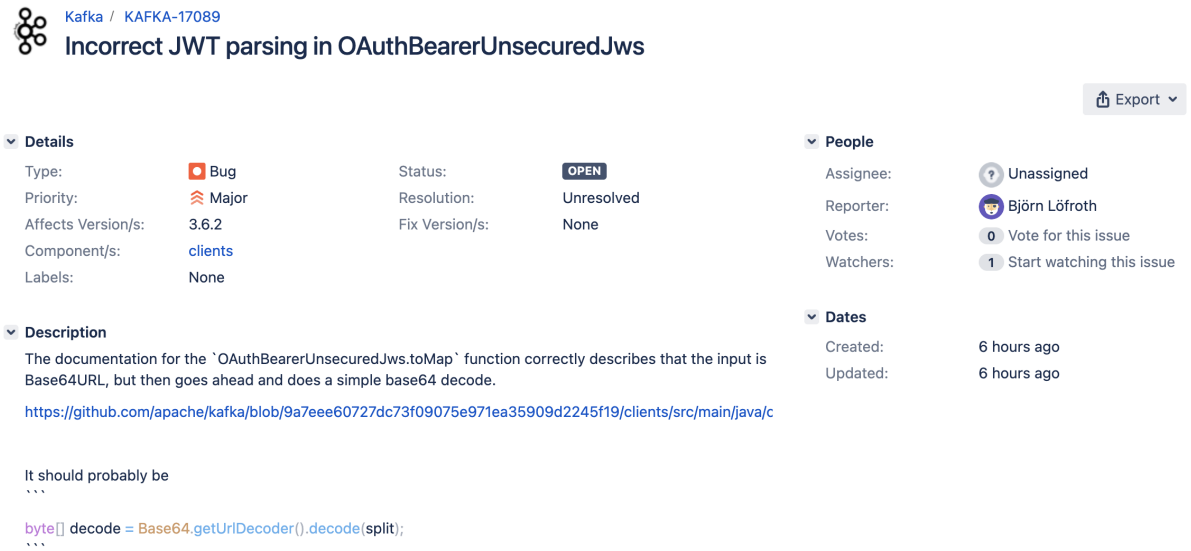

Figure 1.1: Sample issue report from the Apache software repository.

[1] `https://issues.apache.org/jira/browse/KAFKA-17089`
[2] https://www.bugzilla.org/
[3] https://jira.atlassian.com/

1.1.2 Code Clone Management

Source code analysis involves examining the codebase of a software system to assess its quality, identify potential issues, and ensure its long-term maintainability. This process is crucial for maintaining code quality by detecting issues such as coding standards violations, security vulnerability and code clones (i.e., identical or highly similar source code fragments) [229]. There exist four types of code clones: Type 1, Type 2, Type 3, and Type 4. Type 1 clones are exact copies of code fragments with no modifications, except for variations in whitespace and comments. Type 2 clones are syntactically identical fragments, with differences limited to variations in identifiers, literals, types, layout, and comments. Type 3 clones are copied fragments with more substantial modifications, such as added, modified, or deleted statements. Finally, Type 4 clones represent code fragments that are functionally similar but implemented differently, meaning they achieve the same functionality through distinct coding structures or algorithms. Code clone detection tools, such as Nicad [230] and SourcererCC [234], are instrumental in this process, helping developers locate these redundant fragments. Additionally, extracting code metrics of code clones, such as complexity, to gain deeper insights into the maintainability of source code and pinpoint areas with potential maintainability problems.

Addressing the issues associated with code clones early helps prevent future problems and ensures the overall quality and stability of the software system. For instance, developers can detect code clones and take appropriate actions, such as refactoring or consolidating duplicate code segments, thereby improving code quality in the long run [127].

1.1.3 User Reviews Analysis

User reviews typically consist of textual feedback and a star rating, representing the user's sentiment or satisfaction level. The textual feedback contains rich information, such as opinions, experiences and suggestions regarding the user experience, providing valuable insights for developers. Additionally, the star rating system—often ranging from a single star (very dissatisfied) to five stars (highly satisfied)—indicates overall user satisfaction. By analyzing both the textual content and the star rating sentiment, developers can gain a

deeper understanding of user needs and identify areas for improvement and opportunities for feature enhancements in a software system [10, 213, 236, 255, 288]. Figure 1.2 shows an example of a user review from the Skype[4] mobile application.

Analyzing user reviews can be used to conduct competitive analysis. By comparing reviews of similar or competing software products, developers can understand what features users find valuable which can inform strategic decisions, driving the development of new features or improvement of existing ones.

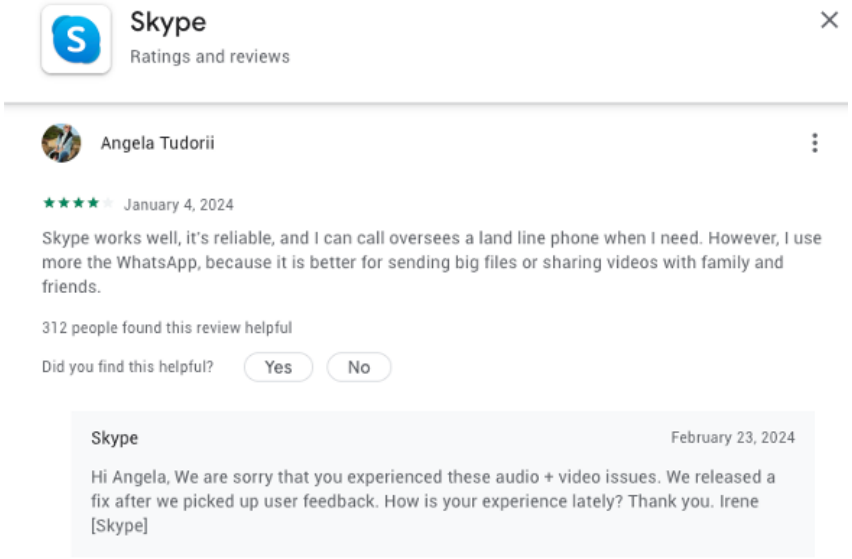

Figure 1.2: Sample user review from the Skype mobile app.

1.1.4 Machine Learning for Software Maintenance

While software artifacts are a valuable data source for developers in facilitating the maintenance of their software products, manually analyzing them is time-consuming and labour-intensive [93]. Therefore, automating software maintenance tasks by leveraging software artifacts has drawn research attention. Researchers have developed various techniques and tools aimed at enhancing the efficiency and productivity of developers [114, 133, 246, 280, 309]. These automated techniques replace manual effort with mined or learned information, streamlining the maintenance process and allowing developers to focus on more critical tasks [302].

[4]https://play.google.com/store/apps/details?id=com.skype.raider&hl=en_CA

Natural Language Processing for Software Maintenance. Natural Language Processing (NLP) techniques are a powerful branch of ML specifically designed to analyze textual information [132]. In the context of software maintenance, NLP techniques are increasingly being used to automatically analyze textual information extracted from software artifacts such as issue reports and user reviews. By automating the analysis of textual information in software artifacts, NLP empowers developers to gain insights from user feedback and reported problems, leading to more effective maintenance. For instance, NLP can automatically analyze user reviews to extract sentiment (positive, negative, neutral [92, 94]), summarize reviews [80, 276] or extract features [131, 245]. Additionally, NLP can analyze the textual descriptions in issue reports to classify bug types [6, 44] and facilitate defect assignment [296]. NLP approaches encompass statistical methods such as Latent Dirichlet Allocation (LDA) [34], ML techniques (e.g., Support Vector Machines (SVMs) [258]), Deep Learning (DL) techniques (e.g., Recurrent Neural Networks (RNNs) [237]), and generative AI techniques (e.g., Large Language Models (LLMs) [323]).

Applying ML for Code Evolution. Software code constantly evolves as a system undergoes maintenance and feature additions. Code evolution analysis involves analyzing the historical changes in the codebase to identify trends and potential issues. ML techniques can be applied to predict code evolution-related tasks such as the likelihood of bugs or the effort required to implement a particular change [248]. For instance, predictive models can leverage historical data to predict bug occurrence [322], or estimate the effort associated with an issue [308]. Additionally, ML can be used to analyze trends and patterns in code evolution [310], and identify areas with frequent changes or potential code smells. This allows developers to focus maintenance efforts on sections with high churn or potential maintainability issues.

1.2 Research Problems

Despite existing research efforts that apply ML techniques to maintain software systems effectively, we envision that developers continue to encounter challenges in two key areas of the software maintenance process:

Area 1: Managing code changes and clone evolution in code maintenance

- **Challenge 1: Lack of approaches predicting change impact with high accuracy for defect resolution.** There are limited human resources, *i.e.*, developers, available to work on a software project [135]. Therefore, estimating the human labour and change impact, i.e., the amount of change needed to fix a defect, is crucial for efficient defect assignment and prioritization in software maintenance [172, 329]. Existing change impact analysis research [186, 313] predicts the amount of change needed to fix a defect in terms of code churn using the summary and the description of the issue report only, and is unable to achieve a high accuracy of prediction, *i.e.*, Area Under the Curve (AUC) of 0.61 [266]. The AUC measures a model's predictive power, with 1.0 being perfect and 0.5 indicating random guessing. The low AUC reveals the need for improved change impact analysis methods. Issue reports contain valuable information and share common characteristics [205, 296, 297, 305]. Hence, it is important to provide approaches that predict the change impact by leveraging the common characteristics of the issue report to predict the change impact with high accuracy.

- **Challenge 2: Lack of approaches to understanding code clone dynamics in DL frameworks.** DL frameworks, like other systems, are prone to code clones [123, 188]. Code cloning can have positive and negative implications for software development influencing maintenance [134, 152, 167]. DL frameworks play a role in advancing artificial intelligence. Therefore, it is essential to develop approaches that understand the impact of code clones on the software quality and maintainability of these critical systems. Traditional studies on code clones may not be valid for DL problems because DL frameworks have unique characteristics, such as complex dependencies and rapidly evolving libraries [13], which require further analysis approaches. While a few existing studies [124, 188] focus on studying clones in DL-based applications, no work has been done investigating clones, their evolution and their impact on the maintenance in the DL frameworks.

Area 2: Feature Enhancement for Mobile Applications

- **Challenge 3: Lack of approaches conducting comprehensive competitor feature analysis for mobile app improvement.** The mobile application (app) market is experiencing explosive growth, with global downloads reaching a staggering 257 billion in 2023 [45]. In such a competitive landscape, developers must continually understand and meet user needs while also differentiating their apps from competitors'. Therefore, developers need to read reviews from all their competing apps and summarize the advantages and disadvantages of each app. Such a manual process can be tedious and even infeasible, with thousands of reviews posted daily. Few prior studies [60, 151, 243, 246] have shed light on the potential of automatically extracting useful information, i.e., user opinions (sentiment and rating) associated with specific app *features*, for app comparisons. However, these extracted app features are fine-grained and explicitly mentioned in reviews via word pairs such as "*photo edition*" and "*upload video*". Comparing competing apps on fine-grained features can be challenging and time-consuming as the number of extracted fine-grained features can reach up to hundreds or even thousands per competing group [60, 244]. Therefore, it is important to provide an approach that conducts a comprehensive, non-fine-grained comparison of the features of competing apps.

- **Challenge 4: Lack of approaches generating feature enhancement suggestions leveraging competitor analysis.** Researchers have explored various approaches to conduct competitor user review analysis [20, 60, 151, 160, 243, 247]. However, existing work presents some limitations. First, the existing approaches often generate an overwhelming number of fine-grained features [60, 243, 247] due to comparing the apps' features based on word pairs, making it hard to conduct competitor user review analysis with thousands of features [244]. Second, competitor user review analysis is only conducted by identifying explicit expressions of comparison (e.g., *"Zoom's screen sharing is way smoother than Skype's"*) [151] and fails to take into consideration implicit insights derived from user reviews. Third, existing

work on competitor user review analysis [20] offers only feature rating comparisons lacking the ability to suggest concrete improvements for specific features based on competitors' user feedback. Therefore, it is essential to develop approaches that automatically generate suggestions for mobile app feature improvements by leveraging user feedback on similar features from competitors.

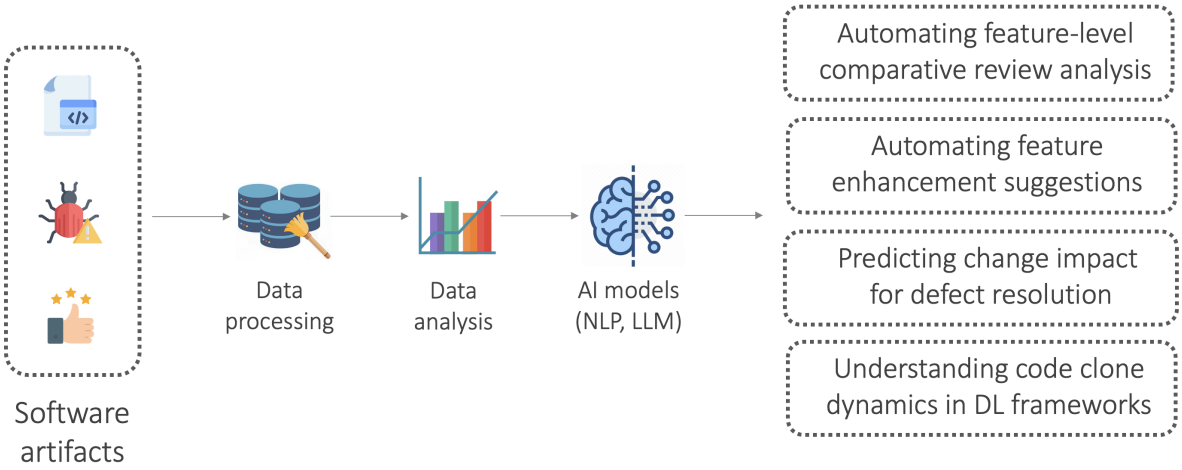

Figure 1.3: Overview of our proposed approaches in this thesis.

1.3 Thesis Statement

Software maintenance remains a complex challenge despite ongoing research efforts. Developers face constant pressure to maintain a robust codebase while continually enhancing features to stay competitive. This thesis proposes novel approaches leveraging software artifacts and ML to support developers in two critical areas within software maintenance: code maintenance and feature enhancement. Figure 1.3 In the code maintenance part, our research tackles two perspectives: (1) enhancing the accuracy of predicting change impact for defect resolution by leveraging the common characteristics of issue reports and (2) understanding the dynamics and impact of code clones in DL frameworks to ensure long-term code quality and maintainability. For feature enhancement, our contributions include (3) conducting automatic comprehensive competitor feature analysis and (4) generating suggestions for feature enhancement based on competitor user review analysis for mobile applications. This thesis aims to empower developers with novel approaches for

streamlined and effective software maintenance, thereby fostering a high-quality maintained codebase and competitive and sustainable software systems.

Summary of Thesis Statement

Developers continue to face significant challenges in software maintenance despite existing research. These challenges include: 1) the lack of approaches predicting accurate change impact for defect resolution, 2) the lack of approaches to understanding code clone dynamics in DL frameworks, 3) the lack of approaches to conducting comprehensive competitor feature analysis for mobile app improvement, and 4) lack of approaches generating feature enhancement suggestions leveraging competitor analysis. To guide software developers in effective software maintenance, we propose approaches that support developers through two main objectives: code maintenance and feature enhancement.

1.4 Thesis Objectives

Figure 1.4 provides an overview of the proposed approaches in this thesis. These approaches target the challenges outlined in Section 1.2 by addressing defect resolution through predicting change impact, conducting code clone analysis and enhancing software features.

- **Objective 1: Predicting change impact for defect resolution leveraging issue report information.** Software systems are prone to defects. Predicting the change impact needed to fix a defect is a crucial step in optimizing the defect resolution process. However, accurate prediction of change impact for defect resolution remains a significant challenge in software maintenance (i.e., Challenge 1). To address this challenge, we propose a predictive model capable of estimating the change impact of resolving a defect by leveraging historical data and Natural Language Processing (NLP), i.e., state-of-the-art topic modelling algorithm ETM [65]. Instead of relying on the individual textual description of each defect, our approach leverages the collective information of semantically similar defects, i.e., defect topic,

extracted through ETM. We construct several machine learning models to predict the change impact using the identified defect topics and other information extracted from issue reports and identify the defects requiring small or large change impact along two dimensions, i.e., the code churn size and the number of files changed.

- **Objective 2: Understanding code clone dynamics in DL frameworks.** While code cloning saves time initially, it results in higher maintenance and lower software quality over time, compromising system reliability. To address the challenge of limited understanding of code clone evolution in DL frameworks (i.e., challenge 2), we conduct a longitudinal empirical study on nine popular DL frameworks code clone evolution and its implications for reliable systems. Our approach identifies four trends, i.e., *"Serpentine"*, *"Rise and Fall"*, *"Decreasing"*, and *"Stable"*, in the evolution of clones observed in DL frameworks. To better understand the development practices that lead to these different trends, we conduct a within-release development investigation. Additionally, we explore the presence of cross-project code clones within DL frameworks and how the collaborative community size evolves across different cloning trends.

- **Objective 3: Automating feature-level comparative user review analysis for competing mobile applications.** To address the challenge of lack of comprehensive competitor mobile apps feature analysis (i.e., Challenge 3), we propose FeatCompare, an approach to automatically mine and compare features of competing mobile applications from user reviews without any manually annotated resources. Our approach tool adapts Attention-based Aspect Extraction (ABAE) [98], the state-of-the-art neural network-based model for feature extraction. FeatCompare is capable of extracting and differentiating, through a Global-Local sensitivity ratio, global common application features (i.e., performance issues) and local domain-specific ones (i.e., accurate forecasts for weather applications) to provide software engineers with full-spectrum features comparison. Then, FeatCompare creates a comparative table that summarizes users' opinions for each identified feature across

competing apps. The comparative analysis guides developers toward making well-informed decisions about their app's features and functionalities.

- **Objective 4: Proposing an automatic feature enhancement suggestions approach through competitor user review analysis.** The mobile application (app) market is witnessing explosive app adoption, fostering a highly competitive environment among competitors. To stay competitive, developers need to learn from their competitors' behaviours to maintain a competitive edge. To address the challenge of generating effective feature enhancement suggestions (i.e., Challenge 4), we propose a *Large Language Model (LLM)-based Competitive User Review Analysis for Feature Enhancement)* (*LLM-Cure*), an approach powered by LLMs to generate suggestions for mobile app feature improvements automatically. More specifically, *LLM-Cure* identifies and categorizes features within reviews by applying LLMs. When provided with a complaint in a user review, *LLM-Cure* curates highly rated (4 and 5 stars) reviews in competing apps related to the complaint and proposes potential improvements tailored to the target application. By analyzing user reviews from competing apps, developers can uncover insights into features that address unmet needs, potentially giving their app a significant advantage.

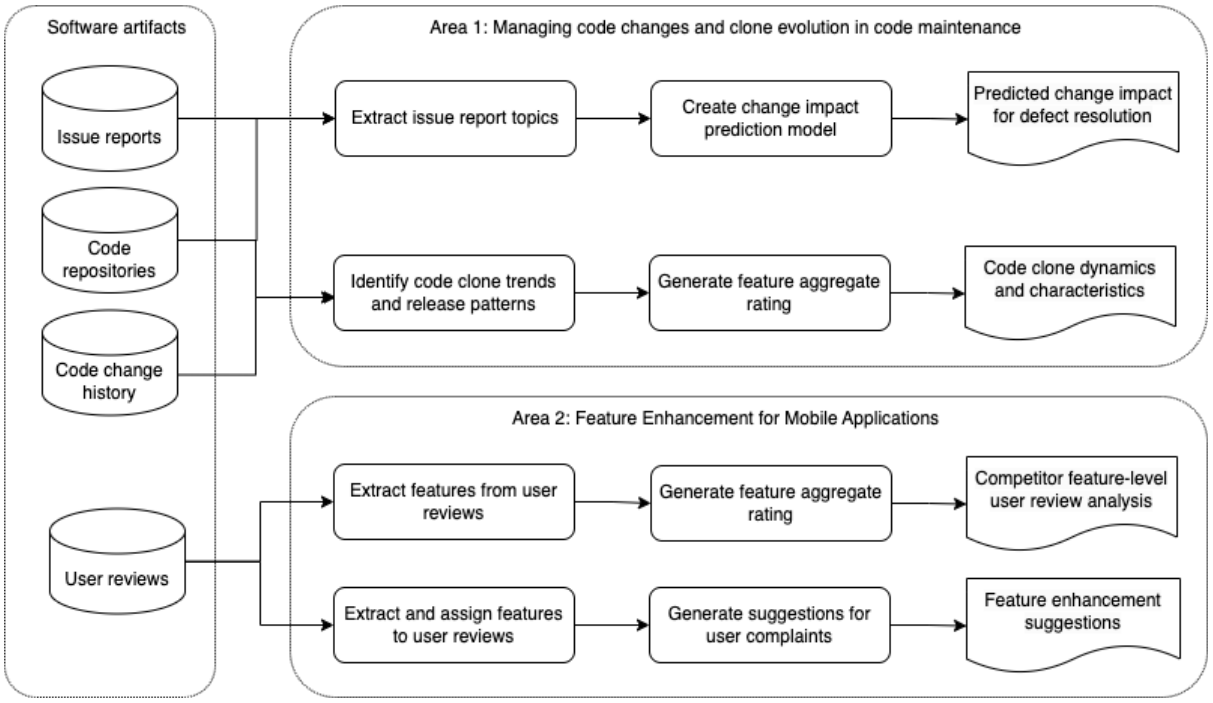

Figure 1.4: Overview of our proposed approaches in this thesis.

1.5 Thesis Overview

We present an overview of this thesis in the following.

Chapter 2 provides a comprehensive literature review of related work.

Chapter 3 details our predictive model, which estimates the change impact of resolving a defect by leveraging historical data and the state-of-the-art topic modelling algorithm Embedded Topic Model (ETM).

Chapter 4 explores our approach to investigating code clone dynamics in DL frameworks, presenting findings on long-term cloning trends and within-release clone evolution characteristics. We discuss practical implications for practitioners and researchers.

Chapter 5 describes our approach for conducting comprehensive competitor feature analysis aimed at mobile app improvement.

Chapter 6 outlines our automated approach to generating feature enhancement suggestions based on competitor user review analysis.

Chapter 7 concludes the thesis and discusses potential future research directions.

Chapter 2

Literature Review

2.1 Change Impact Prediction

In this section, we present the plethora of available literature closely related to our research and summarize the contributions in each area: defect taxonomies and classifications, defect fixing effort prediction, defect localization, and change impact analysis.

2.1.1 Classifying software defects.

Generic defect taxonomies. To understand possible defect types, researchers suggest taxonomies targeting general defects but focusing on one specific dimension or several dimensions, *e.g.*, impact, category, root cause and life cycle phase. Several studies propose *fine-grained taxonomies* [103, 154, 209] that are defined based on code patterns that represent defects, *e.g.*, conditional statements and initialization errors. Ni et al. [209] exploit defect fixes from source code to an AST level and categorize defects into fine-grained root cause categories, such as conditional test errors and data verification errors. Some researchers propose to map code-related fine-grained categories to coarse-level dimensions. For example, Li et al. [154] manually mapped fine-grained code-related root causes to three coarse-level categories, *i.e.*, memory, concurrency and semantic.

Other researchers introduce *coarse-level defect taxonomies.* Catolino et al. [44] perform an empirical study to find coarse-level defect categories. The authors manually analyze 1,280 defects and build a taxonomy of 9 root cause categories: configuration, GUI, Performance, Program anomaly, Test-code, Database, Network, Permission and

Security issues. Catolino et al. then model an automated supervised classifier to label defects. Although the classifier achieves 64% as F-measure in general across all the defect categories, it fails to predict the defects for the configuration and network categories. Ahmed et al. [6] also present a framework, CapBug, which classifies defects according to the coarse-level categories. CapBug is similar to Catolino et al.'s approach in two ways. CapBug follows a supervised technique to categorize issue reports. It relies on manually predefined categories. Unlike Catolino's work, CapBug studies six defect categories, *i.e.,* Program anomaly, GUI, Network or Security, Configuration, Performance, and Test code. CapBug achieves 88% accuracy in predicting the category of the defects by using the Random Forest model and SMOTE technique to deal with class imbalance.

Specific defect taxonomies. A rich collection of research work focuses on a specific programming language [4, 55, 95] or defects discovered in a specific system [112, 119, 216, 218, 219, 240, 267, 278, 318, 321] or even a specific class of defects [5, 71, 261, 268, 282, 312]. For example, Ciborowska et al. [55] shed light on the differences in the characteristics and localization of the defect across COBOL and non-COBOL software. To achieve this, Ciborowska et al. refine the taxonomy introduced by Catolino et al. [44] by considering the opinion of surveyed developers.

Some researchers have examined the types of defects related to specific systems, such as AI-based systems [112, 119, 267, 321]. Others amend existing taxonomies to classify AI defects. For example, Thung et al. [267] study the characteristics of 500 defects belonging to three machine learning systems (Apache Mahout, Apache Lucene, and Apache OpenNLP). Thung et al. manually categorize defects leveraging the taxonomy designed by Carolyn et al. [240] and include an additional category, 'configuration'. Other researchers design new taxonomies. For example, Zhang et al. [321] propose a new taxonomy for TensorFlow defects by analyzing 175 issue reports.

Researchers also investigate specific classes of defects. For instance, the research community shed light on understanding the characteristics of performance defects [261, 312, 322] since they lead to user dissatisfaction. For example, Sanchez et al. [261] propose

a taxonomy of three dimensions, *i.e.,* effects, causes and contexts of defects for real-world performance defects. To support practitioners in building more secure software, researchers propose taxonomies for security defects [5, 268, 312].

Summary

Prior research has proposed various approaches to classify software defects. However, these approaches have limitations: (1) they often present a *fine-grained taxonomy* focused on code-related defects, which is out of the scope of our study, and (2)their coarse-level matching for fine-grained code-related root causes is restricted to a few categories, such as memory, concurrency, and semantics. These taxonomies represent only a subset of the potential software defects and require an expensive amount of human effort to categorize the issue reports manually. Consequently, they offer limited support for developers in understanding other types of defects. In contrast, our study (1) encompasses all types of defects without restricting itself to any specific class, and (2) does not require any labelled data.

2.1.2 Defect fixing effort

Defect fixing effort refers to the amount of time and code modifications required to identify, address, and resolve defects or bugs in a software system. It specifically involves defect fixing time and code churn size.

Defect fixing time. Several studies leveraged the issue report information (*i.e.,* component, priority and severity) to predict the defect fixing effort in terms of time required for fixing defects [17, 18, 87, 106].

Giger et al. [87] carry out an empirical study to investigate the possibility of predicting the defect fixing time from the issue report attributes. In their study, Giger et al. aim to classify the issue reports into two classes: requiring *Slow* or *Fast* fixing effort. The authors build a decision tree model that achieves a precision of 0.654. Giger et al. find that assignee, reporter and monthOpened are the top 3 issue report attributes influencing the fixing effort time prediction. Similarly, Ardimento and Mele [18] also treat the fixing time prediction as a binary classification problem (*i.e.,* Slow and Fast classes). The

authors treat the problem as a supervised text categorization task but used Bidirectional Encoder Representations from Transformers (BERT) [63] with a classifier to predict the fixing time. In their approach, the authors propose a new set of features, including the description of the issue and developers' comments on which they perform transfer learning.

In contrast to the above, some researchers treat the fixing effort prediction as a regression problem. Yuan et al. [308] propose an approach to predict Bug Fixing Time with Neural Networks, named BuFTNN. BuFTNN leverages several features (*i.e.,* developers' activities, developers' sentiments, the semantics of bugs, and efforts caused by understanding and analyzing source code) to predict a defect's fixing time. The proposed model is validated on four real-life projects, including Eclipse, and outperforms the state-of-the-art deep learning-based model (called DeepLSTMPred) [242], effort prediction model by 7.3% in F1-score, on average.

Another line of work handles the fixing effort prediction problem as information retrieval. For instance, the time required to fix a defect is anticipated by querying textually similar issue reports marked as fixed. The fixing time of the new issue report is then estimated by relying on the fixing time of similar retrieved reports. Weiss et al. [291] rely on the k-Nearest Neighbors algorithm combined with textual similarity to identify similar defects. Zhang et al. [315] follow a similar approach to Weiss et al.; however, they focus on commercial projects.

In the above studies, the fixing time is calculated as the time elapsed between the issue being reported/assigned and resolved. However, the estimated time might not indicate the actual fixing time. In some cases, developers might be working on several tasks or working part-time. In other instances, the issue report status might not be updated on time. Different from the above studies that predict the fixing effort in terms of time, we measure the change impact, i.e., code churn size and the number of files changed, as a proxy for the fixing effort.

Code churn size. Measuring lines of code has been considered a standard way of estimating the developer effort [36]. Nevertheless, the possibility of predicting the defect

fixing effort in terms of code churn (*i.e.,* number of lines of code modified) remains scarcely explored. To the best of our knowledge, Thung's study [266] is the only existing work that predicts defect fixing effort by considering the code churn size. Thung designs a supervised machine learning approach to classify issue reports into *low* and *high* categories. The author fed the summary and description of the issue report as input to a Support Vector Machine model. Thung evaluates the model on 1,029 bugs from *hadoop-common* and *struts2* and achieves an AUC of 0.61.

Summary

Prior research estimates the fixing effort in terms of fixing time, i.e., the time elapsed between the issue being reported/assigned and resolved. However, the estimated time might not indicate the actual fixing time. In some cases, developers might be working on several tasks or working part-time and in other instances, the issue report status might not be updated on time. Alternatively, Thung's work predicts fixing effort in terms of code churn size. Our work aligns with Thung's approach but introduces several key differences. First, we use different features to predict the change impact size. Second, we implement multiple ML models and utilize a larger dataset. Our study complements the existing approaches.

2.1.3 Defect localization

Defect localization is one of the crucial yet most costly and time-consuming steps of software maintenance [66]. There are three widely used families of defect localization techniques: spectrum-based (SBBL), mutation-based (MBBL), and information retrieval-based (IRBL). SBBL techniques [41,292,317] employ a series of test cases to identify the buggy code spectra. SBBL is considered a fine-grained defect localization as it can point to the buggy statement. MBBL techniques [46,105,298] employ test results after mutating the program to determine the buggy code. IRBL techniques employ information from the issue report to localize the defect. IRBL techniques focus on the textual similarity between the issue report description and the source code [295]. In IRBL techniques, program entities of different granularity might be identified as faulty components. Some

research work [306] points to the files containing the defect. For instance, Chen et al. [48] propose Pathidea, an information retrieval defect localization approach that relies on both log snippets and stack traces in issue reports to localize the defect file. Other research work [30,202,293,295] focuses on identifying the commit that caused the defect. One team at Facebook proposes Bug2Commit [202], an IRBL approach that leverages unsupervised techniques to find commit-level defects. Both IRBL localization approaches and our work leverage the issue reports information to support the developers with the software maintenance task. Our research work complements the defect localization line of work. The defect localization approaches seek to identify the elements of a program causing the failure with different granularity, i.e., statement, method and file. However, our work focuses on predicting the change impact, i.e., the amount of change, in terms of lines of code and the number of files changed.

Summary

Prior research has suggested information retrieval-based (IRBL) defect localization approaches to localize a defect within a software system. Both IRBL approaches and our work leverage information from issue reports to support the developers in software maintenance tasks. Our research complements the defect localization work. The defect localization approaches seek to identify the elements of a program causing the failure with different granularity, i.e., statement, method and file. However, our work focuses on predicting the change impact, i.e., the amount of change, in terms of lines of code and the number of files changed.

2.1.4 Change impact analysis

Software Change Impact Analysis (CIA) represents a collection of techniques that help developers identify the effects of a change or to estimate the amount of change needed [37] during the software maintenance and evolution phase. Depending on the technique and its application, CIA can be helpful before implementing the change (*e.g.,* predicting the change impact) or after implementing it (*e.g.,* performing change propagation) [148]. Researchers define metrics and prediction methods to conduct CIA along four main CIA

quantification parameters: (1) instability (*i.e.,* likelihood that various software artifacts change simultaneously), (2) amount of change (*i.e.,* the size of changes affecting a software artifact), (3) change proneness (*i.e.,* likelihood that a software artifact will change due to bug fixes or requirement changes), (4) and changeability (*i.e.,* the level of ease related to the affecting the changes to a software artifact) [142]. The existing research related to the amount of change is the closest to our work. Existing work quantifies the amount of change in terms of code churn [102, 187], the number of changed artifacts (*i.e.,* modules, operations, members, classes, or files) [252] and incremental changes [12]. Similar to our goal, some existing work estimates the change set, *i.e.,* the amount of change, using static analysis [238, 257], dynamic analysis [40, 110] or Mining Software Repositories (MSR) [86, 313]. MSR techniques leverage historical information from the defect artifacts and source code repository to predict the potential code change. Gethers et al. [86] leverage the issue report textual information and the source code to estimate the amount of change. The authors adopt information retrieval methods to couple the description of the issue report to the potential software entity, *e.g.,* method. Zanjani et al. [313] couple information retrieval, machine learning and source code analysis to build a corpus of source code entities queried when an issue report description is submitted.

Summary

Prior research has proposed various methods to predict the change impact of code changes in software systems. However, these existing approaches estimate the amount of change only after identifying the specific change set by examining the source code. In contrast, our work adopts a simpler approach that predicts the amount of change by only leveraging information from issue reports, without the need to first identify the actual change set in the source code.

2.2 Code Clone Dynamics in DL Frameworks

In this section, we present a review of the existing literature on analyzing DL frameworks. Additionally, we touch upon the studies relevant to code clone research.

2.2.1 Deep Learning framework-related empirical studies

Deep learning, emerging as a prominent field in recent years, has attracted considerable attention within the software engineering community. A growing number of researchers are actively empirically investigating the characteristics of DL applications and frameworks. While some researchers focused their interest on DL applications, such as identifying the challenges of building DL-based systems [197,319] and exploring the taxonomies of faults in DL applications [113, 123], others directed their attention toward the exploration of the DL frameworks that are the building blocks of these applications.

Bug taxonomies and characteristics. Researchers contributed to understanding DL framework bugs comprehensively in several ways. TensorFlow, one of the most popular DL frameworks, gained the attention of several researchers [126, 321]. For instance, Zhang et al. [321] analyzed TensorFlow bugs sourced from StackOverflow QA pages and GitHub. The authors provide taxonomies along several dimensions, including root causes, symptoms, and bug detection strategies. Other researchers expanded their work to investigate multiple DL frameworks. Chen et al. [49] conduct a comprehensive analysis of four frameworks, i.e., TensorFlow, PyTorch, MXNet, and DL4J, to identify root causes and symptoms and provide actionable guidelines for improved bug detection and debugging. In addition, the authors developed a tool called TenFuzz to identify bugs in Tensorflow. Similarly, Islam et al. [119] analyzed the characteristics of bugs in five DL frameworks. In addition, the authors identify the stages of the DL pipeline that are more bug-prone and investigate antipatterns. Tambon et al. [262] shed light on silent bugs in DL frameworks. The authors classified the bugs according to their impact on users' programs and the specific components where these issues originated, drawing upon information found in the issue reports. et al. [70] provide a classification of the fault-triggering conditions of bugs. The authors manually investigated 3,555 bug reports collected from three TensorFlow, MXNET and PaddlePaddle and analyzed the frequency distribution of different bug types and their evolution features. Long et al. [165] presented an exploratory study on performance and accuracy bugs in ten popular DL frameworks, revealing insights such

as the primary root cause for reporting performance bugs, and they offered actionable implications for researchers, maintainers, and submitters to improve the bug management process of performance bugs.

Bug fixing patterns. Jia et al. [126] explore TensorFlow, offering insights into the bugs and their fixing process within the framework's components. Ho et al. [104] replicate the study by Jia et al. to explore another popular DL framework, PyTorch. In addition to identifying the bug-fixing patterns in PyTorch, the authors provide a detailed comparison between TensorFlow and PyTorch, highlighting the similarities and differences between the frameworks' bugs. Islam et al. [120] conducted a larger scope study encompassing five DL frameworks. The authors investigated the challenges associated with 970 bug-fix patterns from Stack Overflow and GitHub and found that the most common patterns are related to data dimension and neural network connectivity issues. Li et al. [153] follow a different direction by focusing on multi-language DL frameworks. Apart from exploring the bug types and their impacts on DLF development, the authors discover that addressing bugs in multi-language frameworks involves significantly greater complexity in code changes compared to single-programming-language bug fixes.

Technical debt. Liu et al. [161] investigate the DL framework from a technical debt perspective. The authors analyze the comments indicating technical debt (self-admitted technical debt) of seven popular DL frameworks and identify seven types of technical debt in DL frameworks, i.e., design, defect, documentation, requirement, test, compatibility, and algorithm debt.

Summary

Prior research has conducted empirical studies on DL frameworks to examine bug taxonomies, bug-fixing patterns, and technical debt. These studies offer valuable context for understanding the maintenance challenges in DL frameworks. Our work on code clones complements this by focusing on the identification, evolution, and impact of code clones, thus providing additional insights into maintaining and improving the quality and sustainability of DL frameworks.

2.2.2 Code Clones Analysis

Code clones analysis in traditional systems Researchers have extensively explored code clones in traditional systems [325]. These studies encompass a multifaceted exploration of code clones, including their impact on software maintenance, bug-proneness, change-proneness, and evolution.

Impact on software maintenance: Existing work demonstrated that code cloning could result in increased maintenance challenges for software systems [22, 101, 121, 192]. Mondal et al. [192] conduct a comparative empirical study to investigate the maintenance efforts required for cloned and non-cloned functions. The study found that cloned code is associated with an increased maintenance cost, in particular with Type 2 and Type 3 clones. Higo et al. [101] demonstrated that among web-based systems developed from the same specifications, those projects with a high prevalence of code clones present a greater challenge for project testing. In particular, the study demonstrated an increased effort in bug detection during unit testing and an increased number of clones during integration testing. Islam et al. [121] conducted a comparative analysis of code clones with and without bugs, considering 29 code quality metrics across 2,077 revisions of three Java software systems and offering insights for cost-effective clone management and clone-aware software development.

Bug proneness: Several previous studies have examined the bug proneness of code clones, revealing that clones make code more bug-prone and increase maintenance costs

[117, 118, 127, 220, 277]. Islam et al. [118] compared the bug-proneness of clone and non-clone code. This paper presents a comparative study on the bug proneness of code clones and non-clones, finding that clone code has a significantly higher percentage of files changed due to bug-fix commits and a greater likelihood of severe bugs, suggesting that bug-fixing changes in clone code require extra attention. In another study, Islam et al. [117] delved into the bug-proneness of micro-clones, i.e., small code fragments of 1 to 4 lines of code and compared them with regular code clones in diverse open-source systems written in C, C#, and Java. The study findings show that micro-clones are significantly more prone to bugs, exhibit more consistent changes due to bug-fix commits, affect a higher percentage of files, and contain a greater percentage of severe bugs than regular clones, hence underscoring the importance of managing and maintaining micro-clones in software development. Jiang et al. [127] study also validated that code clones had a higher likelihood of containing bugs, primarily originating from inconsistencies within cloned code segments.

Change proneness: The negative impacts of code clones on software maintenance have led to extensive research on clone stability and change proneness. [167, 168, 190, 193, 194, 199]. For example, Mostafa [199] focused on analyzing clone evolution with respect to clone location, i.e., Inter-File and Intra-File, and clone lifetime. The study reveals that Intra-File clones are more prevalent, suggesting that developers tend to duplicate code within the same file, and these clones are also more dynamic, indicating a preference for refactoring or altering clones within the same file. Mondal et al. [190] found that cloned code tends to be more change-prone and unstable during the maintenance phase than non-cloned code. The study also identified differences in stability among various types of clones, programming languages, and programming paradigms. In a more recent study, Mondal et al. [194] conducted a comprehensive study on 12 subject systems to compare the stability of clone and non-clone codes. The authors demonstrated that code clones generally exhibit higher change-proneness as compared to non-clones by referring to the eight stability measuring metrics, implying increased maintenance effort and cost.

Clone evolution: Some researchers constructed clone genealogies by tracing the history

of code clones [23,265,271]. Barbour et al . [23] derived six distinct evolutionary patterns by building the clone genealogies of four open-source Java systems. In addition, the authors leveraged the clone genealogy information to enhance the effectiveness of fault prediction models. Similarly, Thongtanunam et al. [265] conducted an empirical study on six open-source Java systems genealogies, revealing that a significant proportion of clones have short lifespans. In addition, the authors predicted, using random forest classifiers, whether a newly introduced clone would be short-lived based on factors extracted from the genealogy. In a recent work, Bladel and Demeyer [271] conducted a study on eight open-source systems genealogies and revealed that code clones are more prevalent in test code compared to production code due to the recurring pattern of unit test code.

Summary

Prior research has studied the challenges of code cloning in traditional software. However, due to the distinct nature of DL frameworks, these findings cannot be directly applied. Our work aims to bridge this gap by specifically examining code cloning issues in DL frameworks.

Code clones analysis in DL systems. Despite the increasing surge in the development of DL software, only two studies have investigated code clones within this domain, highlighting the need for more comprehensive investigations into code cloning practices in DL software.

Jebnoun et al. [124] introduced the first study on clones in DL development. The authors analyzed code clones in 59 Python, 14 C#, and 6 Java-based DL systems and an equal number of traditional software, highlighting the frequency, distribution, and effects of code clones. In addition, they provided a taxonomy to identify phases with a higher risk of cloning-related faults. Their findings indicated that code cloning is prevalent in DL systems, with developers often copying code from files located in other directories, and that code cloning is more common during DL model creation, training, and data preprocessing phases. In addition, the authors find that code clones in DL code are likely to be more defect-prone compared to non-cloned code.

Similarly, Mo et al. [188] also found that code clones are prevalent in DL systems, exhibiting nearly twice the rate in traditional projects. However, the authors conducted their study on an optimized dataset of Python projects with only 45 DL and 45 traditional projects. Different from the work of Jebnoun et al., Mo et al. focused on co-changed clones and investigated the distribution of Type 1, Type 2, and Type 3 co-changes and their bug-proneness. The authors also conducted a comparative analysis of the DL applications based on the underlying DL frameworks.

Summary

Prior research has studied code clones in DL systems. However, existing work focuses on DL applications and considers only one system snapshot. Our work diverges in two key aspects: first, we examine code clones within DL frameworks instead of applications, and second, we emphasize the evolutionary and comparative aspects of code clones rather than just their distribution at a single point in time. Additionally, we analyze code clones across multiple DL projects, providing a broader perspective beyond individual projects.

2.3 Mobile App Review Analysis

In this section, we first introduce the existing mobile app review analysis approaches for competing apps. We then briefly mention approaches for extracting fine-grained features from app reviews and other studies on mining mobile app reviews.

Mining User Reviews for Mobile App Comparison. While automated tools are increasingly being proposed to analyze the reviews of a specific mobile app [92, 94, 114, 131, 245], few researchers have centered their focus on extracting useful knowledge from reviews of competing apps [60, 151, 243, 246]. For example, Shah et al. [246] propose the task of mining mobile app reviews for competitor analysis as a tool named REVSUM [246]. REVSUM takes as input reviews from a set of competing apps and compares users' sentiments on each fine-grained feature among competing apps. First, REVSUM identifies reviews that contain feature evaluation, bug reports, or feature requests. Next, it extracts

fine-grained features from selected reviews using an approach named SAFE [131]. In the end, REVSUM applies the sentiment score prediction function offered in the Stanford CoreNLP library on sentences that mention fine-grained feature(s) reviews and compares the average sentiment score of each feature across competing apps. Shah et al. did not evaluate the accuracy of the sentiment score prediction component and the fine-grained feature extraction component in REVSUM. Thus, it is unclear how REVSUM performs in practice.

Dalpiaz and Parente design a similar tool named RE-SWOT that can extract fine-grained feature requirements from user reviews of competing apps and generate a Strength-Weakness-Opportunity-Threat (SWOT) matrix supporting competitor analysis [60]. RE-SWOT takes as input all reviews of competing apps and identifies fine-grained features by finding word pairs that co-occur frequently in reviews. Several hand-craft rules are applied to filter out meaningless co-occur word pairs. To further reduce the number of fine-grained features, RE-SWOT merges semantically similar features by invoking a closed NLP service[1]. After grouping similar fine-grained features, each review is then assigned to identified features based on words appearing in the review. Different from REVSUM, RE-SWOT does not apply any sentiment analysis tool to reviews. It aggregates ratings of reviews associated with each feature and creates a SWOT table for each app based on the average rating per feature compared to the average rating per competing group. RE-SWOT suffers from three main issues: 1) the identified fine-grained features have not been evaluated; 2) only a small-scale interview was conducted to validate the usefulness of the generated SWOT tables; 3) it does not filter out non-informative reviews.

Different from RE-SWOT and REVSUM, which first identify fine-grained features from user reviews and then summarize users' opinions on each feature among competing apps, Li et al. [151] propose a tool that can compare features of competing apps via identifying comparative reviews. Comparative reviews are reviews such as *"Slower page loading than chrome"* for the Firefox mobile app, which directly compares Firefox with Chrome. Their approach is good at identifying explicit app comparisons provided by

[1] `https://www.cortical.io/`

users. However, it fails to compare other features that are not mentioned in user reviews in a comparative fashion. Besides, there might be a limited number of comparative reviews available among a target set of competing apps.

Shah et al. [243] introduce an approach that compares two apps' features pairwise but fails to provide a comparison of more than two competing apps as FeatCompare is capable of. First, the authors extract fine-grained features from 25 collected apps by relying on the two-word collocations in user reviews. Second, the authors choose a base app and identify its features. Then, the authors identify the list of competing apps based on the common low-level features shared among the set of 25 apps and the selected base app. The competitor analysis is conducted by comparing the sentiments of the features of the base app and the selected competitor.

Summary

Prior research has explored various approaches to conducting competitor user review analysis. However, existing methods present some limitations. They often generate an overwhelming number of fine-grained features based on word pairs, making it hard to conduct competitor user review analysis. Additionally, they focus on explicit expressions of comparison and overlook implicit insights from user reviews. To address these limitations, we propose an approach that facilitates the analysis by focusing on high-level features across multiple competing apps.

Extracting Features from Mobile App Reviews. Extracting features from user reviews for a specific app is a trending and fundamental task in app store mining research [92, 94, 115, 131, 245]. Identified features could be used to summarize user opinions on app features and thus provide actionable insights for developers to improve their apps. Depending on how to identify words describing app features in reviews, these approaches can be categorized into two groups: rule-based approaches and collocation-based approaches.

Rule-based approaches such as MARA [114, 115] and SAFE [131] mine fine-grained features based on manual defined linguistic rules. However, these linguistic rules are often

created by investigating a limited number of reviews. Thus, they suffer from a potential loss of features due to the bias in sampled reviews. A recent study [243] shows that the most advanced rule-based feature extraction approach, SAFE, is sensitive to the density of the annotated app reviews in a review dataset and may lead to poor performance in practice.

To catch more fine-grained features and reduce manual work in rule-based approaches, researchers have proposed another line of tools that can automatically identify fine-grained features from reviews without linguistic patterns [92, 94]. These approaches are collocation-based, i.e., mining word pairs such as *"picture view"* that co-occur unusually often in reviews. Guzman and Maalej propose the first collocation-based feature extraction algorithm [94]. Their approach removes words that are not nouns, verbs, or adjectives from reviews. Next, they calculate the co-occurrence of all word pairs in the preprocessed reviews and treat those that appear in at least three reviews and have less than three words distance between them as fine-grained features. A sentiment analysis tool is then applied to each sentence that contains at least one fine-grained feature, and sentiment scores of sentences are aggregated to indicate user opinions on each feature. Gu and Kim propose SUR-Mine, which aims to answer "what parts are loved by users" for app developers [92]. Unlike previous work, SUR-Mine extracts feature and opinion word pairs such as *"prediction, accuracy"* together and then uses co-occur frequency to identify fine-grained features and their associated sentiment.

Summary

Prior research has investigated methods for extracting features from user reviews. However, these often rely on manually crafted linguistic patterns. To overcome this limitation, we propose an approach that leverages an unsupervised neural model to automatically identify the most semantically relevant features in each review, eliminating the need for manual effort.

Other studies on mining mobile app reviews. Researchers have investigated summarizing, categorizing, and prioritizing user reviews. Below, we outline the existing

work.

Summarizing reviews: Fu et al. [80] assume that negative reviews (associated with 1-star or 2-star ratings) are most interesting to developers. They apply Latent Dirichlet Allocation [35] (LDA) on negative reviews to identify the major reasons why users dislike an app and learn how users' complaints change over time. Vu et al. [276] believe that a set of keywords could capture developers' interest. Thus, they propose a framework that takes input from a set of keywords from developers and then ranks all reviews based on their relevance to the specified interest and groups the most relevant reviews to summarize user opinions.

Categorizing reviews: Plenty of taxonomy methods and corresponding classifiers are proposed to automatically categorize reviews based on user intent or software engineering topics. Panichella et al. [214] use natural language processing (NLP) and sentiment analysis techniques to automatically classify user reviews into four types: information seeking, information giving, feature request, and problem discovery. Villarroel et al. [274] propose a tool named CLAP that mines app reviews for the release planning of mobile apps. CLAP features a supervised learning algorithm that can categorize reviews into three categories: bug reports, suggestions for a new feature, and others. Consequently, CLAP clusters similar reviews and provides the developers with suggestions for future releases. Mcilroy et al. [178] introduce a fine-grained categorization of reviews. They identified 14 types of issues in reviews and found that up to 30% of the reviews raise various types of issues in a single review. SURF [64] considers a combined categorization of user intent and SE topics. Man et al. [173] define seven types of issues appearing in reviews across multiple app stores and propose a corresponding review classifier. Lu et al. [169] propose a review classifier that can categorize reviews into three main types: non-functional requests (related to reliability, usability, portability, and performance), functional feature requirements, and others.

Prioritizing and filtering non-informative reviews: Keertipati et al. [137] rank feature requests extracted from reviews based on four feature attributes, including frequency, rating, negative emotions, and deontics. Recently, Gao et al. [83] propose a tool named

IDEA to identify emerging issues from user reviews. They define the emerging issue as an issue in a time slice that rarely appears in the previous slice but is mentioned by a significant proportion of user reviews in the current slice.

Despite the potential usage of user reviews for improving mobile apps, many user reviews contain less valuable information, such as pure user emotional expression, questions, etc. To solve this issue, Chen et al. [50] proposed an app review analyzing framework named AR-Miner. AR-Miner first identifies non-informative reviews by training a semi-supervised algorithm on a small scale of labeled reviews along with a mass of unlabeled reviews. Next, it groups the informative reviews using LDA and further prioritizes them with an effective review ranking scheme. FeatCompare adopts the review filtering component AR-Miner in the data preprocessing step to filter non-informative sentences from reviews.

Summary

Prior work has investigated mining information from user reviews, focusing on aspects like summarization, categorization, and prioritization. Our research complements and leverages this prior work by using the mined information to conduct mobile app competitive analysis.

2.4 Feature Enhancement Suggestions

Feature Enhancement. Prior approaches have explored automatic feature extraction from user reviews for feature enhancement [62, 85, 160, 284]. For example, Scalabrino et al. [62] introduce CLAP, a web application facilitating mobile app release planning by analyzing user reviews. CLAP categorizes reviews from the target app, prioritizing user concerns to be addressed. Wang et al. [284] present UISMiner, which supports UI-related feature enhancement by mining user review suggestions about UI. Gao et al. [85] propose a method for analyzing user reviews to extract requirements and update app goal models, including feature improvements and additions. Liu et al. [160] present an approach considering market trends, guiding developers on feature update strategies by

comparing features of similar apps. The proposed approach primarily suggests which features to update rather than offering suggestions for improvement.

Summary

Prior research has explored methods for feature enhancement using user reviews. However, these efforts focus solely on individual target apps, lacking consideration of competing apps. To address this limitation, we propose an approach that provides a competitive landscape for feature improvement by harnessing user reviews from competitors.

LLMs for Mobile Apps. De Lima et al. [157] propose a method utilizing LLMs to autonomously identify risk factors from app reviews and prioritize them to anticipate and mitigate risks. Roumeliotis et al. [227] conduct an evaluation study comparing the effectiveness of LLMs like Llama and GPT 3.5 in predicting sentiment analysis related to e-commerce. Similarly, Zhang et al. [320] assess the performance of three open-source LLMs in zero-shot and few-shot settings for predicting sentiment in user reviews. Xu et al. [299] design a prompt instructing ChatGPT to extract aspect-category-opinion-sentiment quadruples from text. Wei et al. [289] propose Mini-BAR, a tool integrating LLMs for zero-shot mining of bilingual user reviews in English and French. Dos Santos et al. [69] analyze accessibility reviews using LLMs. While the above work leverages user reviews, Huang et al. [289] introduce CrashTranslator, which automatically reproduces mobile app crashes from stack traces guided by LLMs to predict the exploration steps for triggering the crash. Liu et al. [163] propose InputBlaster, leveraging LLMs to generate unusual text inputs for mobile app crash detection.

Summary

Prior research has utilized state-of-the-art ML techniques, particularly LLMs, for mobile app analysis, such as risk assessment and sentiment analysis. Our approach employs LLMs to offer suggestions for feature enhancement, thereby complementing the existing contributions in supporting mobile app analysis.

Chapter 3

Change Impact Prediction for Defect Resolution

In this chapter, we present our study on predicting change impact for defect resolution. Section 3.1 provides an introduction and the motivation for our study. Section 3.2 details the design of our work. Section 3.3 reports the results of our experiments. Section 3.4 discusses the implications of our findings. Section 3.5 examines the threats to validity. Finally, Section 3.6 summarizes the chapter and suggests directions for future research.

3.1 Introduction

Software systems are prone to defects. Issue reports are usually employed to ensure an efficient defect reporting process. Upon receiving a new issue report, developers start by investigating the nature of the software failure. In the literature, researchers support developers in fixing defects by automating various defect-related processes including defect assignment [125], duplicate defect detection [256], defect localization [145], change impact analysis [142], defect prioritization [6], and defect fixing time prediction [291].

Estimating the human labour and the change impact, *i.e.*, amount of change, needed to fix a defect, plays a crucial factor in the efficiency of the defect assignment and prioritization [172, 329] given that there are limited human resources, *i.e.*, developers, available to work on a software project [135]. In the defect fixing effort prediction research, the existing work leverages the status of the issue reports and the textual information of an issue, e.g., the title and the description, to predict the fixing effort in terms of

fixing time [87, 308, 315] and code churn [266]. Similarly, change impact analysis research [186, 313] predicts the amount of change needed to fix a defect in terms of code churn and the number of files changed, leveraging the historical data of the application source code and the textual information of issue reports.

Nevertheless, the predicted defect fixing time might not be accurate since the prediction relies on the status of the issue reports, which might not be updated on time to reflect the real fixing time [122, 314]. For example, developers might not start fixing the defect right after the corresponding issue report was assigned to them. Moreover, the calculated predicted time represents the calendar days or hours, not the working days or hours that reflect the real effort to fix a defect [17]. In addition, none of the existing change impact techniques focusing on predicting the amount of change leverages the common characteristics of the issue reports to quantize the size of the change into small and large change impact categories without relying on the source code.

In this study, we assess the change impact that predicts the amount of fixing needed in terms of (1) the code churn size and (2) the number of files changed. In existing work, researchers leverage the collective knowledge of issue reports by identifying shared topics among defects and use them in defect assignments [205, 296, 297, 305]. In our work, we consider the collection of defects belonging to the same topic, which can provide common characteristics of the defects and leverage the topics of issue reports to estimate the change impact rather than relying on the content of individual issue reports. By leveraging the topics of issue reports, we could improve the accuracy of the change impact prediction models. More specifically, we conduct our study in two steps. First, we automatically assign a topic to each issue report leveraging the state-of-the-art topic modelling technique, *i.e.,* Embedded Topic Model (ETM) [65]. Second, we train eight predictive models capable of distinguishing between defects requiring a small or large change impact while leveraging the identified topics.

We conduct an empirical study on 298,548 issue reports belonging to three known ecosystems, *i.e.,* Mozilla, Apache, and Eclipse. We predict the change impact along two dimensions, *i.e.,* the code churn size and the number of files changed. In addition, we

investigate the most influential features affecting the prediction. We structure our study along by answering the following research questions (RQs):

RQ3.1: Can we accurately assign topics to issue reports of software systems using ETM?

Developers spend time manually investigating an issue report to understand the nature of a defect. Automatically identifying the topic of an issue report can save the developers' effort and time. In this RQ, we demonstrate that it is feasible to leverage ETM [65] to automatically assign topics to issue reports with a promising average accuracy of 79% over the three ecosystems.

RQ3.2: Can we predict the change impact of resolving defects in terms of code churn size? What are the most influential metrics for predicting the change impact in terms of code churn?

Practitioners estimate the change impact of resolving a defect to prioritize the list of defects efficiently. The change impact can be measured by calculating the lines of code changes. In this RQ, we demonstrate that it is feasible to leverage the topics of issue reports to predict the change impact in terms of code churn with a high accuracy, achieving an AUC score of up to 0.84. We find that the number of attachments, the number of issues blocked, and the number of comments per developer are the most influential metrics for the change impact prediction.

RQ3.3: Can we predict the change impact of resolving defects in terms of the number of files changed?

Certain defects propagate to several source code locations and require a fix in several files. The number of modified files represents the change impact of resolving the defect. In this RQ, we demonstrate that it is feasible to predict the change impact in terms of the number of modified files with an AUC score up to 0.82.

3.2 Study Design

In this section, we present the dataset and the steps to collect and prepare the metrics from issue reports and source code repositories. We also discuss ETM, a state-of-the-art topic modelling technique that we adopt to taxonomize the collected issue reports. Figure 3.1 depicts the overview of our study.

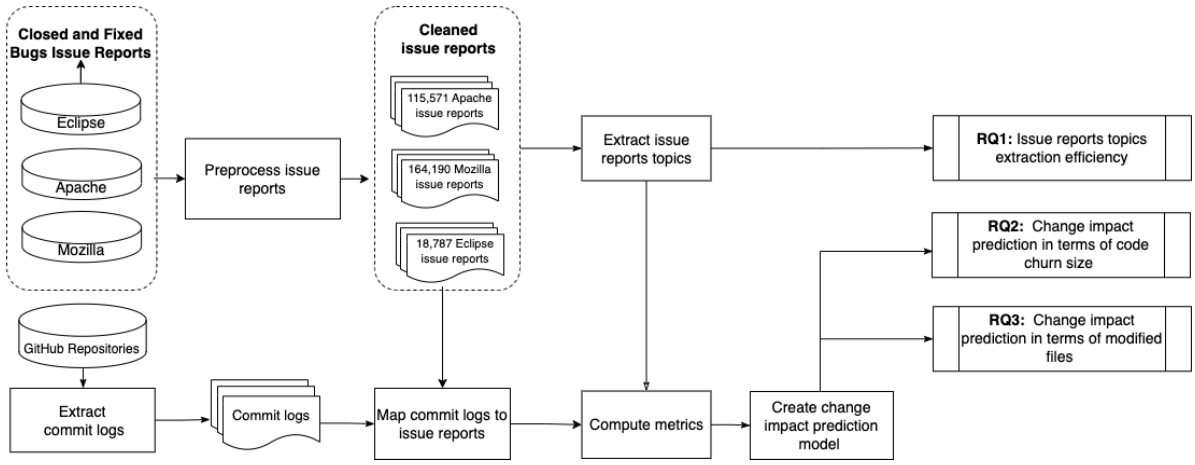

Figure 3.1: Overview of our experiment.

3.2.1 Data collection

Our study focuses on three large open-source software ecosystems, *i.e.*, Mozilla, Apache, and Eclipse. We specifically select issue reports belonging to these three ecosystems as they are widely used in the realm of software engineering [111, 328], and they are rich in issue reports.

Mozilla and Apache. For Mozilla and Apache, we utilize the 20-MAD dataset provided by Claes and Mäntylä [56] that encompasses 20 years of issue tracking and commit information existing between 1998 to January 2020. Issue reports for the Apache projects are extracted from Jira, while those for Mozilla are obtained from Bugzilla. The dataset includes meta-data information about commits (*e.g.*, hash and commit date), issues (*e.g.*, summary, description, and status), and comments (*e.g.*, author and date created). In addition, several other comment-related metrics (*e.g.*, emoticons, sentistrength) processed with various NLP tools are included in the dataset. The data extracted is stored in Parquet file format. First, we convert the data to CSV format and filter out metrics

unrelated to our study. As we aim to map the issue reports to their respective commits, we keep only a subset of the reports that represent defects and filter out all the reports whose *final resolution* is not "FIXED" and *status* not "CLOSED". Since the commit information is needed to identify the code churn, we exclude the issue reports that are not associated with any commit. Among the metrics used in our study and listed in Section 3.2.3, the number of attachments, the number of CC and the number of issues blocked do not exist in the 20-MAD dataset. Therefore, we parse issue report data found on the bug tracking systems of Mozilla[1] and Apache[2] and collect the missing metrics from the 20-MAD dataset. We end up with 151 Mozilla projects and 334 Apache projects.

Eclipse. For Eclipse, we download the issue reports belonging to five popular software projects (*e.g.*, Eclipse JDT and Eclipse Platform) from a web-based issue tracker, called Bugzilla.[3] The collected data spans 20 years, from October 2001 until February 2021. Similar to Mozilla and Apache, we only keep the reports representing defects, and that are "CLOSED" and "FIXED". Next, as shown at the bottom of Figure 3.1, we download the GitHub Repositories of the projects and extract their corresponding commit logs. Next, we download the source code repositories of the projects and their corresponding commit logs. For all the downloaded issue reports, we extract the issue ids and map the id value of each report to the commits, using the numerical id value in the commit log message. To realize the mapping, we adopt a heuristic that matches patterns in the commit messages that include the issue report number using regular expressions [72]. For example, we look for patterns such as *"Bug #1346"* or *"Fix for #6742"*. Then, we use GitPython library[4] to interact with the git repositories and extract the commit information, *i.e.*, number of modified files, number of lines of code added and removed, and commit timestamp of the associated commit. Table 3.1 summarizes the statistics of the collected issue reports of the three ecosystems.

[1]https://bugzilla.mozilla.org/home
[2]https://issues.apache.org/jira/
[3]https://bugs.eclipse.org/bugs/
[4]https://github.com/gitpython-developers/GitPython

Table 3.1: Dataset Statistics.

Ecosystem	Issue reports
Mozilla	164,190
Apache	115,571
Eclipse	18,787
Total	298,548

3.2.2 Issue report labels inconsistency

Issue tracking systems, such as, Bugzilla and Jira, have a field known as keyword and label, respectively, dedicated to tagging issue reports with a meaningful keyword, including the issue report topic. While this field could be helpful to practitioners to better understand the defect, we find that only 13% out of 298,548 issue reports in our study contain at least one keyword. We manually investigate a statistical random sample (*i.e.*, 288 reports in total, 96 from each ecosystem) from the 13% issue reports that have keywords. We notice that: keywords represent various dimensions that do not represent the defect type. For instance, for Eclipse issue reports, 64% of the keywords are irrelevant to the topics of issue reports and include keywords, such as "contributed", "noteworthy", "helpwanted", "greatfix", "bugday". For Apache, 90% are irrelevant: "pull-request-available", "windows", "easyfix", "ready-to-commit", "iOS". For Mozilla, 84% are irrelevant: "fixed1.9.1", "regression", "verified1.9.2", "reproducible".

3.2.3 Metrics collection

As shown in Figure 3.1, after we associate the issue reports to their commits, we compute the metrics. In total, we collect 11 metrics, *i.e.,* the code churn size from the associated commit and another 10 metrics from the issue reports. The 10 metrics from the issue reports constitute textual factors (*e.g.*, the title of issue reports) and characteristic factors (*e.g.*, the number of comments per developer).

Issue report textual metrics

- *The length of the issue report title:* is the count of words encompassed in the title

field of the issue report. A longer title might represent a more complex defect that might have a larger change impact.

- *The length of the issue report description:* is the count of words encompassed in the description field of the issue report. Zhang et al. [314] explain that the length of the issue report might indicate the complexity of understanding the issue report. In addition, Huang et al. [111] explain that a more extended description of the issue report provides more elaborate information about the issue. Therefore, we assume that a complex defect that has a large change impact is more likely to have a lengthy issue report description.

- *The number of comments per developer:* is the ratio of the count of comments posted prior to fixing the defect and the count of developers involved in posting the comments on an issue report. Comments are an indication of increased communication [75]. Increased communication might indicate that the defect is complex and hence has a larger change impact [273].

Issue report characteristics metrics

- *The number of CC:* is the count of distinct developers added to the list of carbon copy (CC). A developer added to the list of CC is a developer interested in the progress of the defect. A large number of CC might indicate that the defect is a bottleneck in the maintenance process [270], which can consequently suggest that the defect might have a larger change impact.

- *The number of issues blocked:* is the count of issue reports that can be fixed only after the issue report in question is resolved. The larger the number of issue reports blocked by an issue, the more likely it is that the issue has more change impact [270].

- *The number of votes:* is the count of users who would like the issue report resolved. Developers vote for an issue report if they favour fixing the defect and consider it an important issue [281]. Also, the number of votes unveils the effort invested in discussing an issue [75]. Since the number of votes can be perceived as an indication

of the importance of the issue [164], an issue of high importance might affect a larger scale of the software product, *i.e.*, several modules, and hence have a higher change impact [8]. Therefore, we assume that an issue report with a larger number of votes might have a larger change impact.

- *The number of attachments:* is the count of attachments added to the issue reports. Attachments can include testing cases, stack traces and screenshots. A larger number of attachments might be an indicator that the report may have a large change impact [31].

- *Has version:* is a boolean variable that describes whether the issue report has a version (*i.e.,* the version of the software). The tracking information is likely to affect the change impact. Related work [215] demonstrates that the version metric can influence the defect fixing time.

- *Has milestone:* is a boolean variable that describes whether the issue report has a milestone target (*i.e.,* target to fix). Similarly to *has version*, a report with more tracking-related information might affect the change impact.

- *Is severe:* is a boolean variable that describes whether the issue report is considered severe or not. We consider the defect severe (*i.e.,* value 1 is assigned) if the severity level specified in the issue report is *blocker*, *critical*, *major*, *urgent*. A more severe defect might have more change impact [196].

Code churn. A code churn is the number of modified lines of code, which is the sum of the lines of code added and removed [147, 162, 203]. Every issue report is associated with one or more commits. Thus, for every report, we extract the number of lines of code modified (*i.e.*, added or removed), which is the sum of the changed lines as the code churn.

The number of changed files. The number of changed files represents the spread of the propagation of the code change [232, 314] in the file level. We calculate the total number of files changed for every issue report as the sum of the added, deleted, and

modified files. If an issue report is associated with several commits, the total number of changed files equals the sum of the number of files changed in all the commits.

3.2.4 Data preprocessing

As depicted in Figure 3.1, we apply the same preprocessing steps to Mozilla, Apache, and Eclipse issue reports, resulting in cleaned issue reports.

Removing the noise. A manual investigation of the issue reports shows that the description field of an issue report could contain noise. For example, it is common for developers to include code fragments and log traces in the description. Therefore, we use regular expressions to identify the lines that contain noise. We exclude the code fragments by identifying the lines embedded in the `Code` or `noformat` tags by using the regular expression `(Code:).*(Code)` and `(noformat):.*(noformat)`. For the log traces, some lines start with timestamps of the forms `hh:mm:ss` and `hhmmss`. Hence, we identify the log lines using two regular expressions `^\d{2}[:]+\d{2}[:]+\d{2}.*` and `^\d{2}+\d{2}+\d{2}.*` and remove them from the descriptions of the issue reports. Some other log traces start with the expression `Caused by`. Therefore, we use another regular expression `(^Caused by)` to exclude them. In addition, we remove the hyperlinks and the automatically generated code related to automated tests using the following respective regular expressions:
`(https|http).*?[\t\s\n]` and `^(TEST-INFO|TEST-START|Build ID:|User Agent:)`.

Normalizing the text. We apply the following text normalization steps on the title and description of an issue report. We remove English non-description stop words (*e.g.*, "the", "was" and "of") using the `nltk` package [32] that contains a defined corpus of English stop words. We also apply text tokenization to exclude punctuation and special symbols. Lastly, we bring the words to their ground forms by converting them to their stemmed and lemmatized versions. This step is essential for Topic Modeling as it maps the words "connections", "connecting" and "connected" to their basic form "connect" [174] which reduces the vocabulary size.

3.2.5 Embedded Topic Model (ETM)

In 2003, Bengio et al. [26] introduce the concept of word embeddings, a distributed learned representation for the text that represents words with similar meanings in close proximity in a vector space. Word embedding plays a vital role in the realm of natural language processing. In particular, ETM adopts the continuous bag-of-words (CBOW) [183] word embeddings. Given a corpus of documents D with V unique words, let represent the n^{th} word in the d^{th} document. The CBOW likelihood of the word w_{dn} is:

$$w_{dn} \sim softmax(\rho^T \alpha_{dn}) \tag{3.1}$$

where α_{dn} represents the transpose of the embedding matrix that contains the embedding representations of the vocabulary. α_{dn} represents the context embedding which is the sum of the vectors of the words surrounding w_{dn}.

In 2003, Latent Dirichlet Allocation (LDA) [34], a statistical model that generates topics, is developed. LDA considers that each topic is characterized by a full distribution over the vocabulary of a corpus. Each document is represented by a unique mixture of topics. However, despite its popularity, LDA suffers in learning interpretable topics when the vocabulary size becomes immense. To overcome the limitation mentioned above for large vocabulary and obtain good quality of topics, practitioners should omit words to reduce the vocabulary size. However, pruning the vocabulary could also threaten the quality of a model. To mitigate the limitations of LDA, Dieng et al. [65] propose ETM. ETM has been shown to outperform LDA by being robust to even large vocabularies.

ETM is a technique that combines properties from both topic modelling and word embedding. First, it relies on the topic model to identify interpretable latent semantics of the corpus. Second, it leverages word embedding to efficiently represent the meaning of the words in the vector space. Similar to LDA, ETM is a generative probabilistic model that represents each document as a probability of topics. Compared to LDA, ETM represents not only the words using embedding vectors but also the topics. For instance, in LDA, the k^{th} topic is a distribution over all the words in a vocabulary. In

contrast, ETM represents the k^{th} topic as an embedding vector, *i.e.,* α_k, in the embedding space. Also, ETM presents an improvement over LDA in the process of reconstructing the words from an assigned topic. It relies on the topic embedding and the embeddings of the vocabulary to assign words to each topic using the CBOW likelihood. However, in ETM, the context embedding is selected from the document context instead of the surrounding words, as in standard CBOW.

There are two parameters in ETM: the word embedding ρ and the topic embeddings α. In the fitting process, ETM is trained to maximize the marginal likelihood of the documents as follows:

$$\mathcal{L}(\alpha, \rho) = \sum_{d=1}^{D} \log p(w_d|\alpha, \rho) \tag{3.2}$$

where for a given corpus of documents $\{w_1,...,w_D\}$, w_d is a collection of N_d words. Thus, the word embedding and the topics are found concurrently by ETM. However, the word embeddings can be prefitted. In that case, the topics can be identified in a specific embedding space.

Similar to LDA, ETM represents each document as a probability of topics, and each topic is a distribution over words. ETM will assign each document one topic.

3.3 Experimental Results

In this section, we evaluate the feasibility of clustering issue reports into topics and the viability of leveraging the topics of issue reports in predicting the change impact. Precisely, we discuss motivation, approach and findings for our research questions.

3.3.1 RQ3.1: Can we accurately assign topics to issue reports of software systems using ETM?

Motivation. In practice, developers manually investigate the issue reports to complete development activities such as defect assignment and prioritization [14, 224, 300]. Issue reports contain rich information that can help developers understand the nature of the defects and, therefore, save development effort and time. Lili et al. [155] demonstrate

that defects of the same categories tend to have the same trend of fine-grained change operations, *e.g.,* if statement, while statement, assignment statement and function call statement, and the same frequency of fine-grained change operation use. The collective knowledge in issue reports, *i.e.*, topics, can guide developers in the defect fixing activities and in predicting the change impact of fixing a defect. As depicted in Section 3.2.2, most issue reports do not contain keywords representing the topics of issue reports. Therefore, in this RQ, we leverage the state-of-the-art topic model, ETM, to automatically discover the hidden common topics across the issue reports. Having a high accuracy in identifying the topics provides valuable data to the predictor models to estimate the change impact. We also study the relationship between the identified topics and their severity.

Approach. Since the effectiveness of our proposed issue report classification approach relies on the accuracy of the adopted embedding-based topic model, we additionally conduct a quantitative evaluation of the effectiveness of ETM in extracting topics associated with every issue report in our dataset. We use ETM implementation[5] provided by its authors. Our approach consists of three steps.

Step 1: Non-textual data processing. To obtain optimal results, we use ETM authors' script[6] to filter out words with a *maximum document frequency* above 70% and remove low-frequency words appearing in only a few documents, referred to as the *minimum document frequency*. Setting a *minimum document frequency* to remove the low-frequency words helps eliminate the rare words that are not important to the topic model. It also reduces vocabulary size [65] and leads to better computing time. After varying the value for the minimum document frequency by increasing starting and 1, we set it to 15 as for the values higher than 15, only a small amount of improvement in the vocabulary pruning and the computing time can be achieved. Hence, we exclude the words that appear in less than 15 documents from the vocabulary. Then, we exclude the reports that have a combined length for description and summary of fewer than three words as previous studies find that short textual information rarely conveys meaningful information [21, 43, 149].

[5]https://github.com/adjidieng/ETM
[6]https://github.com/adjidieng/ETM/tree/master/scripts

Step 2: Topic modeling hyperparameter tuning. Selecting the optimal number of topics plays a pivotal role in the quality of the topic modelling results. We rely on the topic coherence (*i.e.,* the interpretability of a topic [184]) and topic diversity (*i.e.,* unique words in the top 25 words of all topics as defined by the ETM authors) to quantitatively measure the quality of the model in respect to the selected topic number. Additionally, we rely on the human judgement that aligns with the quantitative metrics [260]. We run the model with different numbers of topics by increasing and decreasing the value of the topic number and finally set it to 15 as it provides the best quality of topics. We also set the number of epochs to 300.

Step 3: Classification accuracy. After applying ETM on the issue reports of all three ecosystems, we obtain 15 clusters of keywords, each representing an issue report topic. The author of this thesis and another collaborator followed an open coding approach [235,241] to manually and independently assign labels to the clusters generated by ETM. To evaluate if ETM can accurately assign topics to issue reports, we select for each ecosystem a statistically representative random sample of issue reports with a confidence level of 95% and a confidence interval of 10%. In total, we randomly select 288 issue reports that belong to Mozilla, Apache and Eclipse. We perform the below three steps to evaluate the performance of the approach:

1. The first and the third authors independently assign one topic from the 15 topics obtained by ETM to each of the 288 sample reports.

2. The Cohen's kappa agreement score [176] is calculated on the annotated testing issue reports using the "irr" package[7] provided in R[8]. We achieve a score of 0.78, which indicates a substantial level of agreement. Next, the annotators resolve the disagreements after discussing the conflicts case by case. Finally, all the issue reports were assigned to one issue report topic.

3. We calculate the final accuracy as the percentage of true positive (TP), where a TP represents a topic assigned correctly by ETM that matches the topic assigned

[7]https://cran.r-project.org/web/packages/irr/index.html
[8]https://www.r-project.org/

by an annotator.

The three ecosystems adopt different severity level schemas, *e.g.*, Apache adopts an 8-level severity schema, whereas Eclipse adopts a 5-level schema. Therefore, to conduct the severity analysis across the different topics, we map the various severity levels of the three ecosystems to a three-level severity schema, *i.e.*, *low*, *medium* and *high*, introduced by Thung et al. [267]. Table 3.2 exhibits the three-level severity schema mapping. Then, we conduct the following statistical tests.

1. To check if the severity is significantly different among the topics of issue reports, we conduct a comparison using the Kruskal-Wallis test [143], a non-parametric statistical test used to compare more than two samples of data. If we obtain a *p-value <= 0.05*, we reject the null hypothesis and conclude that not all the topics of issue reports have the same median severity.

2. If the null hypothesis is rejected, we investigate if the differences among the topics are of strong significance by calculating the epsilon squared effect size [307]. The effect size represents the relationship of the variables, such as the topic of issue reports and severity, on a numeric scale. A value close to zero indicates a negligible effect, whereas a value close to 1 indicates a very strong effect. We refer to Table 3.3 to identify the effect size.

3. To further differentiate the topics of issue reports with different severity levels without ambiguity, we conduct the Scott-Knott Effect Size Difference (SK-ESD) [263, 264]. The SK-ESD uses hierarchical clustering to compare the means in the dataset to form statistically distinct groups. The SK-ESD clusters the topics of issue reports in a way where the intra-group difference in severity level is negligible, and the inter-group difference is non-negligible.

Results. Our approach identifies 15 topics of issue reports commonly existing in the three ecosystems but with different distributions. Table 3.4 provides a closer look at the topics obtained by ETM along with the top 10 keywords. We observe

Table 3.2: Three-level severity schema mapping.

Severity level	Mapping
trivial, minor, low, and normal	Low
major	Medium
blocker, critical, and urgent	High

Table 3.3: Epsilon square effect size interpretation reference.

Epsilon squared	Effect size
0.00 < 0.01	Negligible
0.01 < 0.04	Weak
0.04 < 0.16	Moderate
0.16 < 0.36	Relatively strong
0.36 < 0.64	Strong
0.64 < 1.00	Very Strong

that some of the topics are considered common topics equally present in the three ecosystems. A few topics are system specific and are widely present in a specific ecosystem compared to the others. For example, as we can see in Table 3.5, *GUI* and *User experience* are predominantly present in Mozilla and Eclipse and represent combined more than 30% of the topics of issue reports as opposed to 4% for Apache. This is intuitive since the browser and IDE domains heavily rely on user interaction and navigation through a graphical interface. On the other hand, *Server issues*, *Security*, and *Database* exceed 50% of the topics of issue reports in Apache, which is common for web servers. *Performance*, for instance, is uniquely dominant for Mozilla, which can be explained by the importance of performance for web accessibility. Also, we observe that *Platform compatibility*, *Testing*, *Interprocess communication*, and *Release and update* have similar distribution across the three ecosystems. In fact, these topics are expected to be present in almost any kind of software.

Our approach achieves an accuracy of 83%, 72% and 77% for Eclipse, Mozilla and Apache. As shown in Table 3.6, our approach achieves high accuracy across the three ecosystems with an average of 79%.

Defects of the different topics have different median severity levels for the three ecosystems. Table 3.7 shows that the Kruskal-Wallis test's p-value <= 0.05

Table 3.4: Inferred fifteen issue report topics with their representative keywords and an example of issue report

Topic	Keywords	Sample report title
Platform compatibility	'app', 'devic', 'sync', 'web', 'android'	Remove usage of nsIDOMWindowUtils.goOnline() in mobile's netError.xhtml
Testing	'test', 'fail', 'run', 'unit', 'expect'	Update .hgignore to ignore Loop unit test files
User experience	'open', 'page', 'step', 'tab', 'window'	Dialog opens up off the screen
File management	'file', 'packag', 'depend', 'instal', 'path'	Export bundle should create the description file
Build and deployment	'build', 'warn', 'compil', 'make', 'consol'	javaCompiler*.args generated just once per session
API related issues	'instanc', 'class', 'method', 'interfac', 'context'	SWIG interface doesn't support CASException thrown from some common APIs
Security	'user', 'password', 'client', 'group', 'permiss'	Password reset issues tokens w/ "&" in them, URL not escaped
Release and Update	'updat', 'version', 'releas', 'patch', 'branch'	Add license header to RELEASE_NOTES
Performance	'cach', 'memori', 'size', 'buffer', 'alloc'	Portable spark: thread/memory leak in local mode
Database	'tabl', 'data', 'queri', 'schema', 'record'	Delete table does not remove the table directory in the FS
Parallel event processing	'event', 'frame', 'process', 'thread', 'call'	Convert formSubmitListener.js to a process script instead of a frame script
General program related anomaly	'use', 'implement', 'code', 'want', 'work'	Implement automatic bookmarks backup for 1.1
GUI	'background', 'imag', 'font', 'style', 'display'	Scrollbar handle is not colored correctly when selected and dragged in gtk3
Server issues	'thread', 'run', 'connect', 'session', 'node'	Secondary socket of "tee" socket is not threadsafe
Interprocess communication (IPC)	'url', 'request', 'input', 'header', 'pars'	Olingo2's batch process generates the invalid request

for Mozilla, Apache and Eclipse. Moreover, we observe that for Mozilla, the Epsilon squared effect size is moderate and weak for Apache and Eclipse. Table 3.8 shows the different severity groups obtained by the Scott-Knott test. We notice that *GUI* and *IPC* constantly belong to the groups representing the low severity across the three ecosystems. Figures 3.2, 3.3 and 3.4 show the distribution of severity levels across the different topics of issue reports for Mozilla, Apache and Eclipse, respectively.

By observing these figures, we come up with two findings:

Table 3.5: The extracted topics of issue reports percentages distribution across the three ecosystems

Topics	**Apache(%)**	**Mozilla(%)**	**Eclipse(%)**
Platform compatibility	1	11	1
Testing	6	7	3
User experience	3	14	20
File management	12	2	21
Build and deployment	3	10	5
API related issues	12	2	14
Security	13	2	1
Release and Update	4	8	9
Performance	2	10	1
Database	15	2	2
Parallel event processing (PEP)	1	8	1
General program anomaly (GPA)	2	6	5
GUI	2	12	10
Server issues	16	2	3
Interprocess communication (IPC)	8	4	4

Table 3.6: Accuracy of ETM calculated on 288 manual labelled issue reports.

Ecosystem	**Manually labelled issue reports**	**Accuracy**
Mozilla	96	76%
Apache	96	77%
Eclipse	96	83%
Average	288	79%

1. We notice that different topics of issue reports have different severity levels within the same ecosystem. In Mozilla and Eclipse, for example, the number of defects with high severity belonging to the *Parallel event* topic is at least double the number of high-severity defects in other topics. For Apache, more than 20% of the *Security* and *Server* defects are highly severe, whereas less than 10% are severe for *API* and *GUI* topics.

2. The same topics of issue reports have different severity levels distribution across the studied ecosystems. When considering Apache, we notice that *Security* and *Server* topics have the highest proportion of highly severe defects. This could be explained by the fact that Apache applications fall under the web server domain, in which reliability is crucial. As for Eclipse and Mozilla being IDE and client/browser applications respectively, *Parallel event* defects that represent processes and threads

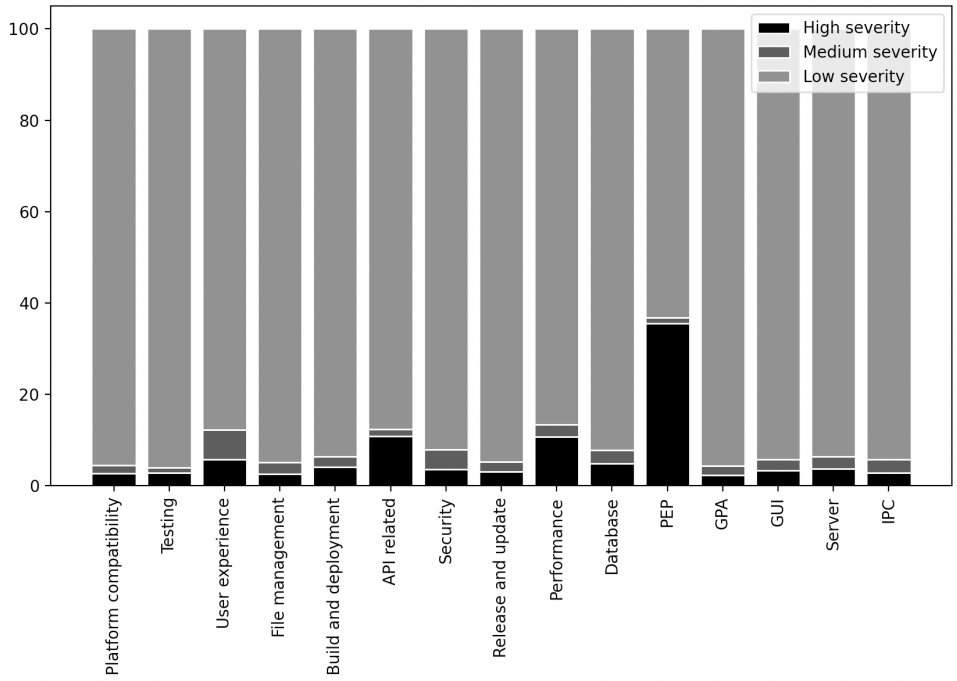

Figure 3.2: The distribution of severity levels for Mozilla issue reports.

are more severe than others.

Table 3.7: Kruskal-Wallis p-value and Epsilon squared effect size

Ecosystem	p-value	Epsilon squared	Effect size
Mozilla	0.000000e+00	0.088	Moderate
Apache	0.000000e+00	0.024	Weak
Eclipse	3.004759e-41	0.012	Weak

Summary of RQ3.1

ETM achieves a promising average accuracy of 79% on 288 manual annotated reviews. The extracted topics of issue reports can support the developers in better understanding the nature of the defect.

3.3.2 RQ3.2: Can we predict the change impact of resolving defects in terms of code churn size? What are the most influential metrics for predicting the change impact in terms of code churn?

Motivation. Given the time constraints and limited resources, during defects assignment, practitioners, *e.g.,* developers and project managers, estimate the defect fixing

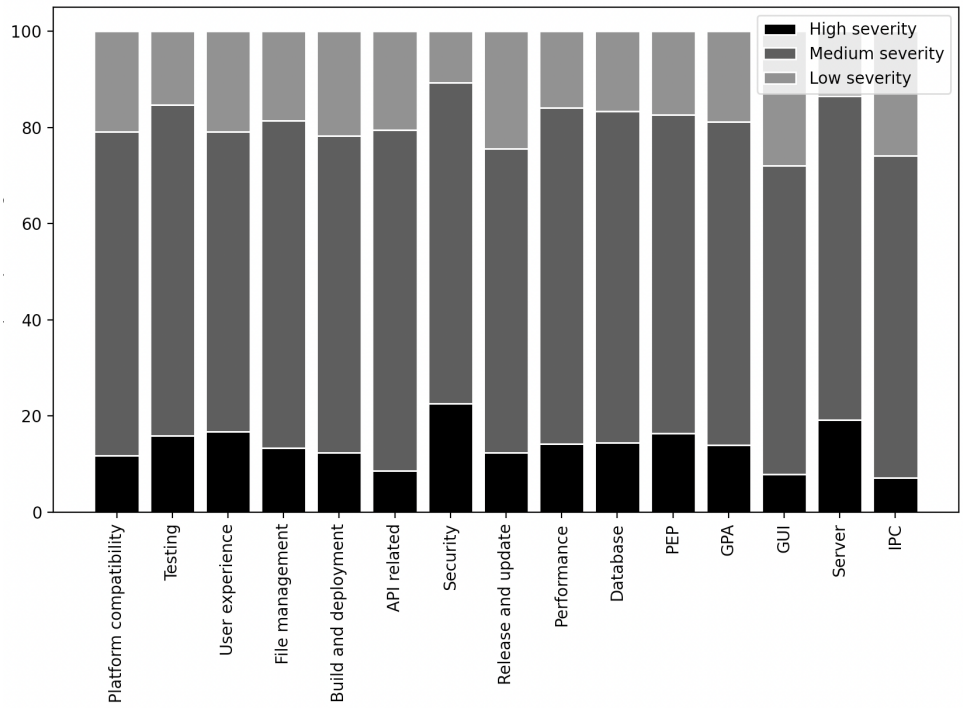

Figure 3.3: The distribution of severity levels for Apache issue reports.

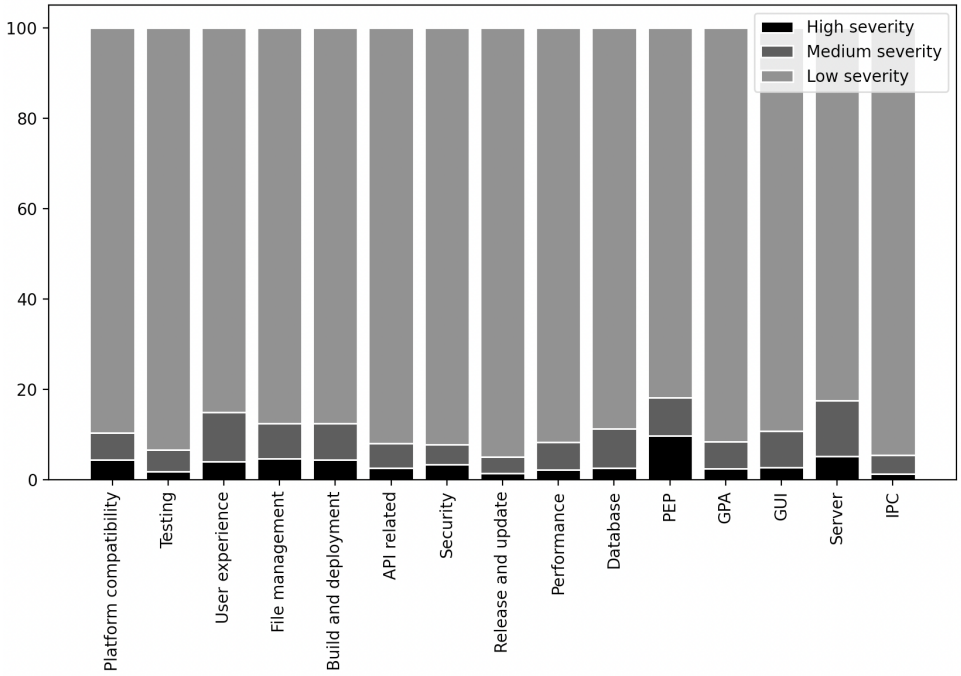

Figure 3.4: The distribution of severity levels for Eclipse issue reports.

Table 3.8: The severity group ranking obtained by SK-ESD.

Mozilla		Apache		Eclipse	
Group	**Topic**	**Group**	**Topic**	**Group**	**Topic**
G1	Parallel event	G1	Security	G1	Parallel event
G2	Performance	G2	Server	G2	User experience
	API related	G3	Testing		Server
G3	User experience		Parallel event	G2	File management
G4	Database		Performance		Build and deployment
	Security		Database	G3	Platform compatibility
	Build and deployment	G4	User experience		Database
	Server		GPR		GUI
G5	GUI		File management		Security
	IPC	G5	Platform compatibility		GPR
	Release and update		Build and deployment		API related
	File management		API related		Performance
	Platform compatibility		Release and update		Testing
	Testing	G6	IPC		IPC
	GPR		GUI		Release and update

effort before assigning the defect to a developer. When prioritizing defects, practitioners take into consideration the effort level required to fix the defect [135]. The developer effort can be measured by the size of the required change, *i.e.,* lines of code modified that is the code churn [36]. In this RQ, we want to predict the change impact of defects in terms of code churn size. We do not only use information extracted from issue reports but also the topics of issue reports extracted in RQ3.1. In addition, we identify the most influential metrics for predicting the change impact. This knowledge can help developers better understand the prediction results and guide them on which metric they should focus on to estimate the change impact.

Approach. Our goal is to predict the amount of change needed to fix the defect in terms of code churn using only the issue report information. Thus, we formulate the dependent variable of our prediction model (a.k.a., the prediction output Y) to be a boolean variable representing whether the issue report requires a large or a small change impact. If the report is classified to require a large change impact the prediction output Y = 1.

Dependent variable. In this RQ, we define the dependent variable as the change impact of fixing a defect measured in terms of code churn size (*i.e., small* and *large*). We apply two different steps to achieve the report classification to change impact. First, we apply the log transformation to the numerical code churn value collected from the commit to correct the skewness in the data. Second, we sort the issue reports by the

increasing order of its code churn size and select the lower 10% as issue reports requiring *small* change impact and the upper 10% as the ones requiring a *large* change impact. To obtain the dependent variable, we assign the value of 1 for the reports with *large* change impact and 0 otherwise.

Independent variables. In total, we have 25 independent variables. While 10 of these metrics are directly extracted from issue reports as explained in Section 3.2.3, we synthesize another 15 metrics that represent the topics of issue reports. In RQ3.1, we extracted 15 different topics. To leverage these extracted topics, we created 15 different boolean independent variables: *Is platform*, *Is testing*, *Is user experience*, *Is file management*, *Is build & deployment*, *Is API*, *Is Security*, *Is Release*, *Is Performance*, *Is Database*, *Is Parallel event*, *Is GPR*, *Is GUI*, *Is Server*, *Is IPC*. Each of the aforementioned variables represents an issue report topic for the collected report. As explained in Section 3.3.1, each report belongs to only one of the 15 topics. Therefore, for every report, only one of the 15 independent variables has the value 1 and the others is assigned 0.

Correlation and redundancy analysis. The presence of correlated metrics might affect the performance of the model [128]. Therefore, we apply *varclus*[9] function in R to detect the existence of highly correlated metrics in our dataset. We consider any pair of metrics that achieves a coefficient of 0.7 [179] and higher as highly correlated. Figure 3.5 illustrates the Spearman correlation of the metrics of Apache projects. We only present the correlation of one ecosystem due to space limitations. Mozilla and Eclipse correlation analyses give similar results, *i.e.,* the absence of highly correlated features; therefore, we keep all the metrics and exclude none.

Prediction models. We train eight different machine learning models to automatically classify the issue reports based on the *small* and *large* change impact they require. We use the following models that are widely utilized in the binary prediction and in the realm of Software Engineering [75, 111, 303]: *Logistic regression*, *Naive Bayes*, *SVM*, *Random Forest*, *XGboost*, *Catboost*, *LGBM* and *Multi-layer Perceptron*. All the ML models are implemented using the scikit-learn[10] library in Python. We adopt the 10-fold

[9]https://search.r-project.org/CRAN/refmans/Hmisc/html/varclus.html
[10]https://scikit-learn.org/

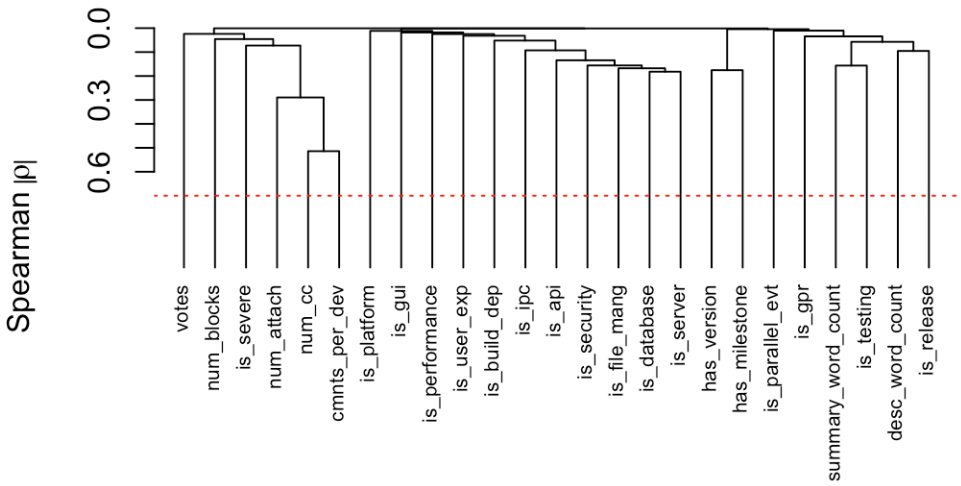

Figure 3.5: The hierarchical clustering of independent variables for Apache. The dotted red line represents the threshold value of 0.7.

cross-validation approach to validate the models and ensure reliable performance. To configure the machine learning models, we use two automated parameter optimization techniques Random Search and Grid Search. Grid Search is a brute-force approach that finds the best hyperparameters for the machine learning model [28]. However, computing all possible combinations of parameters is time-consuming. Therefore, to tune our machine learning models efficiently, we first use RandomSearch, which is capable of testing a random wide range of parameters very fast. After obtaining the best values for every parameter through Random search, we adopt Grid search on a smaller search space of parameters. Grid Search identifies the best combination of the parameters after applying the cross-validation score. To achieve the best performance for our models, we adopt the RandomizedSearchCV and the GridSearchCV from sk-learn to tune the hyperparameters of models.

To provide a robust model evaluation and to imitate the real-life settings, we adopt a time-based train and test data splitting. Instead of randomly splitting the data into 80% training and 20% testing, we sort the issue report by their creation date, select the most recent 20% of reports as testing, and leave the rest for training. The Grid Search is performed on the training set with cross-validation. Once the optimal hyperparameters are obtained, we evaluate the model on the test set. We use the 25 metrics described above as dependent variables for all the models.

Evaluation metrics. To quantify the performance of the predictive models, we consider precision, recall, and F_1-Score (*i.e.,* the harmonic combination of precision and recall) as the evaluation metrics. Equations 6.3, 6.4, and 6.5 show the computation for precision, recall, and F_1-Score.

$$Precision = \frac{TP}{TP + FP} \tag{3.3}$$

$$Recall = \frac{TP}{TP + FN} \tag{3.4}$$

$$F_1\text{-}Score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3.5}$$

We also use Area Under the Receiver Operating Characteristic Curve (AUC) [158] to evaluate the effectiveness of our machine learning models. The AUC values range from 0 to 1. The performance of a prediction model is considered promising if the AUC-ROC is 0.7 and above [90, 206] and 1 denotes a perfect predictive power.

Sensitivity Analysis. Our prediction approach has one hyper-parameter, the small-large code churn change impact threshold. In this section, we select the prediction model that can achieve the best performance to conduct the sensitivity analysis. Then, we experiment with three additional thresholds, *i.e., 25%*, *40%*, and *50%*. The lower 25% reports are considered as defects with *small* change impact and the upper 25% as *large*. A similar classification applies to the *40%*, and *50%* thresholds.

Feature importance. To find the most influential metrics, first, we investigate if the 15 metrics, *i.e.,*, topics of issue reports, synthesized from the results obtained in RQ3.1 improve the prediction model. We construct a baseline model derived from the best performing model and best small-large code churn threshold that considers only 10 metrics, *i.e.,* 3 issue report textual metrics, 7 issue reports characteristic metrics. We exclude the 15 issue report topics metrics. We also conduct a complementary study that seeks to identify the most influential metrics among the 25 metrics for identifying the change impact. Second, since every ecosystem is made of several software projects, *e.g.,*

HIVE, Firefox and JDT are three popular software projects belonging to Apache, Mozilla and Eclipse respectively, we create individual prediction models per software project. To avoid working on toy software projects, we randomly select 4 software projects from each ecosystem having at least 2,000 issue reports [250].

We employ the permutation feature importance[11] from sk-learn to detect the influence, a.k.a. importance, of every metric in our model. In the permutation feature technique, the metric importance (*i.e.,* feature importance) is calculated by considering the drop in the model performance score when the values of the metric in question are randomly shuffled [222]. Shuffling the values of the metric leads to breakage between the metric and the dependent variable and, therefore, indicates to what extent the model depends on this metric. As done in previous work [111], to attain a reliable result, we apply the permutation test on all 25 metrics repeatedly 10 times. To statistically identify the magnitude of the difference between the importance score of the metrics, we compute the SK-ESD test. The SK-ESD clusters the 25 metrics into groups, *i.e.,* ranks, based on their importance score.

Results. It is feasible to predict the issue reports requiring large change impact in terms of code churn based on the information of the issue reports and the topics of issue reports. Table 3.9 shows the performance in terms of precision, recall, F1-Score, and AUC of the eight constructed prediction models. As we can notice, XGBoost achieves the best performance. For instance, the XGBoost model achieves AUC values of 0.84 for Mozilla, 0.76 for Apache, and 0.74 for Eclipse ecosystems. There exists a minuscule difference between the AUC values of the gradient boosting tree-based models (*i.e.,* XGboost, Catboost, LGBM) and Multi-layer Perceptron model AUCs. This can be explained by the fact that all of these models are very efficient in interpreting the complex relationships in our issue reports tabular data [136].

Our approach is not sensitive to the threshold of the selection of the code churn classes (*i.e.*, *small* and *large*). We select XGBoost, the best performing model, to conduct the sensitivity analysis. Table 3.10 show that for Mozilla, the AUC drops by

[11]https://scikit-learn.org/stable/modules/permutation_importance.html

Table 3.9: The performance of our prediction models for the considered datasets. The values shown represent the prediction of the issue reports that require a large change impact in terms of code churn size. Prec. represents the precision.

	Ecosystem	**Prec.**	**Recall**	**F-score**	**AUC**
Logistic Regression	Mozilla	0.93	0.64	0.76	0.80
	Apache	0.74	0.65	0.69	0.71
	Eclipse	0.77	0.72	0.74	0.70
Naive Bayes	Mozilla	0.86	0.44	0.58	0.68
	Apache	**0.80**	0.54	0.64	0.71
	Eclipse	0.77	0.47	0.58	0.63
SVM	Mozilla	**0.94**	0.62	0.74	0.79
	Apache	0.74	0.65	0.69	0.72
	Eclipse	**0.78**	0.71	0.74	0.70
Random Forest	Mozilla	0.93	0.73	0.81	0.83
	Apache	0.75	0.68	0.71	0.74
	Eclipse	0.74	0.74	0.74	0.68
XGboost	Mozilla	0.93	0.72	0.81	**0.84**
	Apache	0.74	0.68	**0.71**	**0.76**
	Eclipse	0.76	0.77	0.76	**0.73**
Catboost	Mozilla	0.93	0.73	**0.82**	0.83
	Apache	0.74	0.68	0.71	0.74
	Eclipse	0.75	0.75	0.75	0.71
LGBM	Mozilla	0.93	0.73	0.82	0.83
	Apache	0.74	0.69	0.71	0.73
	Eclipse	0.71	0.81	0.76	0.67
Multi-layer Perceptron	Mozilla	0.92	**0.74**	0.82	0.83
	Apache	0.72	**0.69**	0.71	0.72
	Eclipse	0.71	0.81	0.76	0.66

Table 3.10: AUC performance of the three ecosystems with three different thresholds for the code churn size.

	XGBoost-25%	**XGBoost-40%**	**XGBoost-50%**
Mozilla	0.78	0.73	0.70
Apache	0.67	0.64	0.61
Eclipse	0.67	0.63	0.60

8% to 16% when the threshold is 25% (*i.e., XGBoost-25%*) and 50% (*i.e., XGBoost-50%*) respectively. Similarly, for Apache, a drop of 7% to 13% when the threshold is 25% and 50% respectively. As for Eclipse, the AUC drops by 6% to 13% when the threshold is 25% and 50%, respectively.

The model fit on the metrics, including the topics of issue reports, improves

the base model's AUC by 5%. The model fit on all the metrics, including the topics of issue reports metrics extracted in RQ3.1, achieves an AUC value of 0.78 on average (AUC 0.84 for Mozilla, AUC 0.76 for Apache and AUC 0.73 for Eclipse) across the three ecosystems for classifying the issue report change impact. This obtained AUC of 0.78 outperforms the AUC of the based model *XGBoost-10-base* fit on the textual and characteristic metrics only, present in the issue report, that achieves an average of 0.73 (AUC 0.80 for Mozilla, AUC 0.71 for Apache and AUC 0.67 for Eclipse). This result suggests that the topics of issue reports can support developers in predicting the change impact of fixing a defect. For instance, we observe in Table 3.11, that *Is GUI* and *Is User experience* are among the top 5 influential metrics for Eclipse and *Is database* ranks 6th for Apache.

The number of attachments and the number of comments per developer are among the top 3 influential metrics across the three ecosystems. Table 3.11 depicts the top 6 influential metrics along their importance scores and ranks. The importance score is the average importance score of the corresponding metrics. **As we can notice, the top 6 metrics are ranked differently across the systems. However, the number of attachments and the number of comments per developer are among the top 3.** This suggests that issue reports with more attachments and developers in their discussions have a larger change impact. The presence of attachments, including test cases, may indicate the complexity of a defect, thus leading to a larger change impact. Similarly, a higher number of comments per developer in an issue may also be associated with a more complex defect that is harder to solve.

We observe that the *number of issues blocked* ranks in second place for Mozilla. The association between the number of issues blocked and the change impact can be attributed to the fact that a defect that blocks several other defects may be used in several packages or in several modules, which may have a larger change impact. In fact, Valdivia-Garcia et al. [270] quantified the effect caused by blocking defects and found that blocking defects require between 1.2–4.7 more lines of code changes than non-blocking defects.

The severity of the issue report, which represents the importance of the issue report, is also an important metric for Apache projects. This result hints that severe defects may be treated meticulously where additional coding practices are implemented, thus leading to a larger change impact.

Table 3.11: The top 6 most influential metrics in the XGBoost-10% model for Mozilla, Apache and Eclipse ranked by their importance. Imp. represents the importance score obtained by the feature permutation approach. The Rank column represents the clusters raking obtained by SK-ESD.

	Metric	**Imp.**	**Rank**
Mozilla	# of attachments	0.1424	1
	# of issues blocked	0.0271	2
	# of comments per developer	0.0269	3
	Has milestone	0.0114	4
	Title of issue report	0.0093	5
	Has version	0.0089	6
Apache	# of comments per developer	0.0559	1
	# of attachments	0.0323	2
	Is severe	0.0195	3
	Description of issue report	0.0193	4
	# of CC	0.0092	5
	Is database	0.0066	6
Eclipse	# of comments per developer	0.1086	1
	# of issues blocked	0.0365	2
	# of attachments	0.0107	3
	Is GUI	0.0090	4
	Is User experience	0.0072	5
	Description of issue report	0.0063	5

Similar to the ecosystem-level prediction, the performance of the software project-level change impact prediction model is improved when the issue report topics metrics are included. We perform a per software project prediction for the software projects of the sampled ecosystems shown in Table 3.12 using XGBoost-10%. Table 3.12 shows that the per-software project prediction model performance is improved by up to 5% in terms of AUC when the topics of issue reports are fed to the machine learning model. Including the topics of issue reports metrics improves the recall by up to 11% (*i.e.,* JDT and AMBARI). The results suggest that the topics of issue reports have a positive impact on the prediction model.

Table 3.12: The performance measures of our XGBoost-10 prediction models per software project. The values shown represent the prediction of the issue reports that require a large change impact.

	Software project	Precision	Recall	F-score	AUC
Mozilla	Core	0.91	0.77	0.84	0.86
	Core (without defect types)	0.90	0.76	0.82	0.85
	Firefox	0.87	0.51	0.65	0.72
	Firefox (without defect types)	0.86	0.55	0.67	0.72
	Firefox OS	0.98	0.79	0.88	0.87
	Firefox OS (without defect types)	0.97	0.78	0.86	0.85
	Toolkit	0.90	0.64	0.75	0.80
	Toolkit (without defect types)	0.89	0.63	0.75	0.68
Apache	Ambari	0.60	0.78	0.68	0.61
	Ambari (without defect types)	0.71	0.33	0.45	0.60
	Hbase	0.86	0.78	0.81	0.83
	Hbase (without defect types)	0.85	0.78	0.81	0.82
	Hive	0.75	0.90	0.82	0.82
	Hive (without defect types)	0.69	0.94	0.79	0.78
	Spark	0.67	0.86	0.75	0.78
	Spark (without defect types)	0.58	0.76	0.65	0.69
Eclipse	Platform	0.73	0.67	0.70	0.72
	Platform (w/o defect types)	0.75	0.58	0.65	0.69
	JDT	0.76	0.87	0.81	0.83
	JDT (w/o defect types)	0.78	0.76	0.77	0.79
	PDE OS	0.71	0.51	0.59	0.66
	PDE OS (w/o defect types)	0.66	0.44	0.53	0.62
	Equinox	0.65	0.61	0.63	0.69
	Equinox (w/o defect types)	0.72	0.59	0.65	0.71

Summary of RQ3.2

Our results propose that machine learning models such as XGBoost have the potential to predict the change impact in terms of code churn size with a high AUC of 0.84, 0.76 and 0.73 for Mozilla, Apache and Eclipse. The topics of issue reports could be leveraged to achieve higher accuracy for predicting the change impact for the three ecosystems. To benefit from the change impact prediction models, we suggest to the reporters to attach relevant documents to the issue reports, list the issues blocked by the defect in question and indicate the accurate severity of the defect.

3.3.3 RQ3.3: Can we predict the change impact of resolving defects in terms of the number of files changed?

Motivation. The number of files changed represents the amount of effort required to fix a defect [233]. Previous work demonstrates that the code churn size and the number of files changed in a defect fix are not highly correlated [100]. The amount of changed files is an indication of the propagation of a code change. Some commits impact many files and require a change in several locations, whereas others require a local change in a single function of one file. Furthermore, certain defect types might present fewer or more dependencies between files. Therefore, we predict the change impact in terms of the number of files changed in this RQ.

Approach. In this RQ, our goal is to predict the change impact in terms of the number of files changed. We follow the similar approach adopted in RQ3.2. We treat the problem as a binary classification. We assign 1 to the prediction output Y if the required change impact is large and 0 if small.

Dependent variable. The amount of files, *i.e., small* and *large*, needed to fix the defect is considered the output of the prediction. To obtain the dependent variable, we sort the issue reports by the number of changed files in ascending order. We assign the lower 10% issue reports a value of 0 to the change impact, meaning that it requires a small change impact. We assign to the upper 10% a value of 1.

Independent variables and prediction models. We use the same 25 independent variables (*i.e.,* textual metrics, characteristic metrics and topics of issue reports metrics) used in RQ3.2 as independent variables. We split the data into training and testing following the time-based approach. We train the eight machine learning models mentioned in the approach of RQ3.2, and we follow the same hyperparameter pipeline (*i.e.,* Random Search, Grid Search, and the 10-fold cross-validation) to attain the best model performance. To evaluate the models, we refer to the precision, recall, F1Score and AUC performance metrics.

Sensitivity Analysis. We run the best performant model, with different thresholds for the small-large binning of the number of files changed. We predict the change impact

using three thresholds *25%*, *40%*, and *50%*.

Results. It is possible to successfully predict the change impact in terms of the number of changed files by leveraging the issue reports information. In Table 3.13, we report the precision, recall, F1-Score, and AUC of the most performant model, XGboost. XGboost achieves an AUC of 0.71, 0.82, and 0.73 for Apache, Mozilla and Eclipse respectively.

Our approach is not sensitive to the threshold of the selection of the *small* and *large* the number of files changed. We observe in Table 3.14 that the AUC drops when the upper and lower threshold increases from 25% to 50% by 7%, 5% and 6% for Mozilla, Apache and Eclipse, respectively.

Table 3.13: The performance of our prediction models for the considered datasets. The values shown represent the prediction of the issue reports that require a larger change impact in terms of the number of files changed. Prec. represents the precision.

	Ecosystem	**Prec.**	**Recall**	**F-score**	**AUC**
XGboost	Mozilla	0.92	0.71	0.80	0.82
	Apache	0.72	0.62	0.66	0.71
	Eclipse	0.84	0.74	0.79	0.73

Table 3.14: AUC performance of the three ecosystems with three different thresholds for the number of files changed.

Ecosystem	**XGBoost-25%**	**XGBoost-40%**	**XGBoost-50%**
Mozilla	0.75	0.71	0.68
Apache	0.66	0.64	0.61
Eclipse	0.67	0.64	0.61

Summary of RQ3.3

It is feasible to predict the change impact of defects in terms of the number of files changed. XGboost achieves an AUC between 0.71, 0.73 and 0.82 for Apache, Eclipse and Mozilla respectively.

3.4 Implications

We discuss in this section the possible implications of our findings that could be useful to practitioners and researchers.

Leveraging the topics of issue reports. Identifying the topic of an issue report could help the researchers and practitioners in many ways:

- Researchers could benefit from the topics of issue reports to improve the defect fixing process. Some topics are easier to be fixed, and some others introduce challenges and require more developers' effort. For example, GUI defects might require manual investigations and validations. As shown in Table 3.11, for Eclipse, the *Is GUI* and *Is User experience* are among the top 5 influential metrics predicting the change impact. Thus, integrating the topics of issue reports in the process of the defect fixing could lead to more satisfying results.

- Practitioners could benefit from the topics of issue reports to improve the software project from a specific perspective (*i.e.*, defect topic). As shown in Table 3.5, for one specific system, certain topics might be more frequent than others. Identifying the most frequent topics could guide developers in mitigating these frequent defects in the system and put in place test cases to detect these defects and hence improve the software.

- Practitioners could benefit from the topics of issue reports to improve the defect prioritization process. The relationship between the topics of issue reports and the severity level of the defect could help the developers recognize which topics should be treated with higher importance. For instance, as we observe in RQ3.1, *Server* or *Security* defects could possibly be given more priority than GUI defects in a software project belonging to the Web Server domain such as Apache.

- Practitioners could benefit from the topics of issue reports to improve the defect assignment process. Practitioners believe that defects belonging to the same topic

tend to have similar solutions [329]. Therefore, practitioners could assign to developers defects that belong to the same topic which could consequently lead to a more efficient fixing process while avoiding the context switching overhead.

Leveraging the change impact. Predicting the change impact could help the researchers and practitioners in many ways:

- Researchers could benefit from the estimated *small* or *large* amount of change to improve the existing automated triage tools. Practitioners expressed their need for an automated triage tool that automatically assigns small defects, *i.e.*, requiring *small* amount of change, and require manual intervention when the defect requires *large* amount of change [329]. Therefore, we encourage researchers to design an automated bug triage tool that incorporates both change impact prediction models, *i.e.*, code churn and the number of changed files, proposed in our work.

- Practitioners could benefit from the estimated change impact in terms of the number of files changed, to implement an efficient manual defect triage process. Practitioners could assign the defects that require a change in a large number of files (1) to developers that have knowledge of a large part of the codebase and its modules' dependency and (2) to developers that have access to the various codebases.

- Practitioners could benefit from the two change impact dimensions, *i.e.*, code churn and the number of files changed proposed in our work to design an efficient defect triage process that adeptly manages the limited human resources.

3.5 Threats to Validity

Construct validity relates to a possible error in the data preparation. In RQ3.2, our results depend on the metrics extracted from the issue reports and the source code of the associated commits. First, we follow a widely used approach to map issue reports to commits [72,141]. The used metrics have been extensively used in prior empirical software engineering research. Second, we estimate the change impact of defects in terms of code

churn and the number of files changed. Although there are other ways of estimating the change impact (*e.g.*, the number of modules, operations, or classes changed), measuring the lines of code and the number of files has been considered a common way of estimating the amount of change [102, 187]. Therefore, we assert a strong construct validity.

To classify the issue reports as requiring small or large change impact, we select the lower 10% and upper 10% of the issue reports as requiring small and large change impact, respectively. While this hyperparameter might be dependent on the selected dataset, we demonstrate that our approach is not sensitive to the selected threshold

To automatically assign topics to issue reports, we adopt ETM, the state-of-the-art topic modelling. Dieng et al. [65] demonstrate that ETM, the state-of-the-art topic modelling, outperforms LDA and its variants in extracting topics in terms of topic quality and predictive performance. LDA is one of the most used unsupervised topic modelling [51, 65, 98, 99]. Existing work in the realm of software engineering used LDA, and other variations of LDA to tackle research questions related to defect categorization [44, 52, 251], and bug report categorization [251, 327]. Therefore, we adopt ETM in our study.

Internal validity relates to the concerns that might come from the internal methods used in our study. The first concern comes from the manual assignment of topics to issue reports. To calculate the accuracy of ETM, we manually assign topics to a statistically representative sample of issue reports with a confidence level of 95% and a confidence interval of 10%, *i.e.,* 288 issue reports in total. To mitigate the possible human errors, we calculate Cohen's kappa agreement score between the two annotators, indicating a substantial agreement level. Although the annotators are not the owners of the issue reports, we highlight that the annotators have five years of work experience as software developers making them familiar with the reporting process and the topics of issue reports. The second concern is related to the textual free-text field of the issue reports, particularly the description. The description field might contain different noise types (*e.g.*, steps to reproduce, log execution and code fragment). To alleviate the noise, we manually check a sample of reports and ensure we exclude all unwanted patterns from the descriptions.

Third, the performance of the predictive models might be influenced by the training and testing datasets. To alleviate this risk, we adopt the 10-cross-validation technique in addition to adopting a time-based splitting strategy."

External validity relates to the potential of generalizing our study results. To maximize the ability to generalize our results, we include in our study more than 290,000 issue reports belonging to three popular open-source ecosystems, *i.e.,* Mozilla, Apache and Eclipse. Although our approach is evaluated on open source projects, it can be easily applied to other projects as long as the issue report information can be scraped. Similarly, the predictive model can be applied to other projects as long as the metrics are extracted and preprocessed, and the identification of influential metrics on a model can be easily performed. In addition, our results are limited to the types of software projects selected within each ecosystem. To minimize bias related to project characteristics, we gathered issue reports from 490 distinct software projects spanning the Apache, Mozilla, and Eclipse ecosystems. Specifically, our dataset includes 151 Mozilla projects, 5 Eclipse projects, and 334 Apache projects. These projects cover a broad range of domains, such as web servers, browsers, mail clients, integrated development environments (IDEs), and databases. However, while these projects represent a variety of areas, the selection might not fully capture the characteristics of the entire ecosystem. Furthermore, the selected issue reports come from two widely used issue tracking systems: Bugzilla and Jira.

3.6 Summary

In this chapter, we present an approach that predicts the change impact of defect resolutions. We conduct an empirical study on 298,548 issue reports belonging to three large-scale open-source systems, *i.e.*, Mozilla, Apache and Eclipse, to estimate the change impact in terms of code churn or the number of files changed while leveraging the topics of issue reports. First, we adopt the Embedded Topic Model (ETM), a state-of-the-art topic modelling algorithm, to identify the topics. Second, we investigate the feasibility of predicting the change impact using the identified topics and other information extracted

from the issue reports by building eight prediction models that classify issue reports requiring small or large change impact along two dimensions, *i.e.*, the code churn size and the number of files changed. Our results show that:

- Our approach successfully identifies 15 topics of issue reports and achieves an accuracy of 79%, on average.
- XGBoost is the best-performing algorithm for predicting the change impact, with an AUC of 0.84, 0.76, and 0.73 for the code churn and 0.82, 0.71 and 0.73 for the number of files changed metric for Mozilla, Apache, and Eclipse, respectively.
- The topics of issue reports improve the recall of the predictive model by up to 45% per software project and the AUC by 5% on average across the ecosystems.
- The number of attachments and comments per developer is among the top 3 influential metrics across the three ecosystems in predicting the amount of change size.
- To make the most of the change impact predictive model, 1) the developers attach relevant documents to the issue reports, 2) list the issues that are blocked by the defect in question and 3) indicate the accurate severity of the defect.

Chapter 4

Unraveling Code Clone Dynamics in Deep Learning Frameworks

In this chapter, we present our empirical analysis of DL frameworks to reveal insights into the long-term code clone trend over releases, and within-release clone patterns. Section 4.1 provides an introduction and the motivation for our study. Section 4.2 details the design of our work. Section 4.3 reports the results of our experiments. Section 4.4 discusses the implications of our findings. Section 4.5 examines the threats to validity. Finally, Section 4.6 summarizes the chapter and suggests directions for future research.

4.1 Introduction

DL frameworks play a pivotal role in the AI software development landscape. The Global DL Market is anticipated to grow a Compound Annual Growth Rate (CAGR) of 51.1% from 2022 to 2030[1]. Empirical data highlights the substantial popularity of DL repositories on GitHub, providing further evidence of the significance of DL frameworks. For instance, the *TensorFlow* framework has attracted remarkable attention, accumulating over 150,000 stars within an eight-year span [3]. Additionally, *TensorFlow* attains an impressive count of over 88,000 forks, making it among the top 5 globally recognized repositories on GitHub.

Code cloning might alleviate developers' workload [134], multiple studies have demonstrated the potential negative impact of code clones on the development and maintenance

[1]https://www.acumenresearchandconsulting.com/deep-learning-market

of software systems [152, 167]. Therefore, developers should be aware of the costs of the risk of code cloning. The evolution and impact of code clones are well studied in traditional systems [24, 192, 220, 277].

Despite the rising popularity of DL software, only two prior studies have explored code clones within the DL domain. Jebnoun et al. [124] study clones in Python, C#, and Java-based DL applications, shedding light on the prevalence of code clones during model creation, training, and data preprocessing. Mo et al. [188] emphasize co-changed clones within DL applications. A recent study demonstrates that clone occurrences are higher in DL systems as compared to traditional code [124]. DL code differs from traditional system code in terms of the development paradigm and coding practices [13]. Hence, the clone evolution analysis of traditional software could not be adapted to DL software.

We conduct experiments on nine popular DL frameworks, i.e., *TensorFlow*[2], *Paddle*[3], *PyTorch*[4], *Aesara*[5], *Ray*[6], *MXNet*[7], *Keras*[8], *Jax*[9] and *BentoML*[10] to gain a better understanding of the evolutionary dimension of code clones in DL frameworks. More specifically, we analyze long-term code cloning trends over releases and short-term code cloning patterns within releases. Studying long-term and short-term patterns is crucial as it enables developers to identify persistent cloning issues that could affect code maintainability and understand immediate impacts of development practices. By recognizing these patterns, developers can implement more effective clone management strategies to improve code quality.

Our empirical investigation yielded the following findings answering the studied research questions:

RQ4.1: What are the characteristics of the long-term trends observed in

[2]https://github.com/tensorflow/tensorflow.git
[3]https://github.com/PaddlePaddle/Paddle.git
[4]https://github.com/pytorch/pytorch.git
[5]https://github.com/aesara-devs/aesara.git
[6]https://github.com/ray-project/ray.git
[7]https://github.com/apache/mxnet.git
[8]https://github.com/keras-team/keras.git
[9]https://github.com/google/jax.git
[10]https://github.com/bentoml/BentoML.git

the evolution of code clones within DL frameworks over releases?

Our goal is to explore the characteristics of the long-term trends of the evolution of code clones over releases within DL frameworks. We observe that the four cloning trends, i.e., *"Serpentine"*, *"Rise and Fall"*, *"Decreasing"* and *"Stable"* exhibit distinct and common characteristics. The reduction in cloned code size in the *"Rise and Fall"* and *"Decreasing"* trends is attributed to factors, such as code refactoring, the reuse of third-party libraries, and the removal of code clones linked to feature elimination. Furthermore, our findings indicate that bug fixing is a consistent activity persisting throughout framework lifespans among all the trends but has a higher presence in the frameworks of the *"Serpentine"* trend.

RQ4.2: What are the characteristics of within-release code cloning patterns and do these patterns contribute to the overarching long-term trends in code cloning?

Our goal is to identify the characteristics of short-term code cloning patterns within-release and explore their potential impact on the long-term trends in code cloning. Our results demonstrate that within-release code cloning patterns, i.e., *"Ascending"*, *"Descending"*, and *"Steady"* patterns, impact long-term code cloning trends. In addition, we unravel the characteristics of the within-release patterns. For instance, *"Ascending"* within-release code cloning pattern is associated with decreased committer involvement in clone pairs, suggesting that fewer committers may lead to a higher likelihood of an increase in code cloned size.

RQ4.3: How do code clones manifest and evolve across different DL frameworks?

We aim to conduct a cross-framework clone detection to identify and analyze similarities in code across different DL frameworks. We find that cross-framework file-level code clones exist within DL frameworks, and they fall into two categories: *functional* and

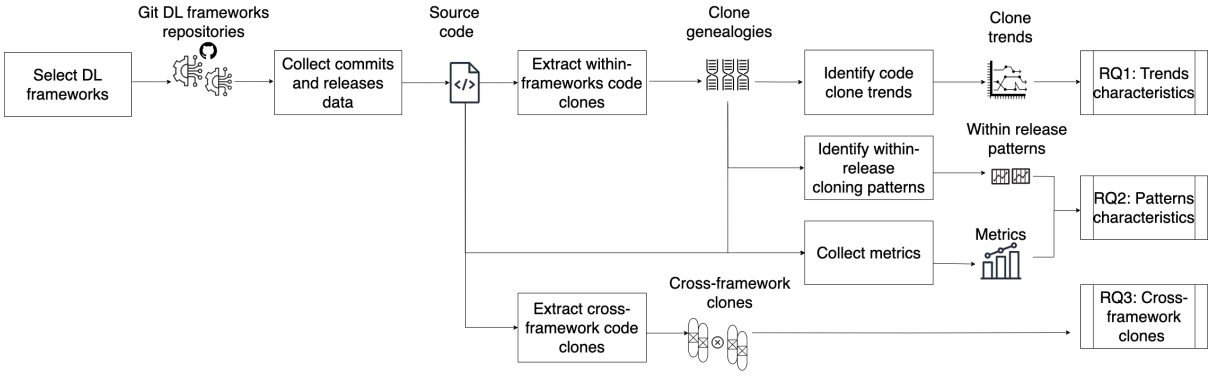

Figure 4.1: An overview of our approach for analyzing code cloning in DL frameworks.

architectural adaptation code clones. Cross-framework code clones undergo a gradual disappearance, attributed to functionality evolution, code divergence, function deprecation and framework restructuring.

4.2 Study Design

In this section, we present the study design. Figure 4.1 depicts the overview of our approach for analyzing code cloning in DL frameworks. In the first step, we select the DL frameworks. Then, we collect the commit and release data to construct the source code version history for each framework. Next, we extract the code clones within individual frameworks to build the clone genealogies and collect relevant metrics. In the final step, we extract the code clones across all the DL frameworks to answer RQ4.3.

4.2.1 Data collection

Deep Learning frameworks selection. DL frameworks provide a robust foundation for implementing DL applications, driving significant technological advancements such as imaging and autonomous vehicles. To identify the target DL frameworks for our study, we follow Du et al. [70] approach and refer to the widely studied frameworks in the recent existing work [49, 113, 119, 124, 139, 144, 249, 285, 286]. We investigate 14 candidates, including *TensorFlow*, *Chainer*, and *Keras*. Python has emerged as the most widely adopted programming language for DL applications [25]. Therefore, we tailor our data collection approach to include only Python frameworks. To ensure the robustness of our

dataset, we manually check the GitHub repositories of the DL frameworks and read the framework readme file to ensure it is DL-specific. We end up with the following nine DL frameworks: *TensorFlow*, *Paddle*, *PyTorch*, *Aesara*, *Ray*, *MXNet*, *Keras*, *Jax* and *BentoML*.

Commit and releases data collection. We extract the historical data of each repository to study the evolution of code clones and identify the evolution trends. In particular, we collect the commit history to extract the corresponding source code over time and the release information to be able to observe the trends in code cloning across different releases.

- *Commit collection and source code extraction.* For every framework, we obtain all commits by utilizing the `git log -- pretty=format:"%h,%ae, %ai, %s"` command, which includes commit information, such as the commit log message, author and committer details, and commit dates. For every commit, we compute the snapshot size in terms of Python source lines of code (SLOC) using the Cloc library [11] version *v-1.96*.

- *Release collection and pre-processing.* We gather the framework releases through the GitHub API, including their version numbers, release dates, and the associated release notes. A version string is composed of three numerical components delimited by periods, e.g., "1.7.2". We follow the existing work [304] and adopt semantic versioning (i.e., version strings represented by three numbers separated by dots) [39] to categorize the releases in our study into three main types: *major*, *minor*, and *patch*. *Major* releases are associated with software's API changes, *minor* releases to the addition of new features and *patch* to backward-compatible bug fixing. A major release corresponds to a modification in the first number, a minor release to a modification in the second number, and a patch release occurs in the third number. To determine the release type, we compare the version strings of the current release R_i and the preceding release R_{i-1}. If there are changes in multiple numbers, the precedence follows major over minor or patch, and minor over patch. In our study,

[11] https://github.com/AlDanial/cloc

we exclude *patch* releases and include only *major*, *minor* as they represent stable releases. Table 4.1 summarizes the statistics of the collected DL frameworks.

Table 4.1: The Descriptive Statistics of the Dataset.

Framework	First commit	# of commits	# releases
Aesara	2008-01-06	29,600	14
Keras	2015-03-27	7,787	9
MXNet	2015-04-30	11,865	20
Ray	2016-02-07	16,403	19
PyTorch	2016-05-02	48,561	18
Paddle	2016-08-29	36,531	22
TensorFlow	2018-03-04	131,854	46
Jax	2018-11-17	13,868	7
BentoML	2019-04-01	2,100	11

4.2.2 Within-framework code clone extraction

To identify code clones within one framework, we employ NiCad tool [230], a well-known and most-used text-based clone detection tool [311], which exhibits strong capabilities to detect both exact and near-miss clones (i.e., segments of code that are similar but not identical) at the block and function level with high precision and recall [231]. We use the latest available version *NiCad6.2* of NiCad [12]. We follow the standard configuration of NiCad, i.e., dissimilarity threshold of 30% and minimum cloned code size of 5 lines, used in recent related work [124,188] as with these settings, NiCad is reported to be very accurate in clone detection. We detect three code clone types: Type 1, Type 2, or Type 3. Type 1 clones are exact copies of code fragments without any modifications except for variations in whitespace and comments. Type 2 clones are syntactically identical fragments except for variations in identifiers, literals, types, layout, and comments. Type 3 clones have copied fragments with a few modifications (i.e., added, modified, or deleted statements). We choose clones at the functions level.

Snapshot code clone detection. We apply the following steps to every commit of the studied DL framework to detect clones. Firstly, we use the `git checkout` command

[12]https://www.txl.ca/txl-nicaddownload.html

to extract the framework snapshot corresponding to a specific commit from GitHub. Secondly, since in our study, we focus on code clones within production code, we follow the existing work [23] to exclude test-related code by identifying files and folders with the "test" keyword. Lastly, we use NiCad to detect the code clones. The output of the clone detection is pairs of functions, where two functions that are cloned, i.e., highly similar to each other, constitute a clone pair.

Clone genealogy generation. Clone pairs can undergo code changes throughout the development and maintenance stages of a framework. As a result of a change, a clone pair can maintain a *consistent* state, i.e., cloned status, or diverge to an *inconsistent* state. A genealogy represents the historical evolution of a pair of code clones. To generate the clone genealogy of each clone pair, we track its modification in every commit along the framework commit list. Similar to the existing work [23, 73], we conduct the below steps to generate the genealogy of every clone pair:

1. For a given commit C_i, we identify the files changed. If a changed file changes a clone pair, we proceed by comparing C_i to the previous commit C_{i-1} to verify if the change was made to the clone pair. We use the `diff` command and a python third party library *whatthepatch*[13] to map the lines corresponding to the beginning and end of the function in both commits C_i and C_{i-1}. To handle the case where a file name was modified, we map the renamed file to the original file by performing the following steps as introduced in the existing work [23, 73]. First, for each commit, we extract the pairs of newly added and deleted files between the current and preceding commit. Second, we compare the code similarity and consider that a file was renamed if the content of the new file is similar to the content of the old file. We use the below git command to construct the pairs of renamed files between two commits: `git diff [old-commit] [new-commit] --name-status -M`

2. We check if a code change, i.e., code lines addition or deletion, is made in C_i within the boundaries of the clone pair. If so, we verify if this clone pair exists in the clone

[13]https://pypi.org/project/whatthepatch/

snapshot corresponding to commit C_{i+1}. If the clone pair exists in C_{i+1}, we attribute the state *consistent* to it; otherwise, we consider the pair state as *inconsistent*.

3. We reiterate this process, i.e., steps 1 and 2, for every commit in the framework repository or until at least one of the clones is deleted.

4.2.3 Cross-framework file-level code clone extraction

To identify code clones across several DL frameworks, we employ SourcererCC [234], a token-based clone detection tool capable of identifying both exact and near-miss clones within extensive inter-framework repositories. We select SourcererCC as it outperforms other clone detectors, e.g., Nicad, for large-scale cross-projects clone detection [88, 226] and it was used by recent cross-project clone related work [88, 221, 310] for its high performance. We use the publicly available version[14] on GitHub. SourcererCC performs clone detection in two steps. Firstly, it employs a tokenizer where the programming language parameters, such as file extension, are set. In our case, we set the file extension to ".py". Secondly, the actual clone detection is executed. For this step, the similarity thresholds parameter is specified. Existing work used different values for the similarity threshold. For example, Rahman et al. [221] used 70% whereas Chochlov et al. [54] used 50%. In our study, we vary the value between 60% and 80%. We adopt the 60% similarity threshold as it detects meaningful code clones across frameworks.

4.2.4 Metrics collection

After we generate the code clone snapshots and genealogies, we collect cloned metrics for every framework. The collected clone metrics capture information about the snapshot and the genealogy of a clone pair to investigate the relationship between the collected metrics and the cloned code size. In total, we collect 29 metrics, representing (1) clone code, (2) process metrics, (3) clone genealogy metrics and (4) code metrics. These metrics were used in existing work [23, 73, 265]. We describe below each of the categories and present the collected metrics in Table 4.2.

[14]https://github.com/Mondego/SourcererCC

Clone code metrics. Clone code metrics are the attributes related to the snapshot, i.e., commit. The clone code metrics are collected from the snapshot of the framework that contains the clone pair. Example metrics include the number of siblings a clone has in the version where it is introduced, the size of the clone and the number of clone groups.

Process metrics. Process metrics include metrics representing the involvement of contributors in the development of clone pairs. For example, the number of committers, i.e., the person who applied the commit to the repository, and authors, i.e., the person who originally wrote the changes introduced in the commit, associated with the genealogy of clone pairs that provide insights into the collaborative efforts, i.e., number of individuals that have applied and integrated changes related to the clone pairs, and individual contributions to the codebase over time.

Clone genealogy metrics. Clone genealogy metrics capture the state changes in the history of clone pairs and the history of changes in a clone pair. An example of a clone genealogy metric is the ratio of inconsistent changes in the whole framework code clone genealogy.

Code metrics. Code metrics include metrics related to the code characteristics of the method that contains a clone, such as cyclomatic complexity, the number of declaration and execution statements and . To analyze code metrics, we use the Understand tool[15], a reverse engineering tool capable of analyzing clone code from the source code of software systems. First, we run the Understand tool on all the snapshots of DL frameworks and extract the metrics on the function level. We obtain metrics such as the number of declarative statements in a function and the complexity of a function. Second, we map the functions obtained by the Understand tool to the clone code to compute the metrics for the cloned metrics.

[15]https://scitools.com/

Table 4.2: Collected code clone metrics.

Metrics	Description
	Clone code metrics
NumCP	The total number of clone pairs at any specific commit.
NumCPGroups	The unique number of groups at any specific commit.
MedianCLOC	The median number of cloned lines of code [23].
MaxCLOC	The maximum number of cloned lines of code [23].
MedianCSib	The median number of siblings of the clone pairs at any specific commit [73].
MaxCSib	The maximum number of siblings of the clone pairs at any specific commit [73].
CPMedianAgeDays	The median clone pair age in terms of the number of days [73].
CPMedianAgeCommits	The median clone pair age in terms of the number of commits [73].
NumUniqueFiles	The unique number of files that contain clone pairs.
NumAbstClasses	The number of abstract classes containing code clones.
MedCodeSim	The median of code clone similarity.
RatioSync	The ratio of clone pairs that have a final "consistent" change in the genealogy [23].
RatioLatePropCons	The ratio of clone pairs having late propagation, which end in a consistent change [23].
RatioLatePropIncons	The ratio of clone pairs having late propagation, which end in an inconsistent change [23].
	Process metrics
NumCommittorsGen	The number of committers involved in the genealogy [73].
NumAuthorsGen	The number of authors involved in the genealogy.
MedianNumComChanges	The median number of commits in the genealogy by a specific committer [73].
MedianNumAutChanges	The median number of commits in the genealogy by a specific author.
	Clone genealogy metrics
NumCPG	The number of clone pairs in the genealogy.
NumChanges	The total number of changes in the genealogy.
NumBursts	The number of change bursts on a clone. A change burst is a consecutive change with a maximum distance of one day between the changes [23].
NumInconsChanges	The number of inconsistent changes within the genealogy [23].
NumDiverChanges	The number of divergent changes (pattern "CI") in the genealogy [23].
NumResyncChanges	The number of resynched changes (pattern "IC") in the genealogy [23].
	Code metrics
MedianCompl	The median cyclomatic complexity of the cloned code [265].
MedianNumDeclStmt	The median number of declaration statements in a cloned code.
MedianNumExecStmt	The median number of execution statements in a cloned code.
MedianRatioCommentCode	The median ratio comment to code in cloned code [265].
MedianSumComplAbstMod	The median sum of cyclomatic complexity of all the classes containing cloned functions.

4.2.5 Lifelong code cloning trends identification from DL frameworks

Our goal is to discern the lifelong evolution, i.e., long-term, trends of code clones within DL frameworks to gain insights into the historical and chronological changes in cloned code over multiple releases. This analysis can provide valuable information about the stability, growth, or reduction of code clones over time, aiding in the assessment of software evolution and maintenance practices within DL frameworks. For every DL framework, we build a time series, i.e., a temporal representation, capturing the historical and chronological changes to the cloned code over the releases of a framework. Then, we group similar time series patterns together and identify the code clone trends over the releases. We elaborate on our approach in the following steps.

Step 1: Building time series data. To build the time series that represents the historical evolution of code clones, we start by calculating the clone coverage. Clone coverage represents the proportion of cloned code in relation to the overall codebase [116]. The clone coverage is calculated as follows:

$$\text{clone\_coverage\_}i = \frac{\text{clone\_size}_i}{\text{SLOC}_i} \tag{4.1}$$

where $clone\_coverage_i$ represents the clone coverage corresponding for *commit i*. The $clone\_size_i$ represents the total number of lines of code belonging to clones at *commit i*. The $SLOC_i$ represents the lines of source code of the overall repository at *commit i*.

Our goal is to construct a release-level clone coverage time series for every framework. We choose the release-level because it represents a stable granularity as opposed to the commit-level granularity that would have been a fine granularity, given the small and/or temporary modifications that a commit can introduce [59]. We calculate the clone coverage for every revision, i.e., commit, using the code clone data extracted through the NiCad tool detailed in section 4.2.2 and the SLOC of the framework snapshot at the commit. Therefore, for every release, we compute the median code clone coverage for all the commits within the time interval between two consecutive releases. We refer to the list of releases extracted from GitHub as described in section 4.2.1 and the associated

release dates to isolate the commits falling within the release date intervals. Then, we calculate the median release clone coverage as follows:

$$\text{clone\_coverage}_{R_j} = \text{median (clone coverage of all commits in the release } R_j) \quad (4.2)$$

where R_j is a specific release of the project. By the end of this step, we obtain the median value of code clone coverage for every release of every framework. For each DL framework, we construct a time series representing the evolutionary trajectory of clone coverage across multiple releases. By the end of this step, we obtain nine unique time series, one for each framework.

Step 2: Clustering time series. We utilize time series clustering to group time series presenting similar patterns in code cloning. This approach involves the selection of (i) a method for measuring distances, (ii) a clustering algorithm, and (iii) an optimal number of clusters, as we elaborate on below.

- *Dynamic Time Warping (DTW).* DTW is used as a distance measurement method [29] to align and measure the similarity between two time series that may have different temporal characteristics. In our context, it is used to compare and align the time series representing code clone coverage across multiple releases of DL frameworks. Different DL frameworks may have a distinct number of software releases, resulting in time series of varying lengths. To measure the similarity, i.e., distance, between time series data of different lengths, we adopt the DTW.

- *TimeSeriesKMeans clustering algorithm* is used for time series clustering [287] to identify patterns in the evolution of the nine DL frameworks. K-means clustering is a widely used unsupervised machine learning technique that is particularly effective for partitioning data into distinct groups or clusters based on the similarity of the data points. K-Means treats each time series as a point in a multidimensional space, aiming to group similar time series together in clusters. In our analysis, we employ tslearn [16] implementation of K-means for time series, TimeSeriesKMeans

[16]https://tslearn.readthedocs.io/en/stable/gen_modules/tslearn.clustering.html#module-tslearn.clustering

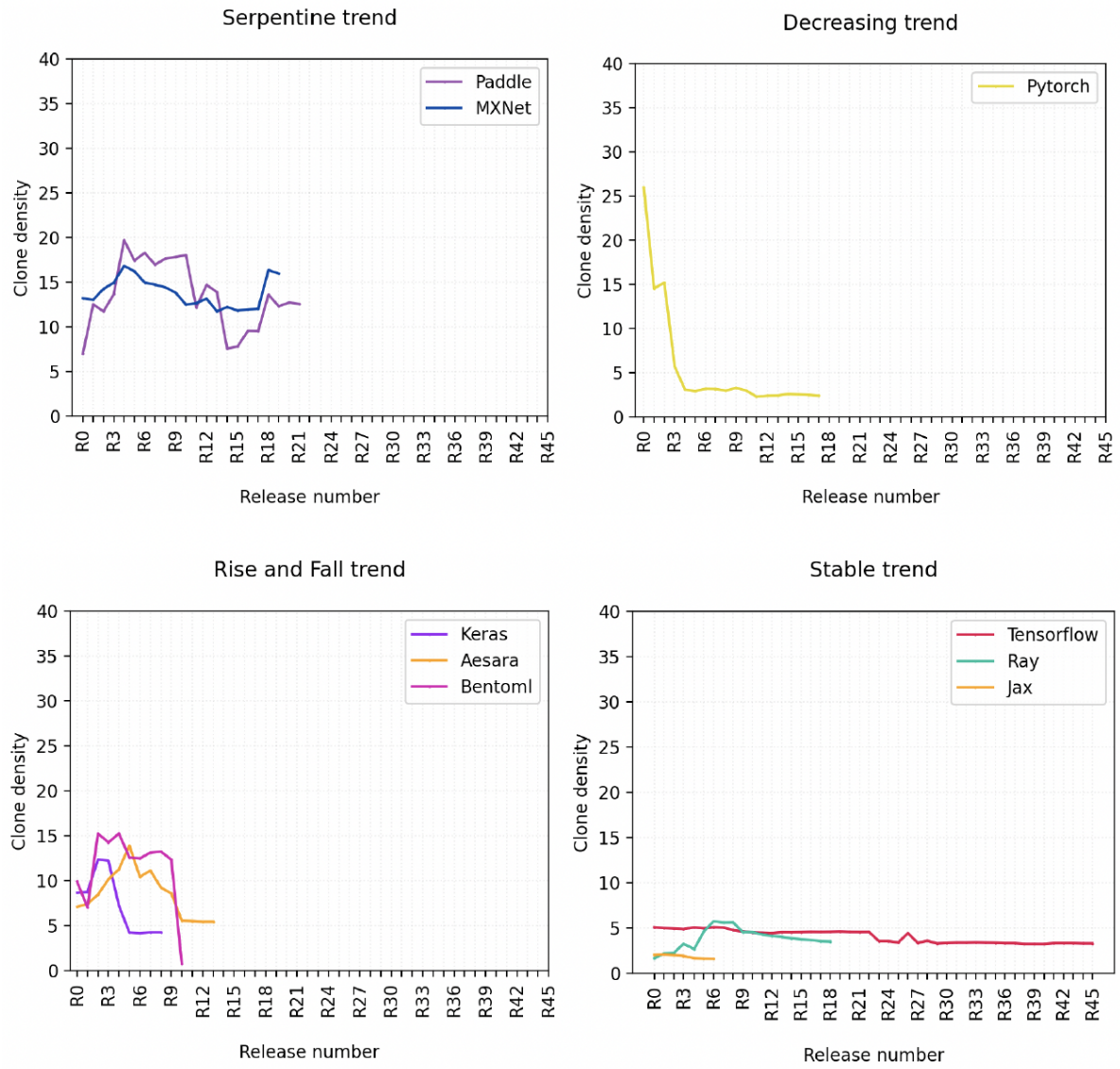

Figure 4.2: The four code clones trends exhibited by DL frameworks.

[17] to identify and categorize the trends present in the time series data representing code clone coverage within DL frameworks.

- *Optimal number of clusters.* Following prior work [211], we employ the silhouette score [228] to identify the most suitable number of clusters. The silhouette score assesses clustering quality by measuring how close data points are to their cluster compared to other clusters. The silhouette score ranges from -1 to 1, where a score close to 1 suggests well-defined clusters, and a score above 0.5 indicates a good clustering configuration. It leverages the output of the chosen clustering algorithm,

[17] https://tslearn.readthedocs.io/en/stable/gen_modules/clustering/tslearn.clustering.TimeSeriesKMeans.html

e.g., K-Means, to determine the optimal number of clusters, aiming for the highest average silhouette score, which indicates well-separated and coherent clusters. We employ the silhouette score function[18] from tslearn. We use a range of numbers of clusters, k, between 2 and 8. As a result, we obtain 4 optimal numbers of clusters with a silhouette score of 0.63, suggesting a good quality of clusters. Additionally, we rely on human judgement that aligns with the quantitative metrics.

Our approach identifies four trends, i.e., *"Serpentine"*, *"Rise and Fall"*, *"Decreasing"*, and *"table"*, in the evolution of clones observed in DL frameworks as illustrated in Figure 4.2. We describe the features of every pattern below:

- The *"Serpentine"* trend reflects a cyclic, oscillatory pattern in code clones, where coverage alternates between highs and lows, presenting fluctuations in code clone coverage over successive releases.
- The *"Rise and Fall"* trend marks intermittent rises in clone coverage followed by an overall prolonged decrease, resulting in a downward trajectory over time.
- The *"Decreasing"* trend represents a consistent and continuous reduction in clone coverage across successive releases.
- The *"Stable"* trend exhibits a relatively constant and consistent clone coverage with minimal fluctuations over releases.

For example, while *Paddle* and *MXNet* display a *"Serpentine"* trajectory, *BentoML*, *Keras*, and *Aesara* follow a *"Rise and Fall"* trajectory that shows a prolonged decrease despite intermittent rises. *PyTorch* exhibits a consistent decreasing clone coverage, i.e., starting at 27% at the first release and dropping to less than 3% after 20 releases. *TensorFlow*, *Jax*, and *Ray*'s clone coverage remain relatively stable, with minimal fluctuations observed across all releases remaining within a narrow range (less than 5%).

[18]https://tslearn.readthedocs.io/en/latest/gen_modules/clustering/tslearn.clustering.silhouette_score.html

4.2.6 Within-release development patterns identification

Our goal is to identify the within-release development patterns, i.e., the patterns of cloned code size evolution from one commit to another within a framework release. This analysis investigates how code clones evolve within each individual release and can help uncover the impact of the within-release patterns on the long-term code cloning trends. Our approach involves two key steps:

Step 1: Building within-release cloned code size time series. The evolution of code cloned code size between the commits of a release can be interpreted as time series data. To build the time series for every release of every framework, we select the commits that belong to every release interval by referring to the release date. Then, we calculate the cloned code size using the code clone data extracted through the NiCad tool detailed in Section 4.2.2 for every commit. In total, we obtain 153 time series corresponding to the nine DL frameworks' releases.

Step 2: Clustering within-release code clone time series. Similarly to the clustering step in section 4.2.5, we leverage DTW and TimeSeriesKMeans to cluster the obtained time series. We determine the number of clusters using the silhouette score, and we obtain 3 as the optimal number of clusters, giving a silhouette score of 0.66, indicating a high quality of clusters. Figure 4.3 shows a subset of the time series clustered into the three *"Ascending"*, *"Descending"*, and *"Steady"* patterns.

The analysis of release-level time series reveals three distinct patterns, i.e., *"Ascending"*, *"Descending"*, and *"Steady"*. As depicted in Figure 4.3, the *"Ascending"* pattern signifies a consistent increase within a release, the *"Descending"* pattern reflects a continuous decrease within a release, and the *"Steady"* pattern suggests a relatively constant behaviour within a release. Table 4.3 shows the distribution of time series among these patterns.

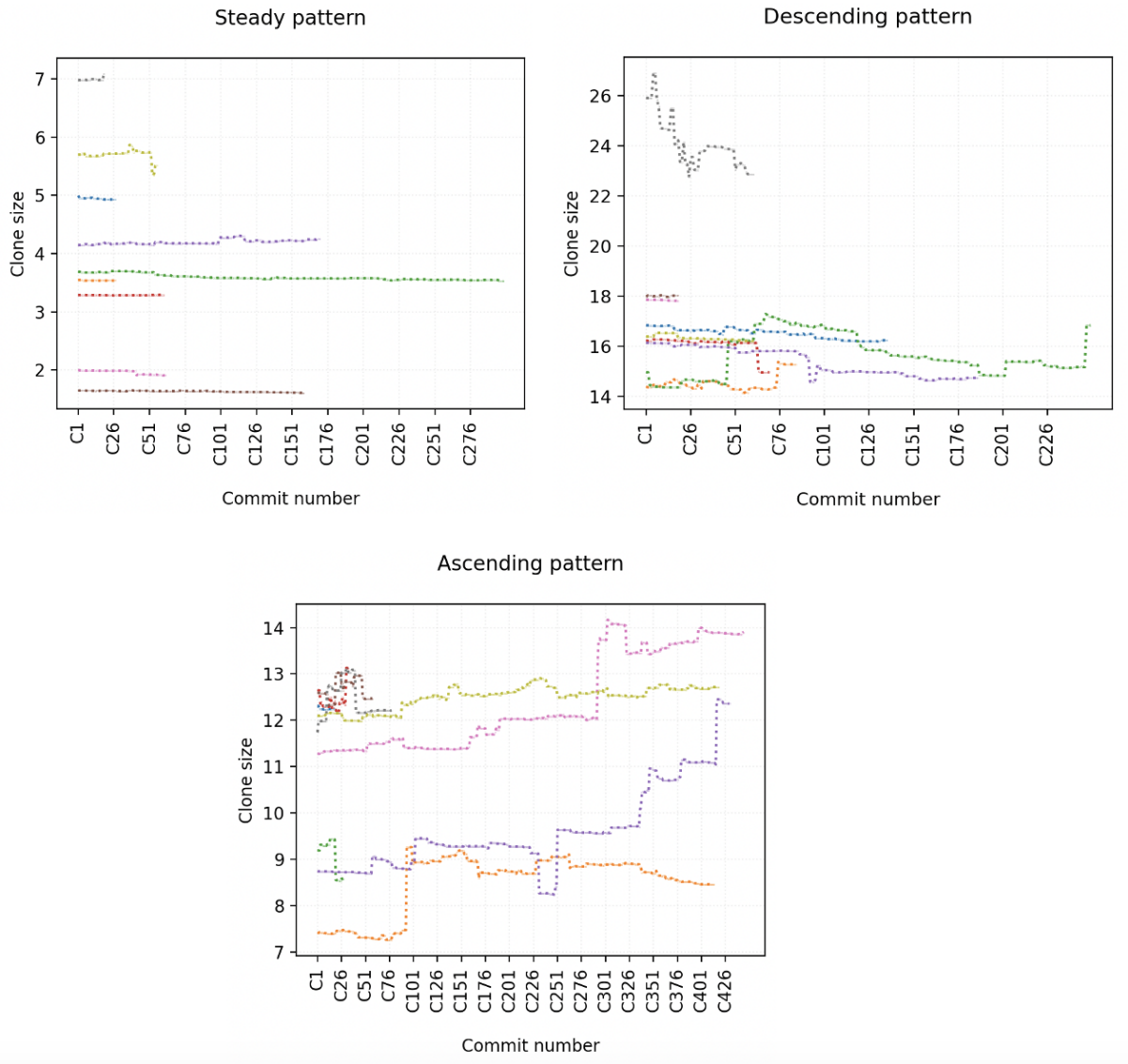

Figure 4.3: Subset of the within-release time series *"Steady"*, *"Descending"* and *"Ascending patterns"*. Every time series represents the evolution of clone size within a particular release of a DL framework.

4.3 Experimental Results

4.3.1 RQ4.1: What are the characteristics of the long-term trends observed in the evolution of code clones within DL frameworks over releases?

Motivation. The field of DL is marked by rapid growth and continual advancements. As DL frameworks rapidly evolve, they are prone to technical debt [123, 279]. Recent work [123, 188] demonstrates that code clones are prevalent in DL applications. Mo et al. [188] find that the choice of underlying DL frameworks influences the frequency of

Table 4.3: The distribution of release-level time series across the three within-release code cloning patterns. TF represents TensorFlow.

Pattern	TF	Paddle	PyTorch	Aesara	Ray	MXNet	Keras	Jax	BentoML	Total
Ascending	0	11	0	8	0	12	4	0	4	39
Descending	0	8	4	0	0	9	0	0	4	25
Steady	38	2	14	4	19	0	4	5	3	89

code clones within DL applications. Jebnoun et al. [123] discover that code clones within DL systems exhibit a higher susceptibility to bugs compared to non-cloned code. In this RQ, we aim to investigate the characteristics (e.g., bug-proneness and community size) of the four identified lifelong code clone evolution trends and uncover any implications that they may have. We aim to enhance our understanding of code quality and maintenance over time by studying these characteristics.

Approach. Our goal is to explore the characteristics associated with the identified code clone trends. We study the characteristics of the trends along three dimensions: cloned code size trend, bug-proneness and community size.

Investigating the decrease of clone coverage. To better understand the rationale behind the decrease in clone coverage in the long-term descending trends, i.e., trends that exhibit a lower long-term clone coverage, we conduct a manual analysis. First, we investigate the evolution of the cloned code size, i.e., the number of lines of cloned code, associated with the median data point of each release to check if the decrease in clone coverage is associated with a decrease in the cloned code size. Then, for each release, we manually read the release notes and the commit messages of the commits associated with the release interval. Release notes typically summarize the major changes, new features, and bug fixes introduced in a specific release, hence providing context about the reasons for code changes that might affect clones. Commit messages offer granular information about specific code modifications, hence could explicitly mention clone removal, refactoring, or other actions that directly impact clone coverage. Analyzing release notes and commit messages may help link changes in code clone to the corresponding modifications in the software codebase thereby gaining insights into the rationale of the evolution of clone coverage over time.

Investigating bug-proneness within trends. We follow a two-step approach to study the bug-proneness of code clones. First, we apply the keyword-based heuristic introduced by Mockus et al. [189] and adopted in related work [24,124,188,191] to identify bug-fixing commits. We consider a commit as a bug-fixing commit if the commit message contains any of the keywords *bug, fix, wrong, error, fail, problem, and patch.* Second, we match the bug-fixing commits to the cloned code. Specifically, we employ the `git diff` command to retrieve the altered lines, i.e., added or deleted, within each bug-fixing commit and compare them to the clones identified by NiCad. If the modified lines for fixing a bug occurred within a clone code, we classify that clone as potentially susceptible to bugs. This approach allows us to pinpoint the bug-fixing commits occurring in clones. After identifying the bug-fixing commits belonging to the cloned code, we plot the evolution of the bug-proneness of cloned code over releases. To quantify bug-proneness, we compute bug-fix density$_i$, i.e., the ratio of bug-fixing commits in the cloned code introduced within a release interval by the total number of all commits to code clones introduced within a given release i. This ratio is calculated as follows:

$$\text{bug-fix density}_i = \frac{\delta\ (\text{clone\_bugs\_fix\_commits}_i)}{\delta\ (\text{all\_clone\_commits}_i)} \tag{4.3}$$

where δ(clone_bugs _fix _commits$_i$) represents the commits introduced within release i interval and classified as bug-fixing and δ(all_clone _commits$_i$) represents the total number of commits that occurred in cloned code within the release $_i$. A bug-fix density ratio of 0 signifies no bug-fixing commits in the cloned code during a release, while a ratio of 1 indicates that all commits within that release for the cloned code are bug-fixing commits. By tracking the evolution of this ratio across releases, we obtain insights into the dynamic relationship between code clone trends and bug-proneness, enhancing our understanding of code quality and maintenance over time.

Investigating bug-proneness in "thin clones" vs. "thick clones". We identify *"thin clones"* and *"thick clones"* based on the distribution of clones according to the percentiles. Firstly, we gather all the clone snapshots from all the frameworks. For each snapshot, the median cloned code size is computed, taking into account the diverse

sizes of clone pairs within each snapshot of the codebase. Subsequently, the 25^{th} and 75^{th} percentiles are calculated, establishing thresholds for *"thin clone"* and *"thick clones"*, respectively. More specifically, code clones falling below the 25^{th} percentile are categorized as *"thin clone"*, while those exceeding the 75^{th} percentile are labelled as *"thick clone"*. Following the identification of bug-fixing commits, we map each change to the corresponding cloned code (as described in previous steps). Through this mapping, we categorize each commit-fix as either fixing a *"thin clone"* or a *"thick clone"*. To assess the statistical significance of bug-proneness differences between "thin clones" and "thick clones," we employ a Mann-Whitney U test on the percentage distribution of the number of changes to cloned code in *"thin"* and *"thick"* clones. This non-parametric test is chosen due to the potential non-normality of the data and its suitability for comparing two independent samples. If we obtain a p-value ¡= 0.05, we reject the null hypothesis and conclude that the distributions of bug-fixing percentages in the two code clone categories are different. Studying bug-proneness in "thin clones" versus "thick clones" provides insights into whether bugs tend to concentrate more on one type of clone over the other, which can help understand how code clones impact software quality and maintenance.

Investigating the bug propagation in clones. Different trends of code clones may exhibit diverse characteristics, such as cloned code size and distribution of thick and thin clones, which could influence how bugs propagate within the codebase. We formulated the following null hypothesis: *H01: Different trends of code clones exhibit different bug propagation trends.* To investigate the bug propagation within code clones, we examine the number of file changes per bug-fixing commit, with the aim of discerning potential differences in bug propagation across various trends. Specifically, we evaluate whether the bug propagation varies significantly among different trends.

To assess significance, we employ Welch's ANOVA test from scipy library [19], a statistical analysis method suited for comparisons of groups with unequal sizes. Additionally, we also compare bug propagation, i.e., by examining the number of file changes per bug-fixing commit, in cloned and non-cloned code. This comprehensive approach allows us

[19]https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.f_oneway.html

to gain insights into the distinctive patterns of bug propagation within code clones and explore potential variations in comparison to non-cloned code.

Investigating the community size evolution in clones. The community size is used to measure the proportion of contributors, i.e., authors of the code, engaged in clone-related activities within DL frameworks relative to the total contributors involved in the entire codebase. The community size represents an indicator of the collaborative involvement of contributors to clones. To derive the community size evolution, we employ the following steps. First, for each release, we calculate the number of contributors engaged in clone-related activities up to the release date. Subsequently, we determine the community size by dividing the number of contributors involved in clones by the total count of contributors to the entire codebase up to the respective release date. This metric provides a measure of how the number of contributors in clone-related efforts evolves over the course of the framework.

Results. The decline in overall clone coverage observed in the *"Decreasing"* and *"Rise and Fall"* trends can be attributed to 1) a decrease in the cloned code size or a slow increase in the number of cloned code size compared to a rapid expansion of the codebase. We conduct a comparison of cloned code size at the initial and final releases of each framework. As shown in Table 4.4, we observe that BentoML presents a reduction in cloned code size over time, resulting from a decrease in clone groups and clone siblings within a clone group. For example, despite a more than twofold increase in framework size between the first and last releases, BentoML's cloned code size decreased from 982 to 174 lines of code. This decrease in cloned code size corresponds to a decrease in the clone groups from 23 to 5. Hence, the decrease in clone coverage of BentoML from 10% in the first release to less than 1% in the last release. Aesara, Keras, and PyTorch also exhibit a decrease in clone coverage between the first and last release. Specifically, the clone coverage for Aesara, Keras, and PyTorch drop from 7% to 5%, 9% to 4% and 26% to 2%, respectively, between the first and last release. However, the decrease for the three frameworks is attributed to the rapid expansion of the codebase, outpacing the growth rate of the cloned code size. For example, while the

codebase for Pytorch increased by a factor of 30 between the first and last release, the size of the clone code only increased by less than threefold.

Table 4.4: Cloned code size analysis of the DL frameworks exhibiting "Decreasing" and "Rise and Fall" trends from first to last Release.

	Metric	First release	Last release	Trend
BentoML	SLOC	9,917	22,494	↗
	Code size (SLOC)	982	174	↘
	Clone coverage	0.1	0.008	↘
	# of siblings	58	18	↘
	# of groups	23	11	↘
Aesara	SLOC	39,021	84,773	↗
	Code size (SLOC)	2,765	4,612	↗
	Clone coverage	0.07	0.05	↘
	Total number of clone siblings	307	396	↗
	Total number of clone groups	117	139	↗
Keras	SLOC	19,382	139,816	↗
	Code size (SLOC)	1,680	5,947	↗
	Clone coverage	0.09	0.04	↘
	# of siblings	117	335	↗
	# of groups	43	98	↗
PyTorch	SLOC	10,567	313,603	↗
	Code size (SLOC)	2,741	7,527	↗
	Clone coverage	0.26	0.02	↘
	# of siblings	275	583	↗
	# of groups	62	202	↗

The decline in cloned code size can be attributed to code refactoring, third-party library reuse, and code clone removal associated with feature elimination. To understand the factors contributing to the decrease in clone code density, we conduct an inter-release manual analysis by reading the release notes and commit messages for releases demonstrating a reduction in cloned code size compared to their preceding release. We find that cloned code size reduction is due to 1) code refactoring, 2) substitution of code fragments due to the use of third-party libraries, and 3) the removal of clone code as a consequence of eliminating certain features. For instance, duplicate code is moved to a shared utility code. For example, in release *2.2.0*[20], the Keras framework underwent a large code refactoring[21] that affected the clones. In another release

[20]https://github.com/keras-team/keras/releases/tag/2.2.0
[21]https://github.com/keras-team/keras/pull/10865/commits

2.4.0[22] of the Keras framework, the cloned code size is reduced due to a redirect of all APIs in the Keras package to tf.keras third party-library. In the release *2.2.0*[23] of the Aesara framework, clones are eliminated as a result of the elimination of the function *local_neg_neg* from the tensor module.

Bug fixing is a persistent activity consistently occurring throughout the lifespan of frameworks, among all the code cloning trends. The *"Serpentine"* trend is more susceptible to bugs. Figure 4.4 represents the evolution of bug-proneness in clone code in DL frameworks over releases. When comparing bug-proneness percentages across frameworks, we notice distinct variations. For instance, frameworks belonging to the *"Serpentine"* trend, i.e., *Paddle* and *MXNet*, demonstrate higher bug-proneness percentages across releases in comparison to the frameworks belonging to the *"Decreasing"*, *"Rise and Fall"* and *"Stable* trends. For example, *Paddle* and *MXNet* frameworks have 50% and 75% of the releases having more than 50% bug-fixing commits in clones respectively, whereas in *TensorFlow* all 100% of the framework releases have less than 50% of the bug-fixing commits in clones as shown in 4.4. ANOVA-Welch test indicates a significant difference between the bug-proneness evolution of *Paddle* and *MXNet* frameworks and the frameworks belonging to the other three trends. These findings suggest that the DL frameworks belonging to the *"Serpentine"* trend characterized by a fluctuating code clone trend could be more susceptible to bugs.

Moreover, the *Keras* and *Aesara* frameworks belonging to the *"Rise and Fall"* clone trend exhibit a decreasing bug-proneness trend where the overall bug-proneness percentage decreases from approximately 40% in initial releases to almost zero in subsequent releases. For the Aesara framework, we notice a spike to 100% in the bug-proneness at release 12, meaning that all of the commits introduced were fixing bugs. Further investigation demonstrates that the 100% rate is due to a short release cycle of 11 days, during which only one commit to a clone occurs, and this solitary commit happens to be a bug fix. Similarly, the abrupt decrease in bug-proneness to zero in the *MXNet* framework at release 17 is linked to the absence of commits to code clones during a relatively short

[22]https://github.com/keras-team/keras/releases/tag/2.4.0
[23]https://github.com/aesara-devs/aesara/releases/tag/rel-2.2.0

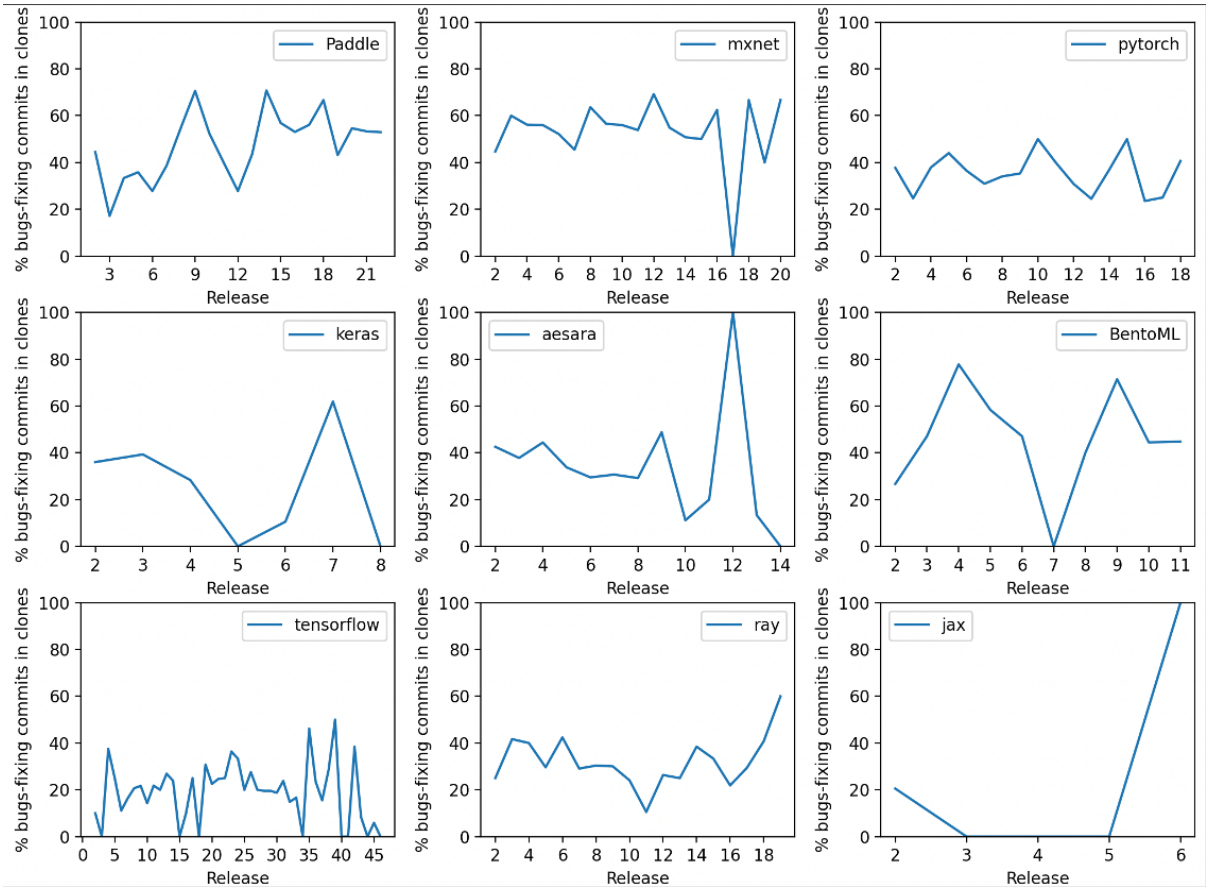

Figure 4.4: The bug-proneness evolution in cloned code over releases.

release cycle of 27 days. The observed descending trends in code clones, coupled with a corresponding reduction in bug-proneness, suggest an enhancement in overall code quality and reliability. These findings underscore the importance of investigating the coding practices followed within the release and their impacts on the code clone in DL frameworks, which we address in the following research question.

Over 50% of the changes implemented in bug-fixing commits predominantly occur within *"thick"* cloned code as opposed to *"thin"* cloned code across all the long-term code clone trends. Figure 4.5 shows the distribution of changes across the frameworks. The Mann-Whitney U test reveals a significant difference in bug-proneness between *"thin* clones and *"thick"* clones with p-value < 0.05. In addition, we investigate if the difference among the clone categories is of strong significance by calculating the Common Rank (Z) score. We obtain a Z value of 3.57 that suggests a substantial and statistically significant difference between the two bug-fixing commits in

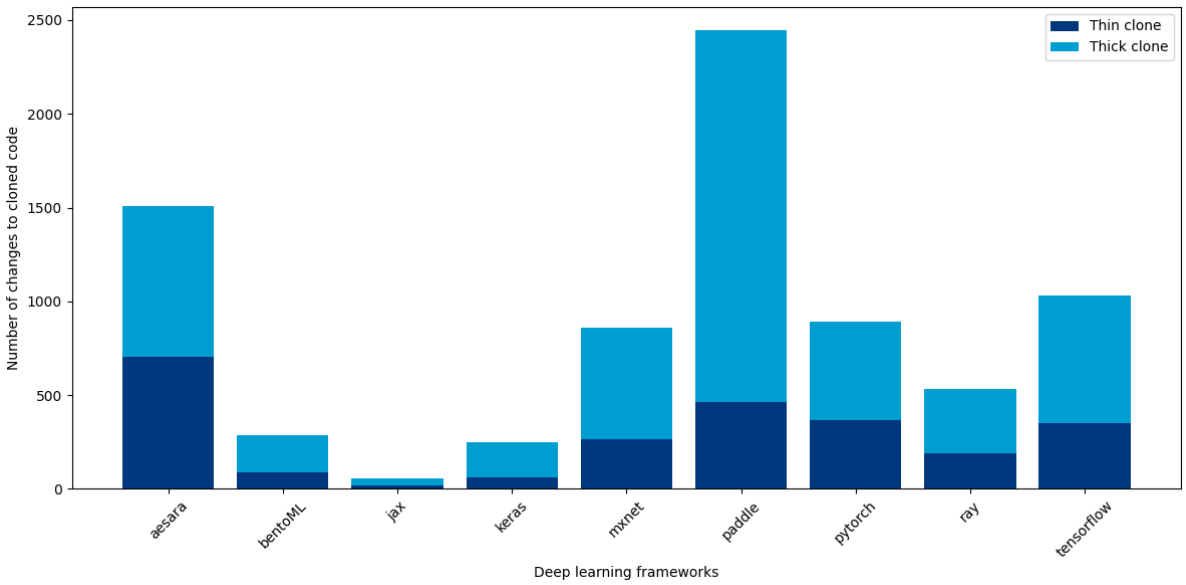

Figure 4.5: Bug-fixing commits distribution by *"thin clones"* and *"thick clones"*.

"thick" and *"thin"* code clones. A z-score [24] of +/- 1.96 or greater is considered statistically significant at the 5% level of significance (i.e., $p < 0.05$). Additionally, we employ the Cliff's Delta as another measure of effect size. A value of 1 indicates a large effect size. We obtain a Cliff's Delta value of 1, which further reinforces the notion that the bug-proneness difference between *"thin"* and *"thick"*. These results provide statistical evidence supporting the assertion that bug-proneness varies significantly between *"thin* clones and *"thick"* clones in our study. This result highlights a substantial distinction in bug-fixing tendencies between the two clone categories with *"thick"* clones being more susceptible to bug-fixing commits. This finding can be valuable for software development practices by emphasizing the importance of identifying and managing thicker clones to improve overall code quality and reduce bugs.

Our analysis reveals no significant difference in the number of files changed per bug-fixing commit across long-term code clone trends, as validated by Welch's ANOVA test, and we reject the null hypothesis. However, comparing the bug propagation between cloned and non-cloned code across the frameworks indicates a significant difference in the number of files changed per bug-fixing commit in clones versus non-clones. Specifically, non-cloned files exhibit a higher number of changes, with

[24] https://www.z-table.com/

an average of 2.4 files changed per commit, compared to 1.5 files changed per commit in cloned code. This could suggest that bug fixes in cloned sections are more localized and targeted.

The community size in clone activities remains consistent throughout the lifetime of the frameworks, with exceptions in *BentoML* and *Ray*, where original authors dominate clone maintenance and the community size witnesses a decrease over time. For instance, a relatively small portion of the community actively contributes to clones, consistently amounting to less than 50% for the majority of releases. Figure 4.6 depicts the community size evolution, i.e., the percentage of contributors to clones across various DL frameworks. As we can notice, the community size involved in clone-related presents a stable and sustained level of contribution to clones over time, among all the long-term cloning trends of the framework. However, there is a distinct reduction in community size in code clones within the frameworks *BentoML* and *Ray*. This decline in community participation can be attributed to the circumstance where the contributors making changes to clones are predominantly the original authors of those clones. To investigate this observation further, we manually check the clone pairs within *BentoML* and *Ray*. We notice that for those frameworks, the original creators of the clones are the ones maintaining them. This may imply that a significant portion of the code clones might have been created by one-time contributors who are less likely to revisit later.

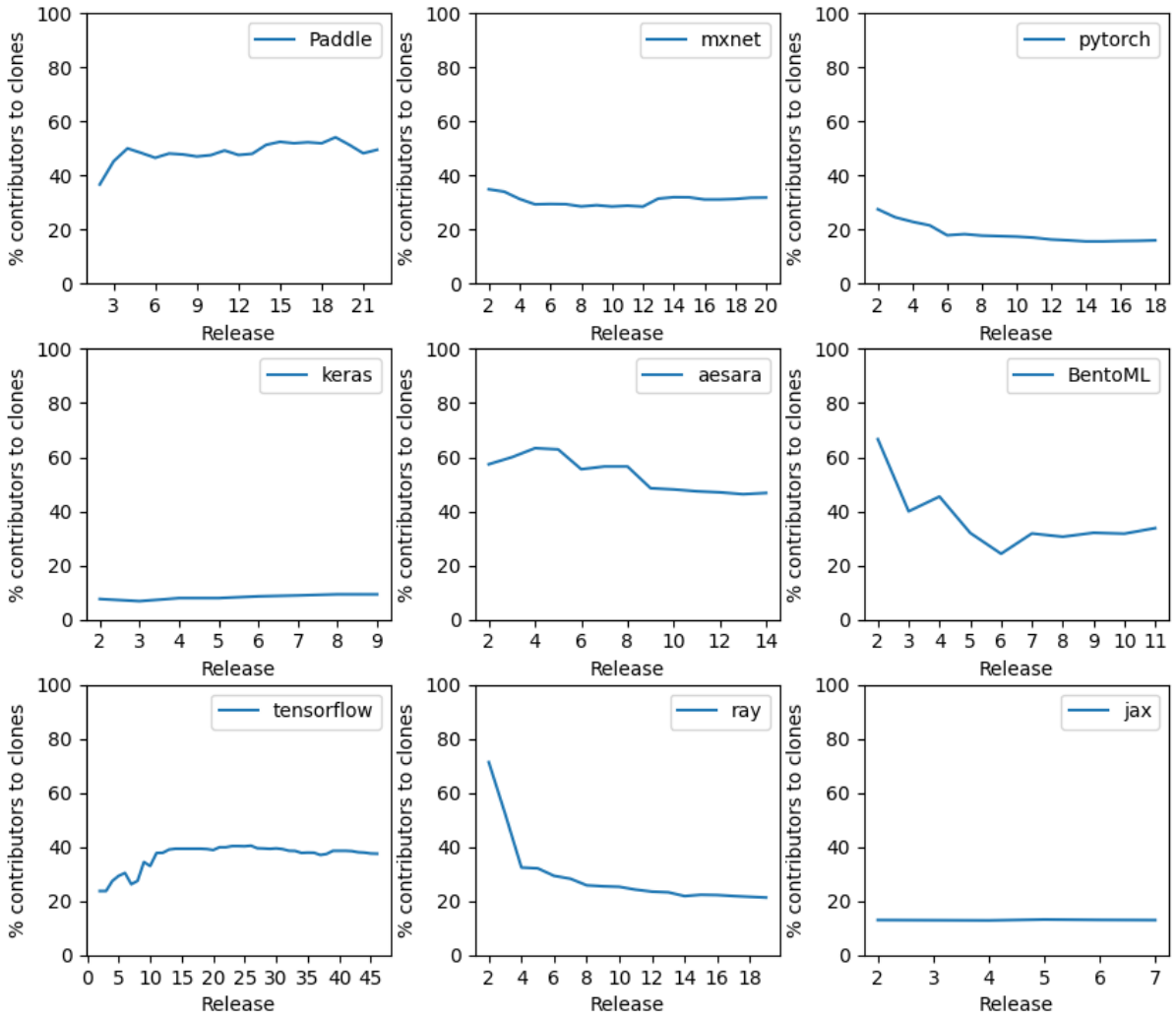

Figure 4.6: The evolution of code clone community size over releases.

Summary of RQ4.1

Long-term code cloning trends exhibit some common and distinct characteristics. The decline in the clone coverage in DL frameworks over time observed in the *"Decreasing"* and *"Rise and Fall"* trends is attributed to 1) a decrease in the cloned code size or 2) a slow increase in the cloned code size compared to a rapid expansion of the codebase. All long-term trends are prone to bugs, and the bug-fixing activities are consistent across the lifespan of the frameworks of all the trends. However, the framework belonging to the *"Serpentine"* trend, i.e., *Paddle* and *MXNet*, could be more susceptible to bugs as they present a higher percentage of bug-proneness across the releases indicated by a higher commit-fixing. *"Thick"* clones present a higher bug-proneness as compared to *"thin"* clones across all the long-term trends. Regarding the community size involved in clone activities, it remains consistent through the lifetime of the frameworks, with exceptions in *BentoML* and *Ray*, where original authors dominate clone maintenance and the community size

4.3.2 RQ4.2: What are the characteristics of within-release code cloning patterns and do these patterns contribute to the overarching long-term trends in code cloning?

Motivation. In RQ4.1, we have investigated the characteristics of long-term clone trends over releases and observed that every trend has distinct characteristics, such as high susceptibility of the *"Serpentine"* trend to bug-proneness. In this RQ, we are interested in investigating the short-term clone patterns within each individual release. Our goal is to discern the impact of these patterns on long-term cloning trends and identify the characteristics of the within-release code cloning patterns. More specifically, we want to understand the factors influencing the evolution of cloned code size and provide insights into the dynamics of development practices within individual releases. In addition, the gained knowledge about within-release patterns, when aggregated over multiple releases, could contribute to understanding how development practices within releases influence clone evolution trends over time. By uncovering these relationships, we seek to formulate insights to enhance the efficiency and maintainability of DL frameworks within the code clone context.

Approach. After we obtain the three within-release code cloning patterns *"Ascending"*, *"Descending"*, and *"Steady"*, we aim to obtain a chronological evolution of how the within-release pattern contributed to the long-term trends. Therefore, we construct, for every DL framework, the evolution sequence of within-release patterns over the framework releases. Table 4.5 depicts the obtained chronological pattern sequences. P_a, P_d and P_s denotes an *"Ascending"*, *"Descending"*, and *"Steady"* pattern respectively.

Our approach to building the regression models comprises the following three steps:

- *Step1: Correlation and redundancy analysis of the independent variables.* we collect 37 metrics as detailed in section 4.2.4. The presence of correlated metrics might affect the performance of the model [129], therefore, we apply varclus[25] function in R to detect the existence of collinearity and exclude one metric of any pair of metrics that achieves a coefficient of 0.7 [180]. Then, we calculate the Variance

[25]https://search.r-project.org/CRAN/refmans/Hmisc/html/varclus.html

Inflation Factors (VIFs) using the VIF[26] function in R for each independent variable to detect multicollinearity issues. We exclude all independent variables with VIF values above 5 because it indicates the existence of multicollinearity [57, 290].

- *Step 2: Constructing the models.* We construct three different regression models for each pattern shown in Figure 4.3: $Model_{Ascending}$, $Model_{Descending}$, and $Model_{Steady}$. For a $Model_i$, we assign to each snapshot revision, i.e., commit, the value 1 if the associated within-release pattern is i and 0 otherwise. For example, for $Model_{Ascending}$, we assign 1 to the commits that are clustered in the release belonging to the *"Ascending"* pattern. Since our dependent variable is binary, we follow the existing work [138, 204, 304] and employ logistic regression models. In particular, we fit a binomial logistic regression model using the glm[27] function provided by the state R package.

- *Step 3: Evaluating the models and assessing the significance of the independent variables.* After constructing the models, we use the Area Under the Receiver Operating Characteristic Curve (AUC) [159] to assess the predictive power of the created logistic models. A predictive model is deemed promising if the AUC-ROC is 0.7 or higher, with 1 signifying perfect predictive power [91, 96]. Next, we employ the ANOVA test [217] to assess the significance, measured in terms of (χ^2) for each independent variable in our model. The (χ^2) values for each variable are calculated as a percentage of the total (χ^2) values for all variables. We use upward and downward arrows to signify direct and inverse relationships between independent and dependent variables, respectively.

Results. Long-term descending trends, i.e., *"Decreasing"* and *"Rise and Fall"*, in addition to *"Stable"* trends consistently exhibit *"Steady"* within-release patterns. As we notice in Table 4.5, the frameworks belonging to the long-term descending trends, i.e., *Keras*, *Aesara*, *BentoML*, *PyTorch*, *JAX*, *Ray* and *TensorFlow* are characterized by a consistent and consecutive repetitive stable pattern P_S. These

[26] https://www.rdocumentation.org/packages/regclass/versions/1.6/topics/VIF
[27] https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/glm

stable patterns constitute over 30% of the total within-release patterns of each framework. This observation shows a connection between the two temporal scales and how the patterns of code clones within releases, i.e., short-term, have long-term implications. Hence, we proceed next with understanding the characteristics of the within-release code clone development practices.

Table 4.5: Within-release patterns. P_a, P_d and P_s denote an *"Ascending"*, *"Descending"*, and *"Steady"* pattern respectively.

DL framework	**Within-release chronological patterns**
MXNet	$P_a, P_a, P_a, P_d, P_d, P_d, P_d, P_d, P_d, P_d, P_a, P_a, P_a, P_a, P_a, P_a, P_a, P_a, P_d, P_d$
Paddle	$P_s, P_a, P_a, P_a, P_d, P_d, P_d, P_d, P_d, P_d, P_d, P_a, P_d, P_a, P_s, P_a, P_a, P_a, P_a, P_a, P_a$
Keras	$P_a, P_a, P_a, P_a, P_s, P_s, P_s, P_s$
Aesara	$P_a, P_a, P_a, P_a, P_a, P_a, P_a, P_a, P_s, P_s, P_s, P_s$
BentoML	$P_a, P_a, P_d, P_d, P_d, P_d, P_a, P_a, P_s, P_s, P_s$
PyTorch	$P_d, P_d, P_d, P_d, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s$
JAX	P_s, P_s, P_s, P_s, P_s
Ray	$P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s$
TensforFlow	$P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s$

The $Model_{Descending}$ and $Model_{Steady}$ demonstrate the most robust explanatory capability among the three constructed models. Table 10 provides an overview of our created models. The logistic regression models for $Model_{Descending}$ and $Model_{Steady}$ exhibit the highest AUC values at 0.92, whereas $Model_{Ascending}$ achieved an AUC of 0.89. Each model highlights a distinct attribute with the greatest explanatory strength, as shown in Table 4.6 (due to the space constraint, the table shows only the top five significant features for all three constructed models). Further insights into the analysis of these constructed models are detailed below.

Table 4.6: A summary of the analysis of the constructed three regression models.

Metrics	Coef.	χ^2(%)	Pr(¡χ^2)	Sign.	Relationship
			$Model_{Ascending}$		
# of committers to a clone pair	-3.62	39.67	$< 2.2e^{-16}$	***	↘
complexity of abstract class	7.74	23.90	$< 2.2e^{-16}$	***	↗
size of cloned code (med)	4.98	21.56	$< 2.2e^{-16}$	***	↗
# clones siblings (max)	-5.30	16.99	$< 2.2e^{-16}$	***	↘
clone life (# commits)	9.50	4.85	$< 2.2e^{-16}$	***	↗
			$Model_{Descending}$		
# changes to clone pairs	-9.08	24.86	$< 2.2e^{-16}$	***	↘
complexity of abstract class	-18.58	24.63	$< 2.2e^{-16}$	***	↘
size of cloned code (med)	8.00	23.83	$< 2.2e^{-16}$	***	↗
similarity	-5.18	14.58	$< 2.2e^{-16}$	***	↘
cyclomatic complexity	3.78	11.94	$< 2.2e^{-16}$	***	↗
			$Model_{Steady}$		
# of declaration statement (med)	-8.54	36.56	$< 2.2e^{-16}$	***	↘
# abstract classes (med)	4.36	25.61	$< 2.2e^{-16}$	***	↗
# changes to clone pairs	3.34	18.08	$< 2.2e^{-16}$	***	↗
# of clones siblings (med)	45.89	6.85	$< 2.2e^{-16}$	***	↗
# of unique file	-2.70	5.78	$< 2.2e^{-16}$	***	↘

"Ascending" code cloning pattern analysis

- **The decreased involvement of committers in the clone pairs clone genealogy is associated with a larger cloned code size**. As shown in Table 4.6, the feature number of committers to a clone pairs accounts for the highest (χ^2) in the $Model_{Ascending}$, suggesting that the fewer is the number of committers changing a clone pair, the higher is the likelihood of cloned code size increase. This could be explained by the fact that fewer committers modifying the clone pairs know the code well and are propagating the changes while maintaining the code clones. When new committers make changes to existing clones, the probability of applying the changes to all clones' copies decreases as they might not be aware of its presence, leading to inconsistent copies, hence a decreased cloned code size. For example, in the *PyTorch* framework, the functions *acc_ops_max_pool2d* and *acc_ops_avg_pool2d* were

modified consistently through the three commits *239b38268b*[28],*0d8a8a2e41*[29] and *e23827e6d6*[30] by a single committer which resukted in stable clone sizes. In contrast, the independently modified *AdagradOptimizer* and *AdamOptimizer* cloned functions exhibited inconsistent growth due to the involvement of multiple committers. For instancee, a first commit *05542f6222*[31] made by one committer made changes to *AdagradOptimizer* only leading to an inconsistent state. Hence, a decrease in the clone size. Then, another commit by another committer *812bc1dde6*[32] propagated the change to the second optimizer, which led a consistent state again. **In addition, we highlight that the positive correlation (in the same model) between clone pairs' longevity (i.e., clone life in terms of the number of commits and the increased cloned code size) suggests that the clone pairs are either changed consistently or not modified**.

- **Code clones size tends to increase when abstract classes exhibit higher complexity**. Cloned code size positively correlates to the complexity of the cloned code in abstract classes. It is the second most explanatory factor in $Model_{Ascending}$. The rationale is likely rooted in the observation that code cloning often occurs under conditions of higher code complexity within a class. When a class is more complex, it may contain a greater number of methods, variables, or functionalities. In such intricate classes, developers might encounter challenges related to code reuse or modularization. As a result, they may resort to code cloning as a quick solution. This also aligned with the fact that the increased cloned code size is positively correlated with a larger cloned code size as it is the third most significant explanatory factor, as shown in Table 4.6.

"Descending" code cloning pattern analysis

- **Code clones pairs in *"Descending"* cloned code size pattern exhibit fewer**

[28]https://github.com/pytorch/pytorch/commit/239b38268b
[29]https://github.com/pytorch/pytorch/commit/0d8a8a2e41
[30]https://github.com/pytorch/pytorch/commit/e23827e6d6
[31]https://github.com/pytorch/pytorch/commit/05542f6222
[32]https://github.com/pytorch/pytorch/commit/812bc1dde6

changes. The $Model_{Descending}$ indicates a significant association between the number of changes to clone pairs and the cloned code size. When fewer modifications are made to existing pairs of code clones, it suggests that these segments of code remain relatively stable over time. This stability implies that developers are refrained from making substantial alterations or additions to the existing code clones.

- **Reduced complexity in abstract classes is associated with less reliance on code cloning**. As depicted in Table 4.6, a reverse association exists between the complexity of the abstract class where the code clone is originated and the cloned code size. This finding aligns with the results obtained in $Model_{Ascending}$ where an increased complexity is associated with the increased cloned code size. This suggests developers probably need to engage in less copying and pasting of code when dealing with simplified code structures in abstract classes.

"Steady" code cloning pattern analysis

- **Code optimization strategies, such as reducing the number of declaration statements and incorporating modular structures, are significantly linked to cloned code size in "Steady" cloned code patterns**. Table 4.6 shows an inverse association between the number of declaration statements and the stable cloned code size in $Model_{Steady}$. A lower count of declaration statements within code clones suggests a more concise and modular structure, indicating a potential effective maintenance strategy where similar functionalities across the code base share the same set of globally declared variables. This finding aligns with the relevance observed in the descending code clone pattern analysis, where an increased number of abstract classes, i.e., the second most significant factor, is also associated with stable cloned code sizes. Smaller and modular code segments are commonly linked to better maintainability and ease of understanding, highlighting the significance of cohesive code structuring practices.

Summary of RQ4.2]

Within-release code cloning patterns impact the long-term code cloning trends. For instance, long-term descending trends, i.e., "Decreasing" and "Rise and Fall," in addition to "Stable" trends, consistently exhibit "Steady" within-release patterns. Within-release patterns also exhibit distinct characteristics. Our results demonstrate that *"Ascending"* code cloning pattern is associated with decreased committer involvement in clone pairs and increased cloned code size, suggesting that fewer committers may lead to a higher likelihood of cloned code size increase. In the *"Descending"* pattern, clone pairs with reduced cloned code size exhibit fewer changes, indicating stability over time, and simplified code structures in abstract classes are associated with less reliance on code cloning. Lastly, the *"Steady"* pattern links code optimization and smaller declaration statements count to stable cloned code sizes, emphasizing the significance of cohesive code structuring practices. Our work shows the characteristics of the within-release clone evolution, which helps developers prioritize their actions based on their within-release pattern.

4.3.3 RQ4.3: How do code clones manifest and evolve across different DL frameworks?

Motivation. Different DL frameworks support distinct features and adopt different design philosophies. For example, while *Jax* focuses on high-performance numerical computing, *PyTorch* is known for its dynamic computation graph. However, some DL frameworks share common functionalities. For example, *TensorFlow*, *PyTorch*, and *MXNet* exhibit overlapping functionalities, such as in neural network definition and training. The convergence of functionalities among these frameworks can result in comparable code patterns for similar tasks, potentially leading to code clones, as developers may employ similar constructs and operations. In this RQ, we conduct a cross-framework clone detection to identify and analyze similarities in code across different DL frameworks, and we track the evolution of the cloned functionalities. This is crucial for understanding how common functionalities of code clones manifest in DL, aiding in the development of

standardized practices and fostering collaborative initiatives within the DL community.

Approach. The investigation process of cross-framework code clones involves the following steps:

Step 1: Building quarterly snapshots and detecting cross-framework code clones. In this step, we organize the commits of each DL framework into quarterly groups and select snapshots corresponding to the latest commit in each group for analysis. We choose the quarterly interval as it provides a fine-grained analysis of the cross-framework clone evolution. Then, we run the clone detection on the snapshots of the frameworks corresponding to the selected commits. As explained in section 4.2.2, we use SourcererCC to detect the cross-framework clones every quarter, generating a comprehensive list of cross-framework clone file pairs for every quarter.

Step 2: Constructing the time series for the evolution of cross-framework clones. Following the cross-framework clone detection, we construct a time series for the quarterly evolution of cross-framework clones. This time series consists of data points representing the count of cross-framework clone files on specific dates. It spans 28 quarters, offering a dynamic record of how cross-framework clones evolve across DL frameworks over time.

Step 3: Investigating cross-framework file-level clone pairs manually. To provide deeper insights into the evolution of cross-framework clones in DL frameworks, we conduct a manual analysis. In particular, we manually investigate the identified cross-framework file-level clone pairs for each snapshot. First, we examine for each file clone the *repository location*, such as the path or module (i.e., logical unit of code within the software framework), to identify the context in which the clones exist. Secondly, for every file pair, we check the *source code* to identify the functionality of these files. Lastly, we track the *code-changing activities* related to these clone files to better understand how they evolve over time and the reason they cease to be clones.

Results. DL frameworks present two main categories of cross-framework file-level code clones: the *functional code clones* and the *architectural adaptation clones*. *Functional code clones* include the clones where specific files, pertaining

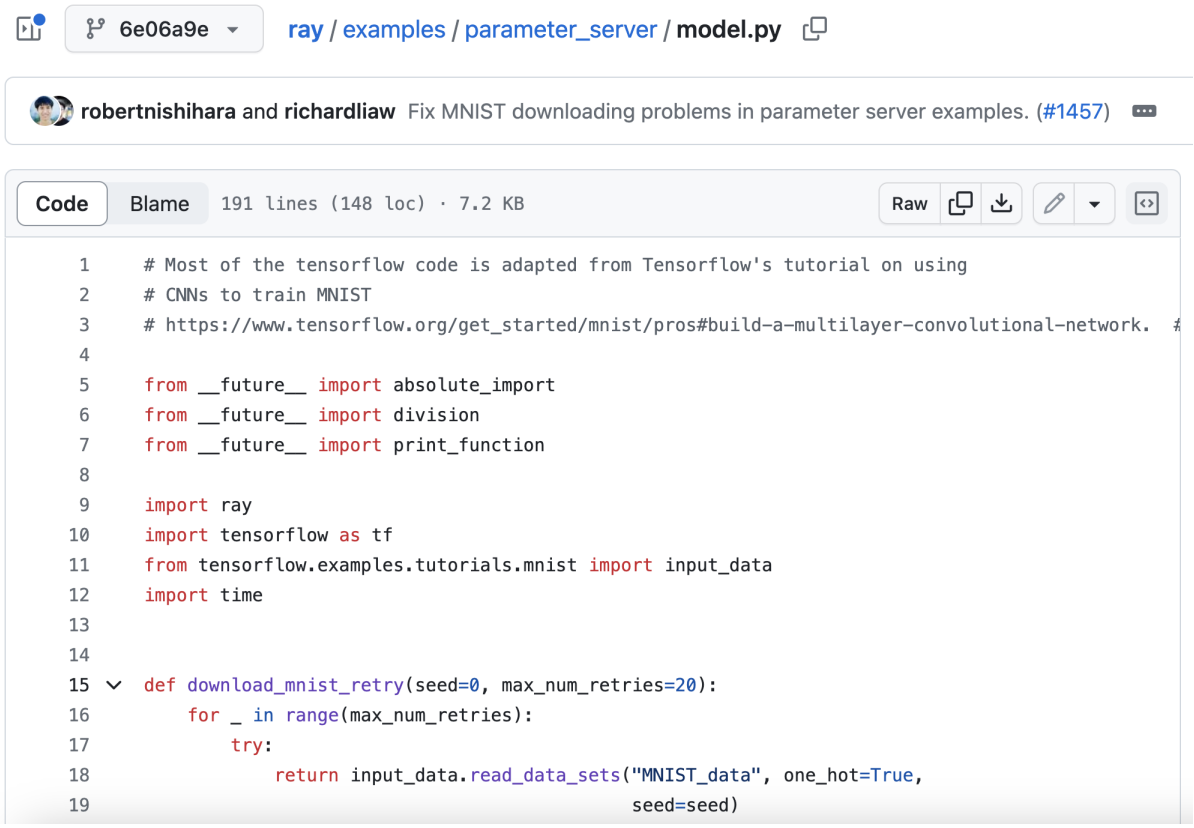

Figure 4.7: Example of a functional code clone instance from TensorFlow to Ray.

to distinct functionalities, are replicated or adapted within a DL framework. Figure 4.7 illustrates an example[33] of the clone belonging to this category where some code was adapted from a TensorFlow tutorial for training CNNs on the MNIST dataset to be included in the *Ray* framework. *Architectural adaptation clones* represent the adaptation and integration of an entire module from one DL framework into another resulting in an architectural adaptation between frameworks. For example, we notice that the increase in clones in March 2018 was due to the integration of the *Keras* module into *TensorFlow*. In particular, the layer and applications modules of Keras were integrated into TensorFlow as exact duplication. We also observe that even though both categories of clones are present in DL frameworks, the *architectural adaptation clones* represent the larger category among all file cross clones in DL frameworks.

Functional code clones in DL frameworks belong to seven main categories

[33]https://github.com/ray-project/ray/blob/6e06a9e338e1045fa0ba73b366bb78a2c7f0fef8

Communication Interface, Distributed Training in TensorFlow, Python Object Serialization, Efficiency in Python code, Version Control, Deep Learning Architectures,* and *Probability and Statistics. We describe below each of the categories, and we include in Table 4.7 the distribution of the file clone pairs across the seven categories with the functionalities of the existing clones and the range of the lifetime of the file clone pairs.

- **Communication Interface:** Clones related to how DL frameworks handle data exchange or interaction, such as the protocol buffers (protobuf) messages and services for communication with a *BentoML* repository[34].

- **Distributed Training in TensorFlow:** Clones associated with distributed training in the *TensorFlow* DL framework, such as the gradient reduction (all-reduce) in distributed training using TensorFlow[35].

- **Python Object Serialization:** Clones related to the serialization of Python objects such as CloudPickle serialization[36].

- **Efficiency in Python code:** Clones related to enhancing efficiency in Python code, such as the LazyLoader class[37].

- **Version Control:** Clones pertaining to version control mechanisms, to manage and track changes in code versions such as Versioneer[38].

- **Deep Learning Architectures:** Clones related to deep learning architectures, suggesting shared design patterns and functionalities, such as Convolutional Neural Networks (CNN)[39].

- **Probability and Statistics:** Clones related to probability and statistics, revealing

[34]https://github.com/bentoml/BentoML/blob/22f1e42084/bentoml/yatai/proto/repository_pb2.py
[35]https://github.com/keras-team/keras/blob/967817c22/keras/distribute/collective_all_reduce_strategy_test.py
[36]https://github.com/bentoml/BentoML/blob/4d201c4cd9/bentoml/utils/cloudpickle.py
[37]https://github.com/tensorflow/tensorflow/blob/ab4762acba/tensorflow/python/util/lazy_loader.py
[38]https://github.com/aesara-devs/aesara/blob/e92e5e309a/versioneer.py
[39]https://github.com/keras-team/keras/blob/967817c226/keras/applications/resnet_v2.py

common approaches in implementing statistical concepts, such as normal distribution[40].

As we notice, *Probability and Statistics* and *Deep Learning Architectures* represent the top two common categories constituting approximately 70% of the file clone pairs. This insight suggests an overlap in specific functionalities and design patterns across different DL frameworks, emphasizing commonalities in the implementation of key features such as Convolutional Neural Network (CNN) and Residual Network for the *Deep Learning Architectures* category and Normal Distribution for the *Probability and Statistics* category. We also notice that the identified DL framework pairs associated with each category reveal cross-framework code cloning instances, and 8 out of the nine studied DL frameworks present at least one cross-framework file-level clone.

The computed lifetimes of cross-framework file-level pairs vary across categories, ranging from 0 to 8 quarters, i.e., three-month period within a calendar year. We compute the lifetimes of cross-framework file-level clone pairs to reflect how long a particular clone pair persists within a specific functional category over time. As shown in Table 4.7, some categories, such as *Communication Interface*, experience short-lived cloning instances lasting less than a quarter, while other categories, such as *Version Control* and *Probability and Statistics* persist over a more extended period between 5 and 8 quarters. This variation suggests that code within different functional categories evolves at different rates. Some categories, such as *Version Control* and *Probability and Statistics*, may involve functionalities with more stable and fundamental design patterns less prone to frequent changes, resulting in longer lifetimes. On the other hand, categories like *Communication Interface* may experience frequent updates or changes, leading to shorter lifetimes. These insights highlight areas where stability or adaptability is crucial when managing and maintaining code reuse.

[40]https://github.com/tensorflow/tensorflow/blob/38df5d8ef4/tensorflow/python/ops/distributions/special_math.py

Table 4.7: The distribution of clone pairs across framework functional code clones.

Functional code clone categories	# of pairs	lifetime range # of file clone pairs	DL framework pairs
Communication Interface	1	0 quarter	(*BentoML*, *Paddle*)
Distributed Training in TensorFlow	1	2 quarters	(*Ray*, *TensorFlow*)
Python Object Serialization	1	1 quarter	(*Aesara*, *BentoML*)
Efficiency in Python	2	3 to 4 quarters	(*BentoML*, *TensorFlow*), (*Paddle*, *PyTorch*)
Version Control	2	8 quarters	(*Ray*, *BentoML*)
Deep Learning Architectures	5	0 to 2 quarters	(*Ray*, *TensorFlow*), (*Ray*, *PyTorch*)
Probability and Statistics	8	5 quarters	(*JAX*, *TensorFlow*), (*MXNet*, *PyTorch*)
Total	20		

Functional code clones across DL frameworks undergo a gradual disappearance, attributed to functionality evolution, code divergence, function deprecation and framework restructuring. Over time, code functionality evolves and leads to a divergence from the initial commonality. For instance, an initial commonality, such as a normal distribution, may undergo changes, as observed in *Jax*, where the file evolves to encompass additional functionalities like special functions, such as the gamma function, leading to a disappearance of the code clone across frameworks. Code divergence is another reason often resulting from adopting third-party libraries. For example, TensorFlow's shift to a new third-party library for object serialization in Python diverges from the code used in *BentoML*, which continues to use Cloudpickle. Additionally, function deprecation also leads to the decline of functional clones as frameworks undergo updates and discontinue specific functionalities. Furthermore, framework restructuring and the creation of new separate modules lead to the removal of certain functionalities. For example, the removal of Residual Network (ResNet) implementation to a standalone module outside of *PyTorch* led to the disappearance of clones between *PyTorch* and *Ray*.

***Architectural adaptation clones* between Keras and *TensorFlow* are performed gradually over multiple releases.** *TensorFlow* and *Keras* are the only two DL frameworks among the studied ones that present *architectural adaptation clones*. The evolution of cloned files between these two frameworks suggests a dynamic and evolving integration process with periods of stability, optimization, and significant updates. For instance, between March 2018 and December 2019, we observe a relatively

stable number of file clone pairs, ranging from 73 to 42 pairs. This suggests an ongoing integration process where *TensorFlow* and *Keras* modules are aligning and adapting. For instance, in release *TensorFlow v1.13.1*[41], a new functionality analogous to "tf.register_tensor_conversion_function" was added. Then, through the module adaptation process, these two frameworks present a decrease in the cloned file that is due to significant refactoring efforts within the *Keras* library, such as the exclusion of the two modules "applications" and "preprocessing" from Keras as described in the release *Keras 2.2.0*[42] notes. At a later stage, *TensorFlow* and *Keras* present a very small number of clones (i.e., between 0 and 5 pairs). The decrease in cloned code file pairs is attributed to optimization efforts within *TensorFlow* focusing on refining and optimizing the previously integrated *Keras* code while the development is discontinued in *Keras*, marking the full transition to the *TensorFlow* codebase.

Summary of RQ4.3

DL frameworks present *functional* and *architectural adaptation* code clones. *Functional code clones* in DL frameworks span seven categories and gradually disappear due to functionality evolution, code divergence, function deprecation, and framework restructuring. The most frequent categories, *Probability and Statistics* and *DL Architectures*, suggest commonalities in key features across different DL frameworks. This implies opportunities for collaboration, code reuse, and understanding best practices in DL framework development. Architectural adaptation clones represent the integration of one framework module into another, such as the integration of *Keras* into *TensorFlow*.

4.4 Implications

In this section, we discuss the possible implications of our findings that could be useful to practitioners and researchers.

[41]https://github.com/tensorflow/tensorflow/releases/tag/v1.13.1
[42]https://github.com/keras-team/keras/releases/tag/2.2.0

Fostering collaboration in clone maintenance. In RQ4.2, our results show that the number of committers to a clone pair is a significant factor influencing cloned code size increase, as indicated by the highest χ^2 in $Model_{Ascending}$, implies that minimizing the number of committers involved in modifying a clone pair may contribute to more consistent and well-maintained code clones. This suggests that a smaller team with a better understanding of the codebase and, more specifically, the cloned code is more effective in propagating changes consistently, thus reducing the likelihood of inconsistent copies. Researchers can leverage these insights to explore strategies for code clone management and team collaboration. In fact, the observation in RQ4.3 that there exists a consistently low community size involvement in code clones complements the findings of RQ4.2 and underscores the need to encourage broader community engagement in clone-related activities. Project maintainers and open-source community leaders should proactively foster collaboration, knowledge-sharing, and contribution initiatives in clone maintenance. For example, enhancing the documentation can promote a wider understanding of clone-related tasks and attract diverse contributors [294], mitigating the reliance on original authors and enhancing the sustainability and robustness of DL framework development communities.

Simplifying abstract class complexity. Addressing code complexity in abstract classes is crucial to mitigating code clone proliferation, as complexity emerges as the second most explanatory factor in $Model_{Ascending}$ and $Model_{Descending}$. By emphasizing code modularization and reuse strategies, developers can potentially reduce the need for code cloning and enhance overall code maintainability. This insight underscores the importance of efficient coding practices and architectural design for DL frameworks to minimize the challenges associated with class structures. For instance, previous studies [195, 198] demonstrate that code clones can lead to a lack of good inheritance structure or abstraction.

Optimizing code maintainability through effective maintenance strategies. As demonstrated in $Model_{Stable}$, the stable cloned code size is associated with effective maintenance strategy in code clones, such as the reduction of the count of declaration

statements and the promotion of a more concise and modular structure by using abstract classes. Hence, developers should prioritize the creation of smaller, modular code segments to enhance maintainability and ease of understanding of clone code.

Addressing quality challenges in *"thick clones"*. In RQ4.1, we observe that a significant number of bugs, i.e., more than 50% of bugs for each DL framework, are associated with *"thick clones"*. This observation emphasizes the potential risks posed by larger and more complex clones, which highlights the importance of thorough code reviews, testing, and bug tracking in sections of code characterized by high clone thickness. Prioritizing bug detection and resolution in *"thick clones"* is crucial for maintaining software quality and reliability.

Standardizing code practices with DL frameworks. The results of RQ4.3 demonstrate that some cross-framework clones are prevalent in specific functional codes, namely *Probability and Statistics* and *Deep Learning Architectures*. These results imply a potential for standardized practices or shared code components in some DL common functionalities, such as architectural design, offering opportunities for collaboration, code reuse, and a deeper understanding of best practices in DL framework development.

4.5 Threats to Validity

Construct validity relates to a possible error in the data preparation. There is no established tool available to identify all existing DL frameworks. In our study, we manually curate DL frameworks from GitHub which could lead to errors. To address this potential concern, we conduct cross-validation on our selected frameworks with the most recent studies [49,113,119,124,139,144,249,285,286] on DL frameworks. Our study stands as one of the most extensive DL framework comparative investigations to date, encompassing a substantial number of Python-based DL frameworks, i.e., nine frameworks, in a single comprehensive analysis, which greatly enhances its contribution to the understanding of this field. Therefore, we assert a strong construct validity.

Another potential threat could be related to the choice of the clone detector in our

study. We employed NiCad to identify within project code clones within each DL framework snapshot, as it has been a common choice in previous research [124,188]. The choice of the detection tool might introduce certain biases or limitations. To address this, we use a popular detection tool, named Nicad, which is evaluated to achieve higher precision in near-miss clone (i.e., type 3) detection [234, 259] when compared to other tools in the evaluation and comparative study of current clone code detection methods. In addition, Nicad supports Python language and has been used in recent studies for code detection in Python [283] and DL related systems [79, 316]. For cross-project clone detection, we adopt SourcererCC tool as it has been proven to outperform other state-of-the-art tools in terms of scalability [234]. Additonally, we acknowledge that NiCad's configuration settings can influence the outcomes of our clone detection process. To address this concern, we tailored the configuration settings based on the methodologies outlined in prior studies [124, 188], which have demonstrated to achieve high precision and recall rates.

Lastly, we adopt the heuristic introduced by Barbour et al. [189] to investigate the code clone bug proneness to identify bug-fix commits. This involves utilizing a predefined set of keywords associated with bug fixes. If any of these keywords are found within a commit's message, it is categorized as a bug-fix commit. This heuristic might lead to inaccurately classified commits. However, it has been successfully used in multiple previous studies from the literature [24, 124, 188, 191], and proven to achieve satisfyingly accurate results.

Internal validity relates to the concerns that might come from the internal methods used in our study. In RQ4.2, we offer explanations for these observed results related to practitioner development practices. However, we refrain from asserting causation and instead provide observations and correlations.

External validity relates to the potential of generalizing our study results. Given that most DL frameworks are developed in Python, our study primarily concentrates on frameworks implemented in Python. Consequently, we cannot ensure the broad applicability of our findings to frameworks developed in different programming languages, such as C++ and Java. However, we have made an effort to encompass a wide range of DL frameworks of different sizes and functionalities.

4.6 Summary

Given the pivotal role of DL frameworks in the rapidly growing field of artificial intelligence and machine learning, its software quality is crucial for the development of DL-based applications. Code clone is a common coding practice that can potentially adversely affect software maintainability. We conduct an empirical analysis of nine popular DL frameworks, including TensorFlow, Paddle, PyTorch, Aesara, Ray, MXNet, Keras, Jax, and BentoML, to reveal insights into the long-term code clone trend over releases, and within-release clone patterns, i.e., short-term, and their implications. In addition, we also conduct a file-level cross-framework clones analysis to identify the cross-functionalities of cloned code within the frameworks.

Our results suggest that:

- DL frameworks follow four distinct trends, i.e., *"Serpentine"*, *"Rise and Fall"*, *"Decreasing"*, and *"Stable"*, in the evolution of clones.
- DL frameworks adopt three distinct within-release patterns, i.e., *"Ascending"*, *"Descending"*, and *"Steady"*.
- Short-term code cloning practices impact long-term cloning trends.
- The cross-framework code clone investigation reveals the presence of *functional* and *architectural adaptation* file-level cross-framework code clones across the nine studied frameworks.

We provide a replication package[43] for our approach, including the list of DL frameworks, the genealogy results obtained, and the scripts necessary to reproduce our study.

[43] https://github.com/mia1q/code-clone-DL-frameworks/

Chapter 5

Feature Comparison for Competing Mobile Apps

In this chapter, we propose an approach named FeatCompare to automatically mine and compare high-level feature-based user opinions for competing mobile apps from user reviews. Section 5.1 provides an introduction and the motivation for our study. Section 5.2 details the design of our work. Section 5.3 reports the results of our experiments. Section 5.4 discusses the implications of our findings. Section 5.5 examines the threats to validity. Finally, Section 5.6 summarizes the chapter and suggests directions for future research.

5.1 Introduction

Mobile applications (apps) have become an integral part of our daily activities. To get an app, users visit a mobile app store, such as the Google Play Store, and search for an app that fulfills their needs [156]. After downloading an app, users can post reviews to express their opinions on the downloaded app. Table 5.1 shows three sample reviews from two popular weather apps. In this example, two users of the *"AccuWeather"* app complain about the low accuracy of the forecast given by the app. In contrast, one user of the *"Weather Live"* app praises the accurate forecast of weather provided by the app.

User reviews are crucial for mobile app developers [92, 213] as it has been reported that mobile users are not likely to download an app with an average rating that is less than three stars [175]. To maintain a high average rating, developers need to rapidly implement the requested features and fix the reported bugs mentioned in user reviews

Table 5.1: User reviews of the *"AccuWeather"* app and the *"Weather Live"* app.

App	User reviews	Rating	High-level feature
AccuWeather	*"**Terrible accuracy**. Keeps telling me it's snowing. No snow. Not a flake."*	★☆☆☆☆	Forecast accuracy
AccuWeather	*"**Inaccurate** weather **forecasts**. Says its partly sunny where it is actually overcast"*	★★☆☆☆	Forecast accuracy
Weather Live	*"**Excellent accuracy** in forecasting the weather for my area"*	★★★★☆	Forecast accuracy

[213]. The manual process of reading a large number of user reviews can be tedious, expensive, and error-prone. Hence, in literature, researchers have proposed approaches to help mobile app developers gain insights from their user reviews by automatically extracting features mentioned in reviews [92, 94, 115, 131, 245], summarizing [80, 276], categorizing [173, 178, 214, 274], and prioritizing user reviews [50, 83, 137].

However, given the highly competitive nature of the mobile app market, analyzing the reviews of a specific app is not enough for developers of the app to succeed. El Zarif et al. [76] find that user complaints can lead to user churn, which means users can switch to start using the competitors' products. Developers need to expand their focus beyond their own apps and understand how users perceive their apps with respect to their competitors' apps [10, 208, 236, 274] via a competitor analysis [246]. Conducting user-based competitor analysis support developers in eliciting requirements requested by users [236, 274], planning the app releases [208] and performing apps enhancements [10]. Competitor analysis can be conducted in many ways and for different purposes. In this work, inspired by the existing user review analysis approaches, we specifically explore the opportunity of supporting mobile app competitor analysis via automatically mining useful information from reviews of closely related apps (i.e., a group of *competing apps*). Studies show that reading about app features in the app description section does not provide any information about the quality of the features and how it is perceived by

users [89,170]. Therefore, researchers show interest in techniques that provide developers with suggestions (e.g., bug fixes and feature requests) extracted from user reviews for future app releases [10,274].

Nayebi et al. [207] suggest that accommodating user opinions is crucial for planning the future releases of the apps in the competitive market. Through the feature-level comparison, developers can focus on either improving the most frequently mentioned low rate features (i.e., requests and complaints) [207] or on enhancing features that are less rated (i.e., most negative as compared to the same feature rating of competitor app. For example, for the "Forecast accuracy" high-level features of the "Weather" app shown in Table 5.1, the developer can know how many users report negative opinions about this particular feature and compare it to the same feature rating across competitors along with the number of satisfied users.

Few prior studies [60,151,243,246] have shed light on the potential of extracting useful information, i.e, user opinions (sentiment and rating) associated with specific app *features*, for app comparisons. These app features are fine-grained, and explicitly mentioned in reviews via word pairs such as "*photo edition*" and "*upload video*". However, comparing competing apps on fine-grained features can be challenging and time-consuming as the number of extracted fine-grained features can reach up to hundreds or even thousands per competing group [60, 244]. Instead of mining the fine-grained features, which describe the details of an app feature, we propose to mine and compare user opinions on *high-level features* from reviews of competing apps. High-level features refer to the main functionalities (e.g., video streaming or daily forecast) and the main characteristics (e.g., UI appearance and stability) of an app. One can regard high-level features as semantic clusters of fine-grained features. For instance, the reviews shown in Table 5.1 mention multiple fine-grained features specified by phrases *"Terrible accuracy"*, *"Inaccurate Forecast"*, and *"Excellent accuracy"*. The three fine-grained features can be grouped into one high-level feature, i.e, *"Forecast accuracy"*, indicating the common concern from all sample reviews. By identifying all high-level features from reviews of competing apps, developers can summarize users' attitude on forecast accuracy among all competing apps.

We propose **FeatCompare**, an approach that facilitates the analysis of high-level features across competing apps. To evaluate the effectiveness of our approach, we collected 14,043,999 reviews from 196 popular Android apps. The studied 196 apps are distributed across 20 functionality-similar groups, e.g., weather apps, music player apps, etc. Our experimental results show that GLFE achieves an average precision of 81% and an average recall of 75%, and outperforms ABAE by 14.7%, on 480 randomly selected reviews from five competing app groups. We then conduct a case study on opinions summarized from the studied 196 competing apps. We observe that using FeatCompare to analyzing the reviews of an app, within the context of its competitors, can help app developers spot the potential opportunities for improving their app. A user study involving 107 developers is conducted, and the results show that our FeatCompare approach is useful for app developers to perform competitor analysis.

5.2 Study Design

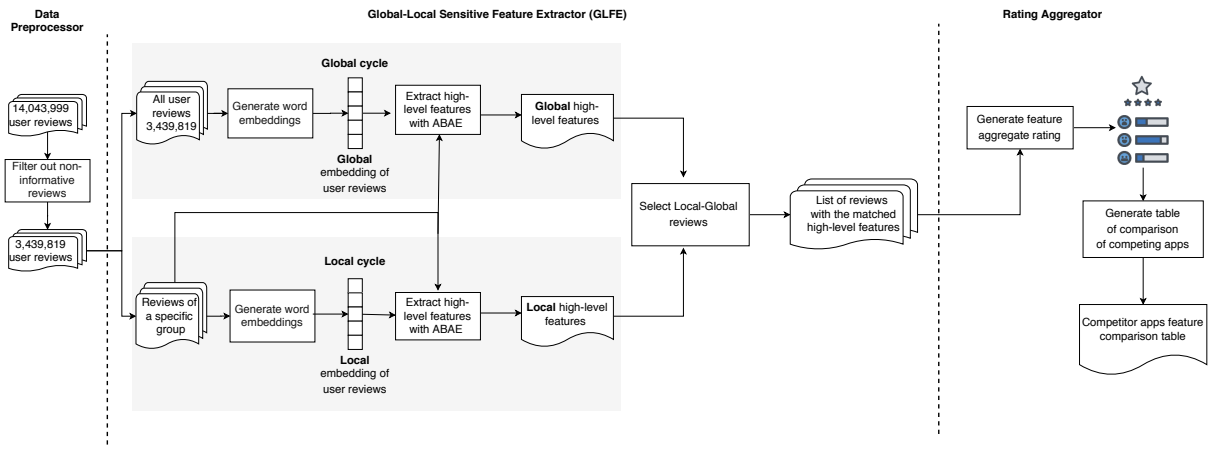

Figure 5.1: FeatCompare three main components.

In this section, we introduce the design of FeatCompare, a neural network-based approach that can identify high-level features and summarize corresponding user opinions for competing mobile apps from app reviews. Notably, FeatCompare is an unsupervised machine learning approach, which means it does not require any manual labeled reviews in training. The only human effort needed in FeatCompare is determining hyper-parameter value and feature names.

As shown in Figure 5.1, FeatCompare contains three components: 1) a *data preprocessor* component that takes as input reviews from multiple competing groups and removes non-informative reviews using AR-Miner [50], 2) an unsupervised neural network-based *feature extractor* component, named **G**lobal-**L**ocal sensitive **F**eature **E**xtractor (GLFE), that extracts high-level features from preprocessed user reviews, and 3) a *review aggregator* component that aggregates ratings for each feature and generates a comparison table summarizing feature-based user opinions for competing apps. GLFE extends a state-of-the-art high-level feature extraction model named Attention-based Aspect Extraction (ABAE) [98] by proposing a threshold mechanism that can identify *global features* (e.g., general software engineering features shared across multiple app groups) and *local features* (e.g., features shared among a specific group of competing apps) discussed in reviews. We elaborate below on each component and on the data collection.

5.2.1 Attention-Based Aspect Extraction (ABAE)

Our work adopts and augments advanced neural network techniques. In this section, we introduce ABAE, the neural network model that we adapt in our approach.

ABAE is one of the state-of-the-art unsupervised neural attention models proposed by He et al. [98] for extracting high-level features (i.e., "aspect" in their paper) from product/service reviews. Prior to ABAE, topic models such as Latent Dirichlet Allocation (LDA) and its variants have been widely applied to user reviews such as Amazon product reviews or Yelp restaurant reviews to extract high-level features [201, 324]. However, researchers find that while LDA can describe high-level features in text corpus fairly well, the mined individual features are of poor quality with unrelated or loosely-related words, which often leads to low accuracy in identifying high-level feature of a specific review [98]. ABAE is then proposed and has been shown to outperform LDA-based approaches. The main idea of ABAE is to learn the semantic meaning of high-level features, reviews, and words as vectors (i.e., embeddings) so that fine-grained features explicitly expressed in reviews can be mapped to their corresponding high-level features.

Figure 5.2 shows an example of the ABAE architecture using a mobile app review.

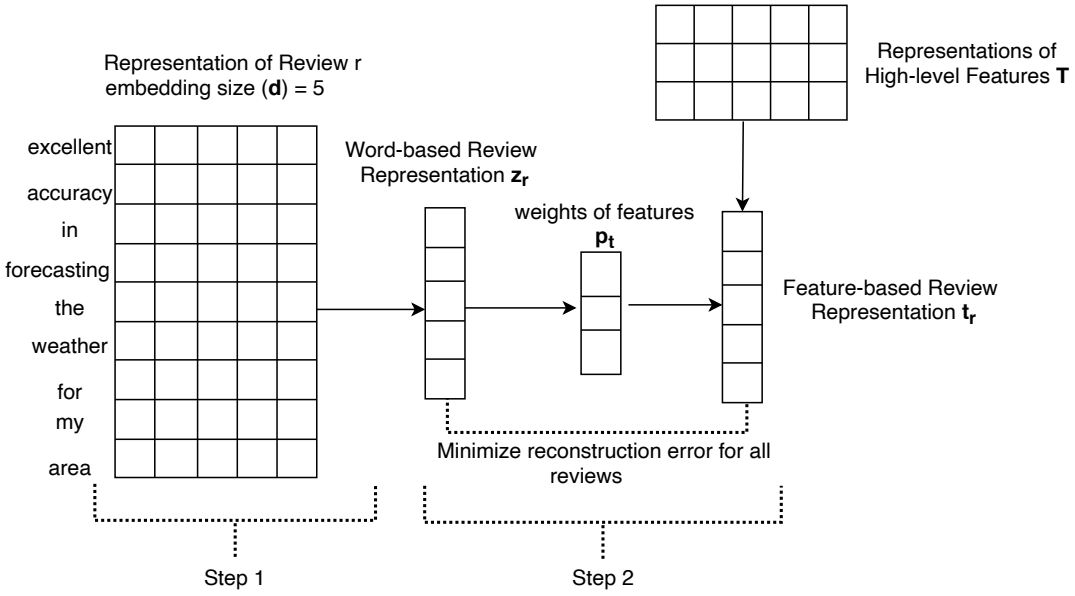

Figure 5.2 An example of ABAE architecture. In the above sample, the size of word embedding is 5, the number of high-level features is 3.

At high-level, ABAE is an autoencoder [275] that can learn the embeddings of reviews and high-level features automatically using only reviews (e.g., "*excellent accuracy in forecasting the weather for my area*"). ABAE requires three input elements: 1) a matrix representing pre-trained embeddings of words, 2) an integer representing the number of high-level features expected to be learned, and 3) a set of reviews. The output of ABAE includes embeddings of reviews and high-level features, probabilities of each review belonging to each of the high-level features.

Given the list of words in a review as input, two steps are performed as shown in Figure 5.2. ABAE first deemphasizes words that are not relevant to high-level features, such as "*the*", "*for*", "*my*" in the example review using an attention mechanism, and constructs a word-based review embedding z_r. Next, ABAE reconstructs the review embedding (t_r) as a linear combination of high-level feature embeddings. All the parameters in ABAE are learned by minimizing the reconstruction loss of the word-based review embedding z_r and the feature-based review embedding t_r, aiming to preserve most of the information of the feature-related words (e.g., "*excellent*" and "*accuracy*") in the embedded high-level features (e.g., "*accurate forecast*"[1]). Next, we introduce the two steps of

[1]Note that all high-level features are hidden, meaning they are represented using embeddings. The semantic meaning of a high-level feature can be identified by searching the most representative words, the embeddings of which are close to the embedding of the high-level feature.

ABAE in detail.

The first step in ABAE computes the embedding $z_r \in R^d$ of a review r that contains n words $w_1, w_2, \ldots, w_n$:

$$z_r = \sum_{i=1}^{n} a_i e_{w_i} \tag{5.1}$$

where e_{w_i} refers to the word embedding e of size d for word w_i in review r. The word embeddings are initialized by applying word2vec [182] over a collection of reviews. word2vec is an unsupervised algorithm that learns meaningful embeddings of words from a text corpus. The weight a_i is conditioned on the embedding of the word e_{w^i} as well as the global context of the review:

$$a_i = softmax(e_{w^i}^T \cdot M \cdot y_r) \tag{5.2}$$

$$y_r = \sum_{i=1}^{n} e_{w^i} \tag{5.3}$$

where y_r is the uniformly-weighted bag-of-words embedding of the review, and $M \in R^{d \times d}$ is a learned attention model. This attention layer is designed to reduce the significance of the words that are not relevant to any high-level features, and focus more on the more high-level feature related words.

The second step computes the high-level feature-based review representation $t_r \in R^d$ in terms of a high-level feature embedding matrix $T \in R^{K \times d}$, where K is the number of high-level features:

$$p_t = softmax(W \cdot z_r + b) \tag{5.4}$$

$$t_r = T^T \cdot p_t \tag{5.5}$$

where $p_t \in R^K$ is the weight vector over K feature embeddings, and $W \in R^{K \times d}$, $b \in R^K$ are the parameters of a multi-class logistic regression model.

The parameters in ABAE are trained to minimize the reconstruction error, i.e., the cosine distance between t_r and z_r. Representative words of a high-level feature can be found by ranking all words based on the cosine similarity between the word embedding and feature embedding. To assign a high-level feature to each review, the learned parameter p_t is used to pick the high-level feature that is of the highest weight.

5.2.2 Data preprocessor

User reviews can be non-informative (i.e., do not contain useful information), such as *"I will give one star"*. Thus the first step in FeatCompare is to filter out non-informative user reviews and keep only the reviews that contain valuable information. First, we filter out reviews without any text. Next, we feed the filtered user reviews of all apps to AR-Miner [50]. We leverage AR-Miner as it is a commonly used approach to filter non-informative reviews [210, 214]. The definition of informative and non-informative reviews in our work aligns with the definition used by AR-Miner's authors. We consider a user review as informative if the review carries potential helpful information that can be leveraged by the developers to enhance the quality of the app or improve the app's usability. For example, "Pro version still needs an ad free version" and "It does not give location" are informative user reviews that bring insight into feature requests or functional flaws. Non-informative reviews such as "This app is pointless" and "I don't like this app, I will give only 3 stars" provide no constructive information for app developers to guide them on how to improve the quality of the app.

AR-Miner classifies reviews into informative reviews and non-informative reviews. We consider only the subset of informative user reviews in the following steps of our approach. Finally, the length of the review content (i.e., the review title and the review description) can impact the usefulness of the user reviews [140, 272]. Shorter reviews are less likely to be meaningful. Among the subset of three million informative reviews, 191,417 reviews, e.g., 5%, are two words in length or less. We manually check a statistical representative random sample, ie., 96 user reviews, with a confidence level of 95% and a confidence interval of 10% of the up to two words user reviews. We find that 87% of the reviews

with less than three words in length, i.e., "useless app", "hate this", "rarely works", and "nothing worked", are non-informative. Therefore, we follow previous work [42, 150] and filter out reviews that contain less than three words in the content.

5.2.3 Global-Local sensitive feature extractor

GLFE is the main component of FeatCompare that elicits high-level features from user reviews. As shown in Figure 5.1, GLFE contains three steps: identifying global high-level features based on all collected app reviews across multiple groups, identifying local high-level features based on only reviews of selected competing apps, and assigning one high-level feature for each review of the selected apps.

Identifying Global and Local Features. We observe that in mobile app reviews, users might choose to comment on either group-specific features, such as "*accurate forecast*" for weather apps, or general features that are common among apps of all categories, such as "*device compatibility issues*" and "*performance issues*". To utilize this unique characteristic of mobile app reviews, we introduce the concepts of local and global high-level features and extract these two types of features simultaneously. *Global high-level features* are general features not related to any specific app domain but rather representing software engineering features shared among all types of mobile apps. *Local high-level features* are specific features shared among the selected competing apps within a particular group.

GLFE contains two cycles, i.e., a global cycle for learning global high-level features and a local cycle for learning local high-level features (ref. Figure 5.1). Both cycles follow the design of ABAE but with separately learned word embeddings. Several embedding methods have been used in sentiment analysis research to construct vector representations of words.

The local cycle in GLFE takes informative user reviews of selected competing apps as input. Next, the reviews are filtered following standard text normalization steps, including tokenization, removing non-English words, and stop words (i.e., frequent but meaningless English words), and lemmatization. As introduced in Section 5.2.1, ABAE

requires three inputs (1) an initial lookup table that contains word embeddings, (2) a set of target sentences to extract features, and (3) a parameter determining the number of learned high-level features. In the local cycle, local high-level features are extracted by applying ABAE on reviews only from competing apps in one specific group. The word embeddings are learned from the same set of reviews via word2vec. Following the paper proposing ABAE [98], we use a heuristic approach to determine the value of the number of features based on our observations on the quality (higher feature coherence and a lower degree of overlapping) of learned features. After running ABAE, the local cycle of GLFE assigns a local high-level feature to each review in the selected group.

In parallel and independently, the global cycle of GLFE utilizes an ABAE model to extract high-level global features from reviews of apps in the selected group. Unlike the local cycle, the word embedding used as input for ABAE in the global cycle is learned from all the 3,439,819 informative reviews of apps across multiple groups. We use 3,439,819 user reviews to generate global embedding. Our intuition is that word embeddings learned from various groups can better capture the semantics of words that are relevant to general features because these words appear relatively more often than domain-specific words (i.e., words describing unique features in a specific type of apps) in the reviews of different apps. Like the local cycle, we use manual observation to determine the best number of high-level features associated with each selected set of similar apps.

After running the local and global cycles, for each word appearing in the reviews of selected similar apps, GLFE learns a global word embedding representing the meaning of the word in the context of all app reviews and a local word embedding representing the meaning of the word in the context of selected apps. Meanwhile, two sets of high-level features and their associated most representative words are also learned. Note that, ABAE does not output a feature name for each learned high-level feature. Following He et al. [98], two annotators manually label the features based on the representative keywords using open coding [235, 239]. The annotators then discuss the conflicts in a consensus meeting where they resolved one by one the disagreements and determine the final feature names.

The design of global feature and local feature learning in GLFE poses a unique challenge for determining the final high-level feature of each review, i.e., each review now is assigned two high-level features, one from each cycle. We explain in the next subsection how GLFE solves this challenge.

Global-Local Feature Selection. We propose a metric, named *mean-norm-tfidf*, to approximate the likelihood of a review being assigned to its local high-level feature, i.e., reviews with high mean-norm-tfidf select the features learned from the local cycle of GLFE. mean-norm-tfidf is the average of values in the L2 normalized tf-idf [223] vector representation of a review. We formally define the notion of mean-norm-tfidf as follows. Given a review r_i containing n unique words $w_1, w_2, \ldots, w_n$ and a review corpus C, the mean-norm-tfidf of the review r_i is defined as:

$$\textit{mean-norm-tfidf}_{r_i,C} = \frac{1}{n}\sum_{k=1}^{n} norm(f_{w_k,r_i} \cdot \log \frac{M}{|d \in C : w_k \in d|}) \tag{5.6}$$

where f_{w_k,r_i} refers to the number of times the word w_k appears in review r_i, i.e., the term frequency of w_k. M is the total number of reviews in C, $|d \in C : w_k \in d|$ refers to the number of reviews in C that contain word w_k. $\log \frac{M}{|d \in C : w_k \in d|}$ is the inverse document frequency of word w_k. *norm* is a function[2] that normalizes a value to the range of 0 to 1. Note that the corpus C refers to all informative reviews of apps across multiple groups, not just the reviews of the selected competing apps.

The main design concerns behind the mean-norm-tfidf metric are two folds. First, the metric should be positively related to the inverse document frequency of words in the review because domain-specific words that do not frequently appear in a variety of app groups have a larger inverse document frequency. If a review contains many domain-specific words, then the review is highly likely to discuss a local high-level feature. On the other hand, if a review contains many words with low inverse document frequency, such as "*ads*" and "*crash*" that frequently appear in reviews of apps across all groups, the review is highly likely to discuss a global high-level feature. Secondly, words that occur many times in the review should be assigned a higher weight as they often present

[2]We normalize document vectors to unit Euclidean length.

the main concern in a review.

However, calculating a mean-normed-tfidf score for each review is not sufficient. We need a threshold value to determine if the local/global feature should be selected. For instance, if we set a threshold at 0.5, then every review with a mean-norm-tfidf value larger than 0.5 is assigned to its local high-level feature, or its global high-level features, otherwise. To simplify the process, we rank all reviews in the selected competing apps in ascending order based on their mean-norm-tfidf scores and pick the 25th percentile (lower quartile), the 50th percentile (median), and the 75th percentile (higher quartile) as the three considered thresholds. In practice, stakeholders could tune this hyper-parameter using few groups of competing apps.

5.2.4 Rating Aggregator

As the final component of FeatCompare, the rating aggregator takes as input the informative reviews extracted from the selected competing apps with their extracted high-level features by GLFE and outputs a comparative table.

We define two main metrics of the rating aggregator: the **F**eature **A**verage **R**ating (FAR) and the **C**ompetitive **F**eature **A**verage **R**ating (CFAR). FAR represents the mean of star ratings of one specific feature of an app. CFAR represents the mean score of one feature across all the apps of the same group. Formally, given the set of w competing apps $A = \{a_1, a_2, ..., a_w\}$, and their p user reviews $R = \{r_1, r_2, ..., r_p\}$ that are relevant to a set of n high-level features $F = \{f_1, f_2, ..., f_n\}$, the rating aggregator first calculates a feature average rating FAR of a specific feature f_j in relation to app a_i as follows:

$$FAR_{a_i,f_j} = \frac{\sum_{k=1}^{m} rating(r_k)}{m} \tag{5.7}$$

where r_k refers to any review of the app a_i that is assigned to the high-level feature f_j. m is the total number of reviews of the apps a_i that are assigned to f_j. $rating(r_k)$ refers to the user rating associated with the review r_k.

FeatCompare also calculates a competitive feature rating $CFAR$ of a specific feature f_j among w competing apps as follows.

$$CFAR_{f_j} = \frac{\sum_{i=1}^{w} FAR_{a_i,f_j}}{w} \tag{5.8}$$

$CFAR$ represents how the overall user-base (R) belonging to a group of competing apps (A) perceives a specific feature f_j. If the FAR value of an app a_i is higher than CFAR, this means, on average, app a_i receives more positive feedback than its competitor's apps on feature f_j.

In the resultant comparative table, FeatCompare provides the following information:

1. Name of each considered high-level feature.
2. Feature Average Rating (FAR) of each considered feature in each selected app.
3. Total number of reviews within each app that are relevant to each considered feature.
4. Distribution of ratings on the relevant reviews to each considered feature.
5. Competitive Feature Average Rating (CFAR) for each considered high-level feature.

5.2.5 Data collection

Identifying Groups of Competing Apps. To begin with, we select the top 2,000 free-to-download popular apps from the Google Play Store as the candidate target apps. The popularity of the apps is decided based on the App Annie report [15]. We then collect all general information available on the Google Play Store, including the number of reviews and the app description for the 2,000 popular apps using a web crawler [9]. Our study mainly focuses on the most popular apps as these apps contain rich review data that facilitate our analysis of user reviews. Moreover, developers may wish to compare their apps to the most successful apps in the market.

From the 2,000 popular apps, we identify *groups of competing apps* (i.e., apps sharing similar functionalities and business domains) by applying the following steps. First, we identify a set of keywords that represent a main app feature to form 20 different apps groups in total. The selected 29 keywords (i.e., main functionality of the app) were

chosen to identify groups of similar apps. The first two authors have attentively chosen the keywords to cover multiple Google apps categories, so a specific app category does not bias our study. Hence, we covered 17 Google categories. For example, we use the keywords *"*weather*"* and *"*browser*"* to represent the weather forecast apps and the browser apps, respectively. Next, for each of the considered main app features illustrated in Table 6.1, we search apps with the corresponding keywords mentioned in their names or descriptions using the collected information from the 2,000 popular apps. We then rank the matched apps by their review numbers. From the top of the ranked list, the first two authors manually read the name and the description of each encountered app to verify whether the app indeed contains the corresponding main feature. For each main app feature, we filter out apps that do not contain the feature and stop checking until we collect ten apps. We choose ten apps for each group, so that our analysis is not biased towards any particular app group. Besides, intuitively, we do not expect each app to have a very large (e.g., 50) number of competing apps, implying that an app can be replaced by as many as 50 other apps that are in the top 2,000 popular apps in the whole market.

Table 5.2: Statistics of 20 competing apps groups.

App group	Google category	Used keywords	# of apps	# of reviews
Taxi and rideshare	Maps and navigation	**taxi*, *rideshare*, *share-ride**	9	337,009
Navigation	Travel and local	**navigation*, *gps*, *map**	10	328,719
Security	Tools	**virus*, *malware*, *security**	10	312,758
Browser	Communication	**browser**	10	292,103
Free call	Communication	**free call**	10	291,034
News	News and magazines	**news**	10	283,711
Weather	Weather	**weather**	10	276,941
SMS	Communication	**sms**	10	264,926
Dating	Dating	**dating**	10	200,236
Wallpaper	Personalization	**wallpaper**	10	183,405
Notes	Productivity	**notes*, *notepad**	8	133,665
Video editor	Video player and editor	**video editor**	10	126,357
Hotel booking	Travel and local	**hotels**	10	69,699
Bible	Books and reference	**bible**	10	64,417
Mobile banking	Finance	**mobile banking**	10	64,371
Music player	Music and audio	**music player**	10	60,685
Sports news	Sport	**sports news**	10	57,582
Cooking recipe	Food and drink	**recipe*, *cooking**	10	41,154
Pregnancy	Health and fitness	**pregnancy**	9	29,816
Coloring	Creativity	**coloring**	10	21,231
Total			196	3,439,819

To validate that the selected candidate apps for each main feature are closely related, the first two authors independently read the page of every selected app in the Google Play Store and check if the other nine apps in the same group appear in the "*similar*" app list recommended by the store. In the end, we find that all the candidate apps satisfy the above requirement. Table 6.1 summarizes the basic statistics of the 20 selected competing app groups along with their categories at the Google Play Store. Some groups, i.e, "*Taxi and ride share*", "*Notes*", "*Pregnancy*", have less than ten apps because there are less than ten apps match with our searched keywords. Expanding the search to consider more than 2,000 popular apps from App Annie, might lead to find ten competing apps for the aforementioned categories. However, we believe that different categories can have a different number of competitors. Since we have covered a sufficient number of groups (e.g., 20 apps domain) and each group contains enough competing apps (e.g., 8, 9 and 10), we only consider the top 2,000 popular apps.

We follow the above practice as a proof of concept. Nevertheless, we expect that even the developers of an app may have different sets of competing apps formed based on different goals. For instance, developers can choose to compare their apps with the similar paid apps or freemium apps.

User Reviews Collection. For each selected app, besides the general information, we collect its user reviews over 3.5 years, starting from April 2016 until January 2019. In total, we collect 14,043,999 user reviews. For each review, we record the title of the user review, the detailed comment text, user rating, and the post date of the review. Table 5.3 summarizes the basic statistics of the collected reviews.

Table 5.3: Dataset Statistics.

Number of studies apps	196
Number of initial user reviews	14,043,999
Number of non-empty text user reviews	13,847,602
Number of informative user reviews	3,631,236
Number of user reviews with more than two words	3,439,819

5.3 Experimental Results

In this section, we evaluate the effectiveness and usefulness of FeatCompare for comparing high-level features among competing mobile apps based on user reviews. Specifically, we discuss motivation, approach, and findings for the following research questions.

5.3.1 RQ5.1: How effective is our global-local sensitive feature extraction approach GLFE?

Motivation. The effectiveness of FeatCompare relies on the accuracy of its data-driven feature extractor component, i.e., GLFE. Thus for FeatCompare to be useful in practice, we need to evaluate the effectiveness of GLFE in identifying the high-level feature associated with each informative review.

Approach. To evaluate the performance of GLFE, we randomly pick five groups from the 20 identified competing app groups (ref. Table 6.1). The selected groups are "*Weather*", "*Sports news*", "*Bible*", "*SMS*", and "*Music player*", respectively.

Then, for each app group, we select a statistically representative random sample of user reviews with a confidence level of 95% and a confidence interval of 10%. In total, we select 480 informative reviews that belong to the selected five groups. Next, the ground truth high-level features of the testing reviews are obtained by performing the following steps:

1. To achieve candidate high-level features, for each of the five selected app groups, we apply the global cycle and local cycle of GLFE on the reviews output by the data preprocessor component of FeatCompare. At the end of this step, we achieve a set of local and global high-level features for each group of competing apps.

2. The first three authors independently annotated the 480 testing reviews using the candidate high-level features of the corresponding app group. For instance, each testing review from the app group "*Weather*" should be assigned to the corresponding high-level features extracted from the "*Weather*" group in the first step. At least two annotators annotate each testing review.

It should be noted that one user review can discuss multiple features. For instance, the following review comment *"Very user friendly app and I find the alerts warning of severe weather conditions very helpful."* contains information about the *"User Interface"* feature and the *"Weather Alert Services"* feature. Hence, in this step, the created gold dataset contains all valid features that are mentioned in the review (i.e., the *"User Interface"* and the *"Weather Alert Services"* feature). We find that only 8.6% of the manually labeled reviews contain multiple features.

3. The Fleiss's Kappa agreement score [177] is calculated on the annotated testing reviews using the "irr" package[3] provided in R[4] and we achieve a score of 0.83, which indicates a high level of agreement among annotators. The annotators then discuss the conflict cases and resolve all disagreements.

Following the original design of ABAE [98], we set the default number of high-level features as 14 and vary the number by increasing and decreasing it. We find that the default setup consistently provides high quality high-level features for all app groups. Thus, we decide to keep the number of expected global/local high-level features as 14.

GLFE has one hyper-parameter, the mean-norm-tfidf threshold for the global-local feature selection step. We considered three values for this hyper-parameter, leading to three models, named (*GLFE-25th*, *GLFE-50th*, and *GLFE-75th*), respectively. The 25th percentile means that for the 25% reviews with the lowest mean-normed tfidf, we assign them the identified global features, and the rest 75% reviews are assigned with their local features. GLFE with a higher value threshold (percentile) represents a model that prefers global high-level features over local high-level features. We select the best value for the mean-norm-tfidf threshold by evaluating the performance of three models (*GLFE-25th*, *GLFE-50th*, and *GLFE-75th*) on a validation set, consisting of two competing app groups, i.e., "Recipe cooking" and "Free call". Table 5.4 shows that on the validation set, GLFE-25th achieves the highest accuracy. Thus we use the 25th percentile as the threshold when applying GLFE on five testing app groups."

[3]https://cran.r-project.org/web/packages/irr/index.html
[4]https://www.r-project.org/

Table 5.4: Accuracy of GLFE for different local global hyper-parameter threshold on two validation app groups. Prec. represents the precision.

App Group	GLFE-25th			GLFE-50th			GLFE-75th		
	Prec.	Recall	F_1	Prec.	Recall	F_1	Prec.	Recall	F_1
Recipe Cooking	0.78	0.70	0.74	0.40	0.36	0.38	0.34	0.31	0.32
Free Call	0.80	0.73	0.76	0.47	0.43	0.45	0.38	0.35	0.36
Average	**0.79**	**0.71**	**0.75**	0.43	0.39	0.41	0.36	0.33	0.34

The rest of GFLE parameters are set based on the best performing parameter sets reported in the original ABAE paper [98]. Specifically, we initialize the word embedding matrix with word vectors trained by word2vec, setting the embedding size to 200, the window size to 10, and the negative sample size to 5. We initialize the feature embedding matrix with the centroids of clusters resulting from running k-means on word embeddings. Other parameters are initialized randomly. During the GLFE training process, we fix the word embedding matrix and optimize other parameters using Adam [130] with the learning rate of 0.001 for 15 epochs and the batch size of 50.

Evaluation Metrics: Since user reviews may contain multiple features, albeit a small percentage, we model the high-level feature identification task as a multi-label classification task. Given a set of testing reviews with *gold* high-level features (i.e., ground truth) and the predicted features, we evaluate the feature extraction approaches as follows. First, we measure the true positive (TP) as the number of successfully predicted features, the false positive (FP) as the number of falsely predicted features (i.e., features predicted by the approach but are not mentioned in the reviews), and the false negative (FN) as the number of features mentioned in the reviews but are not predicted by the feature extraction approach. We consider precision, recall, and F_1-Score as the evaluation metrics. Equations 6.3, 6.4, and 6.5 show the computation for precision, recall, and F_1-Score. Precision measures the percentage of true positive predictions among all the predictions made by the evaluated approach. Recall represents the percentage of the features that can be predicted by the approach among all the features in the ground truth. Finally, F_1-Score is the harmonic combination of precision and recall.

Table 5.5: Accuracy of GLFE and ABAE on five testing app groups. Prec. represents the precision.

App Group	GLFE-25th Prec.	GLFE-25th Recall	GLFE-25th F_1	ABAE Prec.	ABAE Recall	ABAE F_1	ABAE-Global Prec.	ABAE-Global Recall	ABAE-Global F_1	Global embedding reviews	Local embedding reviews
Weather	**0.81**	**0.74**	**0.78**	0.70	0.64	0.67	0.32	0.29	0.31	3,439,819	276,941
Sports news	**0.82**	**0.75**	**0.78**	0.71	0.65	0.68	0.28	0.26	0.27	3,439,819	57,582
Bible	**0.81**	**0.77**	**0.79**	0.72	0.69	0.70	0.36	0.34	0.35	3,439,819	64,417
SMS	**0.80**	**0.74**	**0.77**	0.67	0.62	0.64	0.15	0.14	0.14	3,439,819	264,926
Music player	**0.79**	**0.75**	**0.77**	0.70	0.66	0.68	0.21	0.20	0.20	3,439,819	60,685
Average	**0.81**	**0.75**	**0.78**	0.70	0.65	0.68	0.21	0.20	0.20		

$$Precision = \frac{TP}{TP + FP} \tag{5.9}$$

$$Recall = \frac{TP}{TP + FN} \tag{5.10}$$

$$F_1\text{-}Score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{5.11}$$

Baselines: To investigate the benefit of our adaptation on the original ABAE model, we compare the performance of GLFE with the performance of ABAE using the 480 labeled reviews. Specifically, we consider two different embedding matrices as input for ABAE, reviews from the selected similar apps (i.e., local reviews and the original setup), and reviews from apps across multiple groups (i.e., global reviews). We name the two variants of ABAE on our task as *ABAE* and *ABAE-global*, respectively. Note that ABAE can be treated as *GLFE-0th*, where every review is assigned to its local high-level feature, and ABAE-global can be treated as *GLFE-100th* where every review is assigned to its global high-level feature.

Results. GLFE-25th achieves F_1-Score of 77-79% on five testing app groups, which outperforms the baselines and other variants of GLFE. Table 5.5 shows the results of five considered approaches, i.e., GLFE with three feature selection thresholds and two ABAE models with global and local word embedding matrices. We can observe that the best performing GLFE model is the one with the 25th percentile threshold, with an average precision of 81% and an average recall of 75% across five testing groups. In

this setting of GLFE, 25% reviews are assigned to their global high-level features, and the rest 75% are assigned to their local high-level features. Our experiment results also show that, on average, GLFE model (i.e., GLFE-25th) can improve the F_1-Score of ABAE trained with reviews from competing apps by 14.7% ($(0.78 - 0.68)/0.68$).

Both global high-level features and local high-level features contribute to the overall performance of GLFE. As shown in Table 5.5, the two ABAE models trained from global and local embeddings perform worse than the GLFE-25th model, with an average F_1-Score of 0.20 and 0.68 respectively. As reviews in the ABAE model all choose the learned local high-level features and reviews in the ABAE-global model all choose the learned global high-level features, the above results imply that only relying on one type of features does not lead to a good performance. Another observation we can make is that GLFE-25th performs better than the other two settings, i.e., GLFE-50th and GLFE-75th, on all five testing app groups. This observation indicates that the threshold value used for global-local high-level feature selection can impact the performance of GLFE, and local high-level features are more valuable than global high-level features.

Summary of RQ5.1

The feature extractor GLFE in FeatCompare with 25 percentile feature selection threshold achieves a promising average F_1-Score of 78% on 480 manual annotated reviews, which surpasses the performance of the state-of-the-art neural network-based high-level feature extractor model ABAE by 14.7%.

5.3.2 RQ5.2: Is FeatCompare able to find and compare meaningful high-level features among competing apps? Is FeatCompare useful for mobile app developers?

Motivation. In RQ5.1, we perform a quantitative evaluation of the GLFE component in FeatCompare. The results show that our GLFE model (i.e., GLFE-25th) can successfully identify high-level features discussed in user reviews. However, it remains unclear whether FeatCompare is able to compare features among competing apps and how the comparative table provided by FeatCompare can be utilized for developers. Thus in

this RQ, we conduct a qualitative case study on the effectiveness of FeatCompare in comparing competing apps. Since our work aims to provide an approach that can help mobile app developers perform competitor analysis (i.e., analyze how users perceive an app concerning its competitors) and to better understand developers' practices and validate the usefulness of FeatCompare, we conduct a qualitative study with 107 participants.

Approach. We run FeatCompare on the five groups of competing apps which are randomly sampled from 20 identified app groups in RQ5.1, i.e., group "*SMS*", "*Weather*", "*Sports news*", "*Bible*", and "*Music player*". For each app group, FeatCompare identifies 28 high-level features and generates a comparative table, as described in Section 5.2.4. Among the 28 features, 14 are learned from the local cycle and another 14 from the global cycle. To demonstrate the usage of FeatCompare for app comparison, we consider two specific scenarios: 1) identifying the most commented features in a set of competing apps and 2) comparing feature-wise user opinions among competing apps. For the first scenario, we rank 28 identified features for each app group by calculating the number of reviews associated with each high-level feature within the group. For the second scenario, we explore insights that could be mined from the comparative tables created by FeatCompare.

For the qualitative survey, Table 5.6 shows the survey questions. In general, the survey consists of three parts:

1. **Background questions**: General questions asking about the participant's age, years of work experience in the mobile development field, job role, and the number of developed mobile apps.

2. **Development activities questions**: Development-related questions including whether the participant has conducted any comparative analysis before, the sources used to conduct competitor analysis, opinion on using the overall app store ratings as the only source to conduct comparative analysis.

3. **Validating FeatCompare**: Questions asking the opinion of the participant on

Table 5.6: List of defined questions in the conducted survey.

ID	Question	The possible answers
Background questions:		
Q1.1	What is your age?	"<= 25 years old", "26-35 years old", "36-45 years old", "46+ years old", and "Prefer not to answer"
Q1.2	How many years of work experience do you have?	"1 or less", "2-5 years", "6-10 years", "11-20 years", and "21+ years"
Q1.3	How many years of work experience in the field of mobile development do you have?	"1 or less", "2-5 years", "6-10 years", and "11+ years"
Q1.4	What are your roles in the development of mobile apps? Please select all that apply.	"Development", "Testing and quality assurance", "Release management", "Configuration management", "Product support", "Project/product management", "Other"
Q1.5	How many mobile apps have you developed (including the current one)	"None", "1", "2-5", "6-10", and "11+"
Q1.6	Which of the following industries best describe the category of your app? Please select all that apply.	"Weather", "Bible", "Browser", "Navigation", "Free Call", "SMS", "Music player", "News", "Security", "Wallpaper", "Taxi and rideshare", "Dating", "Recipe cooking", "Coloring", "Pregnancy", "Sports news", "Video editor", "Notes", "Mobile banking apps", "Accommodation booking", "Other"
Development activities questions:		
Q2.1	Do you agree or disagree with the following statement: app's overall rating in mobile stores is sufficient alone for developers to compare their apps to their competitors' apps and discovery areas to improve.	"Strongly agree", "Agree", "Neither agree nor disagree", "Disagree", and "Strongly disagree"
Q2.2	Have you ever compared your app to a competitor app?	"Very frequently", "Often", "Sometimes", "Rarely", and "Never"
Q2.3	How do you identify your app competitors? Please select all that apply.	"Keyword search", "App stores similar app suggestion", "App stores categories", "Customer feedback", "Social media", "Other"
Q2.4	In case you do perform competitor analysis, how do you perform it? Please select all that apply.	"User reviews of the competitive apps", "Check the overall competitive app ratings", "Check the web presence of the competitors", "Check the competitor app number of downloads", "Other"

ID	Question	The possible answers
Development activities questions:		
Q2.5	If you have not conducted a competitor analysis before, what are in your opinion good sources of comparison?	
Q2.6	How many competitors do you compare your app features to? (If you have developed multiple apps, please select a range that represents the average number of competitors that you used to compare with your app)	"None", "1", "2-5", "6-10", "11+"
Validating FeatCompare:		
Q3.1	Do you agree or disagree with the following statement: our research output will be of great benefit for the developers to compare their app to competitors' apps based on the user experience	"Strongly agree", "Agree", "Neither agree nor disagree", "Disagree", and "Strongly disagree"
Q3.2	In the space below, please provide any feedback that you wish to share with us. For example, do you have any recommendations to improve the results of our comparison tool? We would really appreciate your input and feedback.	

Table 5.7: Continuation of the survey questions.

the comparative table (i.e., Figure 5.3) created by FeatCompare.

To ensure that the participants of a specific organization do not bias our survey, we approach participants through multiple communication channels as follows.

- **Surveying developers of apps on F-Droid.** We retrieve a list of 922 open-source Android apps from F-Droid[5]. For each app, we obtain the developers' contact email addresses from the Google Play Store. Then, we send our survey to the 922 email addresses and receive 20 responses (2.2% response rate).

  We also contact the maintainers and the contributors of the 922 F-Droid apps as follows. First, we identify 598 apps providing their corresponding Git repositories on F-droid. Next, we collect 10,146 developers and their email addresses from the git commit history of the identified 598 git repositories. We then rank all developers

[5]https://f-droid.org/en/

based on the number of commits they made in the repositories and select the top 3,000 developers as a target. These 3,000 developers contribute 2,135 valid email addresses. In this end, we send our survey to these 2,135 email addresses and obtain 40 responses (1.9% response rate).

In total, we obtain 60 responses from the developers of F-Droid apps.

- **Surveying participants through the popular apps at the Google Play Store.** We retrieve 2,000 contact email addresses from the Google Play Store pages of the 2,000 popular apps collected in RQ5.1. Then, we send our survey to these 2,000 email addresses and obtain only 5 responses (0.3%).

- **Surveying the development teams of multinational companies.** We send our survey to the technical leaders in five multinational companies and ask them to distribute the survey within their teams. In the end, We obtain 22 responses from the development teams of different multinational companies.

- **Surveying participants using the development chat platforms.** We post our survey to the most popular development chat platforms. In particular, we post our survey to reddit Android development groups[6][7][8], Facebook developer circle Beirut group[9], and Facebook mobile development pages and groups[10][11][12][13]. We obtain 20 responses from participants on development chat platforms.

In total, we survey 107 participants.

[6] https://www.reddit.com/r/mAndroidDev/
[7] https://www.reddit.com/r/appdev/
[8] https://www.reddit.com/r/androiddev/
[9] https://www.facebook.com/groups/DevCBeirut
[10] https://www.facebook.com/groups/1549592438605145/
[11] https://www.facebook.com/groups/260880814006061/
[12] https://www.facebook.com/groups/cs464/
[13] https://www.facebook.com/groups/cs464/?ref=contextual_unjoined_mall_chaining

Table 5.8: Inferred top ten features with their representative words and an example review.

	Feature	Representative words	Sample review
SMS	Scam identification	block, scam, spam, robot, anonymous	*"Its useful to avoid spam calls and to identify the fake numbers"*
	Location connectivity	address, city, state, network, operator	*"It does not give location"*
	App stability	close, reboot, restart, hang, crash	*"Very useful not always 100 updated or synced to my phone"*
	Customization	simple, customizable, design, interface, replace	*"Far better than i expected full of customizations"*
	User experience	section, content, folder, undo, add	*"Much easier to find the people in our contact list"*
	In-app ads	advertiser, interruption, profit, commercial, cost	*"I do not want your notifications of promoting your stuff"*
	Account authentication	sign, login, register, error, connect	*"Not working unable to connect"*
	UI appearance	style, wallpaper, emoji, ugly, front	*"More skin and background options as well as font options"*
	Security	hacking, dangerous, trust, cheater, steal	*"People are misusing it to create a fraud saving fraud number"*
	Premium version	subscribe, pay, lifetime, purchase, membership	*"After i upgraded to premium the call record function disappeared"*
Music Player	App stability	stop, close, randomly, crash, freeze, anonymous	*"Bugged wont let you open music player closes itself"*
	Music library	rapper, metallica, band, artist, song	*"App only shows your list and option to search youtube"*
	Download music	store, mp3, file, storage	*"Won't let me listen to music I've downloaded"*
	Complain	explain, understand, argument, respond, prefer	*"Needs instructions i have no idea how to get music on it"*
	User Interface	toolbar, layout, feature, section, icon, tab	*"Easy to navigate but white colors on notification bar barely visible"*
	Search	search, screen, find, select, change	*"It won't let me search"*
	Premium version	buy, trial, purchase, version, premium	*"Pro version still needs an ad free version"*
	Playlist	order, track, genre, playlist, album, artist	*"I love the way how playlist work keep it up"*
	Offline feature	internet, wifi, connection, offline, data	*"Wish i had wifi i really would give it five stars if i did not need wifi"*
	Sound quality	bass, speaker, headphone, sound, high	*"Works just fine needs more sound output"*
Cooking Recipe	Record keeping	track, maintain, manage, monitor, organize	*"Tracking is so easy with frequent foods favorites and build a recipe"*
	Online reliability	problem, error, internet, server, connection	*"Network error freezes all the time"*
	User experience	thumbnail, item, category, section, search	*"Easy to search and find foods"*
	App practicality	convenient, handy, easy, quickly, helpful	*"My Guardian So handy to keep me on track wherever I am"*
	Bar code scanner	store, product, code, shop, qr	*"Needs to be easier to open the bar code scanner"*
	Recipe diversity	recipe, ingredient, search, menu, choice	*"Endless selection of recipes you can think of and more"*
	Version update	version, update, long, recent, upgrade	*"After the update it's even worse"*
	Fitness tracker compatibility	fitness, sync, gear, fit, tracker	*"I love that it sync with my Fitbit"*
	Diet plan	journey, program, goal, loss, eat	*"Love this app so far it has been instrumental in my weight loss journey"*
	Authentication	sign, log, account, login, email	*"Won't let me log in and will not let me create another account"*
Free Call	Call quality	call, outgoing, audible, echo, voice	*"I can t hear the other side"*
	International calls	korea, japan, abroad, overseas, internationally	*"Very useful instant chat and video call with anyone internationally"*
	Privacy	hide, block, remove, privacy, unwanted	*"Privacy issue no option to hide last seen"*
	Bug reports	exit, shut, freeze, hang, close	*"Keeps on crashing keeps on crashing while calling"*
	Other	cool, fun, fast, nice, interesting	*"Useful app i really like this app but sometime very annoying"*
	User experience	ux, usability, designed, complex, unintuitive	*"It s easy to use than anything else simplicity is whats admirable"*
	User interface	toolbar, section, header, column, icon	*"Not happy with the new UI. Some may like it and it's fine for default"*
	Version update	update, yesterday, upgrade, re-download, patch	*"Always need update but not improvement"*
	Account authentication	admin, login, signup, registered, signin	*"It spammed my contact list with a link from my account dangerous app"*
	Credit services	token, dollar, diamond, lottery, coin	*"It is really useful especially when u have no credit"*
Weather	Detailed weather info	detailed, info, precise, concise, weather	*"Very nice app no problems love the minute by minute forecast"*
	Daily forecast feature	day, prepare, know, weather, clothes	*"Helps me plan my days ahead. Love it!"*
	Accurate forecast	accuracy, overcast, predict, percent, reality	*"Accurate on most days spot on"*
	App stability	sync, update, reload, recently, restart	*"This crappy app won't let you update or do things correctly"*
	User interaction	header, slider, banner, menu, animation	*"Less options of locations in widgets previous version was best"*
	In-app ads	pay, ads, commercial, subscription, dollar	*"Inappropriate advertising adds"*
	Device compatibility	tablet, phone, ipad, app, android	*"This app goes on every device that I have"*
	UI appearance	font, size, color, style, skin	*"New Color scheme makes the information hard to see/read"*
	Weather alert services	warning, flood, dangerous, alert, notify	*"Awesome it gives warnings 10 min earlier than the tv and radio do"*
	Location-aware services	location, enter, save, zip, address	*"Great app lots of info for multiple locations"*
Sport news	Video streaming	watch, behind, ahead, stream, buffer	*"Love that i can watch the hockey feeds but they are so far behind live"*
	Playoff coverage	season, playoff, basketball, nhl, cup	*"At least it works most of the time for the nhl playoffs this season"*
	Subscription service	service, paying, cable, subscription, satellite	*"Excellent way to get access to shows when paying high provider prices"*
	Chromecast	android, phone, chromecast, cast, device	*"Embarrassed that i downloaded this no chromecast having app"*
	Notification	alert, information, notification, push, daily	*"Stupid thing can t disable notifications wow this is bad"*
	Games playback	video, playback, stop, buffering, freez	*"Had great playback but now stutters and skips during live playback"*
	Bug reports	reinstall, uninstall, crash, start, delete	*"Won't load i submitted my carrier and still won't load waste of time"*
	Blackout broadcasting	watch, blacked, restriction, access, blackout	*"Too many blackouts"*
	User interaction	bar, screen, button, page, scroll, menu	*"Headlines don't take you to the story"*
	Version update	ruin, new, version, compare, worse	*"Become worst after updating"*
Bible	In-app ads	ads, shop, pay, flash, interruption	*"It is a great app but the ads are unnecessary they take up my entire screen"*
	Educative	learn, child, teach, help, interesting	*"Handy teaches u a lot of new things"*
	Highlight and bookmark	highlight, content, suggest, bookmark, search	*"Love the highlights bookmarks and how easy it is to get to a specific verse"*
	Guided prayer	lord, soul, pray, god, amen	*"Perfect one cannot ask for anything better the word of god really available"*
	Easy narration	simple, term, adequate, magnificent, keyword	*"There are no sub headings on electronic bibles compared to hard copies"*
	Audio feature	listen, multitasking, hear, play, pause	*"I like the audio reading while im doing other things"*
	Bible versions	kjv, niv, king, james, esv	*"The bible has many different versions i haden't even heard of"*
	Practicality	quick, convenience, useful, portable, ease	*"Easily accessible switches between translations quickly and easily"*
	App stability	issue, error, fix, stop, bug	*"After update app is stuck on start screen after the most recent update"*
	Sharing	post, facebook, friend, link, social	*"I can't share any of my daily scriptures to instagram or snapchat please fix"*

Results. FeatCompare can spot the most frequently mentioned features in a group of competing apps. Table 5.8 show the top ten features with the highest number of reviews associated with five considered app groups. We only present the top-10 representative words and one sample review for each feature due to space limitations. The extracted features show the capability of FeatCompare in automatically finding the most popular (frequently mentioned in reviews) features among competitors. For example, using FeatCompare, we find that "*Scam identification*" is the most frequently discussed feature in the "*SMS*" group. We can also observe that providing detailed weather information is the most popular feature in the "*Weather*" group.

FeatCompare can spot potential opportunities for improving the app. Figure 5.3 presents the comparative table created by FeatCompare for the top three popular weather apps. For each considered app, the comparative table contains the 5-star rating distribution among the reviews associated with each feature and the total number of reviews mentioning the feature. This breakdown helps in identifying how the users perceive every feature of the app. For example, the feature "*Location-aware services*" is mentioned in a similar number of reviews in the *"Weather by WeatherBug"* (aka WeatherBug) app and the *"Weather radar and live maps - The Weather Channel"* (aka WeatherChannel) app. By comparing the star ratings of this feature in two apps, we can tell that WeatherBug users are more satisfied with its location-aware services than the users of WeatherChannel. FeatCompare also provides the average feature rating per app group to support app comparison. For instance, from Figure 5.3, we can discover that regarding the location-aware services, WeatherBug significantly outperforms the average of all competing apps in the same group. We also can spot that the app "*AccuWeather: Live weather forecast & storm raider*" may need to improve its weather alert services as it receives a significantly lower rating on the weather alert services compared to the other apps.

Competitor analysis is a common practice in mobile app development. Figure 5.4 and Figure 5.5 show the job role and the experience of the surveyed participants.

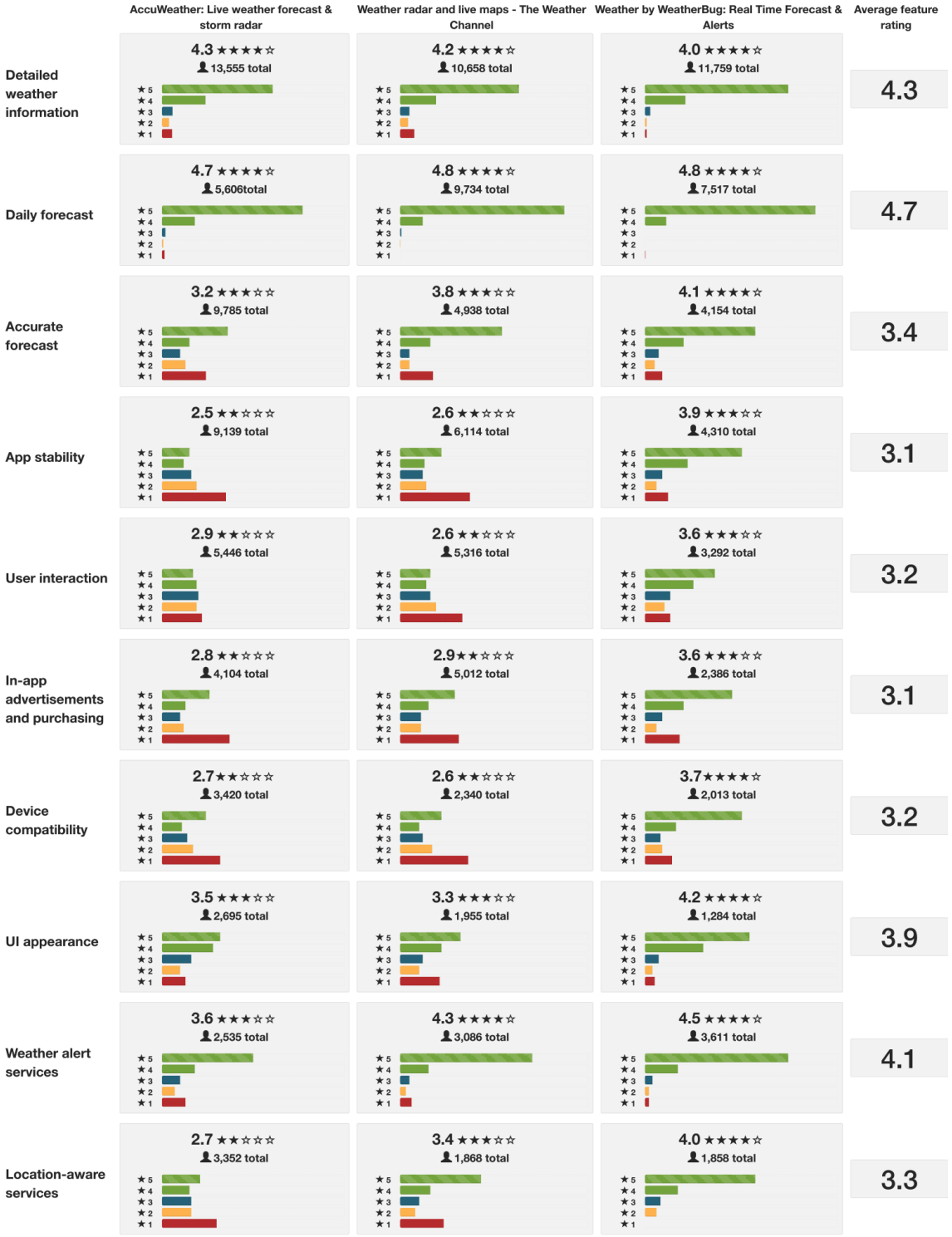

Figure 5.3: Comparison of the top ten features of the three most popular apps of the "*Weather*" group.

As shown in Figure 5.5, 76% of the participants have at least 2 years of mobile development experience, and 91% of them have been enrolled in development tasks. Note that participants are allowed to choose multiple job roles. As shown in Figure 5.6, 94% of the surveyed participants compare their apps to similar ones, while only 6% do not perform apps comparison.

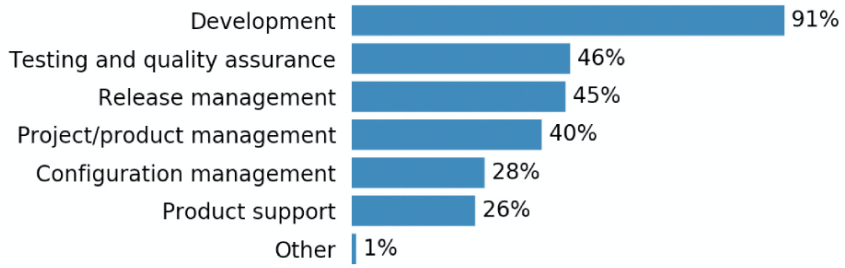

Figure 5.4: The distribution of the job roles of the surveyed participants.

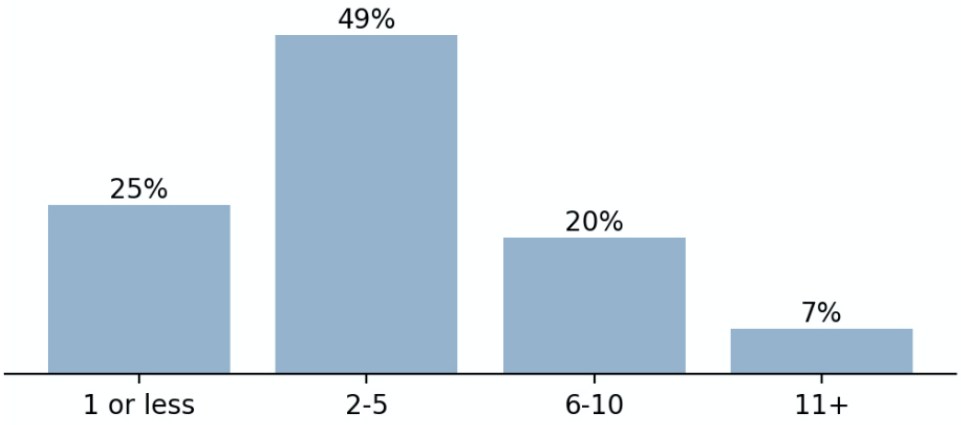

Figure 5.5: The distribution of the years of experience of the surveyed participants.

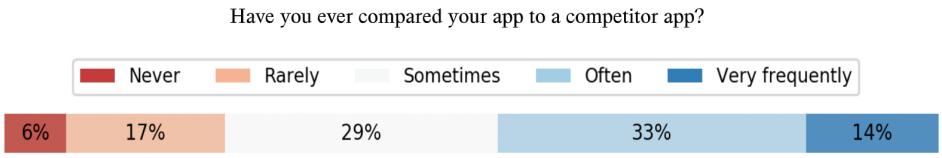

Figure 5.6: Participants' answers about competitor analysis.

We further investigate how participants conduct comparative analysis. Figure 5.7 shows that 68% of the participants treat user reviews as a comparison source to rely on. We also observe that 82% of the participants compare their apps with at least two competing apps, as shown in Figure 5.8. These survey results imply that comparative analysis

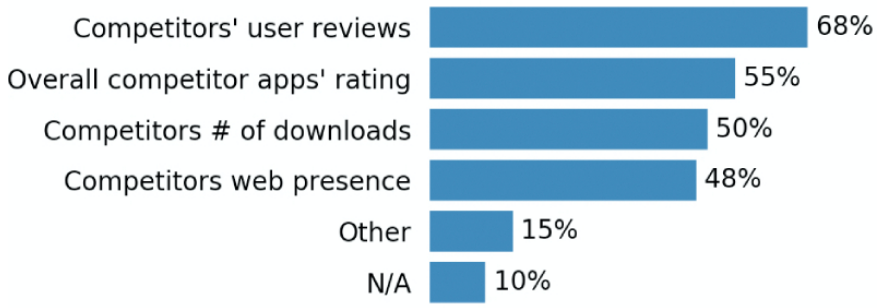

Figure 5.7: Participants' answers about the source used for competitor analysis.

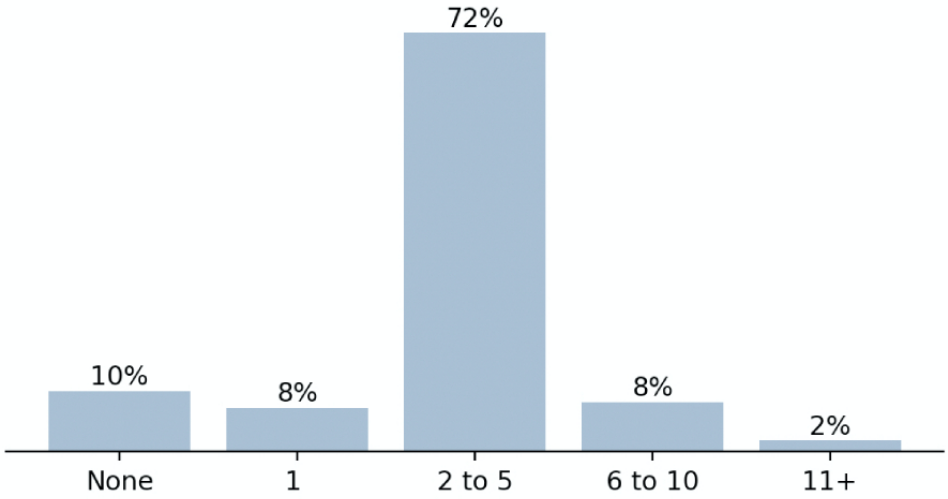

Figure 5.8: Participants' answers about the number of competing apps.

is common among app developers, and user reviews are valuable for performing competitor analysis. Hence, providing an automated approach for similar apps comparison based on user reviews can help app developers perform competitor analysis.

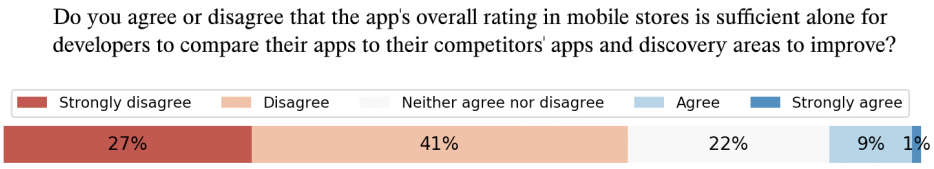

Figure 5.9: Participants' opinion about overall app rating.

The overall app rating alone is not sufficient for the competitor analysis of mobile apps. As shown in Figure 5.9, only 10% of the participants believe that the overall app rating is ample alone to compare among similar apps, i.e., other sources are crucial for the competitor analysis of mobile apps. We also find among the 22% of the survey participants who express a neutral opinion on the use of the overall app rating,

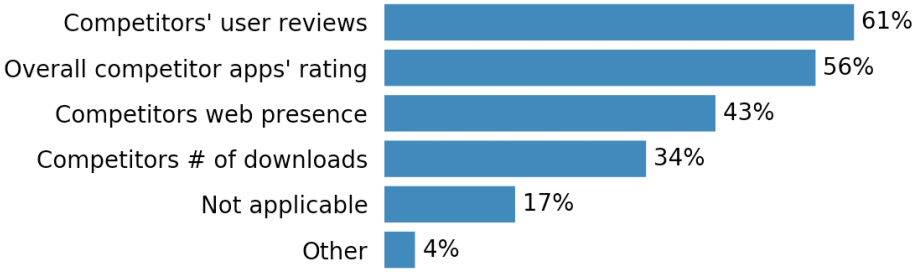

Figure 5.10: Source of competitor analysis for participants having a neutral opinion about the overall general rating of an app being enough to compare similar apps.

61% of them indicate that they use the reviews of similar apps in competitor analysis (Ref. Figure 5.10). As shown in Figure 5.10, among the 22% of participants with neutral opinion, only 56% rely on the overall ratings of competitors' apps to conduct competitor analysis, while the percentage of the participants who depend on the competitors user reviews (61%) in the competitive analysis remains higher. Thus, the above results further emphasize the important role of user reviews in competitor analysis of mobile apps.

Our approach is an asset to mobile developers to conduct a high-level feature analysis of competing apps. We find that 73% of the participants agree that the comparative table created by FeatCompare will be of great benefit (54% agree + 19% strongly agree) for comparing their apps to competitors' apps, as shown in Figure 5.11. Then, we take a closer look at the statistics of the responses and find that the participants who appreciate FeatCompare belong to diverse backgrounds. 74% of the participants have a minimum of 2 years of work experience in the mobile field, as shown in Figure 5.12. Only 5% of them have never performed any competitor analysis before, whereas the rest 95% have, as shown in Figure 5.13. Figure 5.14 shows that 86% of the participants compare their apps to at least two competitors.

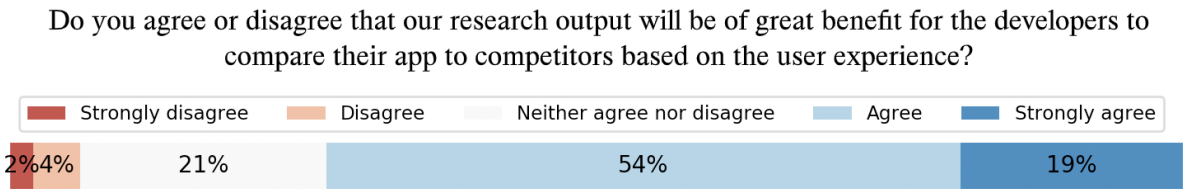

Figure 5.11: Participants' opinion about our research work.

Our findings aligned with existing mobile development related surveys [10, 208] by

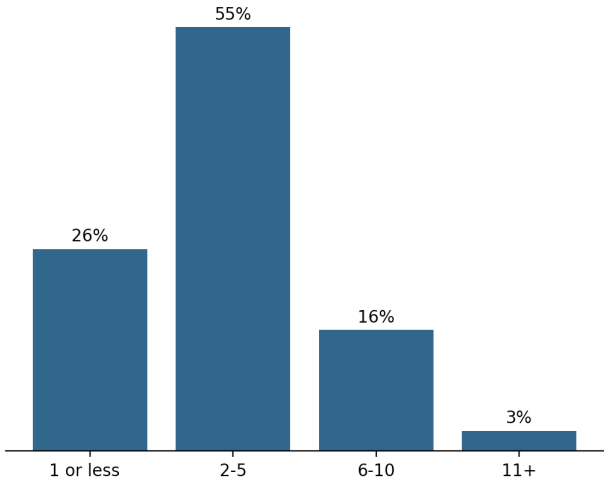

Figure 5.12: Years of mobile experience of the participants who agree on the benefit of our work.

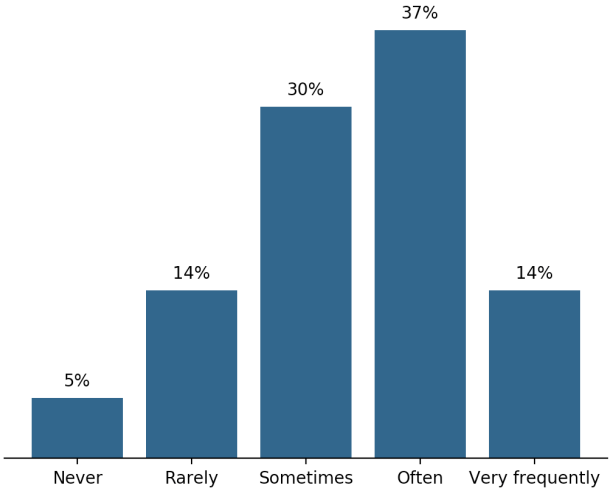

Figure 5.13: The frequency of analyzing the competing apps of the participants who agree on the benefit of our work.

demonstrating that competitor analysis is a point of interest to developers and that user feedback is the main source considered when performing the competitive analysis. The qualitative study by Nayebi et al. [208] gathers information about the market's impact on the mobile apps release planning by interviewing 22 participants. In particular, 20 out of the 22 surveyed participants vote that the customer's feedback is the most important factor for evaluating the success or the failure of mobile apps. Also, A. Al-Subaihin et al. [10] survey 186 participants with mobile development background. The survey results reveal that more than half of the participants rely on competitor analysis to gather

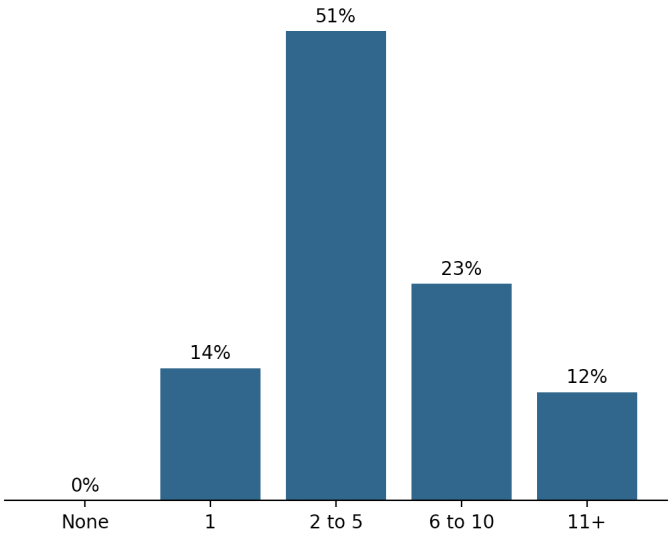

Figure 5.14: The number of competing apps of the participants who agree on the benefit of our work.

requirements. Specifically, 81% of the respondents that conduct competitor analysis select "user feedback" as the main point of interest when looking at similar apps. Similarly, for app enhancements, more than half of the developers examine similar apps in the app store, and 37% rely on similar apps users' feedback. In addition, our survey expanded the knowledge about practitioners' behaviors related to competitor analysis including opinions about the frequency of performing competitive analysis, ways of identifying competitors, and the number of competing apps considered by developers.

Summary of RQ5.2

FeatCompare can help app developers improve their apps by discovering the frequently commented features in competing apps, understanding an app's relative performance on each feature, and spotting the potential areas of improvement. The obtained 107 participants' responses show that competitor analysis is a common practice in mobile app development. 68% of the surveyed participants affirm that the overall app store ratings are not sufficient to conduct a competitor analysis of mobile apps. The survey results show that FeatCompare is an asset to mobile developers and supports high-level feature comparison among competing apps in an automated manner with minimal human effort.

5.4 Discussion

To validate the performance of FeatCompare in extracting high-level features, we implement a Vector Space Model (VSM) baseline to compare against it. For the Vector Space Model (VSM) baseline, the features of each app group are extracted in two steps. First, we create the vector representation of each review from an app belonging to the group using the vector space model (VSM). Specifically, each word appearing in the review becomes one feature, and the weight of the feature is determined by the Tf-Idf (term frequency–inverse document frequency) scheme using TfidfVectorizer function [14]) in the sklearn library to convert the collection of reviews to a matrix of TF-IDF features. Second, we cluster the review vectors using the k-means unsupervised clustering algorithm. The number of clusters, i.e, k, is set to 14, the same as the one used in FeatCompare.

We notice that VSM model produces many non-coherent clusters of reviews, i.e., it is challenging to identify a high-level feature representing the whole cluster. In order to fairly evaluate the coherence of the clusters created by the VSM-based approach, we design an evaluation procedure following Chen et al. [53]. First, for each of the review clusters generated by VSM-based approach, we rank the words appearing in the reviews based on their frequency and select the top-50 words as the representatives of the review cluster (i.e., "high-level feature" identified by the approach). Next, we collect the top-50 representative words for each of the top-10 high-level features identified by FeatCompare and the VSM-based approach respectively. The 20 word groups are then mixed and passed to three judges, who have more than six years' working experiences in IT companies and have developed mobile apps before. We ask judges to rate each of the 20 groups using a 5 point rating scale (i.e., "Strongly agree", "Agree", "Neither agree nor disagree", "Disagree", and "Strongly disagree"), specifying if they consider most of the words in the group representing a high-level feature of a mobile app. After collecting ratings from three judges on six app groups, i.e., 120 word groups, we calculate the number of coherent word groups identified by FeatCompare and the VSM-based approach. We

[14] `https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html/`

consider a word group being coherent if at least two judges "Strongly agree" or "Agree" that most words in the group represent a high-level app feature. Table 5.9 summarizes the evaluation results, i.e., FeatCompare can identify much more coherent word groups representing high-level features than the VSM model. As the baseline approach can not identify high-level features from user reviews, we do consider it in RQ5.1.

Table 5.9: Number of coherent aspects. K (number of aspects) = 10 for all approaches.

App Group	**Coherent aspects**	
	VSM	FeatCompare
Weather	3	8
Sports news	2	8
Bible	2	7
SMS	5	10
Music player	3	8
Recipe cooking	4	7

5.5 Threats to Validity

External Validity. External threats are concerned with our ability to generalize our results. In our study, we analyze the top 196 popular apps that belong to 20 different app groups. The total number of mobile apps in the Google Play Store reached 2.57 million apps by the fourth quarter of 2019 [77]. Hence, our results can be limited to the studied apps. To eliminate the impact of the app selection process on our results, we chose 196 apps distributed across 17 different categories at the Google Play Store. Although the Google Play Store contains more than 17 apps categories, we believe that our apps cover a wide variety of apps in the store. Moreover, our proposed approach can be easily extended to more categories as it does not require manually annotated resources. Similarly, we have only targeted apps in the Google Play Store. However, our approach can be applied to different apps in other stores (e.g., the Apple App Store and the Amazon App Store) as long as the reviews can be extracted.

It is relevant to note that the similarity between apps was defined by identifying keywords reflecting the main functionality across closely related apps. The intuition behind this selection method is that mobile apps that are similar in their descriptions or

titles would behave similarly and present common functionality. Therefore, two of the authors chose the groups keywords in a manner to represent as much diversity in the functionalities and business domains as possible. Unlike other related work [151] that only covered fewer than ten distinct app groups, our study includes 20 groups. Moreover, since our method of identifying competing app groups might be biased as in practice, developers may have specific criteria for selecting their competitors, we have addressed this concern by conducting a manual verification. The first two authors independently validated that each app of a group lists the rest of other apps belonging to the same group under the *"similar"* app list recommended by the Google Play Store.

Another potential threat to the external validity of our study arises from the nature of user reviews. The dataset used for the comparative analysis may not represent the full range of user opinions. Not all users post reviews, and some users may provide ratings without accompanying textual feedback, leading to incomplete insights into their experiences. This introduces potential bias, as the opinions reflected in our analysis are limited to those users who chose to leave detailed reviews. As a result, the findings may not generalize to the broader user population, where silent or less expressive users may hold different views.

Internal Validity. One threat to internal validity is mainly concerned with the manual assignment of features. We manually analyzed the automatically assigned features to user reviews on a statistically representative sample with a confidence level of 95% and a confidence interval of 10% (i.e., 96 user reviews per every group). To mitigate the error of manual identification of features, three annotators independently identify features form the user reviews. The Fleiss's Kappa agreement score on labeled reviews is 0.86, which shows a strong agreement between the annotators. However, we are not the owners of the studied apps, thus our analysis can be limited by our knowledge about the studied apps.

Construct Validity. User reviews may contain multiple features. Our approach assigns a single high-level feature per review. The assumption that every review represents one feature only can impact the accuracy of the feature extraction approach and the

average feature ratings reported for each app. In ection 5.2, we generate the features level average rating per app for the weather group. Although we find that the average percentage of multi-label reviews across the 672 manually labeled reviews is 8.6%, the results in RQ5.2 may be impacted by the fact that a single feature is extracted from every review. To mitigate this problem and validate whether the features' average rating will be significantly impacted if multi-labelling was assigned to reviews, we conduct an experiment to generate the results of RQ5.2 taking into consideration multi-labelling. In the manually labeled statistical sample of user reviews of the weather apps, we find that 9% of the reviews are multi-label. We followed the same approach as in Section 5.2 to calculate the average feature rating, except that we assign multiple labels per review before calculating the number of reviews associated with each high-level feature. First, we randomly assigned an additional label (different from the label already assigned by FeatCompare to the review) to 9% of the user reviews of the weather apps. Second, we calculate the number of reviews associated with each high-level feature within the group. Finally, we generate the features' average rating. We compare the overall rating of the top-10 features in the cases of single label assignment and multi-label assignment. We observe an average of 0.03 rating difference on the top-10 high-level features extracted from 10 apps in the weather group. We perform the Wilcoxon signed-rank test [166] using the wilcox.test function in R[15]. We observed a p-value of 0.3459, indicating no significant difference between the average rating on high-level features before and after considering multi-label reviews.

Another threat relates to the recency of the reviews. The high-level features mentioned in user reviews might belong to different app releases. Our work aims to provide a framework for developers that summarizes the perception of the users with respect to high-level features across similar mobile apps. Although the overall perception of a specific app feature (e.g., the features' ratings of the "Weather" apps presented in RQ2) can be changed from a release to another, the accuracy of extracting features in reviews for competing apps resulting from our intensive validation will not be impacted by the

[15]https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/wilcox.test

apps releases. Existing work [97] proposes approaches to map user reviews to corresponding releases. Hence, future work can benefit from our study and extend FeatCompare to perform a release-based features comparison (i.e., comparing the features of different releases).

5.6 Summary

In this chapter, we introduce FeatCompare, a novel approach that helps developers perform competitor analysis on high-level features based on user reviews with the minimal human intervention. FeatCompare contains three main components: 1) a component to preprocess raw reviews and filter non-informative user reviews; 2) GLFE, an unsupervised model that can identify global and local high-level features from app reviews and select the final feature for each review via a thresholding mechanism, and 3) an aggregator that generates a table of comparison among apps for summarizing feature-level user opinions. Our results suggest that:

- Our proposed approach, FeatCompare, can automatically extract high-level features from user reviews with high accuracy. FeatCompare outperforms the state-of-the-art high-level feature extraction model ABAE by 14.7% on average.
- FeatCompare can generate comparison tables for competing application features.
- 73% of the mobile app developers participants in our survey endorse the benefits of FeatCompare in competitor analysis.

We make our manually labeled data and the scripts available online[16] in an effort to inspire other researchers to investigate the competitor analysis of mobile apps further.

[16]Our data is publicly available in http://shorturl.at/aqCT2

Chapter 6

Competitor User Review Analysis for Feature Enhancement

In this chapter, we present our approach powered by LLMs to automatically generate suggestions for mobile app feature improvements. Section 6.1 provides an introduction and the motivation for our study. Section 6.2 details the design of our work. Section 6.3 reports the results of our experiments. Section 6.4 examines the threats to validity. Finally, Section 6.5 summarizes the chapter and suggests directions for future research.

6.1 Introduction

The surge in mobile apps adoption has fostered a highly competitive environment among competing apps, i.e., apps within the same categories that offer similar functionalities. For instance, WhatsApp[1], leading the messaging app category with 51 million monthly downloads, competes with at least ten other apps, each having millions of downloads [39]. A similar competition is evident in the video conferencing category, where Zoom[2] and Skype[3] are key players [61, 212]. During the pandemic, Zoom swiftly adapted to user demands by optimizing its platform for large-scale virtual meetings and enhancing security features, while Skype struggled to keep pace [225, 253]. To stay relevant and competitive, developers must rapidly respond to user needs. User reviews contain rich information, such as feedback, dissatisfaction and suggestions regarding the user experience of the

[1]https://play.google.com/store/apps/details?id=com.whatsapp
[2]https://play.google.com/store/apps/details?id=us.zoom.videomeetings
[3]https://play.google.com/store/apps/details?id=com.skype.raider

usage of mobile apps. These reviews offer developers critical insights into areas for improvement and opportunities for feature enhancements [10, 213, 236, 255, 288]. To stay competitive, developers need to learn from their competitors' behaviours to maintain a competitive edge [247]. Competitor user review analysis involves comparing user feedback, ratings, and reviews of competing mobile applications to identify strengths and weaknesses relative to competitors. By analyzing user reviews from competing apps, developers can uncover insights into features that address unmet needs, potentially giving their app a significant advantage.

Given the sheer volume of reviews [84], it is challenging to analyze user reviews manually. Practitioners need an automated feedback analysis process [255]. Hence, researchers propose automated approaches to filter informative reviews [50, 82], summarize user reviews [80, 276] and extract features from user reviews [60, 114, 131, 245, 276]. While existing research has been conducted on automated user review analysis, only a limited number of studies focus on competitor user review analysis, where reviews across competing apps are compared [20, 151, 200, 243, 247, 284]. Recent advancements in Large Language Models (LLMs) have demonstrated their capabilities across various natural language processing (NLP) tasks, including text generation, translation, summarization, and question answering [185]. Although LLMs offer promising applications for mobile app review analysis, current LLM-based research primarily focuses on tasks, such as sentiment analysis [227, 320], aspect extraction [299], analyzing multilingual reviews [289] and accessibility-related reviews [69].

Researchers have explored various approaches to conduct competitor user review analysis [20, 60, 151, 160, 243, 247]. However, existing work presents some limitations. First, the existing approaches often generate an overwhelming number of fine-grained features [60, 243, 247] due to comparing the apps' features based on word pairs, making it hard to conduct competitor user review analysis with thousands of features [244]. Second, competitor user review analysis is only conducted by identifying explicit expressions of comparison (e.g., *"Zoom's screen sharing is way smoother than Skype's"*) [151] and fails to take into consideration implicit insights derived from user reviews. Third, existing work

on competitor user review analysis [20] offers only feature rating comparisons lacking the ability to suggest concrete improvements for specific features based on competitors' user feedback.

To address the limitations of existing work in suggesting feature enhancements using competitor user review analysis, we propose an ***LLM**-based **C**ompetitive **U**ser **R**eview Analysis for Feature **E**nhancement (LLM-Cure). LLM-Cure* automatically generates suggestions for mobile app feature improvements by leveraging user feedback on the similar features from competitors. *LLM-Cure* operates through two phases. In the first phase, it leverages its large language model capabilities to extract and assign features to user reviews. In the second phase, it curates underperforming features among those identified in the first phase for the target app and suggests potential improvements for specific complaints by leveraging highly rated similar features in competing apps.

To evaluate the effectiveness of our proposed approach, we conduct an empirical study on 1,056,739 reviews of 70 popular mobile apps from the Google Play store belonging to 7 categories. Our experiment results show that *LLM-Cure* achieves an average F1-score of 85%, an average recall of 84% and an average precision of 86% in assigning features to user reviews, outperforming the state-of-the-art approach by 7%, 9% and 4% in F1-score, recall and precision respectively on average. To ensure the validity of the feature improvement suggestions by *LLM-Cure*, we cross-reference the suggestions with the release notes, verifying if similar improvements have been implemented in the subsequent releases. We find that 73% of the suggested enhancements by *LLM-Cure* are implemented in the release notes.

6.2 Study Design

LLM-Cure is designed to identify user complaints from user reviews and provide suggestions for developers to enhance features that require the developer's attention. More specifically, *LLM-Cure* operates in two distinct phases: (1) *Scalable Feature Extraction and Assignment* that focuses on identifying and assigning features from user reviews of competing apps and (2) *Suggestion Generation with Competitor Reviews* that leverages

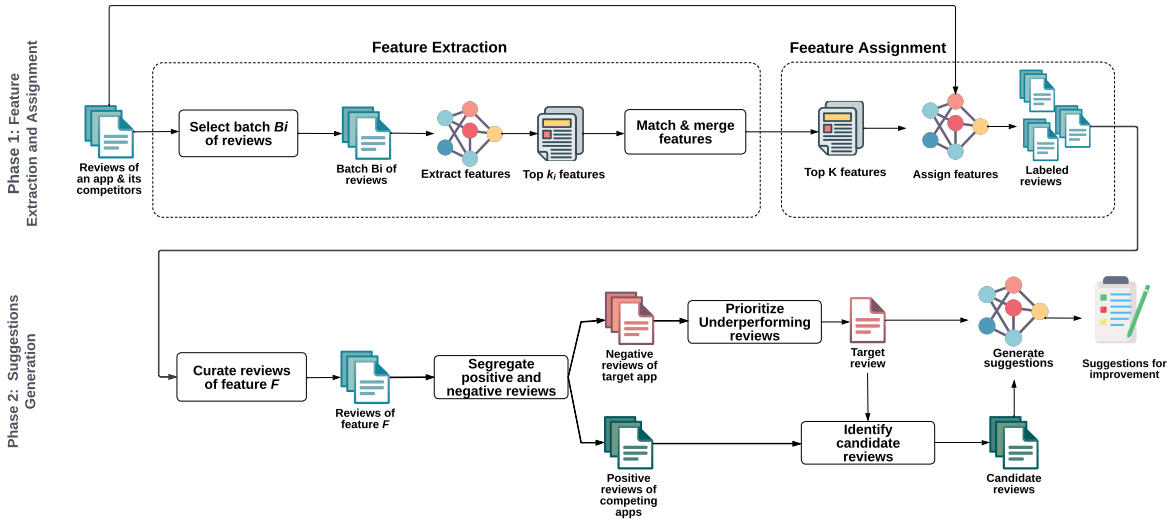

Figure 6.1: Overall approach of *LLM-Cure*

the extracted features to identify user complaints from a target app and generate suggestions for feature enhancement leveraging the competitor reviews. By incorporating user feedback from competitor reviews, *LLM-Cure* helps developers address user complaints with a competitive edge. Figure 6.1 provides an overview of the approach. We provide in the sections below background information, a description of *LLM-Cure* phases, in addition to the dataset description.

6.2.1 Large Language Models

Large Language Models. Pre-trained Large Language Models (LLMs) are deep neural networks that have undergone extensive training on large text data that have enabled them to learn complex patterns and structures of language [323]. In the realm of software engineering, LLMs have gained significant attention and adoption for various apps [107], including code generation [68], code repair [78], and documentation generation [301]. Hence, LLMs offer promising avenues for automating software development processes and enhancing developer productivity. Although LLMs are primarily designed for generating text, their output can be influenced by specific instructions communicated through prompts [254].

Prompting. Prompting is a technique used to communicate expectations and guide

the LLM's vast knowledge and capabilities towards achieving a specific goal [326]. Prompting involves providing instructions to LLMs to guide their generation process and elicit specific types of responses, enforce constraints, or guide the model toward certain stylistic elements. In the software engineering realm, prompting can be used to elicit code snippets, documentation, or other relevant text based on user requirements.

In-Context Learning and Few-Shot Learning. Leveraging pre-trained models, i.e., LLMs, for downstream tasks often requires further fine-tuning on domain-specific labeled data [108]. However, fine-tuning LLMs for specific tasks can be computationally expensive and resource-intensive, requiring substantial amounts of task-specific annotated data [38]. Hence, in-context learning and few-shot learning offer a powerful alternative [67]. In-context learning allows the LLM to adapt within a single interaction, using some initial information, i.e., context. Few-shot learning consists of exposing the model to a few examples (i.e., "*few shots*") to make the model effective for a specific task. For instance, for sentiment analysis in user reviews where the goal is to classify reviews as positive or negative, few-shot learning consists of giving the LLM a few examples for each sentiment, i.e., reviews with associated sentiment. The LLM then uses its existing knowledge and these few examples to categorize new reviews based on the context provided.

Retrieval Augmented Generation. LLMs can suffer from hallucination and generate irrelevant or inaccurate responses [109]. This can occur due to limitations in their training data or the lack of clear context in the prompt. Retrieval-Augmented Generation (RAG) addresses the issue of hallucination by integrating information retrieval techniques into the generation process [181]. Thus, the RAG retrieves relevant knowledge from external sources based on the input context, augmenting the model's understanding. For instance, in the realm of user reviews, RAG can be employed to enhance the quality of review summarization. The RAG model augments the user reviews with additional information extracted from external sources, such as sentiment analysis scores, key phrases or keywords, to generate concise and informative summaries of user reviews.

6.2.2 Scalable Feature Extraction and Assignment

Step 1: Extracting Features with Batching and Matching. This step focuses on identifying the top features that can be summarized by LLMs from an extensive collection of user reviews. However, LLMs have limitations on the amount of context they can process at once. To address this challenge and make our approach scalable, we introduce the so-called *batch-and-match* approach that incrementally extracts the top k features from a large corpus of user reviews. Our approach consists of three processes:

① *Batching reviews and extracting features.* Batching reviews involves dividing a large volume of reviews into manageable batches for incremental processing and feature extraction using LLMs. First, we shuffle the entire collection of user reviews of a group of competing apps to ensure randomness. Let $R = \{r_1, r_2, ..., r_m\}$ represent the shuffled user reviews where m is the total number of reviews. To efficiently process the large volume of reviews, we split the shuffled reviews R into batches of a predefined size s (e.g., 1,000 reviews) that fit within the LLM context size. We denote the batches as $B = \{B_1, B_2, ..., B_n\}$, where n is the total number of batches. Each batch B_i corresponds to a set of individual reviews $R_i = \{r_i, r_{i+1}, ..., r_{i+s}\}$, with each r_i being a user review in batch B_i. We process each batch B_i sequentially using the LLM. For each batch B_i, we prompt the LLM to extract the top k features, denoted as F_i, from the set of reviews R_i. The value of k is a hyper-parameter that can be tuned to optimize the performance of *LLM-Cure* depending on the selected dataset. The *Feature extraction* prompt, illustrated in Figure 6.2, identifies the top k features for a specific app category. It is structured to encapsulate the task description, define features, include a one-shot example, and present the list of user reviews.

② *Matching and merging similar features.* Since we process thousands of reviews in batches due to limitations on the LLM's context handling, the extracted features might sometimes use different wording. For instance, in one batch, the LLM might identify *"Advertisements"* as a feature, while another batch might highlight *"In-app Advertisements"*. Both features represent the same functionality extracted from the user reviews.

To ensure a non-redundant feature identification, we address this challenge by incrementally combining similar features after processing each batch B_i. F_i represents the features extracted from batch B_i. We define M_i as the set of matched and merged features until batch B_i. The merging process starts with the top K_1 features extracted from the first batch B_1. For the first batch, $M_1 = F_1$. For subsequent batches (B_i, $i > 1$), we compare features in F_i with M_{i-1} features already identified from the previous batch B_{i-1} and match then merge the similar features. Word embeddings and cosine similarity are employed to achieve the merging. Word embeddings represent the features in a high-dimensional vector space, capturing their semantic meaning [27]. Cosine similarity, a metric for measuring similarity between vectors, is then calculated between the embedding vectors of features F_i and M_{i-1} from different batches. Features exceeding a predefined similarity threshold tr in cosine similarity are considered highly similar and subsequently matched and merged. The similarity threshold tr is a hyper-parameter. Therefore, we experiment with thresholds ranging from 0.7 to 0.85 on a validation set and choose the value that leads to the highest precision. This incremental process continues with each new batch B_i, merging similar features from F_i with the existing merged set M_{i-1} leading to a unique set of features M_i.

③ *Verifying convergence and stabilizing features.* The challenge in this step is to determine when the incremental *batch-and-match* process has sufficiently captured the top k features, avoiding unnecessary iterations that would consume additional processing time and computational resources to process the entire volume of user reviews. We address this by defining a *convergence threshold* based on the stability of the top k features over a specified number of consecutive iterations N. For instance, the system starts with the initial merged set (M_1). The batch processing continues until the merged set M_j where the merged sets remain unchanged across the last N iterations (from M_j up to M_{j-N}). For example, assuming the *convergence threshold* is set to 3. *LLM-Cure* checks if the top k features identified have remained stable for the last 3 batches. This stability indicates that we have captured the dominant features in the reviews R, and further processing would probably not yield new features. The *convergence threshold* N

```
Below are user reviews, each separated by newline escape. Provide the list of the top 14 distinct, high level features
presented in all the reviews with a brief meaning of each feature.

High-level features refer to the main functionalities (e.g., "Live streaming" or "Sports coverage") and the main
characteristics (e.g., "UI appearance" and "App stability") of an app. For example, the two fine-grained features, "Stanley
Cup playoffs cut" and "Non-restricted playoff coverage," belong to the same high-level feature, "Sports coverage".

Below is an example of user reviews with the corresponding high-level feature.

Review: "The game I want to watch is blacked out."
Feature: Blackout broadcasting - This feature represents the broadcasting restrictions imposed on some games.

{User reviews}
```

Figure 6.2: Template of *LLM-Cure* prompt for extracting features.

is a hyper-parameter. Therefore, we experiment with thresholds equal to 3, 5, and 7 on a validation set. Following this convergence step, we obtain the final set of top k features extracted from the review batches.

Step 2: Assigning Features to Reviews. The prior research on the dataset [20] used in our approach demonstrates that only 8.6% of the user reviews contain multiple features and that multi-labeling does not lead to a significant impact on the results. We leverage prior findings to task a language model to assign one feature to the reviews by constructing a *Feature assignment* prompt. The *Feature assignment* prompt incorporates the task description, the extracted k features with their brief meaning, five few-shot examples demonstrating feature assignment to user reviews, and the list of user reviews to be classified. *Feature assignment* prompt is available in our replication package. As a result, each user review r_i is associated with one designated feature, building the groundwork for targeted analysis and feature enhancement suggestions generation in the second phase.

6.2.3 Suggestion Generation with Competitor Reviews

Prior research [81,269] shows that negative reviews, those with 1 or 2-star ratings, are particularly interesting to developers as they often contain valuable insights regarding feature complaints and areas for enhancement. Conversely, positive reviews, typically rated 4 or 5 stars, offer detailed descriptions of features and positive user experience [171]. These positive reviews are valuable resources as they often showcase successful implementations of similar features and offer potential solutions to address user complaints.

In Phase 2, we leverage the user feedback summarized from competitors' positive

reviews to provide suggestions for the target apps to improve the features associated with negative reviews. Our proposed method uses an RAG approach to dynamically construct prompts for the LLM that are augmented with relevant positive user reviews from competing apps. By analyzing these positive reviews, the LLM can identify successful implementations and suggest potential solutions to user complaints within the target application. *LLM-Cure* generates suggestions based on the following five distinct steps.

Step 1: Curating Popular Underperforming Features. An underperforming feature is defined as one that has the largest percentage of negative reviews. Identifying underperforming features is crucial for developers to prioritize areas for improvement and focus on features with the highest percentage of negative reviews to address user dissatisfaction better. We calculate the *Underperforming Feature Score (UFS)* for each feature by determining the percentage of negative reviews associated with it. The formula for *UFS* for a particular feature F is:

$$\text{UFS}_F = \frac{\text{Number of Negative Reviews}_F}{\sum_{i=1}^{k} \text{Number of Negative Reviews}_i} \times 100 \tag{6.1}$$

where the Number of Negative Reviews$_F$ denotes the number of negative reviews associated with feature F, and $\sum_{i=1}^{k}$ Number of Negative Reviews$_i$ denotes the total number of negative reviews across all k features. Sorting features in descending order of their UFS prioritizes those with the highest percentage of negative reviews. This allows developers to focus on features most frequently associated with user dissatisfaction.

Step 2: Segregating Positive and Negative Reviews. In this step, for a selected underperforming feature F, we select the negative reviews of the target app. Concurrently, we also curate positive reviews rated 4 or 5 stars of the same feature F from competitor apps. We exclude reviews associated with a 3-star rating.

Step 3: Prioritizing Complaint-Rich Negative Reviews. Not all reviews contain the same amount of details. Some might express generic dissatisfaction, while others delve deeper and provide specific details about the issues encountered with the feature. In this step, we aim to guide developers towards the most informative negative reviews, i.e., complaint-rich, for a specific underperforming feature F. To accomplish this, we

implement a ranking mechanism on the negative reviews associated with the target app, utilizing the TF-IDF (term frequency-inverse document frequency) score [223]. TF-IDF analyzes the importance of words within a document (in this case, a user review) relative to their occurrence across the entire dataset of reviews. For a specific feature F of a target app, we use TF-IDF to calculate a score for each negative review. We sum up the TF-IDF scores of all words in a review to get a final TF-IDF score for the entire review. Higher TF-IDF scores indicate that the user review contains terms that are both frequent in the review and relatively unique across the entire dataset, suggesting detailed and specific feedback. This score reflects the review's richness in terms of feature-specific complaints. We select the top n negative reviews based on the calculated TF-IDF scores. These selected reviews $R = \{r_1, r_2, ..., r_k\}$ guide developers, directing their attention towards negative feedback rich in informative content regarding feature complaints specific to the target app.

Step 4: Identifying Candidate Reviews for Relevant Solutions. This step aims to identify potential recommendations from competitors' user reviews that might address the complaint in a specific negative review. This step consists of four processes:

① *Selecting a negative review.* We start by picking one of the top n negative reviews (e.g., r_1) identified in Step 3.

② *Selecting candidate reviews from competitors.* We look at positive (4 and 5-star) reviews P_F for the same feature F selected in Step 2. To mimic a practical environment where developers might only have access to historical data, we further filter out the candidate positive reviews to include only those with a post date equal to or before the selected target complaint review r_1. These reviews represent the candidates for finding suggestions for improvement.

③ *Creating vector embeddings.* We employ an LLM-based word embedding technique to convert the reviews into vector representations to compare them based on their semantic meaning. Using the same embedding model, we generate the vector representations for the selected negative review r_1 and all positive reviews P_F. Let V_{r_1} represent the vector representation of the selected negative review r_1, and V_{p_i} denote the vector representation

of a positive review p_i.

④ *Finding similar reviews.* Since all reviews are represented in the same vector space, we leverage cosine similarity to compare the vectors. The cosine similarity $\text{sim}(V_{r_1}, V_{p_i})$ between the vector representations of the negative review and each positive review is calculated as:

$$\text{sim}(V_{r_1}, V_{p_i}) = \frac{V_{r_1} \cdot V_{p_i}}{\|V_{r_1}\| \cdot \|V_{p_i}\|} \tag{6.2}$$

We rank the positive reviews based on their similarity, and we identify a sample P of positive reviews that exhibit the highest similarity to the negative review. These selected positive reviews represent instances where users discuss similar feature characteristics but in a positive context.

Step 5: Prompting Suggestions Using RAG. This step revolves around constructing the prompt for instructing the LLM to generate the relevant suggestions. Specifically, we design RAG-based prompts by leveraging the positive reviews identified in Step 4 to provide contextual guidance to the LLM regarding where to draw suggestions. Then, we instruct the LLM to suggest top N unique and constructive recommendations to enhance the feature F, discussed in review r_1, using the provided sample of positive reviews identified in Step 4. This RAG-based prompt provides developers with suggestions derived from positive user experiences, facilitating targeted improvements to address user concerns effectively.

6.2.4 Implementation of LLM-Cure

LLM Choice. We select *Mixtral-8x7B-Instruct-v0.1*[4] LLM to conduct our experiments. *Mixtral-8x7B-Instruct-v0.1* is a high-performing, open-weight Sparse Mixture of Experts model. We choose this model as it balances cost with performance. It has been demonstrated that it surpasses open source models, including *Llama 2 70B*[5] while achieving 6x

[4]https://mistral.ai/news/mixtral-of-experts/
[5]https://llama.meta.com/llama2/

faster inference, and it matches *GPT-3.5*[6] performance on standard tasks [7]. Being open-source and free, Mixtral allows other researchers to easily access, understand, and adapt our work. We employ Python scripts to facilitate loading the *Mixtral-8x7B-Instruct-v0.1* model from the Hugging Face Hub[7].

Embedding Model Choice. To ensure consistency within our approach, *LLM-Cure* employs the *mistral-embed*[8] word embedding model from Mistral AI during the process that requires word embedding. Specifically, we used it in the *Matching Similar Features* process of phase 1 and in the *Identifying Candidate Reviews for Relevant Solutions* step of phase 2. We leveraged the Mistral API to retrieve text embeddings efficiently.

Text Preprocessing. Prior to feeding text inputs into the *mistral-embed* model, we conducted standard text normalization processes adopted in previous work [16, 19] to enhance the quality of the input data by applying tokenization, removal of stop words, stemming, and spell-checking. We employed the SpellChecker[9] along with the nltk[10] libraries in Python for these preprocessing steps.

6.2.5 Data collection

We employ the same dataset utilized in chapter 5. The dataset comprises 20 categories of competing apps selected from the top 2,000 popular free-to-download apps from the Google Play Store. The selected apps span diverse categories, e.g., *Navigation*, *Weather*, *Browser*, *FreeCall* and *Dating*, and each category includes a sufficient number of competing apps (e.g., 8 to 10 competing apps) to facilitate competitor user review analysis. To evaluate our approach against the baselines, we use the same five categories used by the baselines [20] to evaluate the precision, namely *Weather, SMS, Bible, Music Player*, and *Sports news*. Similar to previous work, we use the *Free call* and *Cooking recipe* categories for hyper-parameter tuning. Specifically, for each category, we utilize the same statistically representative sample of reviews, i.e., 96 user reviews per category, resulting in a

[6]https://openai.com/chatgpt/
[7]https://huggingface.co/mistralai/Mixtral-8x7B-Instruct-v0.1
[8]https://docs.mistral.ai/api/
[9]https://pypi.org/project/pyspellchecker/
[10]https://pypi.org/project/nltk/

total of 672 ground truth user reviews to evaluate our approach. Table 6.1 summarizes the descriptive statistics of the 70 selected competing app categories. The user reviews of each app are also available, and each user review records the title of the user review, detailed comment, user rating and the posting date. Additionally, we have access to the release notes of the apps which allows us to gain information about the features and updates introduced in each app version.

Table 6.1: Descriptive Statistics of 7 Competing Apps Categories

App category	Number of apps	Number of reviews
Free call	10	291,034
Weather	10	276,941
SMS	10	264,926
Bible	10	64,417
Music player	10	60,685
Sports news	10	57,582
Cooking recipe	10	41,154
Total	70	1,056,739

6.3 Experimental Results

6.3.1 RQ6.1: How effective can LLMs be to extract features from user reviews?

In our work, *LLM-Cure* suggests enhancements to developers to improve their app features by identifying features from user reviews. Therefore, we want to evaluate the capabilities of the LLM in automatically extracting meaningful features from user reviews to understand the feasibility and potential of *LLM-Cure* for real-world applications.

Evaluation Metrics. We assess the performance of *LLM-Cure* by comparing predicted features by the LLM against the ground truth. We employ three key metrics: 1) True Positives (TP) representing the correctly predicted features, 2) False Positives (FP) representing the number of falsely predicted features, and 3) False Negatives (FN) representing features present in reviews but not predicted by *LLM-Cure*. We adopt the

precision, recall and F1-score as evaluation metrics, and we calculate them as follows:

$$Precision = \frac{TP}{TP + FP} \tag{6.3}$$

$$Recall = \frac{TP}{TP + FN} \tag{6.4}$$

$$F_1\text{-}Score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{6.5}$$

To evaluate the performance metrics of feature extraction, the first and second authors, as two independent annotators, manually label 672 testing reviews. The Cohen's Kappa agreement score [58] is computed on the annotated testing reviews, yielding a high score of 0.82, indicative of a high level of agreement.

Experimental Setup. *LLM-Cure* has three hyper-parameters: k the number of the features, the *similarity threshold* and the *convergence threshold*. Previous work that uses the dataset demonstrates that 14 leads to the best results when set as the number of features. Therefore, we set k as 14, aligning with previous work on this dataset [20]. To set the *similarity threshold* and the *convergence threshold*, we conduct experiments on the two validation sets, i.e., *"Recipe cooking"* and *"Free Call"* app categories. The results indicate that a *similarity threshold* of 0.75 coupled with a *convergence threshold* of 5 yields the highest precision. Therefore, we adopt these hyper-parameter values for the testing set. In addition, given the limited context size of LLMs, we set the batch size to 1,000 reviews to accommodate the prompt without exceeding the size limitations of the selected Mistral model.

Baselines. To assess the efficacy of *LLM-Cure* in automatically identifying and assigning features to user reviews, we compare its performance against existing baselines FeatCompare [20] and Attention-based Aspect Extraction (ABAE) [98], using the ground truth of 480 labeled reviews. In addition, we include a baseline called *LLM-Basic*. Similar to *LLM-Cure*, in *LLM-Basic*, we prompt the LLM to extract features from user reviews. However, in this baseline, we do not include the incremental process of batching, i.e., *batch-and-match*, used in *LLM-Cure*. Instead, we select a statistical sample of reviews from the set of all reviews that fits the context size of the LLM (i.e., 1,000 reviews) and

use the same prompt used for *LLM-Cure*. Our goal is to verify that the incremental process adds value to the extraction and improves the performance.

Prompt Construction. To construct the prompts, we adhere to the template provided by Mistral for our selected model[11]. *LLM-Cure* leverages two distinct prompts for Phase 1: the *Feature extraction* prompt and *Feature assignment* prompt. The *Feature extraction* prompt is illustrated in Figure 6.2. The *Feature extraction* and *Review Classification* prompts are available in our research artifact.

Results. *LLM-Cure* is capable of identifying and assigning features with high F1-score, recall and precision. Table 6.2 shows the fourteen features extracted for three sample categories testing apps categories. Across the five testing app groups, *LLM-Cure* exhibits F1-scores ranging from 80% to 91%, with precision between 81% and 92% and recall between 80% and 90%. These findings underscore the capability of LLMs to extract features from user reviews without requiring manual annotation.

***LLM-Cure* significantly outperforms FeatCompare and ABAE baselines in F1-score, recall and precision across the testing apps.** Table 6.4 shows that on average, *LLM-Cure* achieves a 7% improvement in F1-score, a 9% improvement in recall and a 4% in precision as compared to the baselines. We conduct paired t-tests to confirm that the improvement is statistically significant in terms of F1-score, recall and precision.

The *batch-and-match* process of *LLM-Cure* improves the performance of feature extraction. *LLM-Basic*, which only processes a single batch of reviews, achieves lower performance compared to *LLM-Cure*. These results highlight the benefit of *LLM-Cure's* incremental processing, *batch-and-match*, and its ability to extract features more effectively and with higher performance (e.g., F1-score). Instead of processing the entire set of user reviews, *LLM-Cure* processes only a fraction of the total reviews. Table 6.3 illustrates the percentage of user reviews needed by *LLM-Cure* to achieve convergence and extract the features. *LLM-Cure* outperforms all baseline methods while processing only between 3% and 30% of user reviews, achieving feature saturation without the need for processing the entire dataset.

[11]https://huggingface.co/mistralai/Mixtral-8x7B-Instruct-v0.1

Table 6.2: Top fourteen features extracted from the user reviews of three sample app categories.

Feature	Description
	Weather
Accuracy	App's ability to provide accurate weather forecasts
Radar	App's radar and map and visualization features
Weather forecast	App's hourly and daily forecasts
Additional features	App's additional features, such as pollen counts and UV index
Notifications	App's ability to send notifications for weather alerts
Customization	App's ability to be customized, i.e., adding multiple locations
Battery usage	App's impact on the device battery life
Customer support	Users' experiences with the app's customer support
Ease of use	App's ease of use including widgets and how to navigate
Ads	App's use of ads, including how intrusive they are
Device compatibility	App's compatibility with different devices and systems
Design and layout	App's design and layout
Performance	App's stability, including how often it crashes or freezes
Updates	App's frequency and quality of updates
	SMS
Group messaging	App's ability to send messages to multiple recipients at once
Dual SIM support	App's ability to support dual SIM devices
Account authentication	App's ability to identify and authenticate a user's account
Customization	ability to customize the app's appearance and/or functionality
Spam identification	App's ability to identify and filter out spam messages
MMS support	App's ability to send and receive multimedia messages
Integration with other apps	App's ability to integrate with other apps
In-app ads	presence of advertisements within the app
Notifications	App's ability to send notifications for incoming messages
Backup and restore	App's ability to back up and restore messages and settings
User interface (UI)	design and layout of the app
Call blocking	App's ability to block unwanted calls
Caller identification	App's ability to identify the caller's name and/or location
Privacy	App ability to protect the user's privacy
	Sport News
Device Compatibility	App's ability to run on various devices and systems
Live streaming	App's ability to live streaming sports events
Score updates	App's ability to provide real-time scores and updates for games
Sports coverage	App's variety of types of sports and coverage
Notifications	App's ability to send alerts and notifications
User interface	App's design and layout including ease of navigation
Chromecast support	App's ability to cast content to a TV using Chromecast
Performance	App's performance, including general quality and crashes
Ads	The presence and frequency of advertisements in the app
Customer support	App's quality and responsiveness of the customer support
Video quality	App's resolution and overall quality of the video streams
Customization	App's ability to customize the settings
Subscription service	App's requirement for a paid subscription
Providers Integration	App's ability to integrate with other services (cable providers)

***LLM-Cure* performs consistently across different sentiment categories.** To further investigate whether *LLM-Cure* classifies positive versus negative reviews with different precision, we conduct an analysis based on the sentiment categories. We find that positive and negative reviews present similar results across categories, with at most 3% of differences. The average precision for positive reviews across these categories is 85.69%, while for negative reviews, it is 85.78%. These findings indicate that there is no significant difference in classification F1-score between positive and negative reviews, demonstrating that our approach is not sensitive to the sentiment of the reviews.

Table 6.3: The number of batches and percentage of user reviews required to extract the features using LLM-Cure

App category	**Number of batches**	**Percentage of reviews**
Weather	7	3%
Bible	4	6%
SMS	24	9%
Sport News	8	14%
Music player	19	31%

Table 6.4: Performance comparison of *LLM-Cure* and baselines on five testing app groups in features assignment. 'P' denotes Precision, and 'R' denotes Recall and 'F_1' denotes F1-score.

App	**LLM-Cure**			**LLM-Basic**			**FeatCompare**			**ABAE**		
Category	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
Weather	**92**	**90**	**91**	80	77	79	81	74	78	64	64	67
Sports News	81	**80**	**80**	67	65	66	**82**	75	78	71	65	68
Bible	**83**	**83**	**83**	78	75	77	81	77	79	72	69	70
SMS	**86**	**83**	**85**	77	76	77	80	74	77	67	62	64
Music Player	**86**	**86**	**86**	82	82	82	79	75	77	70	66	68
Average	**86**	**84**	**85**	77	75	76	82	75	78	68	65	68

Summary of RQ6.1

LLM-Cure achieves promising F1-scores ranging from 80% to 91% across the five test sets, demonstrating its effectiveness in analyzing user reviews without manual data labeling. The obtained F1-scores surpass the performance of the baselines by an average of 7%. The *batch-and-match* process enables LLM-Cure to achieve a high F1-score with a substantial reduction in the required user review data.

6.3.2 RQ6.2: Can LLMs leverage categorized user reviews to generate suggestions for feature improvements?

RQ6.2 investigates whether *LLM-Cure* can leverage categorized user reviews of competitor apps to generate suggestions for feature improvements. Analyzing competitors' positive user reviews allows developers to identify successful features and user preferences across the market, ensuring their app remains competitive and relevant. By incorporating these insights from competitor reviews, *LLM-Cure* empowers developers to make data-driven decisions about feature enhancement, prioritize user needs, and ultimately create a more competitive app.

Suggestions validation. To assess the relevance of the suggestions provided by *LLM-Cure*, we conduct a retrospective investigation at the app release level. Specifically, we examine the release notes of subsequent versions of the target app following the date of the user review containing the complaint and calculate the *Suggestions Implementation Rate (SIR)*. We define the *SIR* the number of suggestions by *LLM-Cure* matched in the release notes divided by the total number of suggestions provided as follows:

$$\text{SIR} = \frac{\text{Number of Suggestions Matched in Released Notes}}{\text{Total Number of Suggestions by LLM-Cure}} \tag{6.6}$$

Table 6.6: Sample of *LLM-Cure's* Suggestions and Matching Release Notes

LLM-Cure's Suggestions	**Release Notes**
Add a customizable notification sound: Allowing users to set their preferred notification sound can significantly improve their experience	Fixed personalized sound notification
Improve the quality of MMS messages: The app should allow users to send high-quality images and videos through MMS	High quality picture compression when MMS
Reliable MMS delivery: Improve the MMS delivery system to ensure that messages are sent and received reliably	Improved send /receive MMS known issues
Improve international sports coverage: Some users have requested better coverage of international sports, such as rugby and cricket	Expanded golf coverage including schedule leaderboards scorecards tee times and rankings
Add a dedicated tab for NHRA events: To address the user complaint, the app could add a specific tab for NHRA events, making it easier for users to find and access coverage of these events	Enhanced home screen with all your favorite team scores news and live streams in one place

Table 6.5: Suggestions Implementation Rate (SIR) of *LLM-Cure* Feature Improvement Suggestions on the selected three apps.

App name	**# of releases**	**Avr. words per release**	**Underp. features**	**# of neg. reviews**	**# of pos. reviews**	**SIR**
Weather & Clock Widget	44	36	Accuracy	336	8,383	8/9 = 89%
			Ease of Use	159	4,313	5/9 = 56%
			Performance	157	1,951	4/9 = 44%
Handcent Next SMS Messenger	140	30	User Interface	316	686	9/9 = 100%
			Notifications	297	359	6/9 = 67%
			MMS Support	229	162	7/9 = 78%
FOX Sports: Watch Live	36	23	Stability and Performance	27	1,947	8/9 = 89%
			Sports Coverage	26	5,640	4/9 = 44%
			Notifications	18	6,558	8/9 = 89%
Total SIR						**59/81 = 73%**

Experimental Setup. We randomly select one app from each of the three random categories as target apps for feature improvement suggestions. We select the apps with a substantial number of informative release notes to ensure that we have a rich data source

for conducting the manual suggestion validation. Specifically, we choose *Handcent Next SMS messenger*[12] from the SMS category, *FOX Sports: Watch Live*[13] from the Sports News category, and *Weather & Clock Widget*[14] from the Weather app category. As shown in Table 6.5, the chosen Weather, SMS, and Sports News apps have 40, 140, and 36 release notes, respectively, with average word counts of 36, 30, and 23 per release. For each app, we apply steps 1 to 5 of *LLM-Cure*'s Phase 2. We focus on the top three underperforming features that require the most attention from developers. For each feature, we identify three target complaints. For each user complaint, we generate suggestions to improve the app features. We obtain a total of 9 suggestions per feature, resulting in 27 suggestions per app. We then calculate *SIR* for each feature.

Results. *LLM-Cure* achieves a promising *SIR* of 59 out of 81 (i.e., 73%), indicating the majority of the suggestionss from *LLM-Cure* are implemented by the developers. Table 6.5 shows the SIR for each feature across the three apps. The results indicate that some features received higher *SIRs* than others. For example, all the User Interface-related suggestions for *"Handcent Next SMS messenger"* were implemented, while not all Notification-related suggestions for the same app were adopted. This variation can be attributed to different development priorities or challenges associated with certain feature enhancements. Furthermore, the release notes sometimes contained high-level descriptions of updates, such as "Performance improvement," which may not explicitly detail the changes but could reflect the overall enhancement suggested by *LLM-Cure*.

***LLM-Cure* successfully leverages candidate positive user reviews from competitors to provide suggestions for feature improvements related to user complaints.** We demonstrate the process with a case study on the *"Ease of Use"* feature for the *Weather & Clock Widget* app. A user review highlights the issue of the weather widget not updating automatically, requiring manual intervention:

[12]https://play.google.com/store/apps/details?id=com.handcent.app.nextsms
[13]https://play.google.com/store/apps/details?id=com.foxsports.android
[14]https://play.google.com/store/apps/details?id=com.devexpert.weather

"Widget won't update over time it keeps showing same weather and same TIME until I tap on it and open weather app."

★★★★★

LLM-Cure proposes the below three suggestions:

LLM-Cure suggestions for the Ease of Use complaint

Automatic Refresh: Implement an automatic refresh feature for the widget that updates the weather and time data periodically, so users don't have to manually tap on the widget to see the current weather and time.

Improved Time Synchronization: Enhance the clock functionality in the widget to ensure that it stays in sync with the device's internal clock, eliminating the need for manual time updates.

Clear Refresh Icon and Options: Provide a clear refresh icon and add options for users to manually refresh the widget, change clock fonts, and customize the weather update intervals to their preference. This will give users more control over the widget's behavior and improve their overall experience.

The target review identified two weaknesses in the widget: **(1) unreliable time-keeping** and **(2) an unfriendly design**. *LLM-Cure* tackles these issues by proposing automatic refresh functionality, improved time synchronization, and the implementation of a refresh icon. These suggestions directly address the user's frustrations and aim to improve the widget's usability.

Subsequently, we cross-referenced these suggestions with the app's release notes and found that developers implemented recommendations provided by *LLM-Cure* in subsequent releases. For instance, in release *6.1.0.1*, the functionalities *"Added option to show forecast every 3 hours on the widget when you select hourly forecast"* and *"Enabled digital*

font for clock and date" were introduced, aligning closely with the suggestions aimed at enhancing the time synchronization of the widget.

***LLM-Cure's* suggestions often align with functionalities later implemented by the apps documented in the release notes.** Table 6.6 presents a random sample of *LLM-Cure* suggestions alongside corresponding release notes selected from 81 of the total suggestions, demonstrating the alignment between the feature improvement suggestions and the actual implementations. These results provide encouraging evidence that *LLM-Cure* can effectively analyze competitors' user reviews and generate suggestions for feature improvements.

Summary of RQ2

LLM-Cure successfully leverages candidate positive reviews of competitors to generate feature improvement suggestions for user complaints. *LLM-Cure* achieves a promising average of 73% of Suggestions Implementation Rate (SIR), demonstrating its potential for competitive feature enhancement.

6.4 Threats to Validity

Threats to construct validity relate to a possible error in the data preparation. In *LLM-Cure*, we adopt a batch-and-match methodology to accommodate a scalable LLM prompting by employing a subset of reviews to extract features. We evaluate the validity of our approach by testing it across various thresholds and validating against ground truth data, ensuring robustness in representation. To reduce any bias that may be introduced by the order of reviews, we shuffle user reviews before splitting them to ensure a representative sample of user reviews.

In Phase 2, i.e., the Suggestion Generation with Competitor Reviews phase, we suggest addressing the most underperforming features, i.e., having the highest number of negative reviews in the target app. We acknowledge that developers may employ different prioritization (e.g., severity-based and effort-based) techniques in the real world.

However, our methodology remains valid. Our approach is anchored in the actual reviews rather than the selection approach, thus ensuring the reliability of our results. The developer can decide to select complaints from any of the features.

Threats to internal validity relate to the concerns that might come from the internal methods used in our study. One threat may stem from the selection and design of the prompt templates. To address this potential threat, we explore different prompts. When provided with the same shot examples, we observe that the prompt template does not lead to different classification results. Another potential threat relates to the summarization of user reviews by the LLM. Since we do not have a clear measure of how accurate the LLM-generated summaries are, this introduces a degree of uncertainty. However, the final improvement suggestions were implemented and verified through the release notes of the projects, which strengthens the validity of our approach. Additionally, we limit the LLM to providing three suggestions per review, though developers could request more suggestions if needed, allowing them to cover additional items from the LLM's summaries. This flexibility mitigates the risk of missing important improvement opportunities.

Threats to external validity concern the ability to generalize the results. One threat concerns the choice of the LLM utilized in our approach, *LLM-Cure*. We acknowledge that each LLM possesses a distinct architecture, potentially leading to variations in results. However, we opt for Mistral for several reasons. As detailed in Section 6.2.4, Mistral outperforms other models across various benchmarks. Furthermore, Mistral is openly available for researchers, fostering the replication of our work and ensuring transparency in the evaluation process. Despite these considerations, it's important to recognize that the choice of LLM remains a potential source of variability in our findings. Another threat to external validity pertains to the selection of categories and datasets. However, similar to prior research [20], we mitigate this concern by evaluating the validity of our approach across five distinct app categories sourced from ground truth data. This approach aims to minimize the influence of app selection bias. Additionally, our method is platform-agnostic, offering applicability to any mobile app, provided it contains user

reviews. This broad applicability enhances the generalizability of our findings beyond specific categories or datasets.

6.5 Summary

In this work, we introduce *LLM-Cure*, a novel LLM-based approach that conducts Competitor User Review Analysis for Feature Enhancement. *LLM-Cure* generates automated suggestions for app feature improvements by leveraging user reviews from competitor apps to enhance user experience and maintain competitiveness. We evaluate *LLM-Cure* on 1,056,739 reviews of 70 popular Android apps. Our results suggest that:

- *LLM-Cure* achieves high performance in extracting and assigning features to user reviews, outperforming baseline methods significantly by up to 13% in F1-score, up to 16% in recall and up to 11% in precision.
- *LLM-Cure* achieves a promising suggestions implementation rate of 58 out of 81.
- By combining user feedback with competitor user review analysis, *LLM-Cure* empowers developers to make informed decisions, fostering a more competitive landscape.

Data Availability. We provide a replication package including the data and scripts to replicate the analyses at `https://github.com/repl-pack/LLM-Cure`.

Chapter 7

Conclusions and Future Work

In the following sections, we summarize the contributions of this thesis and propose potential research opportunities to further advance our work.

7.1 Contributions

The main contributions of the thesis are outlined below.

(1) Proposing an approach for predicting change impact for defect resolution leveraging issue report information. Issue reports contain valuable information, such as, the title, description and severity, and researchers leverage the topics of issue reports as a collective metric portraying similar characteristics of a defect. Nonetheless, none of the existing studies leverage the defect topic, *i.e.*, a semantic cluster of defects of the same nature, such as *Performance, GUI and Database*, to estimate the change impact that represents the amount of change needed in terms of code churn and the number of files changed. We conduct an empirical study on 298,548 issue reports belonging to three large-scale open-source systems, *i.e.*, Mozilla, Apache and Eclipse, to estimate the change impact in terms of code churn or the number of files changed while leveraging the topics of issue reports. First, we adopt the Embedded Topic Model (ETM), a state-of-the-art topic modelling algorithm, to identify the topics. Second, we investigate the feasibility of predicting the change impact using the identified topics and other information extracted from the issue reports by building eight prediction models that classify issue reports requiring small or large change impact along two dimensions, *i.e.*, the code churn size and

the number of files changed.

Summary

Our results suggest that 1) XGBoost is the best-performing algorithm for predicting the change impact, with an AUC of 0.84, 0.76, and 0.73 for the code churn and 0.82, 0.71 and 0.73 for the number of files changed metric for Mozilla, Apache, and Eclipse, respectively and our results also demonstrate that 2) the topics of issue reports improve the recall of the prediction model by up to 45%.

(2) Understanding code clone dynamics in DL frameworks. DL frameworks play a critical role in advancing artificial intelligence, and their rapid growth underscores the need for a comprehensive understanding of software quality and maintainability. DL frameworks, like other systems, are prone to code clones that can have positive and negative implications for software development, influencing maintenance, readability, and bug propagation. However, no work has been done investigating clones, their evolution and their impact on the maintenance of DL frameworks. We address the knowledge gap concerning the evolutionary dimension of code clones in DL frameworks and the extent of code reuse across these frameworks. We empirically analyze code clones in nine popular DL frameworks, i.e., *TensorFlow*, *Paddle*, *PyTorch*, *Aesara*, *Ray*, *MXNet*, *Keras*, *Jax* and *BentoML*, to investigate (1) the characteristics of the long-term code cloning evolution over releases in each framework, (2) the short-term, i.e., within-release, code cloning patterns and their influence on the long-term trends, and (3) the file-level code clones within the DL frameworks.

Summary

Our findings reveal that 1) DL frameworks adopt four distinct cloning trends: *"Serpentine"*, *"Rise and Fall"*, *"Decreasing"*, and *"Stable"* and that these trends present some common and distinct characteristics. For instance, bug-fixing activities persistently happen in clones irrespective of the clone evolutionary trend but occur more in the *"Serpentine"* trend. Moreover, 2) the within-release level investigation demonstrates that short-term code cloning practices impact long-term cloning trends. The cross-framework code clone investigation 3) reveals the presence of *functional* and *architectural adaptation* file-level cross-framework code clones across the nine studied frameworks. We provide insights that foster robust clone practices and collaborative maintenance in the development of DL frameworks.

(3) Proposing an approach for automating feature-level comparative user review analysis for competing mobile applications. Given the competitive mobile app market, developers must be fully aware of users' needs, satisfy users' requirements, combat apps of similar functionalities (i.e., *competing apps*), and thus stay ahead of the competition. Developers need to read reviews from all their interested competing apps and summarize the advantages and disadvantages of each app. Such a manual process can be tedious and even infeasible, with thousands of reviews posted daily. To help developers compare users' opinions among competing apps on *high-level features*, such as the main functionalities and the main characteristics of an app, we propose a review analysis approach named *FeatCompare*. FeatCompare can automatically identify high-level features mentioned in user reviews without any manually annotated resources. Then, FeatCompare creates a comparative table that summarizes users' opinions for each identified feature across competing apps. FeatCompare features a novel neural network-based model named **G**lobal-**L**ocal sensitive **F**eature **E**xtractor (GLFE), which extends Attention-based Aspect Extraction (ABAE), a state-of-the-art model for extracting high-level features from reviews. We apply FeatCompare to ten million user reviews of 196 popular apps on Google Play that belong to 20 different functional groups. We also

conduct a quantitative evaluation of GLFE is conducted for five randomly picked groups of competing apps.

Summary

The evaluation results show that 1) GLFE outperforms the state-of-the-art high-level feature extraction model ABAE by 14.7% on average. We also survey 107 mobile app developers asking how they perform competitor analysis in practice and how they perceive the comparative table created by FeatCompare. 2) 73% of the mobile app developers participants endorse the benefits of FeatCompare in competitor analysis.

(4) Proposing an automatic feature enhancement suggestions approach through competitor user review analysis. As user satisfaction is paramount to the success of a mobile application (app), developers typically rely on user reviews, which represent user feedback that includes ratings and comments to identify areas for improvement. However, the sheer volume of user reviews poses challenges in manual analysis, necessitating automated approaches. Existing automated approaches either analyze only the target app's reviews, neglecting the comparison of similar features to competitors or fail to provide suggestions for feature enhancement. To address these gaps, we propose a *Large Language Model* ***(LLM)****-based* ***C****ompetitive* ***U****ser* ***R****eview Analysis for Feature* ***E****nhancement)* (*LLM-Cure*), an approach powered by LLMs to automatically generate suggestions for mobile app feature improvements. More specifically, *LLM-Cure* identifies and categorizes features within reviews by applying LLMs. When provided with a complaint in a user review, *LLM-Cure* curates highly rated (4 and 5 stars) reviews in competing apps related to the complaint and proposes potential improvements tailored to the target application. We evaluate *LLM-Cure* on 1,056,739 reviews of 70 popular Android apps.

Summary

Our evaluation demonstrates that 1) *LLM-Cure* significantly outperforms the state-of-the-art approaches in assigning features to reviews by up to 13% in F1-score, up to 16% in recall and up to 11% in precision. Additionally, 2) *LLM-Cure* demonstrates its capability to provide suggestions for resolving user complaints. We verify the suggestions using the release notes that reflect the changes in features in the target mobile app. 3) *LLM-Cure* achieves a promising average of 73% of the implementation of the provided suggestions, demonstrating its potential for competitive feature enhancement.

Even though each study in this thesis targets specific datasets, the approaches developed are broadly applicable to any software product. For example, the competitive analysis of user reviews and feature improvement suggestions can be applied to other software systems such as AlternativeTo [11] where our approach can provide a way to obtain and compare user reviews across various products. Similarly, the change impact prediction using issue reports and the analysis of code clones can be extended to mobile applications, given that many open-source mobile apps use platforms like GitHub for issue tracking, and their codebases are also prone to code cloning. Therefore, the implications of the contributions made in this thesis can be extended beyond the specific datasets studied, offering value to other software product where user reviews, source code, and issue reports are available.

7.2 Future Work

Our proposed approaches positively impact software maintenance, including code maintenance and feature improvement, by providing automated approaches leveraging software artifacts and machine learning. There are several areas where the thesis can be further extended. In this section, we discuss potential extensions for future research.

Incorporating Change Impact Prediction into Defect Assignment Tools. Our proposed change impact analysis approach predicts the amount of changes associated

with fixing a defect, terms of lines of code and the number of files changed using issue report information. A promising direction for future work is to integrate our change impact prediction method into existing defect assignment tools. Defect assignment tools play a crucial role in software maintenance by efficiently assigning open issue reports to the appropriate developer. Incorporating change impact predictions can enhance these tools in several ways. First, it can lead to enhanced assignment accuracy based on the developer's availability. Second, it contributes to resource allocation optimization by knowing the anticipated number of changes needed for effective planning.

Expanding the Scope of Code Clone Analysis in Deep Learning Frameworks. We conducted code analysis for software maintenance by examining code clones within deep learning frameworks, focusing specifically on projects written in Python. Our findings have provided valuable insights into the nature and impact of code clones in Python-based DL projects. However, to achieve a more comprehensive understanding of code cloning across diverse programming languages and ecosystems, further research is necessary. Future work can broaden the scope of our study beyond Python projects to include other programming languages, such as C++ and Java. This will help identify language-specific patterns and challenges in code maintenance within DL frameworks. Additionally, comparing the cloning patterns across languages may reveal unique characteristics and commonalities that may inform better maintenance strategies. Finally, understanding code cloning in multiple languages will foster cross-language insights, enabling the transfer of best practices and lessons learned from one language ecosystem to another. This can lead to improved overall practices in deep learning framework development and maintenance.

Investigating the Evolution of Feature Ratings Across App Releases. We propose an automatic approach for feature comparison across competing mobile apps, focusing on providing feature ratings. Our methodology has provided significant insights into the comparative performance of app features, enabling users and developers to make informed decisions. However, there are several promising directions for extending this work. Future work can expand our study to investigate how high-level feature ratings

evolve over multiple releases among competing mobile apps. This expansion will allow to track feature evolution. For instance, tracking the evolution of features over time may help developers understand how feature improvements, bug fixes, and updates impact user perception and satisfaction. Additionally, studying feature ratings over multiple releases may enable developers to identify trends and patterns in feature development that can help uncover insights into which features consistently improve, which ones decline, and the factors driving these changes.

Enhancing Feature Analysis through LLM Orchestration and Prioritization. We propose an approach that leverages LLMs and competitive user reviews to conduct competitive analysis and suggest feature improvements. Our methodology has demonstrated significant potential in enhancing the understanding and development of app features. We identify several areas for future work that may refine and expand the capabilities of our approach. Future work can aim to explore agent orchestration between LLMs to streamline the feature analysis process and leverage the strengths of different language models. Additionally, *LLM-Cure*'s capabilities can be extended to prioritize suggestions based on factors such as popularity and potential user impact.

Bibliography

[1] This is what your developers are doing 75% of the time, and this is the cost you pay. https://coralogix.com/log-analytics-blog/this-is-what-your-developers-are-doing-75-of-the-time/. Accessed: 2021-06-17.

[2] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[3] Github Ranking, 2024. https://github.com/EvanLi/Github-Ranking#most\protect\discretionary{\char\hyphenchar\font}{}{}stars.

[4] Nurul Haszeli Ahmad, Syed Ahmad Aljunid, and Jamalul-lail Ab Manan. Taxonomy of c overflow vulnerabilities attack. In Jasni Mohamad Zain, Wan Maseri bt Wan Mohd, and Eyas El-Qawasmeh, editors, *Software Engineering and Computer Systems*, pages 376–390, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[5] Nurul Haszeli Ahmad, Syed Ahmad Aljunid, and Jamalul-lail Ab Manan. Taxonomy of c overflow vulnerabilities attack. In Jasni Mohamad Zain, Wan Maseri bt Wan Mohd, and Eyas El-Qawasmeh, editors, *Software Engineering and Computer Systems*, pages 376–390, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[6] Hafiza Anisa Ahmed, Narmeen Zakaria Bawany, and Jawwad Ahmed Shamsi. Capbug-a framework for automatic bug categorization and prioritization using nlp and machine learning algorithms. *IEEE Access*, 9:50496–50512, 2021.

[7] Mistral AI. Mixtral of experts, May 2024.

[8] Shirin Akbarinasaji. Prioritizing lingering bugs. *SIGSOFT Softw. Eng. Notes*, 43(1):1–6, mar 2018.

[9] Akdeniz. Google Play Crawler. `https://github.com/Akdeniz/google-play-crawler`, 2013. (Last accessed: March 2020).

[10] A. AlSubaihin, F. Sarro, S. Black, L. Capra, and M. Harman. App store effects on software engineering practices. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[11] AlternativeTo. Alternativeto - crowdsourced software recommendations. `https://alternativeto.net/`. Accessed: 2024-09-24.

[12] Theodoros Amanatidis and Alexander Chatzigeorgiou. Studying the evolution of php web applications. *Inf. Softw. Technol.*, 72(C):48–67, apr 2016.

[13] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300, 2019.

[14] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 361–370, New York, NY, USA, 2006. Association for Computing Machinery.

[15] AppAnnie. App Annie. `https://www.appannie.com/`, 2016. (Last accessed March 2020).

[16] Nimasha Arambepola, Lankeshwara Munasinghe, and Nalin Warnajith. Factors influencing mobile app user experience: An analysis of education app user reviews. In *2024 4th International Conference on Advanced Research in Computing (ICARC)*, pages 223–228, 2024.

[17] Pasquale Ardimento and Andrea Dinapoli. Knowledge extraction from on-line open source bug tracking systems to predict bug-fixing time. In *Proceedings of the 7th International Conference on Web Intelligence, Mining and Semantics*, WIMS '17, New York, NY, USA, 2017. Association for Computing Machinery.

[18] Pasquale Ardimento and Costantino Mele. Using bert to predict bug-fixing time. In *2020 IEEE Conference on Evolving and Adaptive Intelligent Systems (EAIS)*, pages 1–7, 2020.

[19] Maram Assi, Safwat Hassan, Stefanos Georgiou, and Ying Zou. Predicting the change impact of resolving defects by leveraging the topics of issue reports in open source software systems. *ACM Trans. Softw. Eng. Methodol.*, 32(6), sep 2023.

[20] Maram Assi, Safwat Hassan, Yuan Tian, and Ying Zou. Featcompare: Feature comparison for competing mobile apps leveraging user reviews. *Empirical Software Engineering*, 26(5), 2021.

[21] Maram Assi, Safwat Hassan, Yuan Tian, and Ying Zou. Featcompare: Feature comparison for competing mobile apps leveraging user reviews. *Empirical Software Engineering*, 26, 07 2021.

[22] 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, 2007.

[23] Liliane Barbour, Le An, Foutse Khomh, Ying Zou, and Shaohua Wang. An investigation of the fault-proneness of clone evolutionary patterns. *Software Quality Journal*, 26:1187–1222, 2018.

[24] 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, 2011.

[25] Houssem Ben Braiek, Foutse Khomh, and Bram Adams. The open-closed principle of modern machine learning frameworks. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 353–363, 2018.

[26] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3(null):1137–1155, mar 2003.

[27] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155, 2003.

[28] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13(null):281–305, feb 2012.

[29] Donald J. Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, AAAIWS'94, page 359–370. AAAI Press, 1994.

[30] Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Adithya Abraham Philip. Orca: Differential bug localization in large-scale services. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 493–509, USA, 2018. USENIX Association.

[31] Pamela Bhattacharya and Iulian Neamtiu. Bug-fix time prediction models: Can we do better? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, page 207–210, New York, NY, USA, 2011. Association for Computing Machinery.

[32] Steven Bird and Edward Loper. Nltk: The natural language toolkit. In *Proceedings of the ACL 2004 on Interactive Poster and Demonstration Sessions*, ACLdemo '04, page 31–es, USA, 2004. Association for Computational Linguistics.

[33] Tegawendé F. Bissyandé, David Lo, Lingxiao Jiang, Laurent Réveillère, Jacques Klein, and Yves Le Traon. Got issues? who cares about it? a large scale investigation of issue trackers from github. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 188–197, 2013.

[34] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3(null):993–1022, mar 2003.

[35] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent Dirichlet Allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[36] Barry W. Boehm, Chris Abts, A. Winsor Brown, Sunita Chulani, Bradford K. Clark, Ellis Horowitz, Ray Madachy, Donald J. Reifer, and Bert Steece. *Software Cost Estimation with COCOMO II*. Prentice Hall Press, USA, 1st edition, 2009.

[37] Shawn A. Bohner and Robert S. Arnold. *Software change impact analysis*. IEEE Computer Society Press, 1996.

[38] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[39] Published by Laura Ceci and Mar 4. Messaging apps: Most popular by global downloads 2024, Mar 2024.

[40] Haipeng Cai and Douglas Thain. Distia: A cost-effective dynamic impact analysis for distributed programs. ASE 2016, page 344–355, New York, NY, USA, 2016. Association for Computing Machinery.

[41] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. Gzoltar: an eclipse plug-in for testing and debugging. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 378–381, 2012.

[42] L. V. G. Carreño and K. Winbladh. Analysis of user comments: An approach for software requirements evolution. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 582–591, 2013.

[43] Laura V. Galvis Carreño and Kristina Winbladh. Analysis of user comments: An approach for software requirements evolution. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 582–591, 2013.

[44] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software*, 152:165–181, 2019.

[45] Laura Ceci. Number of mobile app downloads worldwide from 2016 to 2023. `https://www.https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/`, 2024. (Last accessed May 2024).

[46] N. B. Chaleshtari and S. Parsa. Smbfl: slice-based cost reduction of mutation-based fault localization. *Empirical Software Engineering*, 25:4282–4314, 2020.

[47] Ned Chapin, Joanne E. Hale, Khaled Md. Khan, Juan F. Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30, 2001.

[48] An Ran Chen, Tse-Hsun Peter Chen, and Shaowei Wang. Pathidea: Improving information retrieval-based bug localization by re-constructing execution paths using logs. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.

[49] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. Toward understanding deep learning framework bugs. *ACM Trans. Softw. Eng. Methodol.*, 32(6), sep 2023.

[50] Ning Chen, Jialiu Lin, Steven C. H. Hoi, Xiaokui Xiao, and Boshen Zhang. AR-miner: mining informative reviews for developers from mobile app marketplace. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 767–778, 2014.

[51] Tse-Hsun Chen, Stephen W. Thomas, and Ahmed E. Hassan. A survey on the use of topic models when mining software repositories. *Empirical Softw. Engg.*, 21(5):1843–1919, oct 2016.

[52] Tse-Hsun Chen, Stephen W. Thomas, Meiyappan Nagappan, and Ahmed E. Hassan. Explaining software defects using topic models. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, page 189–198. IEEE Press, 2012.

[53] Zhiyuan Chen, Arjun Mukherjee, and Bing Liu. Aspect extraction with automated prior knowledge learning. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, 06 2014.

[54] Muslim Chochlov, Gul Aftab Ahmed, James Vincent Patten, Guoxian Lu, Wei Hou, David Gregg, and Jim Buckley. Using a nearest-neighbour, bert-based approach for scalable clone detection. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 582–591, 2022.

[55] Agnieszka Ciborowska, Aleksandar Chakarov, and Rahul Pandita. Contemporary cobol: Developers' perspectives on defects and defect location, 2021.

[56] Maëlick Claes and Mika V. Mäntylä. 20-mad: 20 years of issues and commits of mozilla and apache development. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 503–507, New York, NY, USA, 2020. Association for Computing Machinery.

[57] J. Cohen, P. Cohen, S.G. West, and L.S. Aiken. *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*. Routledge, 3rd edition, 2002.

[58] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[59] Riccardo Coppola, Maurizio Morisio, and Marco Torchiano. Scripted gui testing of android apps: A study on diffusion, evolution and fragility. In *Proceedings of the*

13th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE, page 22–32, New York, NY, USA, 2017. Association for Computing Machinery.

[60] Fabiano Dalpiaz and Micaela Parente. RE-SWOT: from user feedback to requirements via competitor analysis. In *Proceedings of the 25th International Working Conference on Requirements Engineering: Foundation for Software Quality*, volume 11412 of *REFSQ '19*, pages 55–70, 2019.

[61] Evan DeFilippis, Stephen Michael Impink, Madison Singell, Jeffrey T Polzer, and Raffaella Sadun. The impact of covid-19 on digital communication patterns. *Humanities and Social Sciences Communications*, 9(1), 2022.

[62] Peter Devine, James Tizard, Hechen Wang, Yun Sing Koh, and Kelly Blincoe. What's inside a cluster of software user feedback: A study of characterisation methods. In *2022 IEEE 30th International Requirements Engineering Conference (RE)*, pages 189–200, 2022.

[63] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[64] Andrea Di Sorbo, Sebastiano Panichella, Carol V Alexandru, Corrado A Visaggio, and Gerardo Canfora. SURF: summarizer of user reviews feedback. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 55–58. IEEE, 2017.

[65] Adji Dieng, Francisco Ruiz, and David Blei. Topic modeling in embedding spaces. *Transactions of the Association for Computational Linguistics*, 8:439–453, 07 2020.

[66] Nicholas DiGiuseppe and James Jones. Fault density, fault types, and spectra-based fault localization. *Empirical Software Engineering*, 20, 08 2014.

[67] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, Lei Li, and Zhifang Sui. A survey on in-context learning, 2023.

[68] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt, 2023.

[69] Paulo Sérgio Henrique Dos Santos, Alberto Dumont Alves Oliveira, Thais Bonjorni Nobre De Jesus, Wajdi Aljedaani, and Marcelo Medeiros Eler. Evolution may come with a price: analyzing user reviews to understand the impact of updates on mobile apps accessibility. In *Proceedings of the XXII Brazilian Symposium on Human Factors in Computing Systems*, IHC '23, New York, NY, USA, 2024. Association for Computing Machinery.

[70] Xiaoting Du, Yulei Sui, Zhihao Liu, and Jun Ai. An empirical study of fault triggers in deep learning frameworks. *IEEE Transactions on Dependable and Secure Computing*, 20(4):2696–2712, 2023.

[71] Felipe Ebert, Fernando Castor, and Alexander Serebrenik. An exploratory study on exception handling bugs in java programs. *Journal of Systems and Software*, 106:82–101, 2015.

[72] Osama Ehsan, Lillane Barbour, Foutse Khomh, and Ying Zou. *Is Late Propagation a Harmful Code Clone Evolutionary Pattern? An Empirical Study*, pages 151–167. Springer Singapore, Singapore, 2021.

[73] Osama Ehsan, Foutse Khomh, Ying Zou, and Dong Qiu. Ranking code clones to support maintenance activities. *Empirical Softw. Engg.*, 28(3), apr 2023.

[74] Omar El Zarif, Daniel Alencar Da Costa, Safwat Hassan, and Ying Zou. On the relationship between user churn and software issues. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 339–349, New York, NY, USA, 2020. Association for Computing Machinery.

[75] Omar El Zarif, Daniel Alencar Da Costa, Safwat Hassan, and Ying Zou. On the relationship between user churn and software issues. MSR '20, page 339–349, New York, NY, USA, 2020. Association for Computing Machinery.

[76] Omar El Zarif, Daniel Alencar da Costa, Safwat Hassan, and Ying Zou. On the relationship between user churn and software issues. In Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup, editors, *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, pages 339–349. ACM, 2020.

[77] eMarketer. Number of apps available in leading app stores as of 4th quarter 2019. `https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/`, 2020. (Last accessed March 2020).

[78] Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. Improving automatically generated code from codex via automated program repair. *ArXiv*, abs/2205.10583, 2022.

[79] Siyue Feng, Wenqi Suo, Yueming Wu, Deqing Zou, Yang Liu, and Hai Jin. Machine learning is all you need: A simple token-based approach for effective code clone detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[80] Bin Fu, Jialiu Lin, Lei Li, Christos Faloutsos, Jason Hong, and Norman Sadeh. Why people hate your app: Making sense of user feedback in a mobile app store. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1276–1284, 2013.

[81] Bin Fu, Jialiu Lin, Lei Li, Christos Faloutsos, Jason Hong, and Norman Sadeh. Why people hate your app: making sense of user feedback in a mobile app store. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, page 1276–1284, New York, NY, USA, 2013. Association for Computing Machinery.

[82] Cuiyun Gao, Yaoxian Li, Shuhan Qi, Yang Liu, Xuan Wang, Zibin Zheng, and Qing Liao. Listening to users' voice: Automatic summarization of helpful app reviews. *IEEE Transactions on Reliability*, 72(4):1619–1631, 2023.

[83] Cuiyun Gao, Jichuan Zeng, Michael R Lyu, and Irwin King. Online app review analysis for identifying emerging issues. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 48–58, 2018.

[84] Cuiyun Gao, Wujie Zheng, Yuetang Deng, David Lo, Jichuan Zeng, Michael R. Lyu, and Irwin King. Emerging app issue identification from user feedback: Experience on wechat. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 279–288, 2019.

[85] Shanquan Gao, Lei Liu, Yuzhou Liu, Huaxiao Liu, and Yihui Wang. Updating the goal model with user reviews for the evolution of an app. *Journal of Software: Evolution and Process*, 32(8):e2257, 2020. e2257 JSME-19-0105.R2.

[86] Malcom Gethers, Huzefa Kagdi, Bogdan Dit, and Denys Poshyvanyk. An adaptive approach to impact analysis from change requests to source code. pages 540–543, 11 2011.

[87] Emanuel Giger, Martin Pinzger, and Harald Gall. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, page 52–56, New York, NY, USA, 2010. Association for Computing Machinery.

[88] Y. Golubev and T. Bryksin. On the nature of code cloning in open-source java projects. In *2021 IEEE 15th International Workshop on Software Clones (IWSC)*, pages 22–28, Los Alamitos, CA, USA, oct 2021. IEEE Computer Society.

[89] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 1025–1035, New York, NY, USA, 2014. Association for Computing Machinery.

[90] Florin Gorunescu. *Data Mining: Concepts, models and techniques*. 06 2011.

[91] Florin Gorunescu. *Data Mining: Concepts, Models and Techniques*. 2011.

[92] Xiaodong Gu and Sunghun Kim. "What parts of your apps are loved by users?" (T). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, pages 760–770, 2015.

[93] Emitza Guzman, Mohamed Ibrahim, and Martin Glinz. Prioritizing user feedback from twitter: A survey report. In *2017 IEEE/ACM 4th International Workshop on CrowdSourcing in Software Engineering (CSI-SE)*, pages 21–24, 2017.

[94] Emitza Guzman and Walid Maalej. How do users like this feature? a fine grained sentiment analysis of app reviews. In *Proceedings of the 22nd International Requirements Engineering Conference*, RE '14, pages 153–162, 2014.

[95] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Bugsjs: a benchmark and taxonomy of javascript bugs. *Software Testing, Verification and Reliability*, 31(4):e1751, 2021. e1751 stvr.1751.

[96] James A. Hanley and Barbara J. McNeil. The meaning and use of the area under a receiver operating characteristic (roc) curve. *Radiology*, 143(1):29–36, 1982.

[97] S. Hassan, C. Bezemer, and A. E. Hassan. Studying bad updates of top free-to-download apps in the google play store. *IEEE Transactions on Software Engineering*, 46(7):773–793, 2020.

[98] Ruidan He, Wee Sun Lee, Hwee Tou Ng, and Daniel Dahlmeier. An unsupervised neural attention model for aspect extraction. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, ACL '17, pages 388–397, 2017.

[99] Tobias Hecking and Loet Leydesdorff. Topic modelling of empirical text corpora: Validity, reliability, and reproducibility in comparison to semantic maps. *ArXiv*, abs/1806.01045, 2018.

[100] I. Herraiz, G. Robles, J.M. Gonzalez-Barahona, A. Capiluppi, and J.F. Ramil. Comparison between slocs and number of files as size metrics for software evolution analysis. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 8 pp.–213, 2006.

[101] Yoshiki Higo, Shinsuke Matsumoto, Shinji Kusumoto, Takashi Fujinami, and Takashi Hoshino. Correlation analysis between code clone metrics and project data on the same specification projects. In *2018 IEEE 12th International Workshop on Software Clones (IWSC)*, pages 37–43, 2018.

[102] Abram Hindle. Green mining: A methodology of relating software change and configuration to power consumption. *Empirical Softw. Engg.*, 20(2):374–409, apr 2015.

[103] Thomas Hirsch and Birgit Hofer. Root cause prediction based on bug reports. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Coimbra, Portugal, October 12-15, 2020*, pages 171–176. IEEE, 2020.

[104] Sharon Chee Yin Ho, Vahid Majdinasab, Mohayeminul Islam, Diego Elias Costa, Emad Shihab, Foutse Khomh, Sarah Nadi, and Muhammad Raza. An empirical study on bugs inside pytorch: A replication study, 2023.

[105] Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. Mutation-based fault localization for real-world multilingual programs (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 464–475, 2015.

[106] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software*

Engineering, ASE '07, page 34–43, New York, NY, USA, 2007. Association for Computing Machinery.

[107] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review, 2024.

[108] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.

[109] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions, 2023.

[110] Lulu Huang and Yeong-Tae Song. Precise dynamic impact analysis with dependency analysis for object-oriented programs. In *5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007)*, pages 374–384, 2007.

[111] Yonghui Huang, Daniel Alencar Costa, Feng Zhang, and Ying Zou. An empirical study on the issue reports with questions raised during the issue resolving process. *Empirical Softw. Engg.*, 24(2):718–750, apr 2019.

[112] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1110–1121, New York, NY, USA, 2020. Association for Computing Machinery.

[113] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of real faults in deep learning systems. In

Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20, page 1110–1121, New York, NY, USA, 2020. Association for Computing Machinery.

[114] Claudia Iacob and Rachel Harrison. Retrieving and analyzing mobile apps feature requests from online reviews. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 41–44, 2013.

[115] Claudia Iacob, Rachel Harrison, and Shamal Faily. Online reviews as first class artifacts in mobile app development. In *Proceedings of the 5th International Conference on Mobile Computing, Applications, and Services*, MobiCASE '13, pages 47–53, 2013.

[116] Katsuro Inoue. *Introduction to Code Clone Analysis*, pages 3–27. Springer Singapore, Singapore, 2021.

[117] 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, 2019.

[118] Judith F. Islam, Manishankar Mondal, Chanchal Kumar Roy, and Kevin A. Schneider. A comparative study of software bugs in clone and non-clone code. In *International Conference on Software Engineering and Knowledge Engineering*, 2017.

[119] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 510–520, New York, NY, USA, 2019. Association for Computing Machinery.

[120] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. Repairing deep neural networks: Fix patterns and challenges. In *Proceedings of the ACM/IEEE*

42nd International Conference on Software Engineering, ICSE '20, page 1135–1146, New York, NY, USA, 2020. Association for Computing Machinery.

[121] Md Rakibul Islam and Minhaz F. Zibran. On the characteristics of buggy code clones: A code quality perspective. In *2018 IEEE 12th International Workshop on Software Clones (IWSC)*, pages 23–29, 2018.

[122] Hadi Jahanshahi and Mucahit Cevik. S-dabt: Schedule and dependency-aware bug triage in open-source bug tracking systems. *Information and Software Technology*, 151:107025, 2022.

[123] Hadhemi Jebnoun, Houssem Ben Braiek, Mohammad Masudur Rahman, and Foutse Khomh. The scent of deep learning code: An empirical study. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 420–430, New York, NY, USA, 2020. Association for Computing Machinery.

[124] Hadhemi Jebnoun, Md Saidur Rahman, Foutse Khomh, and Biruk Asmare Muse. Clones in deep learning code: What, where, and why? *Empirical Softw. Engg.*, 27(4), jul 2022.

[125] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, page 111–120, New York, NY, USA, 2009. Association for Computing Machinery.

[126] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. The symptoms, causes, and repairs of bugs inside a deep learning library. *Journal of Systems and Software*, 177:110935, 2021.

[127] 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, ESEC-FSE '07, page 55–64, New York, NY, USA, 2007. Association for Computing Machinery.

[128] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Ahmed E. Hassan. The impact of correlated metrics on the interpretation of defect models. *IEEE Transactions on Software Engineering*, 47(2):320–331, 2021.

[129] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Christoph Treude. The impact of automated feature selection techniques on the interpretation of defect models. *Empirical Softw. Engg.*, 25(5):3590–3638, sep 2020.

[130] Diederik P Kingma JLB. Adam: A method for stochastic optimization. In *3rd international conference for learning representations, San Diego*, 2015.

[131] T. Johann, C. Stanik, A. M. A. B., and W. Maalej. SAFE: A simple approach for feature extraction from app descriptions and app reviews. In *Proceedings of the 25th International Requirements Engineering Conference*, RE '17, pages 21–30, 2017.

[132] Karen Sparck Jones. Natural language processing: a historical review. *Current issues in computational linguistics: in honour of Don Walker*, pages 3–16, 1994.

[133] Rafael Kallis, Andrea Di Sorbo, Gerardo Canfora, and Sebastiano Panichella. Ticket tagger: Machine learning driven issue classification. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 406–409, 2019.

[134] Cory Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful. In *2006 13th Working Conference on Reverse Engineering*, pages 19–28, 2006.

[135] Nilam Kaushik, Mehdi Amoui, Ladan Tahvildari, Weining Liu, and Shimin Li. Defect prioritization in the software industry: Challenges and opportunities. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 70–73, 2013.

[136] Guolin Ke, Jia Zhang, Zhenhui Xu, Jiang Bian, and Tie-Yan Liu. Tabnn: A universal neural network solution for tabular data. 2018.

[137] Swetha Keertipati, Bastin Tony Roy Savarimuthu, and Sherlock A Licorish. Approaches for prioritizing feature improvements extracted from app reviews. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16, pages 1–6, 2016.

[138] Taghi M. Khoshgoftaar and Edward B. Allen. Logistic regression modeling of software quality. *International Journal of Reliability, Quality and Safety Engineering*, 06:303–317, 1999.

[139] Misoo Kim, Youngkyoung Kim, and Eunseok Lee. Denchmark: A bug benchmark of deep learning-related software. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 540–544, 2021.

[140] Soo-Min Kim, Patrick Pantel, Tim Chklovski, and Marco Pennacchiotti. Automatically assessing review helpfulness. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, EMNLP '06, page 423–430, 2006.

[141] Pavneet Singh Kochhar, Yuan Tian, and David Lo. Potential biases in bug localization: Do they matter? In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 803–814, New York, NY, USA, 2014. Association for Computing Machinery.

[142] Maria Kretsou, Elvira-Maria Arvanitou, Apostolos Ampatzoglou, Ignatios Deligiannis, and Vassilis C. Gerogiannis. Change impact analysis: A systematic mapping study. *Journal of Systems and Software*, 174:110892, 2021.

[143] William H. Kruskal and Wilson Allen Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association*, 47:583–621, 1952.

[144] Max Langenkamp and Daniel N. Yue. How open source machine learning software shapes ai. In *Proceedings of the 2022 AAAI/ACM Conference on AI, Ethics,*

and Society, AIES '22, page 385–395, New York, NY, USA, 2022. Association for Computing Machinery.

[145] Tien-Duy B. Le, Richard J. Oentaryo, and David Lo. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 579–590, New York, NY, USA, 2015. Association for Computing Machinery.

[146] Meir Lehman and Juan Fernandez-Ramil. Software evolution and software evolution processes. *Annals of Software Engineering*, 14:275–309, 12 2002.

[147] Stanislav Levin and Amiram Yehudai. The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–46, 2017.

[148] Bixin Li, Xiaobing Sun, Hareton K. N. Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Software Testing*, 23, 2013.

[149] Xiaochen Li, He Jiang, Dong Liu, Zhilei Ren, and Ge Li. Unsupervised deep bug report summarization. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, page 144–155, New York, NY, USA, 2018. Association for Computing Machinery.

[150] Xiaochen Li, He Jiang, Dong Liu, Zhilei Ren, and Ge Li. Unsupervised deep bug report summarization. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, page 144–155, New York, NY, USA, 2018. Association for Computing Machinery.

[151] Yuanchun Li, Baoxiong Jia, Yao Guo, and Xiangqun Chen. Mining user reviews for mobile app comparisons. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(3):75:1–75:15, September 2017.

[152] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.

[153] Zengyang Li, Sicheng Wang, Wenshuo Wang, Peng Liang, Ran Mo, and Bing Li. Understanding bugs in multi-language deep learning frameworks, 2023.

[154] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now? an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, page 25–33, New York, NY, USA, 2006. Association for Computing Machinery.

[155] Bo Lili, Zhu Xuanrui, Sun Xiaobing, Ni Zhen, and Li Bin. Are similar bugs fixed with similar change operations? an empirical study. *Chinese Journal of Electronics*, 30:55–63, 01 2021.

[156] S. L. Lim, P. J. Bentley, N. Kanakam, F. Ishikawa, and S. Honiden. Investigating country differences in mobile app user behavior and challenges for software engineering. *IEEE Transactions on Software Engineering*, 41(1):40–64, 2015.

[157] Vitor Lima, Jacson Barbosa, and Ricardo Marcacini. Learning risk factors from app reviews: A large language model approach for risk matrix construction, 07 2023.

[158] Charles X. Ling, Jin Huang, and Harry Zhang. Auc: A better measure than accuracy in comparing learning algorithms. In Yang Xiang and Brahim Chaib-draa, editors, *Advances in Artificial Intelligence*, pages 329–341, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[159] Charles X. Ling, Jin Huang, and Harry Zhang. Auc: A better measure than accuracy in comparing learning algorithms. In *Canadian Conference on AI*, 2003.

[160] Huaxiao Liu, Yihui Wang, Yuzhou Liu, and Shanquan Gao. Supporting features updating of apps by analyzing similar products in app stores. *Information Sciences*, 580:129–151, 2021.

[161] Jiakun Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. An exploratory study on the introduction and removal of different types of technical debt in deep learning frameworks. *Empirical Softw. Engg.*, 26(2), mar 2021.

[162] Jinping Liu, Yuming Zhou, Yibiao Yang, Hongmin Lu, and Baowen Xu. Code churn: A neglected metric in effort-aware just-in-time defect prediction. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 11–19, 2017.

[163] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, Z. Tian, Y. Huang, J. Hu, and Q. Wang. Testing the limits: Unusual text inputs generation for mobile app crash detection with large language model. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 1685–1696, Los Alamitos, CA, USA, apr 2024. IEEE Computer Society.

[164] Yasitha Liyanage, Mengfan Yao, Christopher Yong, Daphney-Stavroula Zois, and Charalampos Chelmis. What matters the most? optimal quick classification of urban issue reports by importance. In *2018 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 106–110, 2018.

[165] Guoming Long and Tao Chen. On reporting performance and accuracy bugs for deep learning frameworks: An exploratory study from github. EASE '22, page 90–99, New York, NY, USA, 2022. Association for Computing Machinery.

[166] Miodrag Lovric, editor. *International Encyclopedia of Statistical Science*. Springer, 2011.

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

[168] 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.

[169] Mengmeng Lu and Peng Liang. Automatic classification of non-functional requirements from augmented app user reviews. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, EASE'17, page 344–353, 2017.

[170] S. Ma, S. Wang, D. Lo, R. H. Deng, and C. Sun. Active semi-supervised approach for checking app behavior against its description. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 2, pages 179–184, 2015.

[171] Walid Maalej and Hadeer Nabil. Bug report, feature request, or simply praise? on automatically classifying app reviews. In *2015 IEEE 23rd International Requirements Engineering Conference (RE)*, pages 116–125, 2015.

[172] Ruchika Malhotra and Madhukar Cherukuri. Software defect categorization based on maintenance effort and change impact using multinomial naïve bayes algorithm. In *2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, pages 1068–1073, 2020.

[173] Yichuan Man, Cuiyun Gao, Michael R. Lyu, and Jiuchun Jiang. Experience report: Understanding cross-platform app issues from user reviews. In *Proceedings of the 27th IEEE International Symposium on Software Reliability Engineering*, ISSRE'16, pages 138–149, 2016.

[174] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[175] Poschenrieder Martin. 77% will not download a retail app rated lower than 3 stars. `https://blog.testmunk.com/77-will-not-download-a-retail-app-rated-lower-than-3-stars/`. (Last accessed: July 2017).

[176] Mary McHugh. Interrater reliability: The kappa statistic. *Biochemia medica : časopis Hrvatskoga društva medicinskih biokemičara / HDMB*, 22:276–82, 10 2012.

[177] Mary McHugh. Interrater reliability: The kappa statistic. *Biochemia medica : časopis Hrvatskoga društva medicinskih biokemičara / HDMB*, 22:276–82, 10 2012.

[178] Stuart McIlroy, Nasir Ali, Hammad Khalid, and Ahmed E. Hassan. Analyzing and automatically labelling the types of user issues that are raised in mobile app reviews. *Empirical Software Engineering*, 21(3):1067–1106, 2016.

[179] Shane Mcintosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Softw. Engg.*, 21(5):2146–2189, oct 2016.

[180] Shane Mcintosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Softw. Engg.*, 21(5):2146–2189, oct 2016.

[181] Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. Augmented language models: a survey, 2023.

[182] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *Proceedings of the 1st International Conference on Learning Representations*, ICLR'13, pages 1–12, 2013.

[183] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, page 3111–3119, Red Hook, NY, USA, 2013. Curran Associates Inc.

[184] David Mimno, Hanna M. Wallach, Edmund Talley, Miriam Leenders, and Andrew McCallum. Optimizing semantic coherence in topic models. In *Proceedings of the*

Conference on Empirical Methods in Natural Language Processing, EMNLP '11, page 262–272, USA, 2011. Association for Computational Linguistics.

[185] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large language models: A survey. *arXiv preprint arXiv:2402.06196*, 2024.

[186] Ayse Tosun Misirli, Emad Shihab, and Yasukata Kamei. Studying high impact fix-inducing changes. *Empirical Software Engineering*, 21(2):605–641, 2016. An erratum to this article can be found at http://dx.doi.org/10.1007/s10664-016-9455-3 .

[187] Ayse Tosun Misirli, Emad Shihab, and Yasukata Kamei. Studying high impact fix-inducing changes. *Empirical Software Engineering*, 21(2):605–641, 2016. An erratum to this article can be found at http://dx.doi.org/10.1007/s10664-016-9455-3 .

[188] Ran Mo, Yao Zhang, Yushuo Wang, Siyuan Zhang, Pu Xiong, Zengyang Li, and Yang Zhao. Exploring the impact of code clones on deep learning software. *ACM Trans. Softw. Eng. Methodol.*, 32(6), sep 2023.

[189] Mockus and Votta. Identifying reasons for software changes using historic databases. In *Proceedings 2000 International Conference on Software Maintenance*, pages 120–130, 2000.

[190] Manishankar Mondal, Md Saidur Rahman, Chanchal K. Roy, and Kevin A. Schneider. Is cloned code really stable? *Empirical Softw. Engg.*, 23(2):693–770, apr 2018.

[191] 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, 2019.

[192] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. Does cloned code increase maintenance effort? In *2017 IEEE 11th International Workshop on Software Clones (IWSC)*, pages 1–7, 2017.

[193] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. A fine-grained analysis on the inconsistent changes in code clones. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 220–231. IEEE, 2020.

[194] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. *A Summary on the Stability of Code Clones and Current Research Trends*, pages 169–180. Springer Singapore, Singapore, 2021.

[195] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proceedings Eighth IEEE Symposium on Software Metrics*, pages 87–94, 2002.

[196] Shuji Morisaki, Akito Monden, Tomoko Matsumura, Haruaki Tamada, and Kenichi Matsumoto. Defect data analysis based on extended association rule mining. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, pages 3–3, 2007.

[197] Mohammad Mehdi Morovati, Florian Tambon, Mina Taraghi, Amin Nikanjam, and Foutse Khomh. Common challenges of deep reinforcement learning applications development: An empirical study, 2023.

[198] M. Monzur Morshed, Md Rahman, and Salah Ahmed. A literature review of code clone analysis to improve software maintenance process. 05 2012.

[199] Md. Jubair Ibna Mostafa. An empirical study on clone evolution by analyzing clone lifetime. In *2019 IEEE 13th International Workshop on Software Clones (IWSC)*, pages 20–26, 2019.

[200] Quim Motger, Xavier Franch, Vincenzo Gervasi, and Jordi Marco. Unveiling competition dynamics in mobile app markets through user reviews. In Daniel Mendez and

Ana Moreira, editors, *Requirements Engineering: Foundation for Software Quality*, pages 251–266, Cham, 2024. Springer Nature Switzerland.

[201] Arjun Mukherjee and Bing Liu. Aspect extraction through semi-supervised modeling. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 339–348, 2012.

[202] V. Murali, Lee Gross, R. Qian, and S. Chandra. Industry-scale ir-based bug localization: A perspective from facebook. *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 188–197, 2021.

[203] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 284–292, 2005.

[204] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality: An empirical case study. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 521–530, New York, NY, USA, 2008. Association for Computing Machinery.

[205] Hoda Naguib, Nitesh Narayan, Bernd Brügge, and Dina Helal. Bug report assignee recommendation using activity profiles. MSR '13, page 22–30. IEEE Press, 2013.

[206] Jaechang Nam and Sunghun Kim. Clami: Defect prediction on unlabeled datasets (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 452–463, 2015.

[207] M. Nayebi, B. Adams, and G. Ruhe. Release practices for mobile apps – what do users and developers think? In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 552–562, 2016.

[208] M. Nayebi, H. Farahi, and G. Ruhe. Which version should be released to app store? In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 324–333, 2017.

[209] Zhen Ni, Bin Li, Xiaobing Sun, Tianhao Chen, Ben Tang, and Xinchen Shi. Analyzing bug fix for automatic bug cause classification. *Journal of Systems and Software*, 163:110538, 2020.

[210] Ehsan Noei, Daniel Alencar da Costa, and Ying Zou. Winning the app production rally. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '18, pages 283–294, 2018.

[211] Shayan Noei, Heng Li, Stefanos Georgiou, and Ying Zou. An empirical study of refactoring rhythms and tactics in the software development process. *IEEE Transactions on Software Engineering*, 49(12):5103–5119, 2023.

[212] Kate O'Flaherty. Zoom beats microsoft teams, google meet with game-changing new features, Oct 2020.

[213] D. Pagano and W. Maalej. User feedback in the appstore: An empirical study. In *2013 21st IEEE International Requirements Engineering Conference (RE)*, pages 125–134, 2013.

[214] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado Aaron Visaggio, Gerardo Canfora, and Harald C. Gall. How can i improve my app? classifying user reviews for software maintenance and evolution. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, ICSME '15, pages 281–290, 2015.

[215] Lucas D. Panjer. Predicting eclipse bug lifetimes. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, pages 29–29, 2007.

[216] Luca Pascarella, Fabio Palomba, Massimiliano Di Penta, and Alberto Bacchelli. How is video game development different from software development in open source?

In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 392–402, 2018.

[217] P. Pinheiro. Linear and nonlinear mixed effects models: Theory and applications, 2010.

[218] A. Rahman and Effat Farhana. An exploratory characterization of bugs in covid-19 software projects. *ArXiv*, abs/2006.00586, 2020.

[219] Akond Rahman, Effat Farhana, Chris Parnin, and Laurie Williams. Gang of eight: A defect taxonomy for infrastructure as code scripts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 752–764, New York, NY, USA, 2020. Association for Computing Machinery.

[220] 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, 2017.

[221] W. Rahman, Y. Xu, F. Pu, J. Xuan, X. Jia, M. Basios, L. Kanthan, L. Li, F. Wu, and B. Xu. Clone detection on large scala codebases. In *2020 IEEE 14th International Workshop on Software Clones (IWSC)*, pages 38–44, Los Alamitos, CA, USA, feb 2020. IEEE Computer Society.

[222] Gopi Krishnan Rajbahadur, Shaowei Wang, Gustavo Ansaldi, Yasutaka Kamei, and Ahmed E. Hassan. The impact of feature importance methods on the interpretation of defect classifiers. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.

[223] Juan Ramos. Using TF-IDF to determine word relevance in document queries. In *Proceedings of the 1st instructional Conference on Machine Learning*, iCML '03, pages 1–4, 2003.

[224] Christian Robottom Reis, Renata Pontin de Mattos Fortes, Renata Pontin, and Mattos Fortes. An overview of the software engineering process and tools in the mozilla project, 2002.

[225] Christian Rigg and Nikshep Myle. Zoom video conferencing service review, Feb 2021.

[226] Stephen Romansky, Cheng Chen, Baljeet Malhotra, and Abram Hindle. Sourcerer's apprentice and the study of code snippet migration. *CoRR*, abs/1808.00106, 2018.

[227] Konstantinos I. Roumeliotis, Nikolaos D. Tselikas, and Dimitrios K. Nasiopoulos. Llms in e-commerce: A comparative analysis of gpt and llama models in product review evaluation. *Natural Language Processing Journal*, 6:100056, 2024.

[228] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

[229] Chanchal Roy and James Cordy. A survey on software clone detection research. *School of Computing TR 2007-541*, 01 2007.

[230] 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, 2008.

[231] 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, 2009.

[232] Ripon K. Saha, Sarfraz Khurshid, and Dewayne E. Perry. Understanding the triaging and fixing processes of long lived bugs. *Information and Software Technology*, 65:114–128, 2015.

[233] Munish Saini and Kuljit Kaur Chahal. Change profile analysis of open-source software systems to understand their evolutionary behavior. *Front. Comput. Sci.*, 12(6):1105–1124, dec 2018.

[234] 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*, ICSE '16, page 1157–1168, New York, NY, USA, 2016. Association for Computing Machinery.

[235] Johnny Saldaña. *The coding manual for qualitative researchers*. Sage, 2015.

[236] Simone Scalabrino, Gabriele Bavota, Barbara Russo, Massimiliano Di Penta, and Rocco Oliveto. Listening to the crowd for the release planning of mobile apps. *IEEE Transactions on Software Engineering*, 45(1):68–86, 2019.

[237] Robin M Schmidt. Recurrent neural networks (rnns): A gentle introduction and overview. *arXiv preprint arXiv:1912.05911*, 2019.

[238] Lajos Schrettner, Judit Jász, Tamás Gergely, Árpád Beszédes, and Tibor Gyimóthy. Impact analysis in the presence of dependence clusters using static execute after in webkit. *Journal of Software: Evolution and Process*, 26(6):569–588, 2014.

[239] C. B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, 1999.

[240] Carolyn Seaman, Forrest Shull, Myrna Regardie, Denis Elbert, Raimund Feldmann, Yuepu Guo, and Sally Godfrey. Defect categorization: Making use of a decade of widely varying historical data. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 149–157, 01 2008.

[241] C.B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, 1999.

[242] Reza Sepahvand, Reza Akbari, and Sattar Hashemi. Predicting the bug fixing time using word embedding and deep long short term memories. *IET Software*, 14(3):203–212, 2020.

[243] Faiz Ali Shah, Yevhenii Sabanin, and Dietmar Pfahl. Feature-based evaluation of competing apps. In *Proceedings of the ACM International Workshop on App Market Analytics*, WAMA '16, pages 15–21, 2016.

[244] Faiz Ali Shah, Kairit Sirts, and Dietmar Pfahl. The impact of annotation guidelines and annotated data on extracting app features from app reviews. *CoRR*, abs/1810.05187, 2018.

[245] Faiz Ali Shah, Kairit Sirts, and Dietmar Pfahl. Is the SAFE approach too simple for app feature extraction? A replication study. In *Proceedings of the 25th International Working Conference on Requirements Engineering: Foundation for Software Quality*, REFSQ 19, pages 21–36, 2019.

[246] Faiz Ali Shah, Kairit Sirts, and Dietmar Pfahl. Using app reviews for competitive analysis: tool support. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics*, WAMA '19, pages 40–46, 2019.

[247] Faiz Ali Shah, Kairit Sirts, and Dietmar Pfahl. Using app reviews for competitive analysis: tool support. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics*, WAMA '19, pages 40–46, 2019.

[248] Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, Indira Vats, Hadi Moazen, and Federica Sarro. A survey on machine learning techniques applied to source code. *Journal of Systems and Software*, 209:111934, 2024.

[249] Ali Shatnawi, Ghadeer Al-Bdour, Raffi Al-Qurran, and Mahmoud Al-Ayyoub. A comparative study of open source deep learning frameworks. In *2018 9th International Conference on Information and Communication Systems (ICICS)*, pages 72–77, 2018.

[250] Leif Singer and Kurt Schneider. It was a bit of a race: Gamification of version control. In *2012 Second International Workshop on Games and Software Engineering: Realizing User Engagement with Game Engineering Techniques (GAS)*, pages 5–8, 2012.

[251] Kalyanasundaram Somasundaram and Gail C. Murphy. Automatic categorization of bug reports using latent dirichlet allocation. In *Proceedings of the 5th India Software Engineering Conference*, ISEC '12, page 125–130, New York, NY, USA, 2012. Association for Computing Machinery.

[252] Srdjan Stevanetic and Uwe Zdun. Supporting the analyzability of architectural component models - empirical findings and tool support. *Empirical Softw. Engg.*, 23(6):3578–3625, dec 2018.

[253] Chris Stokel-Walker. How skype lost its crown to zoom, May 2020.

[254] H. Strobelt, A. Webson, V. Sanh, B. Hoover, J. Beyer, H. Pfister, and A. M. Rush. Interactive and visual prompt engineering for ad-hoc task adaptation with large language models. *IEEE Transactions on Visualization and Computer Graphics*, 29(01):1146–1156, jan 2023.

[255] Yanqi Su, Yongchao Wang, and Wenhua Yang. Mining and comparing user reviews across similar mobile apps. In *2019 15th International Conference on Mobile Ad-Hoc and Sensor Networks (MSN)*, pages 338–342, 2019.

[256] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 45–54, 2010.

[257] Xiaobing Sun, Bixin Li, Chuanqi Tao, Wanzhi Wen, and Sai Zhang. Change impact analysis based on a taxonomy of change types. In *2010 IEEE 34th Annual Computer Software and Applications Conference*, pages 373–382, 2010.

[258] Shan Suthaharan and Shan Suthaharan. Support vector machine. *Machine learning models and algorithms for big data classification: thinking with examples for effective learning*, pages 207–235, 2016.

[259] Jeffrey Svajlenko and Chanchal K. Roy. Evaluating modern clone detection tools. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 321–330, 2014.

[260] Shaheen Syed and Marco Spruit. Full-text or abstract? examining topic coherence scores using latent dirichlet allocation. In *2017 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 165–174, 2017.

[261] Ana B. Sánchez, Pedro Delgado-Pérez, Inmaculada Medina-Bulo, and Sergio Segura. Tandem: A taxonomy and a dataset of real-world performance bugs. *IEEE Access*, 8:107214–107228, 2020.

[262] Florian Tambon, Amin Nikanjam, Le An, Foutse Khomh, and Giuliano Antoniol. Silent bugs in deep learning frameworks: An empirical study of keras and tensorflow. *ArXiv*, abs/2112.13314, 2021.

[263] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. An empirical comparison of model validation techniques for defect prediction models. (1), 2017.

[264] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. The impact of automated parameter optimization for defect prediction models. 2018.

[265] Patanamon Thongtanunam, Weiyi Shang, and Ahmed E. Hassan. Will this clone be short-lived? towards a better understanding of the characteristics of short-lived clones. *Empirical Softw. Engg.*, 24(2):937–972, apr 2019.

[266] Ferdian Thung. Automatic prediction of bug fixing effort measured by code churn size. In *Proceedings of the 5th International Workshop on Software Mining*, SoftwareMining 2016, page 18–23, New York, NY, USA, 2016. Association for Computing Machinery.

[267] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. An empirical study of bugs in machine learning systems. *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, 11 2012.

[268] K. Tsipenyuk, B. Chess, and G. McGraw. Seven pernicious kingdoms: a taxonomy of software security errors. *IEEE Security Privacy*, 3(6):81–84, 2005.

[269] M. Tushev, F. Ebrahimi, and A. Mahmoud. Domain-specific analysis of mobile app reviews using keyword-assisted topic models. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 762–773, Los Alamitos, CA, USA, may 2022. IEEE Computer Society.

[270] Harold Valdivia-Garcia, Emad Shihab, and Meiyappan Nagappan. Characterizing and predicting blocking bugs in open source projects. *Journal of Systems and Software*, 143:44–58, 2018.

[271] Brent van Bladel and Serge Demeyer. A comparative study of code clone genealogies in test code and production code. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 913–920, 2023.

[272] Rajesh Vasa, Leonard Hoon, Kon Mouzakis, and Akihiro Noguchi. A preliminary analysis of mobile app user reviews. In *Proceedings of the 24th Australian Computer-Human Interaction Conference*, OzCHI '12, pages 241–244, 2012.

[273] Renan Vieira, Antônio da Silva, Lincoln Rocha, and João Paulo Gomes. From reports to bug-fix commits: A 10 years dataset of bug-fixing activity from 55 apache's open source projects. PROMISE'19, page 80–89, New York, NY, USA, 2019. Association for Computing Machinery.

[274] Lorenzo Villarroel, Gabriele Bavota, Barbara Russo, Rocco Oliveto, and Massimiliano Di Penta. Release planning of mobile apps based on user reviews. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 14–24, 2016.

[275] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103, 2008.

[276] Phong Minh Vu, Tam The Nguyen, Hung Viet Pham, and Tung Thanh Nguyen. Mining user opinions in mobile app reviews: A keyword-based approach (T). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, pages 749–759, 2015.

[277] Stefan Wagner, Asim Abdulkhaleq, Kamer Kaya, and Alexander Paar. On the relationship of inconsistent software clones and faults: An empirical study. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 79–89, 2016.

[278] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. Bug characteristics in blockchain systems: A large-scale empirical study. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 413–424, 2017.

[279] Zhiyuan Wan, Xin Xia, David Lo, and Gail C. Murphy. How does machine learning change software development practices? *IEEE Transactions on Software Engineering*, 47(9):1857–1871, 2021.

[280] Haoye Wang, Xin Xia, David Lo, John Grundy, and Xinyu Wang. Automatic solution summarization for crash bugs. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1286–1297, 2021.

[281] Jing Wang, Patrick Shih, Yu Wu, and John Carroll. Comparative case studies of open source software peer review practices. *Information and Software Technology*, 67:1–12, 11 2015.

[282] Peipei Wang, Chris Brown, Jamie A. Jennings, and Kathryn T. Stolee. An empirical study on regular expression bugs. In *Proceedings of the 17th International*

Conference on Mining Software Repositories, MSR '20, page 103–113, New York, NY, USA, 2020. Association for Computing Machinery.

[283] Yafang Wang and Dongsheng Liu. Image-based clone code detection and visualization. In *2019 International Conference on Artificial Intelligence and Advanced Manufacturing (AIAM)*, pages 168–175, 2019.

[284] Yihui Wang, Shanquan Gao, Yan Zhang, Huaxiao Liu, and Yiran Cao. Uisminer: Mining ui suggestions from user reviews. *Expert Systems with Applications*, 208:118095, 2022.

[285] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 788–799, New York, NY, USA, 2020. Association for Computing Machinery.

[286] Zhaobin Wang, Ke Liu, Jian Li, Ying Zhu, and Yaonan Zhang. Various frameworks and libraries of machine learning and deep learning: A survey. *Archives of Coputational Methods in Engineering*, 02 2019.

[287] T. Warren Liao. Clustering of time series data—a survey. *Pattern Recognition*, 38(11):1857–1874, 2005.

[288] Jialiang Wei. Enhancing requirements elicitation through app stores mining: Health monitoring app case study. In *2023 IEEE 31st International Requirements Engineering Conference (RE)*, pages 396–400. IEEE, 2023.

[289] Jialiang Wei, Anne-Lise Courbis, Thomas Lambolais, Binbin Xu, Pierre Louis Bernard, and Gérard Dray. Zero-shot bilingual app reviews mining with large language models. In *2023 IEEE 35th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 898–904, 2023.

[290] S. Weisberg. *Applied Linear Regression*, volume 528. John Wiley & Sons, 2005.

[291] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, pages 1–1, 2007.

[292] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and S.C. Cheung. Historical spectrum based fault localization. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[293] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. Locus: Locating bugs from software changes. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 262–273, 2016.

[294] Edmund Wong. Improving software dependability through documentation analysis. 2019.

[295] Rongxin wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. Changelocator: locate crash-inducing changes based on crash reports. pages 536–536, 05 2018.

[296] Xin Xia, David Lo, Ying Ding, Jafar M. Al-Kofahi, Tien N. Nguyen, and Xinyu Wang. Improving automated bug triaging with specialized topic model. *IEEE Transactions on Software Engineering*, 43(3):272–297, 2017.

[297] Xihao Xie, Wen Zhang, Ye Yang, and Qing Wang. Dretom: Developer recommendation based on topic models for bug resolution. PROMISE '12, page 19–28, New York, NY, USA, 2012. Association for Computing Machinery.

[298] Tongtong Xu. Improving automated program repair with retrospective fault localization. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 159–161, 2019.

[299] Xiancai Xu, Jia-Dong Zhang, Rongchang Xiao, and Lei Xiong. The limits of chatgpt in extracting aspect-category-opinion-sentiment quadruples: A comparative analysis, 2023.

[300] Jifeng Xuan, He Jiang, Zhilei Ren, Jun Yan, and Zhongxuan Luo. Automatic bug triage using semi-supervised text classification. *CoRR*, abs/1704.04769, 2017.

[301] Pengyu Xue, Linhao Wu, Zhongxing Yu, Zhi Jin, Zhen Yang, Xinyi Li, Zhenyu Yang, and Yue Tan. Automated commit message generation with large language models: An empirical study and beyond. *arXiv preprint arXiv:2404.14824*, 2024.

[302] Sezin Yaman, Tanja Sauvola, Leah Riungu-Kalliosaari, Laura Hokkanen, Pasi Kuvaja, Markku Oivo, and Tomi Männistö. Customer involvement in continuous deployment: A systematic literature review. volume 9619, 03 2016.

[303] Meng Yan, Xin Xia, D. Lo, A. Hassan, and Shanping Li. Characterizing and identifying reverted commits. *Empirical Software Engineering*, pages 1–38, 2019.

[304] Aidan Z. H. Yang, Safwat Hassan, Ying Zou, and A. Hassan. An empirical study on release notes patterns of popular apps in the google play store. *Empirical Software Engineering*, 27:1–38, 2022.

[305] Geunseok Yang, Tao Zhang, and Byungjeong Lee. Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 97–106, 2014.

[306] Xin Ye, Razvan Bunescu, and Chang Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 689–699, New York, NY, USA, 2014. Association for Computing Machinery.

[307] Soner Yigit and Mehmet Mendes. Which effect size measure is appropriate for one-way and two-way anova models? : A monte carlo simulation study. *REVSTAT-Statistical Journal*, 16(3):295–313, Jul. 2018.

[308] Wei Yuan, Yuan Xiong, Hailong Sun, and Xudong Liu. Incorporating multiple features to predict bug fixing time with neural networks. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 93–103, 2021.

[309] Yuan Yuan and Wolfgang Banzhaf. ARJA: automated repair of java programs via multi-objective genetic programming. *CoRR*, abs/1712.07804, 2017.

[310] R. Yue, Z. Gao, N. Meng, Y. Xiong, X. Wang, and J. Morgenthaler. Automatic clone recommendation for refactoring based on the present and the past. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 115–126, Los Alamitos, CA, USA, sep 2018. IEEE Computer Society.

[311] Morteza Zakeri-Nasrabadi, Saeed Parsa, Mohammad Ramezani, Chanchal Roy, and Masoud Ekhtiarzadeh. A systematic literature review on source code similarity measurement and clone detection: techniques, applications, and challenges, 2023.

[312] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. A qualitative study on performance bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, page 199–208. IEEE Press, 2012.

[313] Motahareh Bahrami Zanjani, George Swartzendruber, and Huzefa Kagdi. Impact analysis of change requests on source code based on interaction and commit histories. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 162–171, New York, NY, USA, 2014. Association for Computing Machinery.

[314] Feng Zhang, Foutse Khomh, Ying Zou, and Ahmed E. Hassan. An empirical study on factors impacting bug fixing time. In *2012 19th Working Conference on Reverse Engineering*, pages 225–234, 2012.

[315] Hongyu Zhang, Liang Gong, and Steve Versteeg. Predicting bug-fixing time: An empirical study of commercial software projects. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1042–1051, 2013.

[316] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794, 2019.

[317] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. Boosting spectrum-based fault localization using pagerank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 261–272, New York, NY, USA, 2017. Association for Computing Machinery.

[318] Pengcheng Zhang, Feng Xiao, and Xiapu Luo. A framework and dataset for bugs in ethereum smart contracts. pages 139–150, 09 2020.

[319] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. An empirical study of common challenges in developing deep learning applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 104–115, 2019.

[320] Wenxuan Zhang, Yue Deng, Bing Liu, Sinno Jialin Pan, and Lidong Bing. Sentiment analysis in the era of large language models: A reality check, 2023.

[321] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 129–140, New York, NY, USA, 2018. Association for Computing Machinery.

[322] Guoliang Zhao, Safwat Hassan, Ying Zou, Derek Truong, and Toby Corbin. Predicting performance anomalies in software systems at run-time. *ACM Trans. Softw. Eng. Methodol.*, 30(3), apr 2021.

[323] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models, 2023.

[324] Xin Zhao, Jing Jiang, Hongfei Yan, and Xiaoming Li. Jointly modeling aspects and opinions with a MaxEnt-LDA hybrid. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*. ACL, 2010.

[325] Yan Zhong, Xunhui Zhang, Wang Tao, and Yanzhi Zhang. A systematic literature review of clone evolution. In *Proceedings of the 5th International Conference on Computer Science and Software Engineering*, CSSE '22, page 461–473, New York, NY, USA, 2022. Association for Computing Machinery.

[326] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations*, 2023.

[327] Minhaz F. Zibran. On the effectiveness of labeled latent dirichlet allocation in automatic bug-report categorization. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, page 713–715, New York, NY, USA, 2016. Association for Computing Machinery.

[328] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643, 2010.

[329] Weiqin Zou, David Lo, Zhenyu Chen, Xin Xia, Yang Feng, and Baowen Xu. How practitioners perceive automated bug report management techniques. *IEEE Transactions on Software Engineering*, 46(8):836–862, 2020.