2020-11-09

Analysis of compatibility in open-source Android mobile applications

Mukherjee, Debjyoti

UNIVERSITY OF CALGARY

Analysis of compatibility in open-source Android mobile applications

by

Debjyoti Mukherjee

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

NOVEMBER, 2020

Abstract

Non-functional requirements form an intrinsic part of any software system. Compatibility between versions or different platforms of a software product is a form of NFRs. In this thesis, we have studied compatibility in Android mobile applications. We are interested in understanding the different aspects of mobile application incompatibility, their frequency of occurrence, how much effort developers have spent on it, and whether the effort is commensurate with the needs of the users.

In this thesis, an analytical compatibility evaluation approach called ACOCUR is proposed. The main characteristics of ACOCUR are: (i) compatibility requirements are automatically identified from user reviews and their types are also determined, (ii) compatibility fixes done by developers are systematically analyzed, and (iii) the requirements from users are linked to the fixes to identify the responsiveness of developers to compatibility requirements.

We have evaluated open-source mobile applications and have analyzed their commits and reviews to identify the compatibility fixes and requirements respectively. Both the commit messages and reviews have been processed by a pipeline of Natural Language Processing steps. App developers have also been surveyed and their responses have been analyzed to establish the state-of-the-practice and the problems currently faced by developers in this respect. Finally, an automated tool has been developed that implements the ACOCUR methodology to support app developers to identify and analyze compatibility requirements.

Publications

Some of the ideas, materials, tables, figures used in this thesis have appeared previously in the following publication:

Referred and published papers:

1. Debjyoti Mukherjee, and Guenther Ruhe. "Analysis of Compatibility in Open Source Android Mobile Apps." In 2020 IEEE Seventh International Workshop on Artificial Intelligence for Requirements Engineering (AIRE), pp. 70-78. IEEE, 2020.

2. Alireza Ahmadi, Debjyoti Mukherjee, and Guenther Ruhe. "A recommendation system for emergency mobile applications using context attributes: REMAC." In Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics, pp. 1-7. 2019.

Acknowledgements

This thesis would not have taken place without the continuous guidance and the help of several individuals who in one way or the other have contributed and extended their valuable assistance in the preparation and completion of this thesis. I am grateful to my supervisor Dr. Guenther Ruhe for his valuable guidance, continuous support, and supervision. Without his inspiring enthusiasm and encouragement, this thesis would not have been completed.

My heartfelt thanks and appreciation go to my colleagues in the Software Decision Support Laboratory for their benevolent effort to put valuable comments on this thesis. Discussing with them, from time to time, has sharpened my thoughts for the research.

Finally, I am indebted to the endless support of my wife and other family members. Without their unconditional encouragement and support, this endeavor would not have been possible.

Table of Contents

List of Figures

List of Tables

List of Symbols, Abbreviations, and Nomenclature

Symbol	Definition
NFR	Non-functional requirement
NLP	Natural Language Processing
App	Application
ML	Machine Learning
NB	Naive Bayes
SVM	Support Vector Machine
RF	Random Forest
LR	Logistic Regression
RE	Requirement Engineering
API	Application Programming Interface
AI	Artificial Intelligence
κ	Kappa coefficient
σ	Standard deviation

Chapter 1

Introduction

1.1 Introduction

In the past few years, we have observed an explosion in the popularity of mobile devices and smartphones. With these, the usage and development of mobile applications have proliferated. These mobile applications are commonly referred to as *mobile apps* or simply, *apps*. To bolster this effort further, there has been an increasing amount of software engineering research dedicated to mobile apps. Nagappan et al. [51] discussed the current and future research trends within the framework of the various stages in the software development life-cycle for mobile applications. Aspects related to functional requirements, non-functional requirements, development, testing, maintenance, and monetization have been discussed in this paper. For our study, we are focused on *non-functional requirements* (NFRs) as this is the core of our research.

Non-functional requirements (NFRs) are an intrinsic part of any software system, irrespective of whether it is a mobile or a non-mobile application. These are often referred to as the quality requirements for the software system. NFRs vary from functional requirements in that while the former is attributed to building the *system right*, the latter is vital for getting the *right system*. However, there has been a lot of debate within the research community as to what all should be considered as NFRs and how should we tackle them.

This is apparent when Martin Glinz [25] stated – *"Although the term 'non-functional requirement' has been in use for more than 20 years, there is still no consensus in the requirements engineering community what non-functional requirements are and how we should elicit, document, and validate them. On the other hand, there is a unanimous consensus that nonfunctional requirements are important and can be critical for the success of a project"*. He further added – *"If you want to trigger a hot debate among a group of requirements engineering people, just let them talk about non-functional requirements"* [25]. The extent of

1

this issue can be gauged in [54] where Niu et al. have referred to non-functional requirements as *"The Pain Point"*. Depending on the nature of the software system, the non-functional requirements that are the most important will vary. For example, the nature and type of quality requirements for a financial web application will be different from that of a desktop ERP application. As such it is important to understand the software system before deciding upon its non-functional requirements. In this research, we have concentrated on non-functional requirements related to mobile applications.

In the case of mobile applications, some of the typical non-functional requirements that are primarily discussed are usability, reliability, security, performance, availability, scalability, and maintainability. Past researches have found that *usability* and *reliability* are the two most sought-after NFRs for mobile app developers [86]. While these are important, we consider *compatibility* to be very pertinent in the recent scenario for mobile apps.

Mobile apps are offered on a diversity of platforms and varying versions of the system. The user often is confronted with reduced functionality or performance when changing between platforms and versions of an application. Software compatibility is a complex and pervasive topic. By software compatibility, we mean – *"the characteristic of the software components or systems which can operate satisfactorily together on the same device, or different devices linked by a network"* [3]. We found that, even though compatibility is critical, it is not well studied, not well defined nor well-documented [26]. Formally, it is classified as a non-functional requirement. However, not much is known about the degree of occurrence and the time spent on it. As such, we would like to investigate the compatibility aspect further in our research.

With the rapid growth in the diverse types of mobile devices and fast upgrading mobile operating systems, it is challenging for the mobile application developers to keep pace with this fast-changing environment. Some of the types of app incompatibilities that are commonly identified are:

- The app crashes in some particular mobile devices, while it works fine on others (device incompatibility).

- The app's behavior and UI changes and works incorrectly after a change of the operating system of the device (version incompatibility).

- The app's performance is affected by some device configurations (performance incompatibility).

According to recent statistics [7], the Android operating system enjoys a majority in the mobile operating systems' market share worldwide. The first version of Android was launched in 2008; by January 2012, it had already released nine different versions of the Android operating system. During that time, Android's market share was less than 25%. However, as of July 2020, Android's market share has increased to 74.6%. Figure 1.1 shows the gradual increase in the popularity of the Android operating system over the years. In
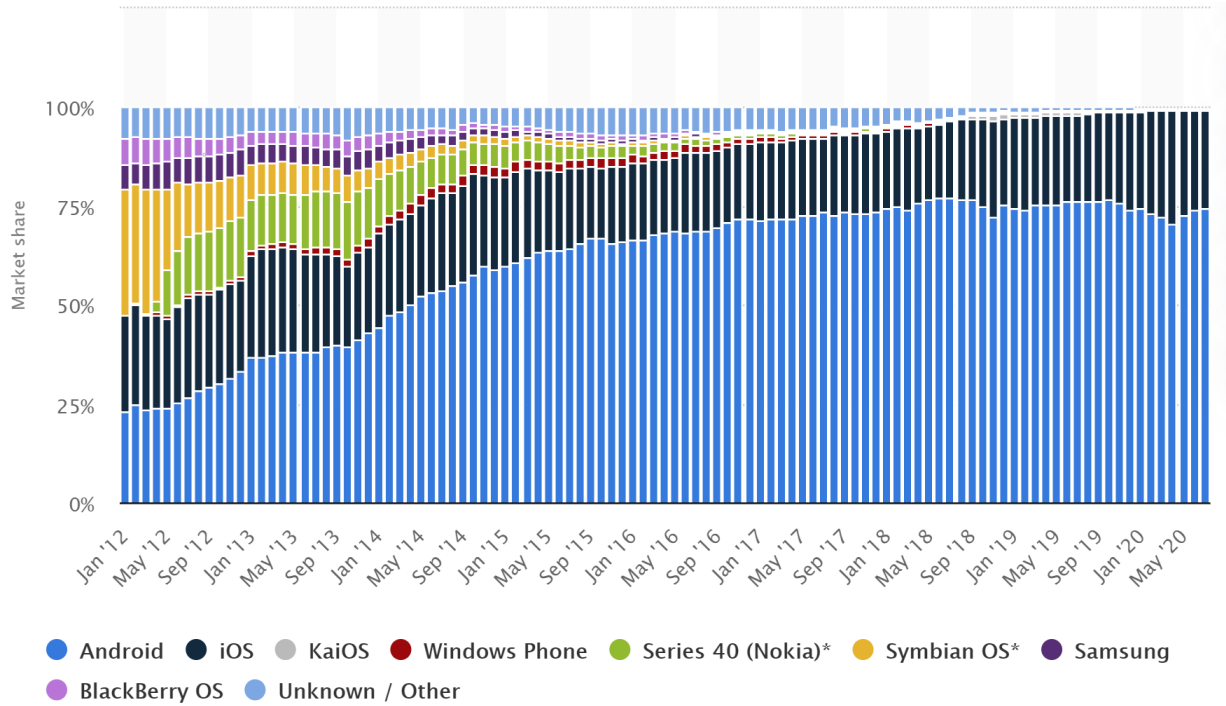
Figure 1.1: Mobile operating system market share (reference: www.statista.com [7])
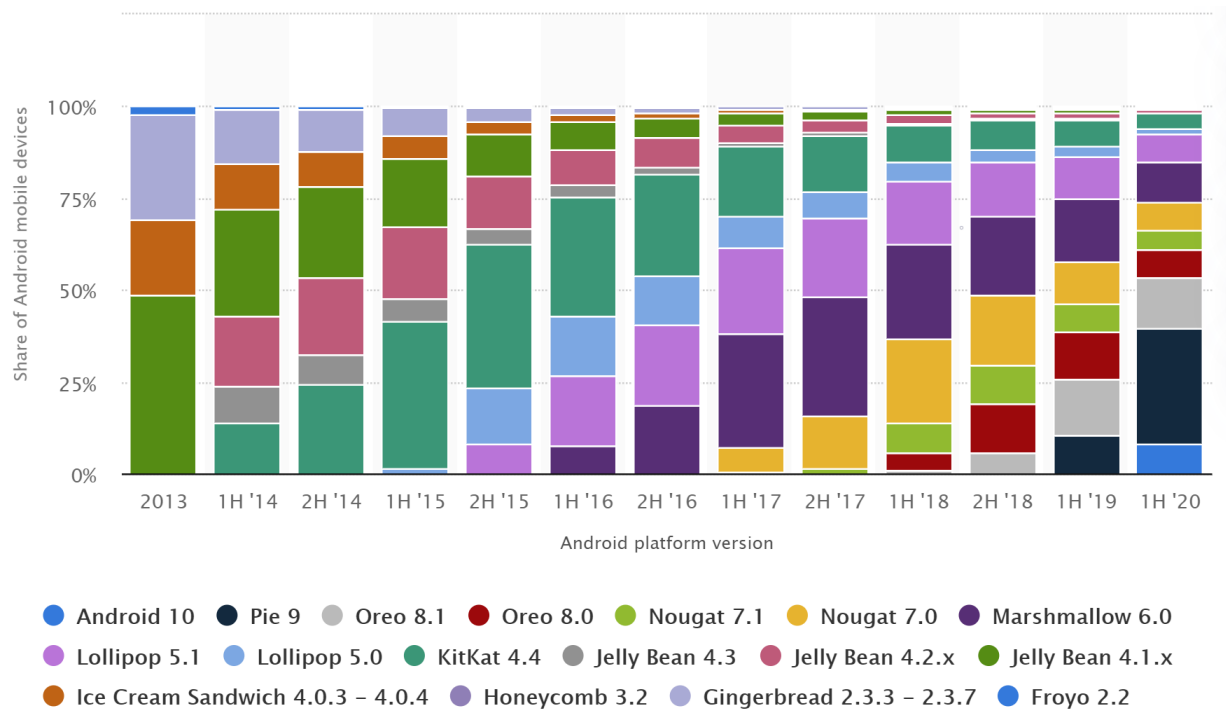
Figure 1.2: Share of different Android versions (reference: www.statista.com [7])

these eight years, Android had released eight major versions. With the frequent release of newer versions and a simultaneously larger number of smartphone users shifting to the Android operating system, the challenge

3

faced by Android app developers is constantly mounting. A large number of versions of the Android operating system simultaneously used by mobile users coupled with a large number of different types of mobile devices has led to a majority of the compatibility issues faced by app developers. Figure 1.2 shows the percentage share of the different Android platform versions that have been simultaneously supported over the years. As such, any Android application developed in such a scenario needs to ensure that the application is compliant with all these different combinations.

Fragmentation is considered as one of the typical causes of incompatibility for Android apps [73]. It resonates with the threat or concern that a proliferation of diverging variants of the Android platform will result in the inability of some devices to properly run apps written with the Android SDK. With a large number of custom versions of the Android platform emerging, the concern is that interoperability will be affected as a result of the potential for applications built specifically for one variant or device not being able to work with others. To cope with this challenge for maintaining compatibility in apps, researchers have formulated different approaches; while some have proposed ways for enhanced testing for identifying incompatibility in mobile apps like Huang [4] and Naith et al. [52], others have devised techniques to automatically detect compatibility issues in the apps (like Li et al. [42]). These are proactive measures that can be used by app developers to identify and resolve incompatibilities in mobile apps. While these techniques are undoubtedly important and useful, it is often very time consuming and difficult to identify and address all types of incompatibilities that mobile applications might encounter. As such, we would like to propose a reactive approach in which compatibility issues can be addressed as and when they are identified. To make this effective, we would like to take advantage of CrowdRE and take help from general app users for identifying these quality issues.

App stores and other social platforms provide a wealth of information derived from users that can be of enormous use for the app developers. Recent studies have shown how information can be retrieved from these sources and effectively used in the development process. These studies have showcased different techniques for extracting requirements from these social forums, and a majority of those are related to identifying requirements from user reviews. We would like to adopt a similar approach and identify non-functional requirements (specifically, compatibility requirements) from user reviews and empower app developers to meet those.

The previous studies that have dealt with this aspect of the non-functional requirement for mobile apps have often prescribed different ways to mitigate the risk arising from this challenge. But there is not enough evidence to judge the current state of the practice. To the best of our knowledge, there has not been enough investigation to understand how much of these approaches have been put into practice. We do not know for certain how much are the app developers concerned about these problems and how much

effort have they put to tackle the issues. In the course of this research, we have also surveyed mobile app developers and the results from the survey have further strengthened the need for an automated process to identify compatibility requirements from user feedback to empower app developers to tackle the mobile apps' compatibility challenge.

In this thesis, we have specifically focused on the compatibility aspect, but the approaches suggested in this study can be applied to other non-functional requirements as well. In the next section, we shall discuss the different research questions that we would like to answer as part of this thesis to get a better understanding of this subject and also address some of the current limitations and problems related to tackling mobile applications' incompatibility issues.

1.2   Research questions

In this thesis, we shall address the following six research questions:

RQ1: In the context of open-source mobile app development, what percentage of developer commits are related to compatibility?

Why and How? There is not sufficient evidence of how much app developers are concerned about compatibility. To get insight into app development, we have opted for open-source mobile apps so that we can access their code, versions, change history, and other relevant information.

Using all these available data and different ML techniques, we would identify the compatible related commits to estimate the percentage of commits that deal with these quality requirements.

RQ2: How much are mobile app users concerned about compatibility?

Why and How? One of the major problems in requirement engineering (RE), as identified by Fernández et at. [23], is insufficient involvement of customers. As such, it would be interesting to identify what percentage of user reviews complain about compatibility. Only then we can judge if the developer's efforts in this area are commensurate to the users' needs.

Different ML techniques can be used for identifying compatibility complaints from user reviews. We would use some of those to identify which all reviews contain compatibility requirements.

RQ3: What are the different types of compatibility fixes done by app developers? What are the types of compatibility issues raised by users in their reviews?

Why and How? We consider compatibility as one of the most important non-functional requirements

for mobile applications. Also, there can be different types of incompatibilities, and identifying the different types would help in resolving them and addressing the concerns appropriately. As such, we would like to identify the different types of compatibility issues raised in reviews and also those that have been addressed by developers. To identify the different types, we would perform exploratory research and analyze user reviews and commits; we intend to perform an card-sorting approach to identify the different types of incompatibilities.

RQ4: What is the degree of alignment between the users' requests and the developers' responsiveness concerning mobile app's compatibility requirements?

Why and How? We are interested in the relationship between demand (articulated in user reviews) and the actual proportion of commits devoted to compatibility. It is important to evaluate if the developer's effort towards tackling the app's incompatibilities are sufficient. There are different types of incompatibilities and each of them has varying importance and effect on the app; as such, the developers must take a systematic approach while addressing the app's incompatibilities. Since we refer to user reviews for identifying compatibility requirements, this comparative analysis between reviews and commits would help developers identify requirements that have higher priority. To evaluate the degree of alignment, we would pursue two approaches; establish a connection between individual user reviews and commits to answer if the issues have been addressed, and analyze the compatibility types to evaluate the alignment of the developers to the users' needs.

RQ5: With respect to the mobile apps' compatibility related requirements and fixes, what is the current state-of-the-practice (methods used for identifying the requirements, challenges faced in the process, importance on fixing these requirements, support from users etc.) followed by the app developers?

Why and How? It is essential to have a deep understanding of the current state of practice related to the compatibility aspect of mobile apps. Only when we can estimate how app developers perceive this non-functional requirement, the importance they associate to it, the approaches used to tackle this requirement, and the challenges faced in this process; then we can attempt to make the process better and propose effective ways to deal with it.

To answer this research question, we shall survey with mobile app developers and request their responses to different aspects related to mobile apps' compatibility. The survey will include questions on the importance of identifying and addressing compatibility issues, the processes followed, the challenges faced by the developers, and what type of support they expect from the researchers.

RQ6: How can we support app developers to automatically identify compatibility requirements and analyze their responsiveness to these requirements?

Why and How? Identifying compatibility requirements and effectively addressing them is key for the success of any application; in the case of mobile applications, compatibility requirements assume a greater priority. However, without an efficient and automated tool, this process can be very tedious, time-consuming, and often difficult to tackle effectively.

As such, as part of this thesis, we would like to propose a tool that can be effective in automatically extracting the compatibility requirements from user reviews; also the tool can help analyze the developer's fixes to identify which all requirements have already been addressed and which all needs further action. With the help of this tool, the developers can also identify other types of non-functional requirements and also check which all of these requirements have already been addressed. This tool would provide them a way to measure the amount of effort dedicated to tackling these different types of non-functional requirements.

1.3 Thesis Contributions

In this thesis, we have conducted the study in two phases; first, an exploratory search to identify how much the users are concerned about app incompatibilities and to what extent the app developers deal with the compatibility aspect, and then we propose an analytical compatibility evaluation approach called ACOCUR (**A**nalyze **Co**mpatibility requirements from **C**ommit messages and **U**ser **R**eviews). The key characteristics of ACOCUR are:

- Automatic mining of relevant data from various sources as necessary for evaluation

- Extraction of compatibility requirements from reviews

- Analysis of compatibility fixes done by developers

- Automatic classification of the requirements and fixes into different categories

- Use of pre-trained models that have been evaluated on a large set of Android mobile apps.

In the initial phase of the research, a large number of mobile apps have been analyzed. We have also created two taxonomies for compatibility types – one based on the user reviews and the other based on the developers' fixes. The classification of categories has been based on these taxonomies. Although ACOCUR has been modeled specifically to identify the compatibility requirements, a similar approach can be used for other non-functional requirements as well.

The primary contributions of this thesis are the following:

- ACOCUR – the proposed method for analyzing compatibility requirements

- Empirically evaluated different machine learners in their capability to classify developers' commit and user reviews for identifying non-functional requirements.

- Answered RQ1 by measuring the percentage of effort dedicated by app developers to tackle incompatibilities in mobile apps.

- Evaluated RQ2 by analyzing the percentage of user reviews that complain about apps' incompatibilities.

- Responded to RQ3 by establishing the taxonomies for the different compatibility types based on the fixes done by the app developers and those reported in reviews by users.

- Analyzed RQ4 by comparing the degree of alignment between the users' compatibility requirements and the responsiveness of the developers in that regard.

- Identified the current state-of-the-practice (related to methods used, challenges faced, support from users, etc. with respect to identifying and fixing compatibility requirements for mobile apps – RQ5) by surveying mobile app developers and analyzing their responses.

- Resolved RQ6 by building an automated tool implementing ACOCUR to identify compatibility requirements from user reviews and commit messages and support app developers analyze their responses to these requirements.

1.4   Thesis outline

The rest of the thesis is organized as follows: Chapter 2 discusses the related work, Chapter 3 depicts our methodology, and Chapter 4 describes the survey with app developers and the results of the survey. Chapter 5 describes the proposed tool, Chapter 6 gives an overview of the data collection process and some of our initial analysis, and Chapter 7 describes the results from the empirical evaluation and the threats to validity. In Chapter 8 we have discussed our conclusion and outlined some future research agenda.

Chapter 2

Background and Literature Review

There has been considerable research done in identifying and tackling non-functional requirements. Although NFRs are an intrinsic part of every software system, there has always been some debate regarding what all should be considered as NFR. In this thesis, we deal with non-functional requirements related to mobile applications. In particular, we have concentrated on one particular non-functional requirement – *Compatibility* – and have studied it in the context of open-source Android mobile applications. There are four main areas of work related to our contributions: (a) studies related to non-functional requirements, (b) analyzing mobile application reviews, (c) comparison to techniques for text classification, and (d) studies on compatibility for mobile applications. All of the work discussed here addresses different facets of NFRs, app store mining and data analysis, natural language processing techniques, and other related subjects.

In this chapter, we shall discuss the work related to each of these areas. We have only mentioned some of the works that are closely related to this thesis; the studies mentioned here are by no means an exhaustive list of all the work done.

2.1   Studies related to non-functional requirements

Since non-functional requirements constitute a very broad area of study, we have divided the background work in this area under two categories: NFR studies on general software systems, and those specifically for mobile applications.

For general software

Non-functional requirements are prevalent in all software systems; as such there have been different studies related to NFRs. In this section, we shall discuss some of the researches that has been done specifically

related to non-functional requirements.

Huang et al. [19] were one of the first to suggest different information retrieval methods for detecting and classifying non-functional requirements from both structured requirement specifications as well as from unstructured text documents. To assess which all non-functional requirements are most focused on by software developers, Zou et al. [86] analyzed the non-functional requirements from Stack Overflow to comprehend the needs of the developers. As per their analysis, the aspects of *usability* and *reliability* are more important from the developers' perspective and consequently most discussed in this discussion forum – *maintainability* and *efficiency* are not their prime concerns. In another research, Hindle et al. [32] studied different software repositories to extract non-functional requirements; specifically, they applied topic modeling (LDA) on commit messages of large-scale projects to automatically label topics to different NFRs. This paper presented a cross-project data mining technique. While previous topic analysis research produced project-specific topics that needed to be manually labeled, in this study, the authors leveraged software engineering standards to produce a method of partially automated (supervised) and fully-automated (semi-unsupervised) topic labeling. In another recent study, Li et al. [43] proposed an ontology-based learning approach for automatically classifying security-related non-functional requirements.

There have also been some studies that have focused on NFRs in conjunction with continuous integration (CI) and continuous deployment (CD) approaches. Paixao et al. [57] have investigated the interplay between NFRs and build statuses in a continuous integration environment. Yu et al. [83] have conducted a literature review to identify the state-of-the-art for utilizing the continuous integration environment for NFR testing. They have also proposed a synthesized CI framework for testing various NFRs – the associated CI tools are also mapped. Similarly, Haindl et al. [29] devised an operationalizable quality model to measure, assess, and evaluate feature-dependent software quality characteristics (NFRs) throughout common DevOps toolchains.

While all of these studies are related to non-mobile software systems, a lot of them can also be extended to the mobile domain. Since our research is related to non-functional requirements for mobile applications, we take inspiration from all of these work and identify new ideas that have led to our contributions. Most of the work in this section has described different ways for classifying non-functional requirements or analyzing which NFRs are most widely discussed by developers. Our work differs from all these existing related work since we have analyzed one particular NFR related to mobile apps and have studies different aspects related to it. So although our work is similar to some of these studies in general, the nature of the study and the related contributions are different.

For mobile applications

To facilitate better management of non-functional requirements during the process of mobile application development, Garba et al. [24] have proposed a data-driven model. This study provides support for mobile application developers in dealing with non-functional requirements for mobile application development using a data-driven approach; the study has proposed a model that can facilitate the management of NFRs just like functional requirements.

Corbalán et al. [5] have performed a comparative analysis of six popular mobile application development frameworks by focusing on three types of NFRs - performance, energy consumption, and storage. Mobile app users heavily weight these non-functional requirements when it comes to deciding whether to install an app on their mobile devices. The results of this study would help app developers decide which development framework to prioritize the use of an approach over others, based on the expected levels of performance, energy consumption and use of storage space.

Similarly, Zahra et al. [84] have focused on the aspect of usability concerning mobile apps. In this study, different usability models have been compared; the authors opine that usability models for mobile applications are relatively unexplored and unproven. Although several usability models for mobile applications do exist, they are isolated and disintegrated. This issue is critical as existing usability guidelines are insufficient to design effective interfaces for mobile applications due to peculiar features and dynamic application context in mobile.

Ahmad et al. [10] investigated the non-functional requirements discussed by iOS developers over stack overflow. The objective of their study was to identify and understand the real problems, trends, and critical non-functional requirements related to iOS mobile application development.

While all of these studies, and few others, have focused on non-functional requirements for mobile apps, none of them have considered compatibility as one of the criteria; our work is related to compatibility in particular. Also, while some of the existing work has proposed ways to tackle NFRs or identify the most discussed NFRs in various forums, none of them have analyzed the current state-of-the-practice that is followed for dealing with NFRs. In our work, we have attempted to identify non-functional requirements from commit messages; this approach is novel and has not been addressed previously. Therefore, our work differs from all the above-mentioned studies is many aspects.

2.2 Comparison of techniques for text classification

The automatic extraction of requirements from text documents has been the focus of several requirements engineering researchers.

Maalej et al. [48] have emphasized the shift toward a data-driven user-centered identification, prioritization, and management of software requirements. As such developers should systematically use explicit and implicit user data in an aggregated form to support requirements decisions. The goal is data-driven requirements engineering by the masses and for the masses.

Knauss et al. [38] used a Naive Bayes (NB) approach to extract clarifications in requirements from software team communication artifacts to detect requirements that are not progressing in a project.

Guzman et al. [28] proposed an approach, ALERTme, to automatically classify tweets for software requirements and evolution. It uses supervised machine learning (Multinomial Naive Bayes) for classifying tweets, topic modeling for grouping related tweets, and a weighted function for ranking the tweets.

Maalej et al. [47] used probabilistic techniques to classify user reviews into four types: bug reports, feature requests, user experiences, and text ratings. They used review metadata such as the star rating and the tense, as well as, text classification, natural language processing, and sentiment analysis techniques. It was found that metadata alone results in poor classification accuracy. However, when combined with simple text classification and natural language pre-processing of the text, the classification performance increases significantly.

Similarly, Lu et at. [45] used ML classifiers in conjunction with bag-of-words (using word2vec) to extract NFRs from user reviews.

In [70], Toth performed a comparative analysis of the performance and applicability of the state-of-the-art techniques used in NLP and ML for text classification and extracting NFRs from the text. The results of their experiments showed that the linear classification algorithm produced the best values - Multinomial Naive Bayes, Support Vector Machine and the Logistic Regression classifiers outperformed all the other classifiers.

Abad et al. [9] investigated how the automated classification of requirements into NFRs can be improved and how different ML approaches perform in this context.

While all of these text classification techniques and many others have been validated on user reviews and other textual documents, none of them have been evaluated on commit messages. A major focus of our research in on classifying commit messages and identify how well do ML classifiers work for commit messages. There have been a few studies that have attempted to classify commit messages but in ways different from ours. As such, it is vital to test the different ML classifiers on commit messages to evaluate

their performance. Also, while these techniques have been applied to classify broad categories, our work is more directed towards one specific non-functional requirement.

2.3   Analyzing mobile applications reviews

Multiple research papers have investigated the nature of available information in mobile app stores, especially user reviews, and tried to automated the process of extracting relevant information from them.

Harman et al. [31] pioneered the concept of app store mining and established the correlation between the rating of an app and the download rank. Khalid et al. [35] manually examined low rated user reviews of iOS apps and identified what the most common user complaints were. Pagano et al. [56] conducted an exploratory study on a large number of reviews from the Apple App Store and built a taxonomy of common topics users talk about in their reviews. The topics identified by Pagano were very general (praise, helpfulness, feature information, etc.) and not specific to mobile applications. They found that users often report bugs and shortcomings of an app in reviews and that those reports have a strong influence on the rating of an app. Tian et al. [69] investigated the characteristics of high rated apps.

AR-MINER [17] represents one of the first automatic approaches to classify user reviews into informative and non-informative content. The paper concluded that only 35.1% of reviews are informative, further motivating the need for tools that automate the process of selecting relevant reviews. Gu and Kim [27] concentrated on analyzing sentiments in user reviews and proposed their approach (SUR-MINER) that summarises sentiments and opinions of reviews and classified them according to five predefined classes (aspect evaluation, bug reports, feature requests, praise, and others).

Panichella et al. [59] used a combination of Natural Language Processing, Text and Sentiment analysis techniques to classify reviews according to the following classes: Information Giving, Information Seeking, Feature Request, and Problem Discovery. Although this is useful, the classification is too general and does not address specific mobile issues. Villarroel et al. [71] proposed another approach, CLAP (Crowd Listener for releAse Planning) – this approach can automatically categorize reviews into a suggestion for new feature request, bug report and other. The tool then clusters the reviews and the developer is presented with a set of clusters that share similar terms. But they still have to analyze each cluster and determine what specific mobile issue they discuss.

Wang et al. [72] conducted an exploratory study to gain a deeper understanding of the nature of NFRs in user reviews and to further compare the differences in the distributions of various NFRs between app user reviews and industrial requirements specifications. The results of this study indicate that users report most frequently on *reliability* and *usability*. However, the study also concludes that most NFRs in user reviews

describe interface behavior of the systems which should be regarded as functional requirements.

Similarly, Jha et al. [33] mined non-functional requirements from iOS app store reviews and found that 40% of user reviews signify at least one type of NFRs. The results also show that users in different app categories tend to raise different types of NFRs.

The closest related work to ours is URR [18]; this approach can classify reviews according to both predefined high and low-level categories. There are six high-level categories one of which is compatibility, the other five being usage, resources, pricing, protection, and compliant. The compatibility class has three low-level categories which are device, Android version, and hardware. In our work, we have dived deep into the compatibility aspect only and have identified more discrete low-level categories. Also, while URR has attempted to identify the potentially associated source code that needs to be updated, we have worked towards mapping the reviews to the commit messages to identify how many of the reviews have already been addressed. Apart from these, some other factors distinguish our work from all of these described here.

2.4 Studies on compatibility for mobile apps

Compatibility is one of the important non-functional requirements when dealing with mobile applications. With the rapid growth in the diverse types of mobile devices and fast upgrading mobile operating systems, it is challenging for the mobile application developers to keep pace with this fast-changing environment. There have been several studies that have focused on the compatibility aspects of mobile apps and have advised various ways to cope up with this challenge. The impact of Android fragmentation has been extensively discussed in [30], where Han et al. have analyzed the bug reports related to two popular mobile device vendors. They have also studied how fragmentation is manifested within the Android project. For Android applications, Android fragmentation remains a compatibility issue for app developers. In [73], Wei et al. have studied over 200 compatibility issues in five popular mobile applications to understand and detect fragmentation induced compatibility issues for Android apps. Studies related to compatibility analysis can be broadly categorized into two subcategories: approaches for effective mobile application compatibility testing, and approaches to automatically detect and identify compatibility issues.

Mobile application compatibility testing approaches

One of the biggest challenges faced by mobile app developers is testing their apps for compatibility. Since the number of different combinations to test is infinitely large, mobile apps often suffer from limited testing when it comes to compatibility related test cases. Most app developers neither have enough resources to test their apps on many different mobile devices nor do they have the time to complete all these tests.

To overcome this, Huang [4] proposed an automated compatibility testing service for mobile apps named AppACTS – an online mobile app compatibility testing cloud facility where developers can upload a mobile app for automated device compatibility testing and obtain the test results.

Naith et al. [52] have proposed a framework for compatibility testing for mobile apps; this framework comprises of a crowd-sourced testing approach that leverages the power of the crowd to perform mobile device compatibility testing. It aims to provide support for testing code, features, and hardware characteristics of the mobile devices and ascertain app developers that the features and hardware characteristics of the device model or features of a specific OS version will work correctly and not cause any problems for their apps.

Another approach suggested by researchers to overcome the limitation of compatibility testing is prioritizing the mobile devices on which the apps should be tested for compatibility. This has been demonstrated in [46], where Lu et al. have proposed a tool named PRADA (approach to prioritizing Android device models for individual apps, based on mining large-scale usage data). PRADA adapts the concept of operational profiling for mobile apps – the usage of an app on a specific device model reflects the importance of that device model for the app. It also includes a collaborative filtering technique to predict the usage of an app on different device models based on the usage data of a large collection of apps.

While other studies have attempted to make the software testing process more effective, these are some of the approaches that have been specifically proposed to tackle the problem of compatibility testing for mobile applications.

Automatic Android version incompatibility detection

One of the major reasons for a large number of compatibility issues in mobile apps is the rapid evolution of the Android Application Programming Interface (API). While the above researches have devised techniques for testing mobile apps for compatibility related issues, another focus has been towards automatically detecting compatibility issues in mobile apps.

In [42], an automated approach named CiD has been proposed by Li et al. that would systematically model the life cycle of the Android APIs and analyze app byte code to flag usages that can lead to potential compatibility issues. It works by analyzing the changes in the history of Android APIs. This process has also been termed as "API side" learning.

To complement CiD, Scalabrino et al. [66] presented an alternative data-driven approach, ACRYL. Contrary to CiD, ACRYL learns from changes implemented in other apps in response to API changes ("client-side" learning). It not only detects compatibility issues in mobile apps but also suggests probable fixes.

While all the previous work in this section has dealt with compatibility in mobile apps, there has been limited research to understand how much of these approaches have been effectively used by the app devel-

opers. To the best of our knowledge, this is the first study that attempts to analyze the developers' effort dedicated to compatibility by extracting compatibility fixes from commit messages and compare it with the compatibility issues articulated by users.

2.5   Summary

In this chapter, we have discussed some of the work and background studies that are related to the different facets of this thesis. The studies have been categorized into groups related to this thesis's contributions. We have also discussed how the nature of our work is different from each of these studies. We concluded that the existing works do not provide an approach for identifying non-functional requirements from commit messages and link those to requirements in user reviews. In our proposed approach, we have identified compatibility fixes done by developers and associated those to requirements in reviews.

Chapter 3

Methodology

In this chapter, we discuss the methodology that we have used to answer our research questions. The different steps discussed in this chapter are the key components of our proposed ACOCUR approach. As already mentioned earlier, although this approach has been specifically fine-tuned for tackling the compatibility aspect, the same approach can be used for other non-functional requirements as well.

The ACOCUR approach mainly performs two types of analysis: (i) identify when the developers make compatibility related changes to apps, and (ii) investigate how do mobile app users react to compatibility related issues. The ACOCUR methodology works by automatically mining the mobile app's commits and reviews. Then it identifies the compatibility requirements from the reviews and the ones that have been already been fixed by the developers. After the compatibility related requirements and fixes have been extracted, those are further classified into the different types; the compatibility types can be a predefined set of types or they can also be dynamically generated for each app. In the final step of the process, the requirements are linked to the commits to analyze the responsiveness of the developers to these non-functional requirements.

Figure 3.1 represents the process flow for the ACOCUR methodology and the relationship to our stated research questions.

3.1 Android app selection, web scraping, and data collection

For this study, we focused on open-source Android mobile applications that are available under F-Droid, an open-source app repository. We extracted the app name, package name, and the address of the source code repository by crawling F-Droid. To limit the process of mining different repositories, we filtered out apps that were not on GitHub.
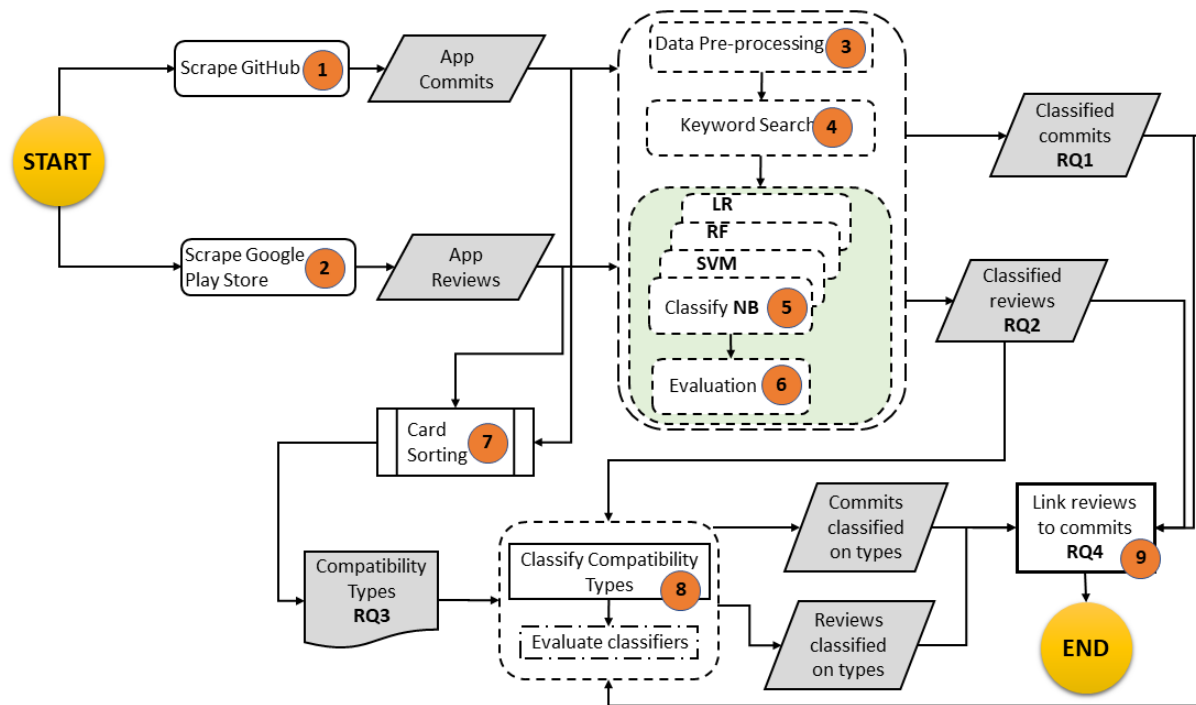
Figure 3.1: ACOCUR methodology process flow

For all the chosen apps, we gathered their GitHub repositories and collected all the app information from GitHub logs (Step **1** in Fig. 3.1). We used this data to mine all the commits associated with the apps. In particular, we extracted the following details for each commit: commit message, commit date, the number of files changed, and the number of lines added and/or deleted for each commit. We used the `GitPython` Python library (listed in Table 3.2 under Section 3.8) for collecting the GitHub repositories and also accessing the logs.

We built another custom web crawler (using `google-play-scraper` Python library – listed in Table 3.2 under Section 3.8) to gather the applications' details and user reviews from the Google Play (Step **2** in Fig. 3.1).

The details of the data collection process and the number of apps selected and the data gathered at each step of this process is described in Chapter 6.

3.2 Data pre-processing and lemmatization

The first step after data collection involves cleaning and pre-processing of the data (Step **3** in Fig. 3.1). This is a very important and significant process as the outcome of the model is dependent on this step.

Figure 3.2 shows the different steps followed to clean and pre-process the raw input data before it can be analyzed.
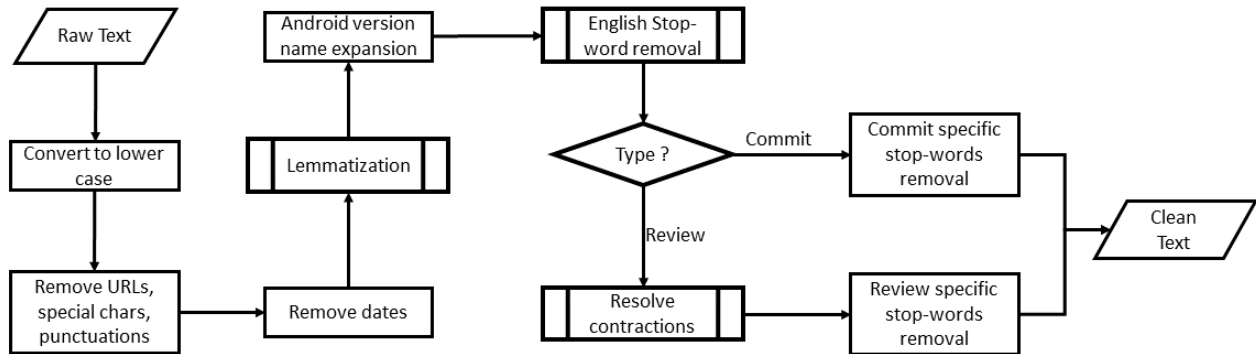
Figure 3.2: Data cleaning, pre-processing and lemmatization flow

Commit messages usually contain a short textual description describing the nature and details of the change performed in the corresponding code commit; it can also include other technical details. Similarly, user reviews often tend to be short and usually contain grammatical mistakes or typos. As such it is essential to clean and pre-process commit messages and reviews before it can be used for any processing or further analysis. Although the basic steps involved in this process are similar for both the commit messages and the reviews, there are some additional steps involved in both as the basic nature of the data is different. Here are some of the common steps used for cleaning the commit messages and user reviews:

- Converted all the text to lower case to ensure there is no case conflict

- Removed all links, URLs, non-ASCII characters, special characters, and all punctuation marks

- Removed all dates from the text

- Lemmatized texts to map words into their dictionary form while retaining the context of the word (using NLTK's WordNetLemmatizer)

- Expanded Android version names so that they are not removed during the cleaning process (for example, Android P is expanded to Android Pie)

- Removed English stop words from the text

For removing the stop words, we used the stop word list available under the `NLTK` Python package (refer to Table 3.2 under Section 3.8) [44]. However, we had to override the list and removed the word *not* from the stop word list so that this term is not removed as part of the stop word removal process. We have found that the word *not* is often used in review for describing issues and as such important for our analysis.

19

In additional to the above, we performed the following steps on user reviews as part of the data cleaning phase:

- Corrected typos and spelling mistakes.

- Resolved contractions (and slang), using `Contractions` Python library (listed in Table 3.2 under Section 3.8).

- Removed words that are very common in user reviews, and, thus, are not highly discriminating - review specific stop words.

We have leveraged the finding of a previous study [58] to identify the review specific stop word list and updated it as per our requirements. This list has been generated by calculating the entropy of all the terms present in the reviews of a large set of 1,000 Android apps. The normalized entropy [22] of a given term t in user reviews is calculated as:

$$E_t = \sum_{r \in R_t} p(t|r) \cdot \log_\mu p(t|r) \tag{3.1}$$

where R_t is the set of apps' reviews containing the term t, μ is the number of reviews on which the terms entropy is computed, and $p(t|r)$ represents the probability that the term t is in the review r. This probability is computed as the ratio between the number of occurrences of the term t in the review r and the total number of occurrences of the term t in all the considered reviews. E_t lies in the range of [0, 1] – the higher the value, the lower the discriminating power of the term. The final list of review specific stop words comprises of all terms whose $E_t > Q_3$ (the third quartile of the distribution of E_t).

Similar to reviews, while processing the commit messages, we removed some words that appear frequently in commit messages (for example: merge, pull, branch, etc.) and do not provide any additional information useful for this study.

For pre-processing, we have primarily used the `NLTK` Python package [44] in additional to our developed modules.

3.3 Keyword search

After the data pre-processing and lemmatization step, we performed a keyword search (Step **4** in Fig. 3.1) to identify the potential compatibility related texts (commit messages and reviews). These keywords are present in commit messages and reviews that represent any compatibility related aspect in mobile apps. This consolidated set of keywords (listed under Table 3.1) is the result of two types of activities – literature

review on mobile app compatibility, and repeated manual iterative search on the data set. We also calculated the *keyword-count* – the number of occurrences of these keywords – within each text (commit message or review).

Keywords
Compatible, Incompatible, Support, Sync, Device, Phone, Integrate, Android, Version, Upgrade, Update, Honeycomb, Ice Cream Sandwich, Jelly Bean, Kitkat, Lollipop, Marshmallow, Nougat, Oreo, Pie, Gingerbread, Froyo, Eclair, Donut, Cupcake, Bluetooth, Platform, Crash, Cloud, API, Samsung, Nexus, Redmi, Galaxy, Moto, Lenovo, Tablet, Pixel, Huawei, Nova, Htc

Table 3.1: List of Keywords

On running the keyword search on the commit messages, we identified all the commits that contained at least one of these words. The maximum value of keyword-count for the commit messages was nine. On evaluating some of the commit messages it was clear that there is a large amount of noise (false-positive) in the data – all the commit messages identified using the keyword match were not related to compatibility. Instead, many of the commits dealt with other aspects but they were obtained by the keyword search as some of the terms are general and have a wide range of applicability. To reduce this high number of false-positive results and attain a higher accuracy we adopted different machine learning classifiers that have been discussed in the next section.

Similar to the commit messages, we also performed a keyword search on the user reviews and identified all the reviews that matched this search. The maximum value of keyword-count for reviews was seven. Just like the commits, these reviews also had a high number of false-positives; i.e. for a large number of the reviews, although it contained one or more of the compatibility related keywords, these reviews did not describe any compatibility aspect. So, even for the reviews, it was necessary to employ classifiers to achieve better accuracy.

The following section describes the different steps related to the classification of commit messages and reviews.

3.4   Text classification

The process of text classification comprises of the following phases:

3.4.1   Building training set

Machine Learning classifiers are part of AI's supervised learning. As such it is imperative to train the classifier models using a training set before it can be used for classification. To achieve optimum results, it

is important to build a good training set such that it is representative of the actual data which would be classified using the classifiers. As we are dealing with two sets of data – commit messages and user reviews – we built two separate training sets.

It is also critical that the training set should be balanced, if possible. A balanced training set is one that has an equal number of records for all the different categories. Our goal for running the classifier is to identify the compatibility related text; i.e., we want to classify the text as either compatibility related or non-related. This is known as binary classification as the classifier has only two possible predictions.

As the keyword searches had resulted in a large number of false-positives, we attempted to make the training sets balanced by increasing the number of related records in the training sets; to achieve this, we mostly included those texts that had higher keyword-count expecting a majority of them to be related.

For the commit training set, we selected all commits with keyword-count of three or more (a total of 3353 commits). This was done to increase the possibility of finding related commits in the training set. We made a random selection of 6,667 commits from the remaining data set (keyword-count of one or two) and included them to build a training set of 10,000 commits.

For the reviews training set, we used a similar approach like that of the commits. We included all reviews with a keyword-count of three or more (5,492 reviews); to this, we randomly included 3,508 reviews with keyword-count of one or two – the reviews training set consisted of 9,000 records.

Once the training sets were identified, our next task was to manually annotate these records as either *related* or *non-related*. For labeling, we considered a commit message or review as *related* if it matched any of the below criteria:

- Discussed any issue/fix related to a particular device

- Discussed incompatibility between Android versions

- Talked about any behavioral discrepancy between versions

- Mentions any other compatibility related concerns/fixes

We took the help of two research interns to annotate the training sets. These interns had good command over different programming languages, and also a fair understanding of versioning control systems (GitHub) and mobile app reviews. They also had a sound understanding of non-functional requirements. Before annotating the training sets, these interns were familiarized with the context and goal of this research. They were also trained on different compatibility aspects related to mobile applications. The two training sets were divided amongst the two interns and also with the author of this thesis in such a way that each record was annotated by two individuals. Each annotator independently labeled the training set assigned to

him/her. This was done to mitigate the risk of any biased annotation. For most of the records, there was unanimity, and the labels associated with the text by the two annotators matched. We analyzed the degree of inter-annotator agreement by evaluating the Kappa coefficient (κ) [20] – a score that expresses the level of agreement between two annotators on a classification problem. It is defined as:

$$\kappa = (p_o - p_e)/(1 - p_e) \tag{3.2}$$

where p_o is the probability of agreement on the label assigned to any sample (the observed agreement), and p_e is the expected agreement when both annotators assign labels randomly [20, 11]. We used the `scikit-learn` Python library (listed in Table 3.2 under Section 3.8) to calculate inter-annotator agreement which showed an almost perfect agreement ($\kappa = 0.93$ for commits and $\kappa = 0.91$ for reviews).

The mismatched labels were discussed among all the annotators and then individually resolved. After resolving all the conflicts and eliminating the duplicate records, the commit training set had the following labels – 2,237 commit messages labeled as *related*, and 6,403 commit messages labeled as *non-related*. Similarly, the final training set for the reviews had the following configuration: 2,370 reviews were labeled as *related*, and 6,583 reviews were annotated to be *non-related*.

3.4.2 Text vectorization

Text Vectorization (a.k.a. Word Embedding) is the process of converting text into a numerical representation; it is an important step when dealing with textual data. There are different techniques available for text vectorization like Bag-of-Words, TF-IDF, Word2Vec, and many other pre-trained embedding models. In this section, we have discussed the four embedding techniques that have been used in our experiment. We have chosen these techniques since these are known methods that have been applied in previous NLP studies and they have shown promising outcome. In our experiment, we have evaluated these four techniques and selected the one that has produced the best results.

TF-IDF

The term TF-IDF [34] stands for Term Frequency-Inverse Document Frequency which gives the importance of the word in the corpus or dataset. This is one of the most frequently used text vectorization techniques. TF-IDF encompasses two concept – Term Frequency(TF) and Inverse Document Frequency(IDF) [79].

Term Frequency (TF): Term Frequency is defined as how frequently the word t appear in the document d. If n denotes the number of times t appears in d, and x denotes the total number of words in d, then TF is defined as:

$$tf(t,d) = \frac{\text{n}}{\text{x}} \qquad (3.3)$$

Inverse Document Frequency (IDF): Inverse Document frequency is a measure of how much information the word provide; it is based on the fact that less frequent words are more informative and important. If N represents the total number of documents, and D denotes the number of documents in which the term t appears, then IDF is represented by the formula:

$$idf(t,D) = \log \frac{\text{N}}{\text{D}} \qquad (3.4)$$

TF-IDF is the combination of these two concepts and is calculated as:

$$tf - idf(t,d,D) = tf(t,d) \cdot idf(t,D) \qquad (3.5)$$

A high value in tf–idf is reached by a high term frequency (in the given document) and a low document frequency of the term in the whole collection of documents. In this research, we have used the `scikit-learn` [60] Python library's implementation of TF-IDF.

Word2vec

Word2vec [50] is a deep learning technique that processes text by vectorizing words. It takes a corpus of text as the input and generates a set of vectors as the output: feature vectors that represent words in that corpus. This model is based on the assumption that "words appearing in similar locations will have similar meanings". It was developed by Google in 2013 and shows good performance in natural language processing studies. Word2vec can utilize either of the two model architectures to produce a distributed representation of words: continuous bag-of-words (CBOW) or continuous skip-gram. In the continuous bag-of-words architecture, the model predicts the current word from a window of surrounding context words. In continuous skip-gram architecture, the model uses the current word to predict the surrounding window of context words. It has been found that while CBOW is faster than skip-gram, the latter produces more accurate results.

Google provides a pre-trained word2vec model[1] using the skip-gram technique that has already been trained on a large data set (Google News Dataset with approximately 100 billion words). This model allows us to use a vector that more accurately represents the meaning of a word since this model has learned a large quantity of data. However, there is a disadvantage in that words that are not trained previously cannot be

---

[1]https://code.google.com/archive/p/word2vec

used unless they are newly learned.

In our experiment, we have used the pre-trained model to obtain vectors from words. Since we are dealing with sentences (a group of words), we have used two approaches to convert word embedding to sentence embedding: Word2vec(Avg) – the vector representation of a sentence is equal to the average of the vectors of all the words in that sentence, and Word2vec(TF-IDF) – a combination of TF-IDF on word vectors. To implement Word2vec in our research, we utilized the `Gensim` Python library (details listed in Table 3.2 under Section 3.8) [1].

Doc2vec

The Doc2vec approach [40] is an extension of the simple Word2vec embedding technique; this provides a vectorized representation of a sentence (a group of words) taken collectively as a single unit. It does not give the simple average of the words in the sentence. Doc2vec computes a feature vector for every document in the corpus. It is an unsupervised framework that learns continuous distributed vector representations for pieces of texts. The texts can be of variable-length, ranging from sentences to documents. This is also referred to as *Paragraph Vector* to emphasize the fact that the method can be applied to variable-length pieces of texts, anything from a phrase or sentence to a large document. Doc2Vec comes in two variants: distributed memory model, and distributed bag of words.

We have used the `Gensim` Python library's Doc2vec implementation in our work.

GloVe

GloVe [62] (coined from Global Vectors) is another unsupervised learning algorithm for obtaining vector representations for words; it was developed by Stanford University in 2014. They noted that the existing statistic-based word embedding techniques (e.g., bag-of-words, TF-IDF) show low semantic inference performance and that the existing prediction-based word embedding techniques (e.g., word2vec) do not include all statistics because they consider only surrounding words; GloVe solves these problems. The GloVe model is based on the idea that semantic relationships between words can be derived from the co-occurrence matrix. The advantage of GloVe is that, unlike Word2vec, GloVe does not rely just on local statistics (local context information of words), but incorporates global statistics (word co-occurrence) to obtain word vectors. In other words, it is a prediction-based word embedding technique that includes all of the statistics.

GloVe also has some pre-trained models[2] that have already been trained on large data corpus (Wikipedia, Twitter data, etc.). However, this model also cannot use words that have not been learned previously

---

[2]https://nlp.stanford.edu/projects/glove/

without additional training. In our experiment, we have used a pre-trained model[3] (trained with Wikipedia data, approximately 6 billion tokens) to obtain vectors from words. To incorporate this model in our code, we implemented the `Gensim` Python library's *glove2word2vec* script that coverts a GloVe format to a Word2vec format. Once converted, we used the same procedure as the Word2vec model described above. We implemented two approaches to build the sentence vectors – GloVe(Avg), and Glove(TF-IDF).

3.4.3   Treating unbalanced training set

It was evident that our training sets were unbalanced as the number of non-related records was considerably higher as compared to the related ones. It is recommended to use balanced training sets for training classifier models, otherwise, the results could be skewed. In our case, as the training set had a large number of non-related text (both commits and reviews), the classifier model could be biased towards the non-related class. There are two approaches proposed to deal with unbalanced data: under-sampling (a.k.a. down-sampling), and over-sampling (a.k.a. up-sampling). In the under-sampling technique, we take a subset of the samples from the class with more instances to match the number of samples of each class. In the case of over-sampling, we randomly duplicate samples from the class with fewer instances or we generate additional instances based on the data that we have, to match the number of samples in each class.

In our experiment, we used *SMOTE* [16] – a process of oversampling by augmenting synthesized data samples. A plausible downside of the approach is that the synthetic examples are created without considering the majority class which might result in the creation of ambiguous examples. We have used the `imbalanced-learn` [41] Python package (details listed in Table 3.2 under Section 3.8) implementation of SMOTE for treating unbalanced training sets.

3.4.4   Classifiers

The final step in text classification was classifying all the commit messages and reviews using *supervised learning*. For that, the Machine learning classifiers (Step ❺ in Fig. 3.1) should be first trained using the training sets. The input to the models is the vectorized texts that were generated in the previous text vectorization steps.

We chose four machine learning classifiers, *Naive Bayes (NB)*, *Support Vector Machine (SVM)*, *Random Forest (RF)*, and *Logistic Regression (LR)*, all of which are known to perform well with text classification [70].

The NB classifier [64] is a simple algorithm that is based on the Bayes' theorem using the naive assumption

---

[3]http://nlp.stanford.edu/data/glove.6B.zip

that each feature makes an independent and equal contribution to the outcome. It finds the probability of an event occurring given the probability of another event that has already occurred. Bayes' theorem is stated mathematically as the following equation:

$$P(y|X) = \frac{P(X|y)P(y)}{P(X)} \tag{3.6}$$

where y is the class variable and X is a dependent feature vector.

The Support Vector Machine [21] algorithm finds a hyperplane in N-dimensional space (where N is the number of features) that distinctly classifies the data points. The Figure 3.3a represents a simplistic SVM classification approach.
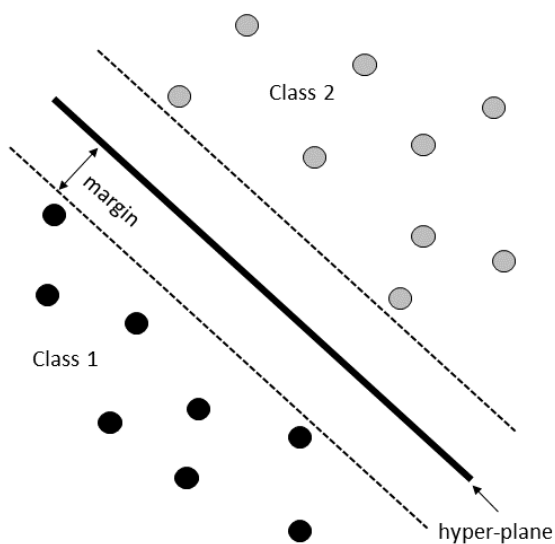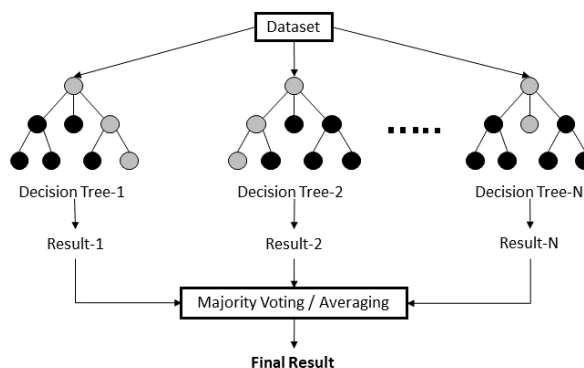
(a) Support Vector Machine   (b) Random Forest

Figure 3.3: Classifier algorithms (figure adapted from [80])

The Random Forest algorithm [14], like its name implies, consists of a large number of individual decision trees that operate as an ensemble (ensemble methods use multiple learning algorithms to obtain better predictive performance). Each tree in the random forest makes a class prediction and the class with the most votes becomes the model's prediction; Figure 3.3b contains a simple representation of the RF classifier.

Logistic regression [37] is a classification algorithm, used when the value of the target variable is categorical. Logistic regression is most commonly used when the data in question has binary output, so when it

belongs to one class or another. It implements a sigmoid function that resembles an 'S' shaped curve when plotted on a graph. The sigmoid function [78] is defined as :

$$y = 1/(1 + e^{-x}) \tag{3.7}$$

where y is the predicted class variable and x is a dependent feature vector.

For implementing these classifier algorithms in our research, we have used the `Scikit-learn` [61] open-source Python library. These machine learning models are defined as mathematical models that have several parameters which are needed to be learned from the data. However, there are some parameters (known as hyper-parameters) that cannot be directly learned. These are commonly chosen by humans based on some intuition or hit and trial before the actual training begins. These parameters exhibit their importance by improving the performance of the model such as its complexity or its learning rate. Models can have many hyper-parameters and finding the best combination of parameters is often a time-consuming and difficult process.

Several studies use optimization techniques to find a set of hyper-parameter values that induces classifiers with good predictive performance. In our experiments, we have used the *Grid Search* technique for identifying these parameter values. This approach works by performing an exhaustive search and tries all the combinations of the different parameter values that have been supplied. We have implemented this by using the *GridSearchCV* class from the `Scikit-learn` library. GridSearchCV takes a dictionary of all the different parameters along with their different values that could be tried on a model to train it. It then performs an exhaustive search and returns the *best_estimator_* and *best_params_* values which is used to identify the ideal combination of hyper-parameter values.

3.4.5   Classifier performance evaluation

In order to assess the classifier models (Step **6** in Fig. 3.1) and evaluate their performance, we used the *10-fold cross-validation* technique. Cross-validation is a statistical resampling procedure used to evaluate machine learning models on a limited data sample [6]. It is commonly used in applied machine learning to compare and select a model for a given predictive modeling problem because it results in skill estimates that generally have a lower bias than other methods. The procedure has a single parameter (k) that refers to the number of groups that a given data sample is to be split into. As such, the procedure is often called k-fold cross-validation. In our experiment, we had chosen $k = 10$; so we have referred to it as 10-fold cross-validation.

This method is primarily used to evaluate a machine learning model on unseen data. The general procedure of cross-validation is as follows:

- Shuffle the data set randomly.

- Split the data set into k groups

- For each unique group:

    - Take the group as a holdout or test data set

    - Take the remaining groups as a training data set

    - Fit a model on the training set and evaluate it on the test set

    - Retain the evaluation score and discard the model

- Summarize the skill of the model across all the groups

It is important to note that each observation in the data sample is assigned to an individual group and stays in that group for the duration of the procedure. This means that each sample is allowed to be used in the test set once and used to train the model k-1 times. The results of a k-fold cross-validation run are finally summarized with the mean of the model skill scores.

We evaluated the performance of classifier models based on the measures of *Precision (P)*, *Recall (R)*, *Accuracy (A)*, and *F1-Score (F1)*, which are defined as:

$$P = \frac{TP}{TP + FP} \tag{3.8}$$

$$R = \frac{TP}{TP + FN} \tag{3.9}$$

$$A = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.10}$$

$$F1 = 2 * \frac{P * R}{P + R} \tag{3.11}$$

Therein, *TP*, *TN*, *FP*, and *FN* denote *true-positive*, *true-negative*, *false-positive* and *false-negative* respectively. We selected the classifier that provided the best results and used it to classify the remaining commit and review messages.

We have used the `scikit-learn` [60] Python library's implementation for k-fold cross-validation and performance metric scores.

3.5 Card sorting for taxonomy generation

To identify the different types of compatibility fixes done by the developers in the respective commits and also the different types of compatibility issues raised by users in their review, we adopted the card sorting approach (Step **7** in Fig. 3.1). This also helped us generate a taxonomy of compatibility types for issues and fixes. Card sorting is an exploratory technique adopted to extract themes from text [65, 85]. It is widely used to create mental models and derive taxonomies from data. In our case, card sorting also helped us to deduce a higher level of abstraction and identify common themes. There are commonly two types of card sorting [13, 85]: *Open* card sort – where there are no predefined groups and the categories emerge and evolve during the sorting process, and *Closed* card sort – where the themes are known in advance and the categories are predefined. Card sorting has three phases: in the *preparation phase*, the cards are created; in the *execution phase*, cards are sorted into meaningful groups; finally, in the *analysis phase*, abstract hierarchies are formed to deduce general categories and themes.

For our study, we performed the card sorting procedure twice; once for categorizing the compatibility related commit messages, and the second time for categorizing the compatibility related reviews. Both these experiments were performed independently of each other as the nature and the source of data were different and we wanted to compare and contrast the taxonomies generated from the two data sets. However, the process followed in both cases was similar. We adopted the *Hybrid Card sort* approach – we started with a representative sample of cards and identified the different categories using an open card sort technique, and then sort the remaining cards into the identified categories based on the closed card sort approach. We performed the open card sort with 2,000 cards and identified the different categories, and subsequently executed a closed sorting of the remaining 1,000 cards. Owing to the global Covid-19 pandemic, we were forced to adopt an online card sorting approach as we could not physically take part in this exercise.

This exercise was conducted with the help of the same two research interns who were instrumental for building the training sets as described above in Section 3.4. Since these interns were already trained earlier and they had successfully annotated the training sets, they had already developed a fair understanding of the context of this study and the expectation from the card sorting exercise. We also performed a workshop with them having the author of this thesis as the facilitator to create a shared understanding of the compatibility related commit messages and user reviews and the objective of the card sort exercise. Within this workshop, each participant was assigned randomly to some commit messages (or user reviews) and reflected what types of incompatibilities had been addressed (or reported, in case of reviews) in them. The reflection was discussed in the group and settled once everyone got a shared understanding.

We mined taxonomies in separate sessions to extract *"What types of compatibility issues are fixed in*

developer commits" and *"What type of compatibility issues are raised in user reviews"*. We limited the number of cards sorted in each session in a way that each session took no more than three hours. Overall, we performed the following four major steps:

- Preparing Cards: We used cards for open sorting from the manually labeled training sets (using the ones that were labeled as compatibility related) for training the classifiers. For closed card sorting, we randomly selected a subset of records (commit messages in one case, and reviews in the other) that were classified by our chosen identifier as compatibility-related.

- Open card Sorting: To identify the different categories, the participants independently categorized and grouped the cards. Each of them dealt with 1,000 cards. After the initial categories were formed by the two participants, the session moderator (author of this thesis) discussed the mutual and different categories until the team agreed upon a final set of categories. To limit the number of categories to a reasonable finite number, some of the categories were further merged with everyone's consensus.

- Closed Card Sorting: The two participants sorted the remaining 1,000 cards into categories identified by open card sorting. 200 cards were categorized by all the members to evaluate their degree of conformity; we used the Kappa measure [49] for this. The Kappa coefficient value was 0.83 on average for the two taxonomies that we generated — this reflects an overall good level of agreement.

- Analysis and Taxonomy design: In this phase, the low-level categories were further grouped and the relationship between the different categories was determined. After some iterative process, the final high-level taxonomy was determined. The disagreements were resolved by discussion.

3.6 Classifying based on compatibility types

The different compatibility types (for commits and reviews) were determined using the card-sorting technique as discussed in Section 3.5. The next step in our research methodology was to again classify all the compatibility related records (commits and reviews) into the different categories identified (Step **8** in Fig. 3.1). The process followed for this classification was similar to what we had done earlier for classifying related vs non-related. The only difference, in this case, was that we performed a multi-label classification (whereas, earlier it was binary classification). We built two training sets (one for commits and the other for reviews) to train the classifiers. This time we used the same training set that we had built earlier (output of Section 3.4.1). We isolated the ones that were identified as related (2,237 and 2,370 for commits and reviews respectively) and labeled them according to their compatibility types.

The training sets were again divided amongst the three annotators (two interns and the author of this thesis) such that each record was labeled by two annotators independently. After the annotation was completed, the training sets were compared. While there was a high-level agreement on the commit training set, the number of disagreements was high for the reviews training set. While resolving the disputed ones and analyzing the review set, it was identified that some of the reviews could be matched to more than one class based on the text; that was the primary reason for the higher number of disagreements in the review training set. We mutually agreed to a single class for all the disputed samples and finalized the training sets.

We used the same four classifiers (*Naive Bayes*, *Support Vector Machine*, *Random Forest*, and *Logistic Regression*) that we had evaluated earlier (Section 3.4.4). These classifiers were again trained with the new training sets and their performances were evaluated similarly; the one with the optimum performance was eventually selected for classifying the remaining commits and reviews into the different compatibility types.

3.7 Measuring the responsiveness of developers to user reviews

The final step of the ACOCUR methodology (Step **9** in Fig. 3.1) involves linking the requirements (from reviews) to the fixes (in commits). In this section, we shall discuss the two approaches that we have undertaken to measure the responsiveness of developers to user reviews. The first approach describes the attempt to map individual review to commits to check if each review complaining about compatibility has been addressed in any commit. The second approach analyzes the percentage of commits and reviews for the different compatibility types.

3.7.1 Linking individual reviews to commits

In this phase, we mapped the commits to the reviews; i.e., we were interested in evaluating if a particular compatibility fix (in commit messages) was the result of a compatibility issue raised in the reviews. Analyzing this aspect would help us answer the research question related to the responsiveness of app developers to user reviews. To effectively evaluate this, we assumed that :

- Users always have the latest version of the app

- The app review is related to the most current version available in the play store

- If a compatibility fix is related to a user review, the fix should be done within a specified period (for our research we set this period to 90 days since most apps release newer versions at quick intervals).

This mapping was done in a two-step process:

1. Identifying all the possible commits that can be linked to the review based on the date

2. Select the most suitable commit based on similarity

The steps followed for this mapping procedure is described below:

1. Identify all the compatibility related issues raised in user reviews of the app

2. Extract all the compatibility related commits of the app

3. For each of the reviews, identify the commits that have been pushed after the review date and within the specified period (90 days)

4. Compute the similarity between the review and the commit messages obtained in the above step

5. Filter out commits based on similarity scores and establish a suitable link between the two data sets

To calculate the similarity between a review and the corresponding commit messages, we have used the following two approaches:

- **Textual similarity computation**

  In order to link a commit to their corresponding review, we computed the textual similarity between the reviews and their corresponding commits. There are different measures for evaluating similarity (like Jaccard coefficient [55], cosine similarity [76], etc.). In our case, we used the asymmetric Dice similarity coefficient to compute a textual similarity between a review and a commit message [12, 58]. We used the asymmetric Dice coefficient instead of other similarity measures because in most cases commit messages are much shorter than user reviews and, as a consequence, their vocabulary is fairly limited. Using this coefficient, the textual similarity (sim_{txt}) between review r_j and commit message c_i is defined as :

$$sim_{txt}(r_j, c_i) = \frac{|W_{r_j} \cap W_{c_i}|}{\min(|W_{r_j}|, |W_{c_i}|)} \tag{3.12}$$

  where W_x denotes the set of words contained in document x and the min function aims at normalizing the similarity score based on the number of words contained in the shorter document between the review and the commit message. The asymmetric Dice similarity ranges in the interval [0, 1].

- **Semantic similarity computation**

  While the above textual similarity works on the basis of word matching, a more advanced method would be semantic similarity matching. In order to evaluate the semantic similarity between a review and the corresponding commit messages, we have calculated the cosine similarity as well as the Word Mover's Distance [39] between the two text vectors. Among the different text vectorization approaches as discussed under Section 3.4.2, we have used the GloVe Text vectorization technique to convert the review and commit messages to vectors and then computed the cosine similarity between them. The semantic similarity between two text vectors (A and B) using cosine similarity [76] is defined as :

  $$sim_{cos} = \cos(\theta) = \frac{A \cdot B}{|A||B|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \sqrt{\sum_{i=1}^{n} (B_i)^2}} \tag{3.13}$$

  Cosine similarity is a measure of similarity between two non-zero vectors of an inner product space that measures the cosine of the angle between them. It is thus a judgment of orientation and not magnitude: two vectors with the same orientation have a cosine similarity of 1, two vectors oriented perpendicular to each other have a similarity of 0. Thus the similarity ranges in the interval [0, 1]. The cosine similarity is advantageous because even if the two similar documents are far apart by the Euclidean distance (due to the size of the document), chances are they may still be oriented closer together. The smaller the angle, the higher the cosine similarity.

  Word Mover's Distance (WMD) [8] uses the word embeddings of the words in two texts to measure the minimum distance that the words in one text need to travel in semantic space to reach the words in the other text. The WMD is calculated by measuring the minimum Earth mover's distance between each word in the two documents in word2vec space; if the distance is small, then words in the two documents are close to each other.

  In order to calculate the cosine similarity and the WMD, we have used their Python implementations from the `SciPy` [2] and `Gensim` Python libraries respectively (details listed in Table 3.2 under Section 3.8).

3.7.2 Linking compatibility types of reviews to those in commits

The second approach that we have used for measuring the alignment of the responsiveness of developers to user reviews is by comparing the distribution of commits and reviews across the different compatibility types. As discussed in Section 3.6, we have classified the reviews and the commits based on the compatibility types. In this phase, we have compared the percentage allocation of the commits and reviews across different

compatibility types to measure the alignment of developer responsiveness to user requests.

We consider a good alignment as the one where the distribution of reviews and commits across the different compatibility types are similar.

3.8   List of ML and NLP tools

In order to implement the ACOCUR methodology, we have used Python 3.6 programming language. The NLP techniques and the ML tools that have been used in the different phases of the ACOCUR methodology have been implemented using open-source python libraries. In Table 3.2, we have listed all the Python libraries that have been used in this thesis.

Libraries	Version
google_play_scraper	0.0.2.2
GitPython	3.1.3
imbalanced-learn	0.6.2
Contractions	0.0.25
NLTK	3.4.5
scikit-learn	0.22.2.post1
Gensim	3.4.0
SciPy	1.4.1
Pandas	0.25.1
Numpy	1.19.0

Table 3.2: List of Python Libraries

3.9   Summary

In this chapter, we have described the methodology for the proposed ACOCUR approach. It includes all the steps right from the data collection process to the final process of mapping reviews to commits. It is worth noting that some of the steps like choosing the word embedding technique or building the taxonomy using a card-sorting approach is a one-time activity and it will not be repeated for each run; these steps are executed only for the first time. But we have mentioned these processes under the methodology section since they are an intrinsic part of the ACOCUR approach. Also, these steps should be performed if the methodology is applied for other types of non-functional requirements.

Following these processes, we can answer the first four research questions that deal with identifying the compatibility fixes and requirements from commits and reviews respectively. The last two research questions have been discussed in the subsequent chapters.

Chapter 4

A Survey with Android Developers

As discussed under Section 1.2, it is very important to understand the current state-of-the-practice that is followed by app developers for dealing with the compatibility requirements. To answer our fifth research question (RQ5), we decided to conduct a survey with app developers and analyze their feedback. In this chapter, we have described the process that we have followed to conduct the survey and also its results.

4.1   Conducting the survey

To gain deeper insight into the compatibility aspects and understand the developers' perspectives better, we surveyed with mobile application developers. It not only helped us answer RQ5 (determine the state-of-the-practice with respect to the methods used for identifying the compatibility requirements, challenges faced in the process, importance of fixing these requirements, support from users etc), but also helped us clarify and validate answers obtained for some of the other research questions.

However, since this survey involved working with humans (app developers), it was imperative that the research should be reviewed by the appropriate board or committee and the necessary approval received before the survey could begin. For this thesis, we approached the University of Calgary's Research Ethics Board (Conjoint Faculties Research Ethics Board, CFREB) responsible for reviewing research applications involving human participants. Since we already had an existing ethics approval for a similar study that also involved working with mobile app developers, this approval was renewed for this thesis before conducting the survey; the ethics approval (Ethics Id: REB15-1986_REN5) is valid till July 30, 2021.

In this section, we describe the main steps for conducting the survey.

4.1.1 Survey participants

This survey was targeted primarily for Android mobile application developers as this research has dealt with Android apps only. We used convenient sampling [36] method among the list of app developers whose apps have been included in this study. We invited 150 Android app developers to respond to this survey. Out of these, 141 developers participated in the survey (response rate = 94%). Five responses were incomplete, and we removed them from the analysis; the survey results have been based on the remaining 136 valid responses.

4.1.2 Survey question format

This survey has been designed in a Google Form [1] so that it can be quickly customized and easily accessible to the participants. We had designed the survey with 14 questions that were divided into three main sections.

Survey Section I: The first section contained four questions that were related to the participant's profile to understand the demography of the developers. Answers to these questions enabled us to understand the app developers better. This section was essential to get a better understanding of the developers' details and also how the answers provided by them in the other two sections were shaped by their experiences.

Survey Section II: The second section of the survey was related to the compatibility issues faced and addressed by the developers. Before conducting the survey, we had already completed analyzing the commit messages to understand the different types of compatibility issues addressed in the apps. This section was designed with the knowledge and understanding that we had already gathered by analyzing the commit messages. This section also contained four questions. Answers to these questions would enable us to measure how close our initial findings are aligned to that of the developers. This section was also designed to get insight into details that were not revealed from the commit messages.

Survey Section III: The last section of the survey contained six questions: these questions were related to the compatibility issues raised by users in review and how the developers deal and respond to them. The primary focus of our research has been to identify how app developers respond to compatibility issues raised in user reviews. This section has been designed to understand this perspective better.

The questions were designed in a manner keeping the participants' ease and comfort in mind, and at the same time ensuring that the survey met its purpose. There were different patterns of questions included in

---

[1]https://forms.gle/vcHFUnYi7oRyVJnE6

the survey — selecting a single option from a list of prescribed answers (radio button or drop-down list), selecting multiple options from a list of answers (checkbox), rating different options in a Likert or Nominal scale, and free flow text option to allow developers provide additional details.

4.1.3   Survey objective

This survey was designed with the objective to:

- Understand how important compatibility issues are for the developers

- Evaluate the different ways used by developers to identify compatibility issues (sources of information) and how important those sources are

- Identify how much importance developers provide to user reviews

- Gauge the different ways used by developers to extract compatibility issues raised in reviews

- Explore the need for automated tools to identify compatibility issues raised by users

- Grab the developers' opinion on how accurately users report compatibility issues in reviews

- Estimate the responsiveness of developers to compatibility issues raised in reviews

4.2   Survey results

In this section, we shall discuss the results of the survey. The analysis is based on the 136 valid responses that were received.

4.2.1   Accessing developers' profile

The first section of the survey was designed to understand the demographics of the developers. Table 4.1 contains the responses of the developers to the questions in this regard.

The app developers had varying years of experience in mobile app development. To categorize their responses, we formulated three groups based on their app development experience: Beginners (developers with less than 1 year of experience), Junior developers (those whose experience varied between 1 and 4 years), and Senior developers (having more than four years of app development experience). Among the respondents, only 14 developers ($\approx 10.29\%$) belonged to the Beginners category. A majority of the developers ($78, \approx 57.35\%$) corresponded to the Junior developers group; and the remaining 44 developers ($\approx 32.35\%$)
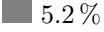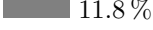
Demographics		
Q1. Total years of experience in mobile application development	Less than one year	10.3 %
	1 − 2 years	26.5 %
	3 − 4 years	30.9 %
	5 − 7 years	17.7 %
	8 − 10 years	9.6 %
	More than ten years	5.2 %
Q2. Total number of developed apps	One app only	20.6 %
	2 − 5 apps	48.5 %
	6 − 10 apps	15.4 %
	More than ten apps	15.4 %
Q3. Average number of commits for each app releases	Less than ten	19.9 %
	10 − 20	26.5 %
	21 − 50	27.9 %
	51 − 100	14.0 %
	More than hundred	11.8 %

Table 4.1: Section I – Developers' profile

were mapped to the Senior developers category. There were 20 developers with more than 8 years of Android app development experience.

In terms of the total number of apps that had been developed by these developers (during the time of this survey), 28 of them (20.58%) had developed only 1 app, while 66 developers (48.53%) had developed between 2 to 5 apps; there were 21 developers (15.44%) who had developed between 6 and 10 apps, while the remaining 21 had developed more than 10 apps.

In total, 63.24% of the app developers had more than three years of experience and 68.38% of them have developed at least three Android apps. The developers have served various roles in the development team, while some have been solo developers responsible for all the activities. To know how much involved they have been during the app development process, we asked on an average how many commits they have performed for each app release. To this 19.86% answered to have performed less than 10 commits on an average (a majority of them are Beginners), 54.41% have performed in between 10 to 50 commits, 13.97% did in between 50 and 100 commits, while the remaining 11.76% have attributed to more than 100 commits for each app release. From these statistics, it is evident that a majority of the app developers have been thoroughly involved during the app development process.

4.2.2 Analyze app compatibility

Mobile app compatibility						
Q5. Importance of identifying and fixing app's incompatibility issues	1 (Not Important)	3.7 %				
	2	4.4 %				
	3	8.8 %				
	4	31.6 %				
	5 (Very Important)	51.5 %				
Q6. Total number of compatibility fixes done	Less than ten	38.2 %				
	10 − 20	22.1 %				
	21 − 30	21.3 %				
	31 − 50	9.6 %				
	51 − 100	4.4 %				
	More than hundred	4.4 %				
Q7. Importance of different sources for identifying compatibility issue		Very Low	Low	Medium	High	Very High
	Testing	13.2 %	15.4 %	29.4 %	30.1 %	11.8 %
	Adhoc	7.4 %	9.6 %	30.9 %	35.3 %	16.9 %
	Review	5.9 %	7.4 %	11.0 %	35.3 %	40.4 %
	Others	21.3 %	10.3 %	44.9 %	16.9 %	6.6 %
Q8. Frequency of occurrence of different types of compatibility issues		Very Low	Low	Medium	High	Very High
	Android versions	3.7 %	13.2 %	30.9 %	33.8 %	18.4 %
	Device	8.1 %	15.4 %	27.9 %	30.1 %	18.4 %
	Software	16.2 %	25 %	35.3 %	15.4 %	8.1 %
	Others	27.9 %	18.4 %	42.6 %	8.8 %	2.2 %

Table 4.2: Section II – Questions related to app compatibility

This section of the survey was designed to elicit the experience of the app developers while dealing with the compatibility aspect. Table 4.2 shows the responses of the developers to the four questions.

The first question in this section was related to the importance on identifying and fixing incompatibilities in mobile apps; on a scale of 1 to 5, more than 83% of the developers responded with a 4 or 5, while only 8% responded with 1 or 2 (a majority of them are Beginners). Taking the developer's experience into account, 86.36% of the Senior developers and 83.33% of the Junior developers have responded with a 4 or 5. This demonstrates that app developers consider compatibility as an important aspect of mobile app development. Experienced developers have associated more importance to compatibility as compared to those with less experience.

The next question in this section was related to the number of compatibility fixes that the developer has addressed. To this question, 38.24% of the responses indicated less than 10 fixes in total, while another 22% believed to have fixed in between 10 and 20 issues; 30.88% of the answers indicated 20 to 50 fixes, 4.41% between 50 and 100, and 4.41% for more than 100 fixes. Unsurprisingly, the percentage of experienced developers gradually increased as we moved from the lower to the higher ranges. Considering how important compatibility aspect is for app developers and the numerous types of issues that occur in apps, the number of fixes done by developers seems rather small. One of the common remarks from the developers regarding this question was that the survey should have included an option as "Do not know" as the majority of them have never analyzed how many of their commits addressed incompatibility issues.

The third question of the survey was related to the different sources for identifying compatibility issues and how important these sources are. About 75.74% of the respondents considered the importance of user reviews for identifying compatibility issues as *high* or *very high* (this proportion increases further if we consider experienced developers), and only 13% of the remaining considered reviews' importance as *low* or *very low* (again this proportion decreases with increase in developer's experience). Among the other sources, only 41.91% considered the importance of dedicated testing as high or very high; on the contrary, they attributed a higher percentage (52.21%) to ad-hoc chances of identifying compatibility issues by the development team. 23.53% considered other sources of information in this regard as important. So, it was again evident that app developers considered user reviews as the most important source for identifying compatibility issues - this sense of importance shows an upward trend along with the experience of the developers. Also, among the issues that were identified by the developers directly (and not from external sources), they attached a higher percentage to finding the compatibility issues by unplanned means than through dedicated testing.

The last question in this section was related to how frequently the different types of compatibility issues occurred. These different types were based on the high-level categories that were identified during the card-sorting exercise. About 83% of them considered Android version incompatibility issues as the most frequently occurring. Also, 48.53% of them opined device-specific incompatibility frequency as high. They attributed a lower percentage to other incompatibility types. To get further information about other types of incompatibility, the survey offered the option to the developers to provide other types of incompatibilities that they frequently face. Around 48.48% of these responses could be associated with different devices and hardware incompatibilities, while one-third of the responses described incompatibilities arising from the operating system versions. Only a small percentage of the remaining issues could be associated with other factors.

Key findings

Here are some of the our key findings from the survey analysis:

- With the increase of experience in app development, the importance associated with compatibility also rises; among the survey participants, senior developers attached the highest importance to compatibility.

- Developers often can not comprehend what percentage of their effort is dedicated to addressing the compatibility aspects.

- Among the various sources for identifying compatibility requirements or issues, developers regard reviews as the most important.

4.2.3   Identify compatibility requirements from reviews

From the answers in the previous section, it has been evident that developers consider user reviews very important for identifying compatibility issues. The last section of the survey focused specifically on user reviews as a source for identifying incompatibility issues and how developers tackle reviews. This section contained six questions (Table 4.3).

The first question in this section was related to the average number of reviews that the developers have studied for each app release to analyze the users' concerns. 44.12% of the developers responded that they have analyzed less than 10 user reviews on average, while another 28.68% answered that they reviewed in between 10 and 50 user reviews. The percentage of developers reading a higher number of reviews has steadily declined. This is strikingly strange considering that developers consider reviews so important.

The next question was related to the different ways adopted by developers to identify compatibility issues reported in user reviews. A majority of the developers (61.76%) reflected that they manually screen user reviews to identify compatibility issues. This can be directly associated with the fact that developers read only a few user reviews considering the amount of time and effort that needs to be devoted to identifying compatibility issues from user reviews manually. Other than manually reading all reviews, 29.41% of the developers stressed that they filter only on low rating reviews and use those to identify compatibility issues. Although this option might reduce the number of reviews, it has been found that often compatibility issues are reported along with other suggestions and compliments, and sometimes the rating of the review is also high. While a similar percentage of developers also referred to using a mixed ad-hoc means for identifying compatibility issues from user reviews, about 18.38% of the responses also concurred to using some kind of automated tools for identifying compatibility issues from user reviews.

Identifying compatibility requirements from reviews					
Q9. Total number of user reviews analyzed for each app release	Less than ten	44.1 %			
	10 − 50	28.7 %			
	51 − 100	15.4 %			
	101 − 200	7.4 %			
	201 − 500	2.2 %			
	501 − 1000	1.5 %			
	More than thousand	0.7 %			
Q10. Methods for identifying compatibility requirements from reviews	Manually	61.8 %			
	Ad hoc means	28.7 %			
	Only check low rating reviews	29.4 %			
	Automated tools	18.4 %			
	Do not read reviews	7.4 %			
Q11. How satisfied are you with the automated tools used for analysing reviews	1 (Very upset)	3.8 %			
	2	5.8 %			
	3	38.5 %			
	4	42.3 %			
	5 (Highly satisfied)	9.6 %			
Q12. Usefulness of an automated user review analysing tool	1 (Not useful)	5.9 %			
	2	5.1 %			
	3	22.1 %			
	4	39.7 %			
	5 (Very useful)	27.2 %			

Q13. Accuracy of users in identifying and reporting compatibility issues in reviews		< 20%	20 - 40%	40 - 60 %	> 80%	Not sure
	Beginner	21.4 %	28.6 %	7.1 %	7.1 %	35.7 %
	Junior	38.5 %	25.6 %	21.8 %	3.8 %	10.3 %
	Senior	29.5 %	27.3 %	22.7 %	11.4 %	9.1 %

Q14. Compared to other sources, compatibility issues from reviews are fixed _____		Much slower	Slower	Same time	Faster	Much faster
	Beginner	22.2 %	33.3 %	22.2 %	22.2 %	0 %
	Junior	6.8 %	31.1 %	36.5 %	18.9 %	6.8 %
	Senior	7.1 %	31.0 %	40.5 %	14.3 %	7.1 %

Table 4.3: Section III – Questions related to identifying app's incompatibilities from reviews

The following question in this section was an optional one only for those who used some kind of automated tool to extract compatibility issues reported in user reviews and we wanted to know how happy they have

been with the tool. To this question we got a mixed response; among the 38.24% of the developers who answered this question, 48.08% of the developers were not satisfied with the tool while the other 51.92% seemed happy. Again considering the experience of the developers who answered this question, 66.67% of the Senior developers and 45.16% of the Junior developers were unhappy with the automated tools. This suggests that the existing tools are not effective in this regard.

The next question was related to whether the developers felt that a tool that would automatically classify user reviews and analyze compatibility issues be helpful. 88.97% of the developers felt that the tool will be useful with 66.91% rating the usefulness as high or very high. This demonstrated the need for a new tool that would help developers with identifying compatibility issues from user reviews automatically.

The fifth question in this section was impressionistic: in the opinion of the developers, what percentage of the compatibility issues reported in the users' reviews were indeed compatibility issues. We wanted to evaluate, according to the developers, how correct the users are while reporting compatibility issues. 33.82% of the developers felt that users are correct only less than 20% of the times (a majority of them were Junior developers), while another 26.47% of the developers believed that the users are correct in between 20 to 40 percent of the times. Only 6.62% of all the developers (11.36% of Senior developers) felt that user reviews describe compatibility issues correctly in more than 80% of the cases.

The final question in this section was a comparative one. We wanted to compare the urgency in fixing compatibility issues identified from user reviews to those identified from any other sources. 39.2% of the developers felt that compatibility issues identified in reviews are fixed slower as compared to other sources, while another 24% responded the opposite - they felt that issues mentioned in reviews are fixed at a faster rate. The remaining 36.8% had a neutral opinion; they believed that issues identified in reviews take the same time to be fixed as those identified from other sources.

Key findings

Here are some of our key findings from this section of the survey

- App developers often are not able to efficiently extract requirements from reviews; most of them use manual techniques which is not possible for a large number of reviews.

- The currently available tools for extracting compatibility requirements from reviews are ineffective. As such, developers are keen on having an automated tool that would help them identify compatibility requirements from reviews.

- Developers do not associate high accuracy on the compatibility issues reported by users.

- Most of the developers opined that the compatibility issues reported in reviews are resolved at a slower rate as compared to other sources. This can be directly attributed to the fact that identifying issues from reviews is difficult for the developers.

4.3 Statistical tests

In order to identify the correlation of the responses of the developers to their respective experiences, we conducted some statistical tests [81]. As the responses consisted of different types of data (numerical, categorical, and ordinal), we have used following statistical measures:

- **Cramér's V**: Cramér's V (denoted as φ_c) gives a measure of association between two nominal variables; it gives a value between zero and one [75].

- **Correlation Ratio** (CR): A measure to calculate the correlation between two variables that have mixed data types; i.e., one of the variables is categorical type and the other is of the numerical data type [74, 67].

- **Kruskal-Wallis** (KWH): It is a non-parametric statistical significance test used for comparing two or more independent samples of equal or different sample sizes [77].

In order to run these tests, we have used the statistical module under `SciPy` Python library [2]. Using these tests, we have identified the following relations:

1. This is a high correlation ($\varphi_c = 0.31$) between the experience of the developers and the importance given to reviews as a source for identifying compatibility requirements.

2. There is a certain degree of correlation (CR value calculated to be 0.14) between the developers' experiences and the importance attributed to identifying and fixing compatibility issues in mobile apps.

3. There is also a dependence between the experience of the developers and their perceptions about the accuracy of the users in reporting compatibility issues ($\varphi_c = 0.12$) and that of the urgency in fixing issues reported in reviews ($\varphi_c = 0.18$).

4. There is a high dependence between the developers' experience and the importance associated with the need for an automated tool for identifying and analysing compatibility requirements. Using the Kruskal-Willis non-parametric statistical significance test, we could reject the null hypothesis ($p = 0.0014$) that the need for a automated tool for extracting and analyzing compatibility requirements is independent of the developers' experience.

4.4 Summary

In this chapter, we have discussed the survey that was conducted with mobile application developers. This survey has been a key component of this thesis as it has not only helped us get a deeper understanding of the state-of-the-practice followed by developers related to mobile app's compatibility, but it has also served as a motivation for proceeding with this research work. From the survey, the need for a tool for analyzing compatibility requirements was evident; also, the need for certain features in the tool were also expressed by some of the developers.

It was clear that although developers hold high esteem for user reviews and consider them as one of the main sources for compatibility requirements, they are not able to effectively use it. By successfully executing this survey, we have answered one of the research questions (RQ5) and this has been a vital piece of this thesis. This survey has also helped us design a tool (discussed in the next chapter) to assist app developers.

Lastly, we would like to thank all the app developers who have participated in this survey and shared their valuable feedback; it would not have been possible to conduct this survey without their support.

Chapter 5

Tool Support

5.1   Introduction

In this chapter, the design and implementation of the proposed ACOCUR approach as an integrated tool has been described. This tool can support app developers to automatically identify compatibility requirements and analyze their responsiveness to these requirements; this answers our last research question (RQ6).

The tool automates the processes such as data collection, data cleaning and preprocessing, data classification, compatibility types extraction, and finally summarizing and mapping data from two separate sources. Tool support allows users (managers or app developers) to evaluate the proposed approach and methodologies in real-life situations. In the context of software development management, providing tool support with the integration of the different platforms is more meaningful and effective. Although ACOCUR currently integrates two data sources (GitHub and Google Play), it can be enhanced to be integrated with other platforms as well. As discussed in the earlier chapters, existing approaches have not sufficiently dealt with identifying how much effort the app developers have already put into certain quality aspects or measure their responsiveness to user demands.

By analyzing some of the existing approaches used for extracting requirements from user reviews and the inputs from app developers (as part of the survey discussed in the previous chapter), we have developed a comprehensive compatibility analysis tool to support software development and management. The tool has the following functionalities:

- F1: The tool integrates the existing platforms so that required data (i.e. user reviews and developers'

commits) for analyzing compatibility can be collected automatically.

- F2: The tool pre-processes and cleans the incoming data so that the accuracy of the process is not affected.

- F3: The tool should provides the option to proceed with pre-trained models so that the users do not need to train it separately. At the same time, there is the provision to run the tool with customized models that have been trained with new data.

- F4: For identifying the different types of compatibility issues, the tool again provides both the options – use the existing types, or include additional types and train the model accordingly.

- F5: The tool summarizes the result in a simplified format for easy analysis.

The compatibility analysis tool presented in this chapter addresses these requirements. The tool is designed and implemented as a desktop application on top of GitHub and Google Play following the guideline presented in [15]. In the following sections, the development platform, architecture, and system modules of the tool are explained.

5.2 Architecture of the tool

The tool has been implemented as a desktop application on top of the existing project management and tracking tool (GitHub) and mobile app store (Google Play). GitHub is one of the most popular distributed project management and source code control systems. Key features of GitHub includes code versioning and tracking, collaboration, integrated issue and bug tracking, code review, and dashboard for monitoring the development activities. Data related to various aspects of project and quality management can be retrieved from GitHub using different APIs that are available for quick and easy access. On the other hand, Google Play is a popular mobile app store from Android applications. Not only does it provide the platform to host or download Android apps, but it also provides a wealth of information and data on different aspects related to the apps. With the continual increase in the popularity of Android apps, Google Play is now the one-stop source for all app-related data. Other than the app's description that explains the features of the app, other relevant details like the last update date, number of installs, app's rating, current version, app's requirements, content rating, permissions, and related information is directly available. Google Play also contains users' reviews about the app which is a direct feedback channel to the app developers and has proved to be of immense use for making the apps better.
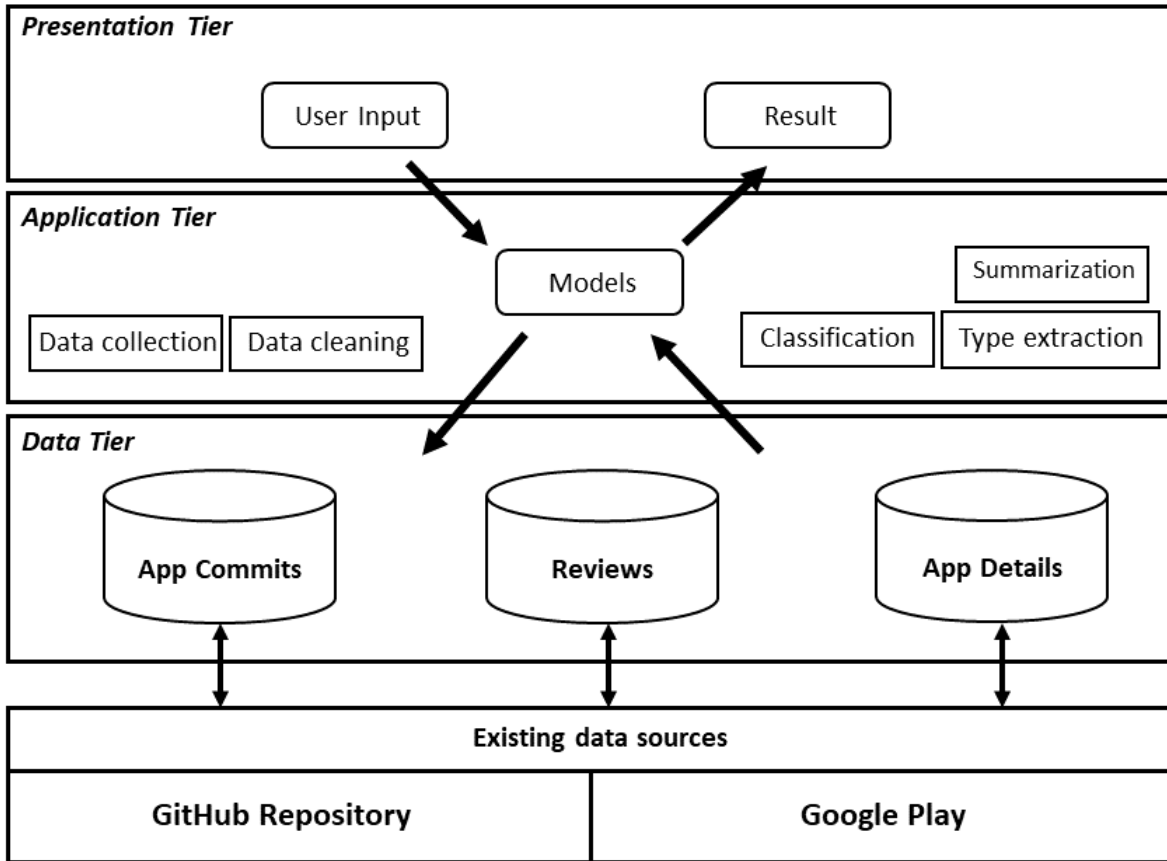
Figure 5.1: Architecture of ACOCUR Tool

Both GitHub and Google Play provides a REST API to support integration with other tools and technologies. As shown in Figure 5.1, the data tier of the tool access the live data from GitHub and Google Play via API and custom libraries and store them in the local database for analyzing compatibility. The application tier of the tool contains the logic and services required to support compatibility extraction and analysis for commit messages and user reviews. It uses the data available in the data tier to implement the services. Key services include data collection, data pre-processing and cleansing, text classification, compatibility type extraction, and summarization. The presentation tier of the tool contains various GUI components of the tool which invoke the services hosted in the application tier. It also allows the stakeholders to interact with the tool via the interface.

The three-tier architecture of the tool is shown in Figure 5.1. The tier-based architecture provides benefits such as reusability, flexibility, manageability, maintainability, and scalability [63]. For instance, the integration of another project management tool and app store requires modification only in the data tier. Similarly, new services can be added easily in the application tier of the tool. Thus, the choice of three-tier architecture was appropriate for developing the tool.

5.3   Development platform

The tool is developed using Python 3 (version 3.6) which is an open-source, interpreted, high-level, and general-purpose programming language. The tool has been developed as a three-tier architecture as explained in Figure 5.1. It internally uses Python's open-source libraries for different types of operations. An overview of the tools and technologies used for managing, developing, and deployment of the tool are shown in Table 5.1. As described in Table 5.1, the source code of the tool is hosted in GitHub repository [1] as open source.

Tools and Libraries	Version	Usage
Python	3.6	Used as development platform
GitHub	–	For managing and hosting source code
google_play_scraper	0.0.2.2	Python library for Google Play API
GitPython	3.1.3	Python library for git repository and logs
pickle	default	Python library for saving or loading models
imbalanced-learn	0.6.2	Python library for unbalanced training set
contractions	0.0.25	Python library for text processing
nltk	3.4.5	Python library for natural language processing
scikit-learn	0.22.2.post1	Python library for various NLP operations
pandas	0.25.1	Python library for mathematical operations
numpy	1.19.0	Python library for mathematical operations

Table 5.1: Components of the ACOCUR tool

5.4   Using the tool

The workflow of the tool (Figure 5.2) has been implemented by following the ACOCUR approach that has been described in Chapter 3. To identify compatibility requirements and analyze them, the development team needs to complete the following steps.

5.4.1   Create input data

The first step (Figure 5.3) of the tool is to create the input data. Since ACOCUR identifies compatibility requirements from user reviews (from Google Play) and compares those to developers' commits (in GitHub), the input data should include details of the app's package name and the GitHub repository name. The tool

---

[1]https://github.com/debmukherj83/ACOCUR
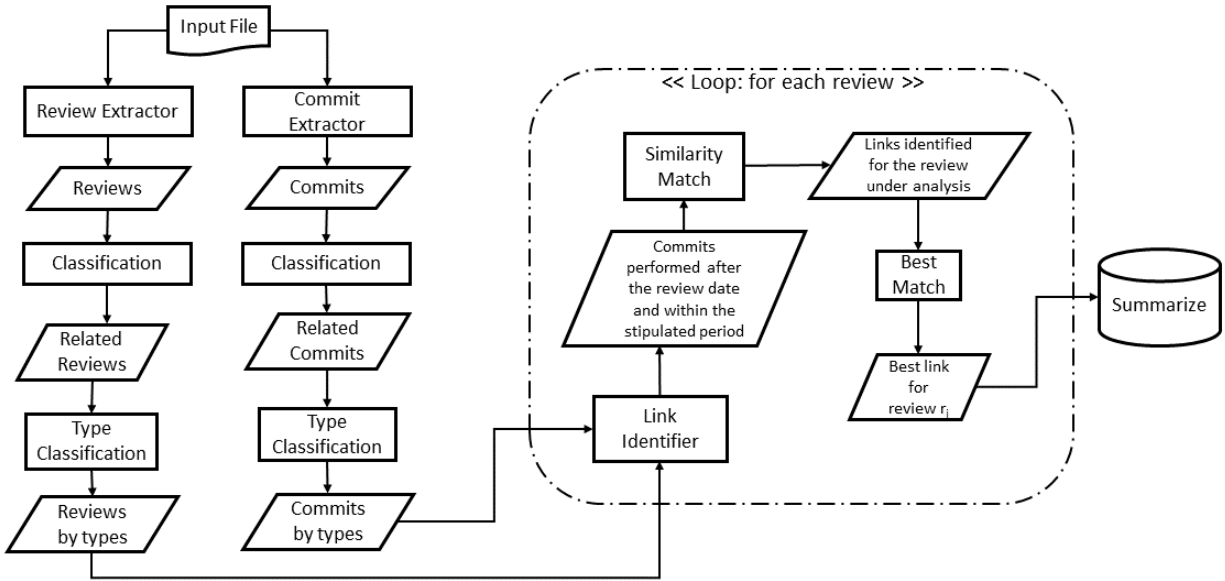
Figure 5.2: Workflow of ACOCUR Tool
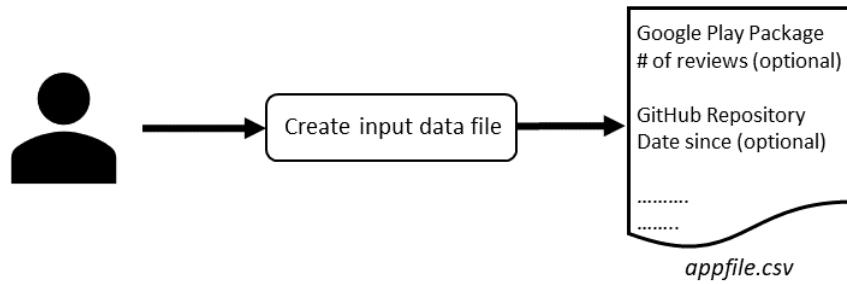
Figure 5.3: Input Data creation for analysis

accepts .csv file as the input data source. The input file (named as *'appfile.csv'* must be placed under the *'Data'* Folder to be accessed by the tool.

5.4.2 Choose analysis type

The tool prompts the user to decide the nature of the analysis that he/she wants to perform (Figure 5.4). There are three options to choose from:

- Option 1: Analyze reviews only

- Option 2: Analyze commits only

- Option 3: Analyze both reviews and commits

Option 1 can be used if the user is only interested in identifying compatibility requirements from the user

52
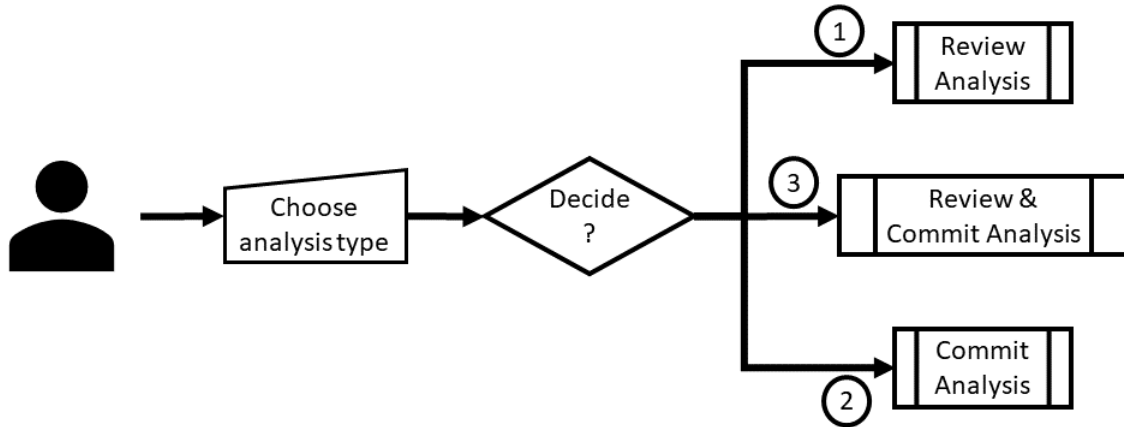
Figure 5.4: What do you want to analyze today?

reviews. In that case, the input data should contain the Google Play package name. The tool does not need the GitHub Repository details. Option 2 is used to identify compatibility fixes from the GitHub repository only; the tool does not look into the user reviews to identify compatibility requirements. In this case, the input data should contain the details of the GitHub repository. Option 3 performs the comprehensive analysis that takes into account both the review and the commits.

5.4.3 Determine classification option
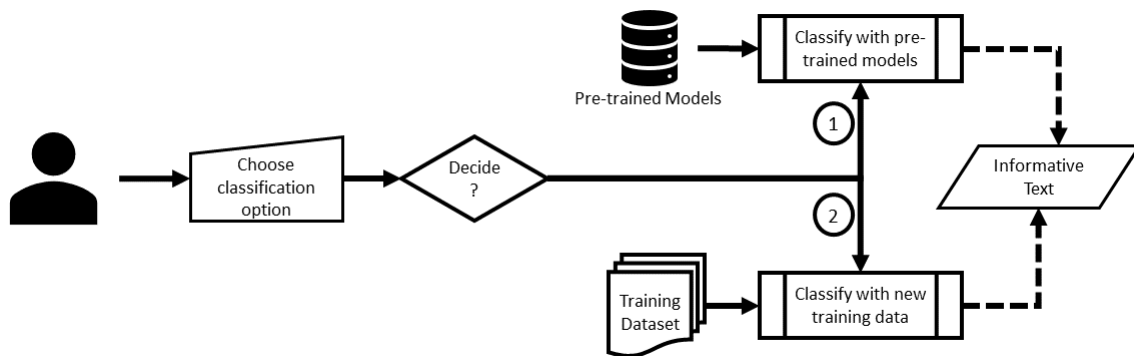
Figure 5.5: Choose classification method

Based on the choice in the previous step, the tool automatically connects the necessary data sources and accumulates the data (reviews and/or commits). This data is pre-processed and cleaned and made ready for further processing. The next step (Figure 5.5) for the tool is text classification where it identifies and segregates the related and informative texts from the uninformative ones. The tool has already been equipped to perform this step automatically with pre-trained models. However, the user also has the option to re-train the tool with custom training data. As such, in this step the user has the option to choose between

two options:

- Option 1: Identify informative text using pre-trained models

- Option 2: Identify informative text after training with custom data

If the user chooses Option 1, the tool automatically classifies the text and identifies the related data. However, if the user chooses Option 2, the tool will again be trained with new sets of training data; as such the new training data set files should be made available to the tool. The training file(s) (named as *'Commit_TrainingSet.csv'* and/or *'Review_TrainingSet.csv'* must be placed under the *'Data'* Folder to be accessed by the tool).

5.4.4 Choose type extraction option
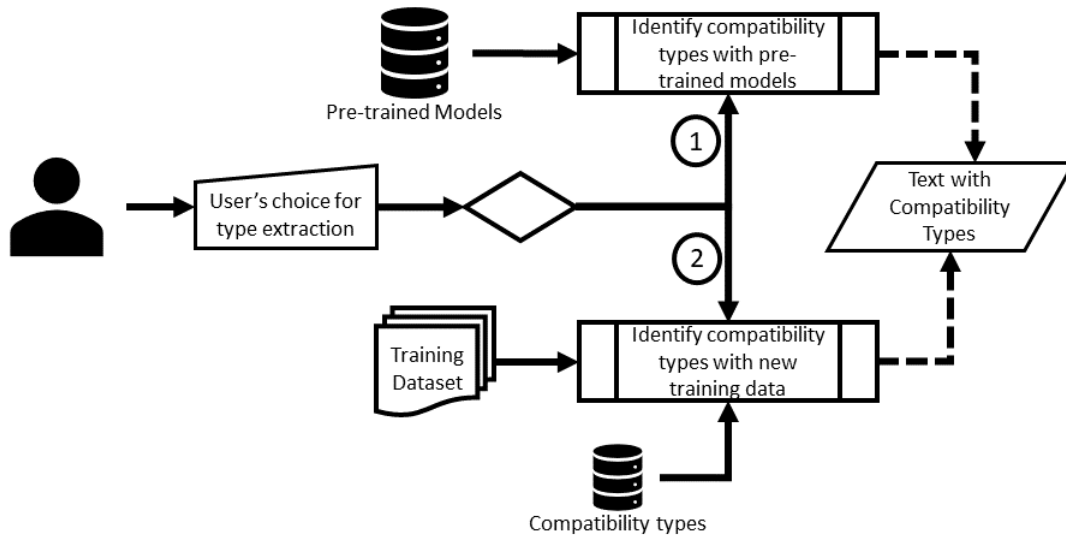
Figure 5.6: Choose type extraction method

The tool identifies the informative texts in the previous step. In this step, the tool further classifies the reviews and commits according to their corresponding compatibility types (Figure 5.6). Just like the previous step, the user again has two options to choose from:

- Option 1: Extract types using pre-trained models

- Option 2: Extract types using new training data

If the user chooses Option 1, the tool automatically identifies the compatibility types from the text based on the pre-trained models. However, if the user chooses Option 2, the tool will again be trained with new

sets of training data; as such the new training data set should be made available to the tool. The training file(s) (named as *'Commit-Type-TS.cs'* and/or *'Review-Type-TS.csv'* must be placed under the *'Data'* folder to be accessed by the tool). The tool also provides the option to update the pre-defined compatibility types to include new compatibility types as deemed applicable by the users.

5.4.5 Summarization and final result
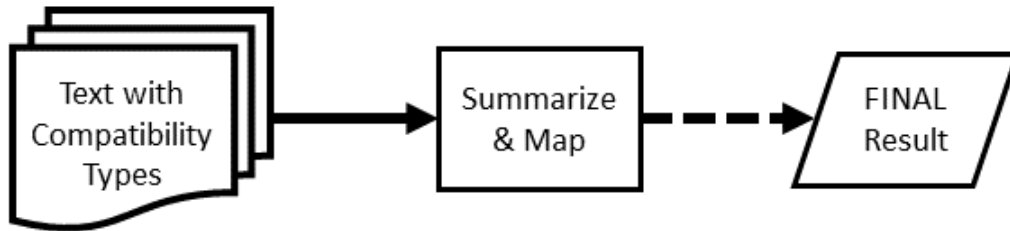
Figure 5.7: Result summarization process

In the final step (Figure 5.7), the ACOCUR tool summarizes the data and present the result in a readable format for the users. The output is presented as an excel file that has a summary of the various compatibility types identified. This file is made available under the folder *'FINAL'*.

The output file in dependent of the nature of analysis that has been performed. If the user has chosen option 1 or 2 during the *choose analysis type* step, the result file has the list of all the compatibility records identified along with a summary describing the different compatibility types amongst the identified records. On the contrary, if the user had chosen option 3 (analyze both reviews and commits), the result file combines the outputs for options 1 and 2 and also links the commits to the reviews.

5.5 High level use-cases

To better illustrate the capabilities of the tool, two high-level use cases (see Figure 5.8) are described from the project manager or app developer's perspective in the following sub-sections.

Use-case 1: Identify compatibility requirements reported in a particular app version

In this use case, the app developers can monitor and analyze the compatibility requirements from reviews using the ACOCUR tool for particular app versions. Based on the pre-trained models or using new training data as discussed in the earlier section, user reviews are classified to identify compatibility requirements. The
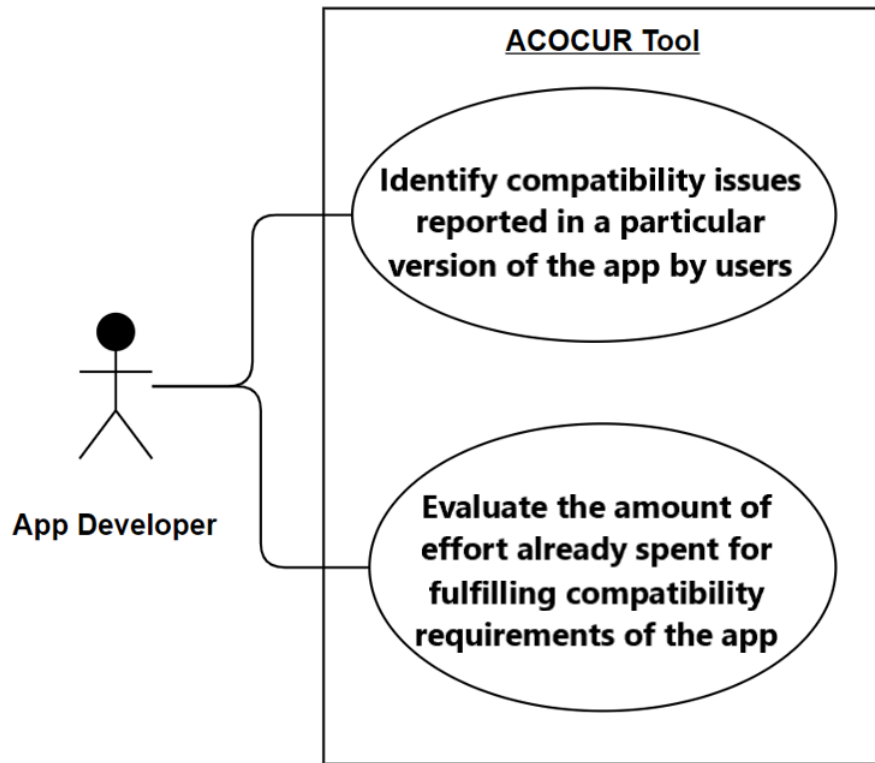
Figure 5.8: High level use cases of ACOCUR Tool

requirements are further categorized under different types. This allows the project manager or developers to know which compatibility requirements have been raised by app users that need to be fixed. This will serve as a ready input for the development team as the set of requirements to act upon.

Use case 2: Evaluate the amount of effort already spent for addressing compatibility requirements

In this use case, the stakeholders can monitor and analyze the various compatibility requirements that have already been completed in the previous releases of the app. Based on the pre-trained models or using new training data as discussed in the earlier section, the commit messages are classified to identify compatibility fixes. The fixes are further categorized under different types. This allows the project manager or developers to estimate how much effort has been dedicated to meet the quality requirements. This is essential as, based on the results, the team can decide whether to pursue additional non-functional requirements. The comparison of this result with the requirements from use case 1 will help the team decide on their future

actions.

5.6   Summary

The tool is developed as a desktop application using the Python 3.6 frameworks. It can be used for identifying compatibility requirements for the mobile Android applications which use GitHub as a source code repository and Google Play as the app store. In this chapter, the requirements of the tool, overall system architecture, process workflow, and two high-level use cases of the tool have been presented. Specifically, analyzing compatibility fixes done by the development team and mapping those to the compatibility requirements in user reviews are unique features of the tool. Overall the tool helps the development team with automated, fast, and continuous non-functional requirement extraction and at the same time measurement of effort in addressing these non-functional requirements.

Just like the ACOCUR approach that can be applied to other non-functional requirements, even this tool can be extended for other non-functional requirements. Specifically, the pre-trained models and the taxonomy have to be generated according to the new non-functional requirement. Since this tool has been built by implementing the ACOCUR approach, it helps us answer RQ6. We have successfully developed a tool that would support app developers to automatically identify compatibility requirements and simultaneously analyze their responsiveness to the requirements.

In the following chapters, the applicability of the tool and the empirical evaluation of the ACOCUR approach has been discussed.

Chapter 6

Data Collection and Initial Analysis

6.1   Data collection

For our analysis we have taken real-world open-source Android mobile applications to analyze compatibility related non-functional requirements. In order to properly evaluate the model, we require different sizes and forms of mobile apps. Open-source projects often come with other additional information which can be handy for other types of evaluation, if necessary.

In this research, we required two kinds of datasets for our study. The first set comprised mobile applications with their source code and version system, and the second set comprised users' reviews for the selected mobile apps.

In this section, we have described the steps for app selection and data collection.

6.1.1   Mobile app selection

To begin the mobile app search process, we started with F-Droid which is a popular open-source Android app repository. In F-Droid the apps are grouped under different categories and some of the apps are also listed under multiple categories.

To retrieve the comprehensive list of all the apps under F-Droid, we built a custom crawler that systematically mined the app details by performing the following steps:

1. **Get F-Droid categories**: The first step of the F-Droid crawling process involved retrieving the names and the url of all the categories so that each of them can be individually crawled.

2. **App list under each categories**: The next step of this activity involved crawling the individual category pages to gather the app name, package name, and the F-Droid address for each of the apps

under each category.

3. **App details**: Using the address obtained from the above step, the crawler then crawled the individual app's pages in F-Droid and obtained the source code address for each of the apps.

In F-Droid there are a total of 17 categories. Using our custom crawler, we mined the following details from F-Droid: app name, package name, the category under F-Droid, address of the app in F-Droid, and address of the source code repository for the apps. In total, we extracted details of 3,460 apps from F-Droid before removing duplicates. After eliminating duplicate apps (those listed under more than 1 category), we had a total of 3,026 distinct apps in F-Droid for our analysis.

The Figure 6.1 shows the process of app selection for this research.
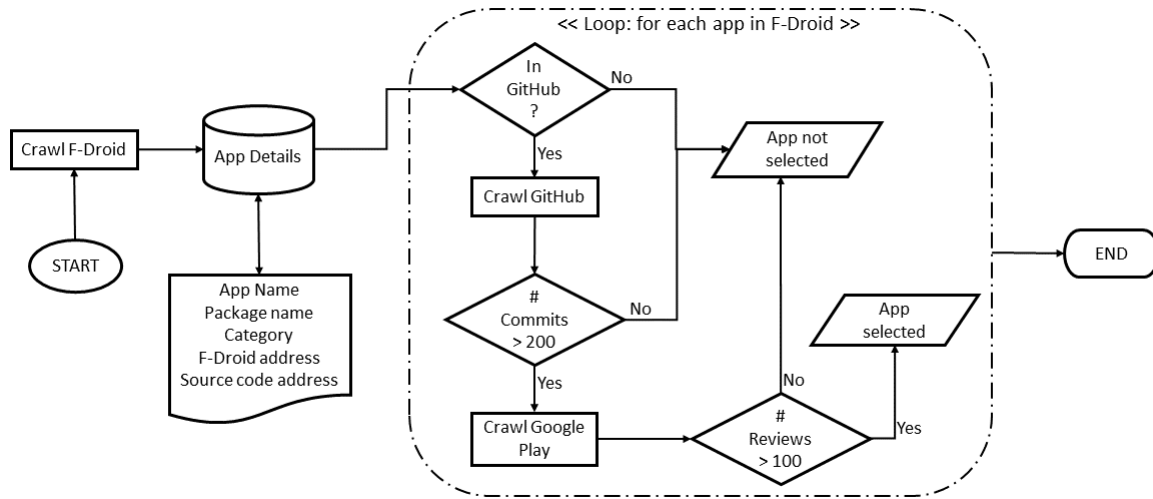
Figure 6.1: Process of mobile app selection

6.1.2 GitHub crawler

To unify the process of mining different repositories, we resorted to only one repository that hosted a majority of the apps. Since GitHub is one of the most popular repositories and it contained a majority of the apps from F-Droid, we decided to proceed with the GitHub repository for selecting apps for our research. After eliminating all apps whose source code was not under GitHub, we had a total of 2,314 apps remaining for consideration.

In order to scrape GitHub, we built another crawler using *git* Python library that is used to clone the GitHub repositories into our local system. Using this process we gathered the GitHub repositories for these 2,314 apps and collected app and release information from GitHub logs. We used this data to mine all the commits associated with the apps. In particular, we collected the following details for each of the commits:

commit message, commit date, number of lines added, number of lines deleted, and the number of files changed.

We collected a total of 17,93,306 commits along with the details from these 2,314 apps. We set a threshold of a minimum of 200 commits for each app to be considered for evaluation in our research. Using this benchmark, we eliminated 1,530 apps as each of them had less than 200 commits in total. As a result, we were left with 784 apps which had a total of 13,78,736 commits; this is the data that we have finally processed and analyzed in our research.

6.1.3 Scrape Google Play

We used another custom web crawler to gather the mobile applications' details and the user reviews from the Google Play. This crawler is built using *google_play_scraper* Python library which uses the Google Play API to connect to the Android app store and retrieve app details and user reviews.

For each app, we collected the following details from the Google Play: app category, score, developer details, app title, the number of reviews, the number of app installations, whether the app is editor's choice, app description, and content rating.

Since some of the apps have a large number of reviews, we restricted the number of reviews collected to a maximum of 20,000 for each app. As such, for apps with less than 20,000 reviews, all the user reviews were collected by our crawler. For the rest, the most recent 20,000 reviews were collected. For each review, we scraped the following details: review text, review date, review rating, and thumbs up count (number of times the review has been marked helpful).

Out of 784 apps, only nine apps had at least 20,000 reviews each (for these apps we gathered the latest 20,000 reviews). However, some apps had a small number of user reviews. We set a minimum threshold for the number of reviews as 100 and only accepted those apps for our analysis that had a minimum of 100 reviews.

With this criterion, 476 apps were further eliminated; we had 308 apps remaining for analyzing. These 308 apps accounted for 7,39,421 reviews in total; out of a total of 13,78,736 commits considered in our study, 8,77,980 commits belonged to these 308 apps.

6.1.4 Data used for empirical evaluation

It is important to note that the initial phase of our research (analyzing commit messages) has been conducted using the 784 apps, the second phase of the research (analyzing user reviews and evaluating the degree of alignment between reviews and commits) had been performed using the 308 apps. These 308 apps are
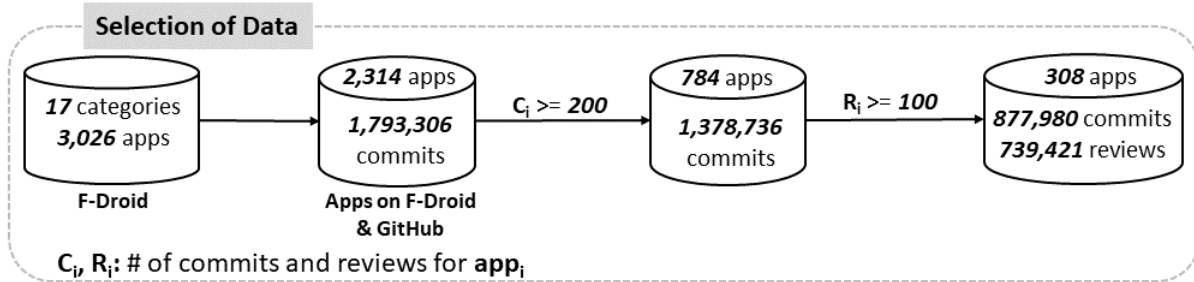
Figure 6.2: Process of data collection and count of apps, commits, and reviews

distributed across 23 app categories from the Google Play (Table A.1 contains the list of all the 23 categories) — the top five categories based on the number of apps per category are *Tools*, *Productivity*, *Communication*, *Games*, and *Books & Reference*. The apps have been listed under Table B.1. Figure 7.1 shows the number of apps and the count of the commits and reviews in the different phases of the data collection process.

Figure 6.3 and 6.4 are the examples of some sample compatibility related commit messages and reviews extracted from GitHub and Google Play respectively.

6.2 Initial Analysis

In this section we shall discuss the results of some of our initial analysis.

6.2.1 Keyword search results

As discussed under Section 3.3, we performed an initial keyword search to identify the compatibility related text (both commit messages and reviews) from the large pool of data. This step had been introduced in our methodology to narrow down the potential results set. Since the set of keywords that are used to identify compatibility related text had been chosen after rigorous manual search iterations and a thorough literature review, we are optimistic that the search result has included all potential compatibility related commits and reviews. Some of the keywords are general and they can be used in a wide variety of contexts; as such the chances that the result set has a large number of false-positives is considerably more as compared to the scenario where it might have missed some valid results.

The keyword search on the commit messages resulted in 2,83,754 commits selected as compatibility related from the total pool of 13,78,736 commits. The keyword-count had the maximum value of nine; this indicated that there was at least one commit message that contained nine of the keywords chosen to describe the compatibility-related text. On closely analyzing the result set, it was evident that the majority of the
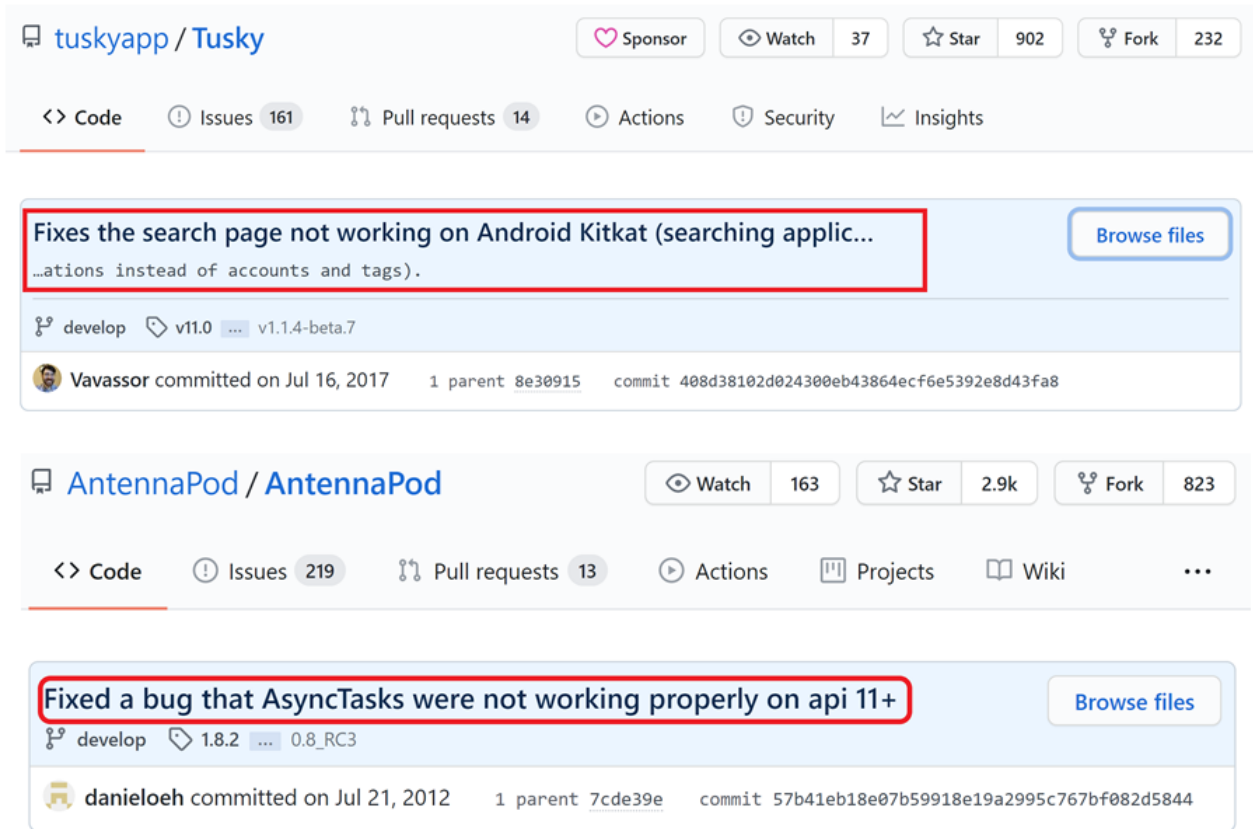
Figure 6.3: Some sample compatibility related commit messages

commit messages selected by this keyword search had the keyword-count of one or two; only a few had the keyword-count of three or more. Also, the number of commits with higher values of keyword-count steadily decreased. For the next phase of this research to identify the compatibility related commits accurately, we used supervised learning – the training set of this process was built using a subset of the commits identified from the keyword search. Table 6.1 has the count of the number of commits for each keyword-count and those used in the training set.

We performed a similar keyword search on the reviews as well. Out of the total of 7,39,421 reviews considered for study in this research, the keyword search selected 1,59,350 reviews which contained at least one of the keywords. The maximum value for keyword-count for the reviews was seven. However, just like the commits, a majority of the reviews coincided with a keyword-count of one or two. Table 6.2 has the count of the number of reviews corresponding to each keyword-count and those used in the training set.
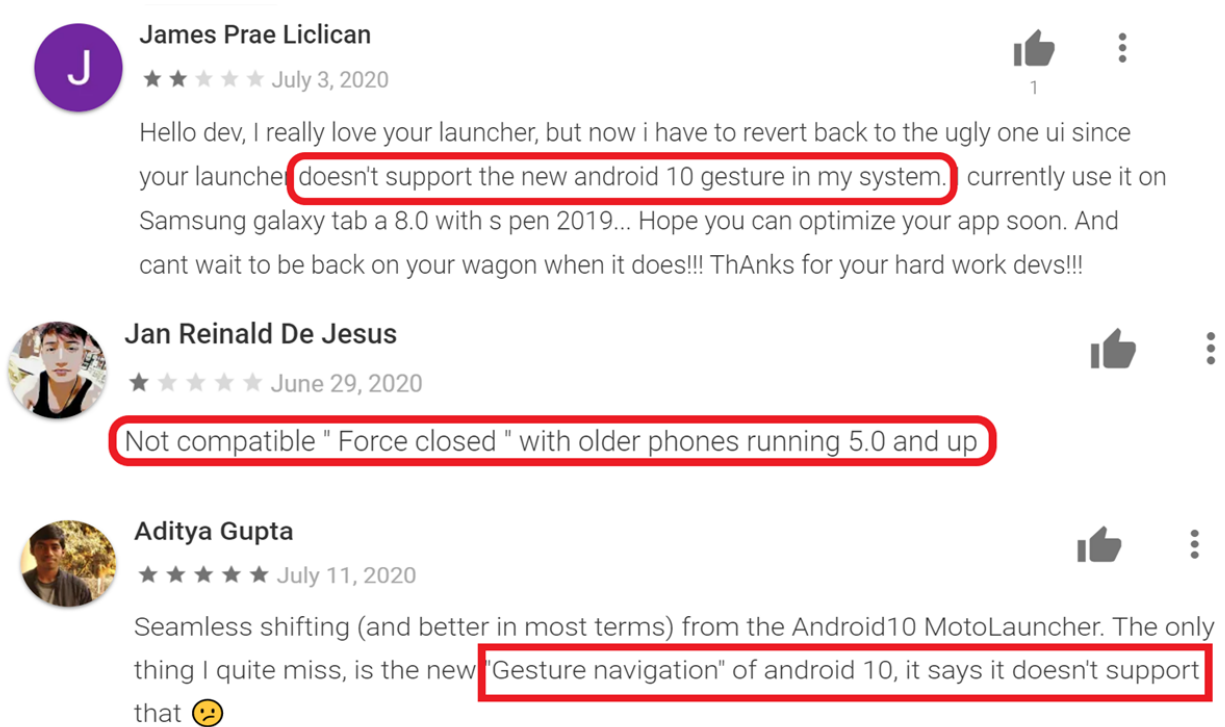
Figure 6.4: Some sample compatibility related user reviews

Commits		
Keyword-Count	**# Matched**	**# Selected in Training set**
1	2,51,004	5,811
2	29,397	856
3	2,980	2,980
4	313	313
5	45	45
6	11	11
7	3	3
8	0	–
9	1	1
SUM	2,83,754	10,020

Table 6.1: Keyword-count match for commit messages

6.2.2 Comparison of different text embedding techniques

Choosing an appropriate Word Embedding technique can be crucial for any machine learning model. In this research, we have used and evaluated four classifiers for classifying commit messages and user reviews.

	Reviews	
Keyword-Count	**# Matched**	**# Selected in Training set**
1	1,29,435	2571
2	24,423	937
3	4,484	4,484
4	829	829
5	142	142
6	31	31
7	6	6
SUM	1,59,350	9,000

Table 6.2: Keyword-count match for user reviews

However, before we can even proceed to run the classifiers, it is important to choose an appropriate word embedding technique. The performance of the machine learners can vary considerably depending on the vectorization technique.

In this research, we have evaluated the following vectorization techniques:

- TF-IDF

- Word2Vec (Avg & TFIDF)

- Doc2Vec

- GloVe (Avg & TFIDF)

To evaluate these word embedding techniques, it was essential to train a classifier with each of these techniques and then determine the accuracy of the model. In our research, we had the option to use any of the four classifiers that we had eventually used for the actual classification activity. We used the Logistic Regression (LR) classifier for this step as we had found in an earlier study that LR performs well with classification of commits and reviews. However, it is worth mentioning that we could have used the other three classifiers as well.

The Table 6.3 and Figure 6.5 shows the performance of the different embedding techniques using the Logistic Regression model on the commit messages. It is evident that the overall performance of all the techniques was very similar; however, TF-IDF performed the best among all the other techniques for each of

Commit Embedding Techniques				
Embedding Techniques	**Precision**	**Recall**	**Accuracy**	**F1 Score**
TF-IDF	88.54	88.22	88.22	88.19
Word2Vec (Avg)	85.25	85.21	85.21	85.2
Word2Vec (TFIDF)	80.65	80.57	80.57	80.56
Doc2Vec	84.02	84	84	84
GloVe (Avg)	85.45	85.41	85.41	85.41
GloVe (TFIDF)	80.26	80.19	80.19	80.18

Table 6.3: Performance of embedding techniques for commit messages
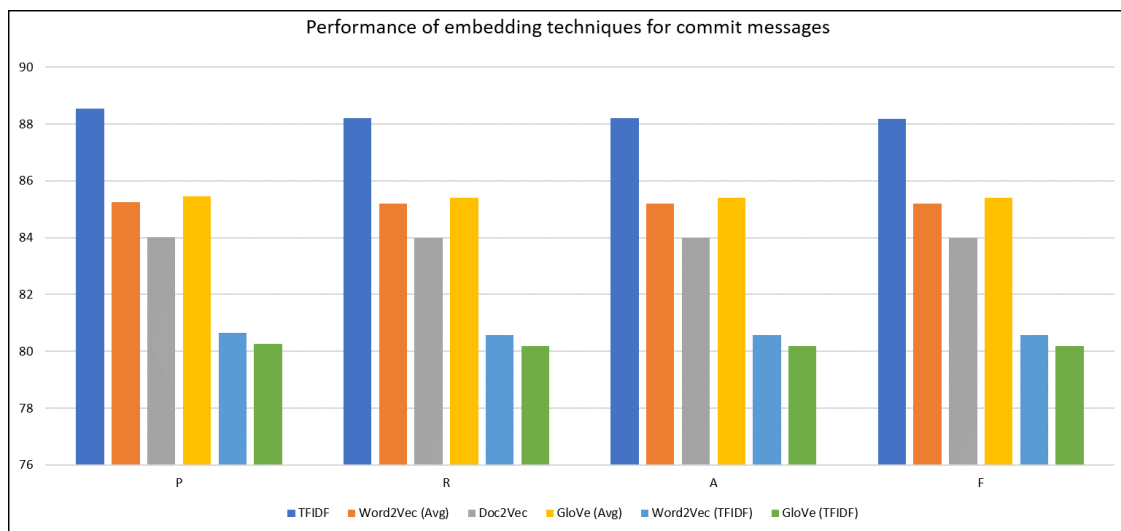
Figure 6.5: Performance of embedding techniques for commit messages

the four performance metrics. So, we have selected TF-IDF as the text vectorization technique for commit messages in this research.

We performed a similar evaluation for the review messages as well. Table 6.4 and Figure 6.6 show the performance of the different text embedding techniques using the LR classifier on the reviews. We observed that even for reviews, TF-IDF vectorization performed considerably better than the other techniques on all the performance measures. As a result, we have chosen TF-IDF as the ideal option for all further text vectorization activities.

Review Embedding Techniques				
Embedding Techniques	Precision	Recall	Accuracy	F1 Score
TF-IDF	90.3	89.9	89.9	89.88
Word2Vec (Avg)	80.56	80.52	80.52	80.51
Word2Vec (TFIDF)	76.2	76.18	76.18	76.17
Doc2Vec	81.54	81.43	81.43	81.41
GloVe (Avg)	80.14	80.09	80.09	80.09
GloVe (TFIDF)	76.81	76.77	76.77	76.77

Table 6.4: Performance of embedding techniques for user reviews
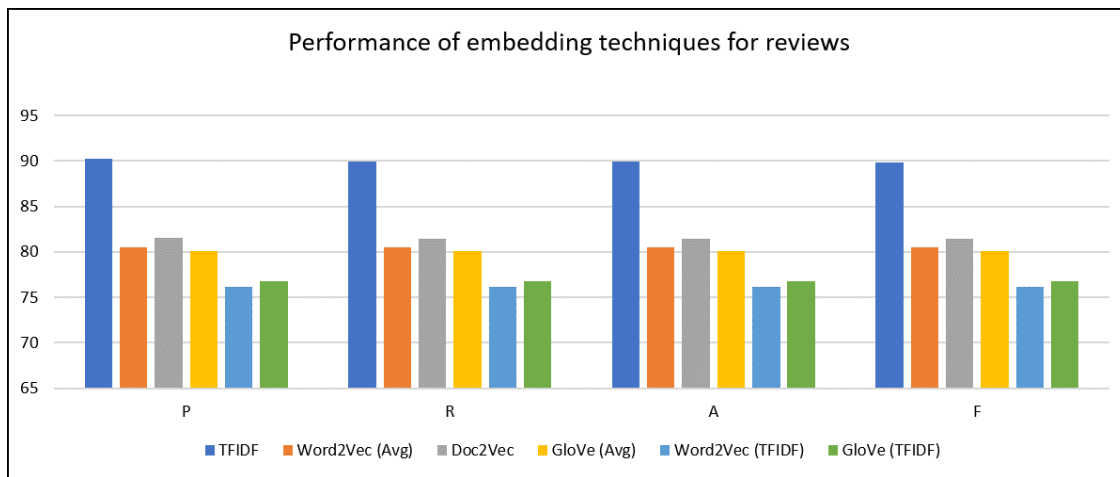
Figure 6.6: Performance of embedding techniques for commit messages

6.2.3 Evaluation of classifiers

In the previous section, we had observed that TF-IDF vectorization produced the best results among the different vectorization techniques. In this section, we have discussed the performance of the different classifiers. As mentioned in Section 3.4.4, we have evaluated four ML classifiers in this research: Naive Bayes, Logistic Regression, Support Vector Machine, and Random Forest. For each of these classifiers, the ideal combination of hyper-parameters was chosen using the GridSearchCV method as described in the earlier chapters. Table 6.5 shows the different combinations of hyper-parameter values supplied to the classifiers and the best parameter combinations identified for both commit messages and reviews.

With the chosen hyper-parameter values, we ran the four classifiers. To evaluate the performance of the classifiers, we have used 10-fold cross-validation (as described under Section 3.4.5) and calculated the four performance metrics (precision, recall, accuracy, and f1-score). In Table 6.6, the average values of these met-

Classifier	Parameter Values	Best Parameter (Commits)	Best Parameter (Reviews)
SVM	C : 0.1, 1, 10, 100 γ : 1, 0.1, 0.01 kernel: 'rbf', 'poly', 'sigmoid', 'linear'	C : 10 γ: 1 kernel : 'poly'	C : 100 γ: 1 kernel : 'poly'
LR	C: 0.001, 0.01, 0.1, 1, 10, 100, 1000	C: 100	C: 100
RF	max_depth: 80, 90, 100, 110 max_features : 2, 3 min_samples_leaf : 3, 4, 5 min_samples_split : 8, 10, 12 n_estimators : 100, 200, 300, 1000	max_depth : 110 max_features : 3 min_samples_leaf : 3 min_samples_split : 10 n_estimators : 1000	max_depth : 90 max_features : 3 min_samples_leaf : 3 min_samples_split : 8 n_estimators : 1000

Table 6.5: Classifier hyper-parameters tuning

rics for the four classifiers after classifying commit messages have been listed; the number in the parenthesis is the measure of their standard deviations (σ).

Commit Messages				
Classifier	Precision	Recall	Accuracy	F1-score
SVM	94.86 (0.71)	94.84 (0.71)	94.84 (0.71)	94.84 (0.71)
LR	91.24 (0.79)	91.12 (0.78)	91.12 (0.78)	91.11 (0.78)
RF	78.76 (2.57)	77.49 (2.74)	77.49 (2.74)	77.23 (2.83)
NB	85.33 (0.98)	83.97 (0.91)	83.97 (0.91)	83.81 (0.92)

Table 6.6: Performance of classifiers on Commit messages

As evident from the above table, the performance of the Support Vector Machine (SVM) classifier was the best among all the four classifiers. SVM had obtained the highest score on each of the four metrics. These values were obtained with the following hyper-parameters: $\mathbf{C = 10}, \gamma = \mathbf{1}$, and kernel = 'poly'. Other than SVM, the performance of the Logistic Regression (LR) classifier was also very good on all the measures. The Random Forest (RF) classifier had obtained the worst results in this case. Figure 6.7 shows the violin plot for precision, recall, accuracy, and f1-score obtained from 10-fold cross-validation for the four classifiers. It can be observed that the performance of SVM was better than the other classifiers for identifying compatibility aspects from commit messages.
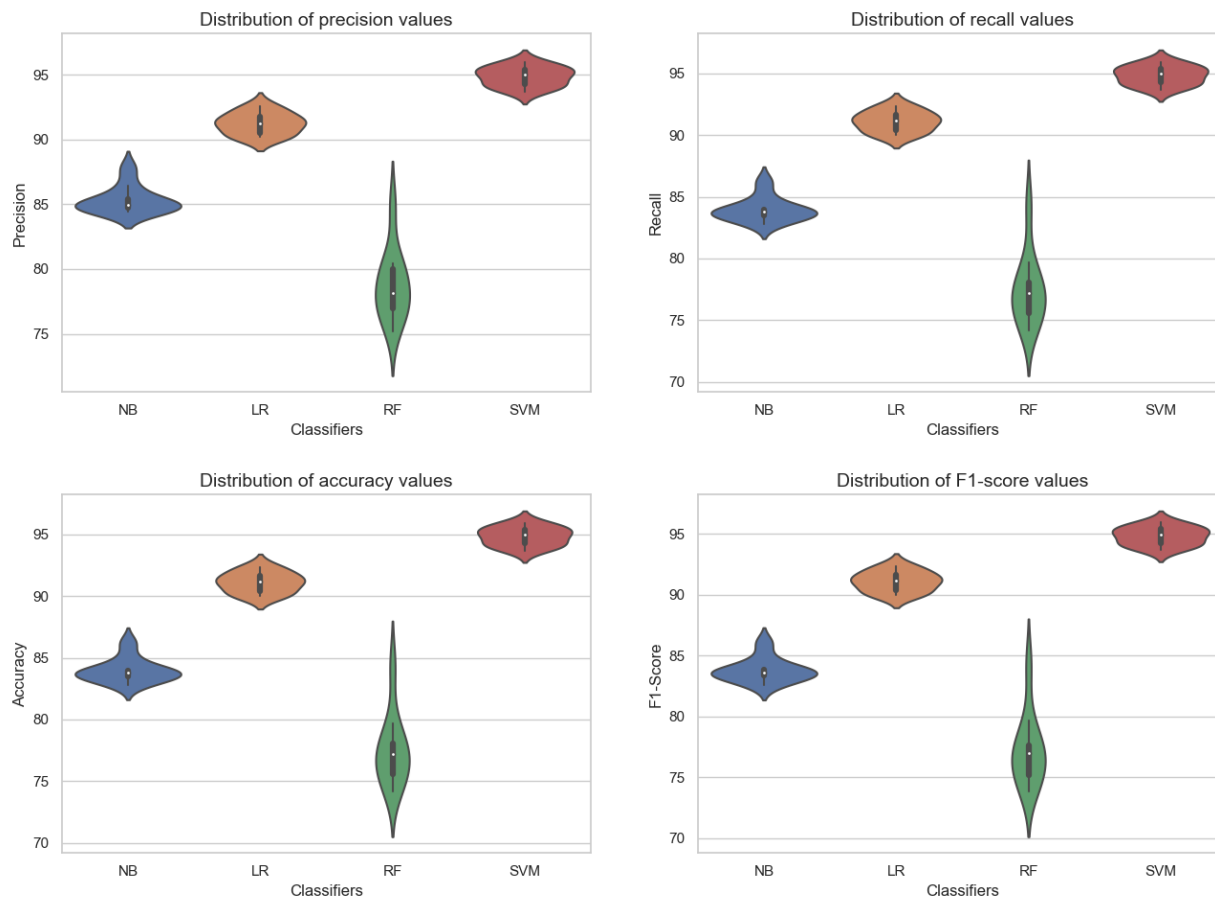
Figure 6.7: Violin plot of performance metrics for the four classifiers on commit messages

	User reviews			
Classifier	**Precision**	**Recall**	**Accuracy**	**F1-score**
SVM	94.33 (0.70)	94.31 (0.70)	94.31 (0.70)	94.31 (0.70)
LR	89.11 (0.64)	88.88 (0.65)	88.88 (0.65)	88.86 (0.65)
RF	81.06 (0.77)	80.88 (0.74)	80.88 (0.74)	80.86 (0.75)
NB	84.18 (0.63)	83.33 (0.56)	83.33 (0.56)	83.23 (0.56)

Table 6.7: Performance of classifiers on user reviews

We used a similar approach for the user reviews as well; we evaluated the performance of the four classifiers to identifying compatibility issues raised in user reviews. Table 6.7 shows the efficiency of the classifier models based on the performance metrics discussed earlier. Again, the values in the table describe the mean of the results obtained from the 10-fold cross-validation and the numbers in parenthesis denote
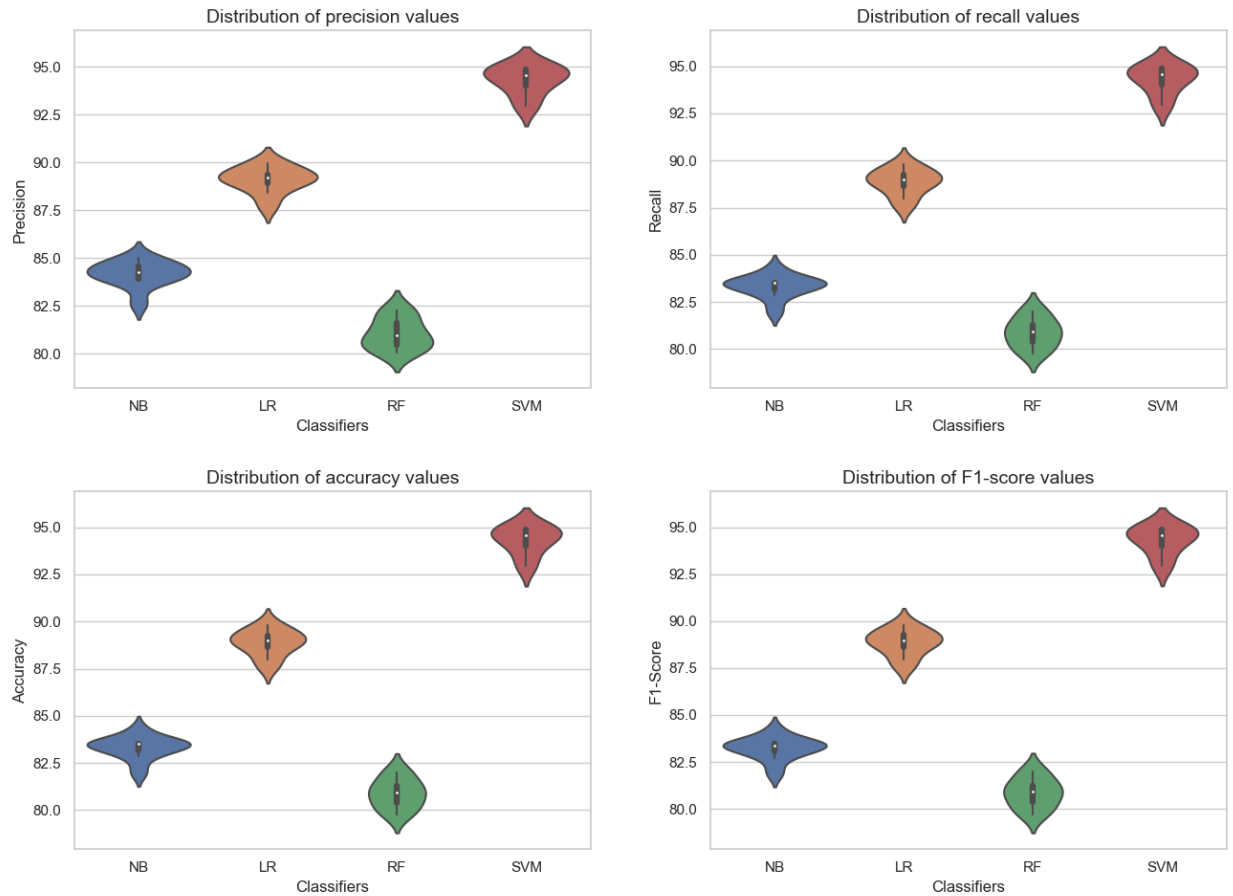
Figure 6.8: Violin plot of performance metrics for the four classifiers on user reviews

their standard deviations. Just like the commits, even for user reviews, the performance of SVM was the best among all the different classifiers; it had obtained the highest value on all the four performance metrics. These values were obtained with the following hyper-parameters: $\mathbf{C = 100}, \gamma = \mathbf{1}$, and kernel = 'poly'. The performance of LR was close to SVM and it had also achieved good results on the different measures. Again, the performance of RF was the lowest among the four classifiers that we have evaluated in this research. Figure 6.8 shows the violin plots for precision, recall, accuracy, and f1-score obtained from 10-fold cross-validation for the four classifiers on user reviews.

This evaluation helped us evaluate the performance of the different ML classifiers and identify the ideal one for our study. We have identified that the Support Vector Machine (SVM) classifier has achieved the best results for classifying compatibility aspects for commit messages and user reviews. Also, for both the cases, the performance of Logistic Regression (LR) was comparable to SVM and can also be used for this type of text classification tasks.

6.2.4 Key findings

In this section, we have listed some of our key findings from this chapter.

- Although the word embedding approaches built using neural networks are more sophisticated and known to produce good results, we obtained the best results with TF-IDF which is one of the traditional approaches. This strengthens the fact that the applicability of NLP and ML techniques is dependent on the context.

- Although Naive Bayes and Random Forest classifiers have been used for other types of text classification, they are not very effective for classifying compatibility related non-functional requirements.

- Our results show that the Support Vector Machine and the Logistic Regression classifiers have consistently produced the best results in all our experiments. As such, these are more effective for classifying compatibility related requirements.

6.3 Summary

In this chapter we have discussed the data collection process in detail – the amount of data (commits and reviews) collected and those retained after each step. We have also discussed the initial findings of some of the key components of the ACOCUR methodology. We have evaluated four types of word embedding techniques; although the more recent techniques are known to work well in other contexts, however in the case of compatibility analysis, we found that the TF-IDF approach provided the best results. We have also evaluated four ML classifiers and observed that two of them have consistently produced good results. As such we have proceeded with these two classification algorithms for all the later analyses. In the next chapter, we shall discuss the empirical evaluation of the results for our research questions.

Chapter 7

Empirical Evaluation

In Chapter 3, we have already discussed the methodology for ACOCUR, and in Chapter 5 we have discussed the tool that has been built by implementing this methodology. Using this tool and the methods discussed, we have analyzed a large number of open source Android apps. In the previous chapter (Chapter 6), we have discussed the data selection process and some of our initial analysis. In this chapter, we shall proceed with the empirical evaluation of the ACOCUR and discuss the answers to our research questions.

7.1   Evaluate RQ1: Commits related to compatibility

To assess the first research question and identify the percentage of developer commits related to addressing the compatibility requirements, we have classified all the commits using the chosen classifier. As discussed in the last chapter, the Support Vector Machine (SVM) and the Logistic Regression (LR) algorithm classifiers produced the best results and their performance metrics were close; as such we have used both the classifiers for identifying compatibility fixes from commit messages.

From our initial data set of 784 apps, we had extracted 283,754 commits using the keyword search. We ran the LR classifier on these commit messages to extract the compatibility related commits. The classifier identified a total of 12,537 commits as related and the remaining as non-related. This corresponds to $\approx 4.42\%$ of the user commits. However, if we consider all the commit messages (including those that did not have any keyword match, i.e., 1,378,736 commits in total), the percentage of compatibility related commit is $\approx 0.91\%$.

We also ran the SVM classifier to extract compatibility related commits. However, this time we did not run it on the entire 784 apps. Instead, we ran the SVM classifier on the 308 apps (which is a subset of the 784 apps) that were selected for the second phase of the research (i.e., considering the user reviews as well). These 308 apps had a total of 877,980 commits, out of which 180,381 were chosen by keyword search. The SVM
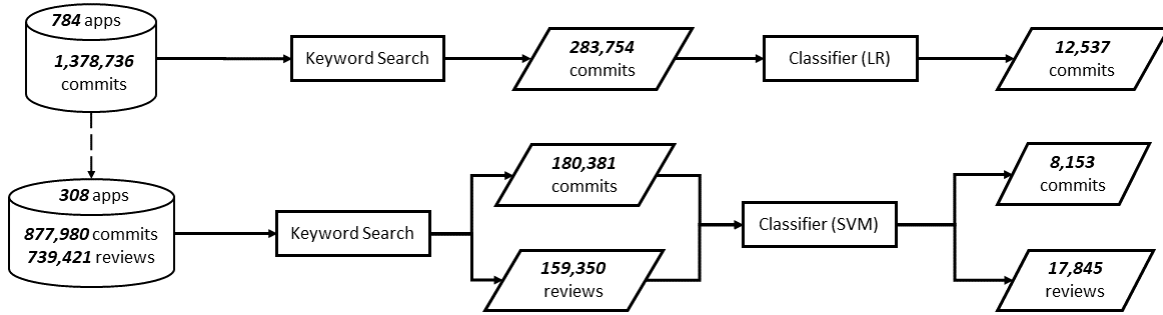
Figure 7.1: Commit and Review classification results

classifier was run on these 180,381 commits and it identified 8,153 of these as compatibility related. This corresponds to $\approx 4.52\%$ of the commits. Again, considering all the commits, these number of compatibility related commits constitute for $\approx 0.92\%$ of the total. Figure 7.1 shows the number of number of commits identified at different stages of the evaluation.

To understand the variation across different apps, we calculated the percentage of compatibility fixes for each app. This value ranges from a maximum of 7.04% (15 compatibility fixes out of a total of 213 commits) to a minimum of 0% (not a single compatibility fix). To understand this better, we further classified the apps based on the total number of commits. Table 7.1 shows the distribution of the percentage of compatibility

Total # of commits	Compatibility-related commits	# of apps
$\leqslant 300$	2.15%	39
$301 - 400$	2.02%	31
$401 - 500$	1.64%	19
$501 - 1,000$	1.62%	67
$1,001 - 2,000$	1.56%	66
$2,001 - 3,000$	1.41%	20
$3,001 - 4,000$	1.09%	21
$4,001 - 5,000$	1.39%	10
$5,001 - 10,000$	1.07%	22
$10,001 - 15,000$	0.93%	5
$15,001 - 30,000$	0.48%	3
$30,001 - 50,000$	0.40%	3
$> 50,000$	0.21%	2
Total	0.92%	308

Table 7.1: Distribution of compatibility related commits

related commits based on the total number of commit messages. It is evident from the table that the percentage of compatibility fixes decreases as the total number of commits increases.

7.2   Evaluate RQ2: Users concerned about compatibility

To assess how much users are concerned about compatibility, we calculated the percentage of user reviews that complain about the apps' compatibility issues. To determine this, we again chose SVM as it had produced good results in the evaluation phase. For the 308 apps selected for this phase of the research, the keyword match extracted 159,350 reviews from a total of 739,421 reviews. The classifier was run only on the keyword searched reviews and it identified a total of 17,845 reviews as compatibility related while the remaining were considered non-related (see Figure 7.1). This corresponds to $\approx 11.20\%$ of keyword matched user reviews. On considering all the user reviews, this compatibility related user reviews constitute for $\approx 2.41\%$.

% of Compatibility-related reviews	# of apps
< 0.5	39
$0.5 - 1$	47
$1 - 2$	81
$2 - 3$	48
$3 - 4$	38
$4 - 5$	19
$5 - 10$	28
> 10	8
Total	308

Table 7.2: Distribution of percentage of compatibility related reviews

To understand the distribution of the compatibility related reviews across apps, we calculated the percentage of compatibility related reviews for each app. The value ranges in between a minimum of 0% (14 apps, for which not a single compatibility related issue raised in the user reviews) to a maximum of 17.43% (19 out of a total of 109 reviews have raised compatibility related issues). Table 7.2 shows the total number of apps for the different ranges of the percentage of compatibility-related reviews.

We have also analyzed the distribution of the percentage of compatibility-related reviews based on the total number of user reviews available; Table 7.3 shows this distribution results. It is clear that the average

73

Total # of reviews	Compatibility-related reviews	# of apps
$\leqslant 200$	3.19%	69
$201 - 300$	2.02%	31
$201 - 300$	3.24%	27
$301 - 500$	2.54%	46
$501 - 1000$	2.50%	49
$1001 - 2000$	2.39%	37
$2001 - 3000$	2.42%	20
$3001 - 5000$	2.13%	21
$5001 - 10000$	2.35%	21
$> 10,000$	2.47%	18
Total	2.41%	308

Table 7.3: Distribution of compatibility related reviews based on total review count

distribution of compatibility related reviews across the different apps based on the total number of reviews is similar; therefore, we can conclude that the percentage of compatibility related reviews is independent of the total number of user reviews.

7.3   Evaluate RQ3: Different compatibility types

In this section, we shall evaluate research question 3 that is related to the different compatibility types for commits and reviews. Since we have performed the card sorting exercise separately for commits and reviews to identify the different types in them, we have created two separate taxonomies – one for compatibility types in commits, and the other for the reviews.

7.3.1   Compatibility types in commit messages

By analyzing the compatibility related commit messages during the card sorting process, we identified different types of compatibility related fixes done by developers. As a result of the open card sorting phase, we identified twelve different categories into which the compatibility fixes could be distributed; however, since some of these categories were very discrete and the number of cards allocated to them was very less, we grouped some of them to create broader categories. As an outcome of this exercise, we developed a two-level taxonomy: the top level has a broad categorization, while the second level contains the low-level types. Table 7.4 describes the taxonomy for the different types of compatibility fixes identified from commit

High Level Taxonomy	Low level Taxonomy	% of Cards	Example
Android version	Support for Android versions	55.86	Android Oreo support; Update targetSdkVersion to the latest (API 22, Android 5.1)
	Support libraries	9.72	Upgrade support libraries to version 26.1.0
	Fix issues on specific Android versions	8.64	Fix crash in Ice cream sandwich Android versions
	Fix for previous versions	7.37	Makes the toolbar shadow visible for pre-Lollipop Android versions
	Remove support for Android versions	4.06	Set minimum required Android version to 12 (Honeycomb 3.1)
Devices	Device specific fixes	6.56	Catch exception on possibly incompatible Samsung tasks version (in Galaxy S3)
	Device configuration	1.22	Support super widescreen Android devices; Remove ANT entries from preferences if the device does not support ANT
	Stop support	0.54	Removing support for armeabi since there are no armeabi devices that support our minimum api level.
	Devices - others	0.40	Bump MK to fix crash on modern devices; Updated to support landscape tablets; Removed bluetooth backport and made other necessary changes to get it working with a more modern Android phone
Others	Other software	1.90	Fix compatibility with new Kanboard version API; Try to fix Dropbox conflicts by waiting on newer versions
	Platform related	0.99	Improve cross-platform compatibility
	General	2.76	Use ResourceCompat to support more Android versions; Updated PermissionsDispatcher to fix compatibility issues.

Table 7.4: Taxonomy of compatibility types in commits

messages. It comprises of three high-level categories (*Android version*, *Devices*, and *Others*) that have been further decomposed into the twelve low-level types.

The low-level taxonomy more specifically refer to the type of fix that the commit is associated with. For example, the fix might have been done to add support for different Android versions, fix some compatibility issues for specific Android versions, resolve compatibility issues with some particular device, remove compat-

ibility support for particular versions or devices, etc. On the other hand, the high-level taxonomy broadly describes the type of fix; i.e., whether it is for Android versions, devices, or other general types. From the table we can see that a majority of the cards (85.62%) belonged to *Android version*, while only 8.73% were related to *Devices* category and the remaining 5.65% were mapped to *Others*. It is evident from the card sort exercise that most of the fixes done by app developers to address the app's incompatibility are related to Android versions, and only a few are related to the other two categories.

7.3.2 Compatibility types raised from reviews

Similar to Section 7.3.1, we have also analyzed the compatibility related reviews using the card sorting technique to identify the different types of compatibility issues expressed in user reviews. After extracting the different groups, we realized that the total number of categories identified were too large to form a

High Level Taxonomy	Low level Taxonomy	% of Cards	Description
Android version	Android-Crash	19.70	App crashes and completely stops working on particular Android versions
	Android-Behavioral	9.03	The app works but some of the functionalities or behaviors are affected
Platform update	Platform-Crash	2.95	App crashes after Android update
	Platform-Behavioral	2.50	Some of the functionalities or behaviors are affected after Android update
App version	Version-Crash	12.95	The app version crashes and does not work
	Version-Behavioral	6.58	Some of the app functionalities are affected after a version update
Devices	Device-Crash	24.36	The app crashes on particular devices
	Device-Behavioral	7.64	Some of the functionalities are affected on particular devices
Others	Irregular crashes	4.51	App crashes on different circumstances
	Performance	5.23	The app's performance is affected in certain configurations
	Other software	2.57	App is not compatible with other related software
	General	1.98	Other types of general incompatibilities

Table 7.5: Taxonomy on compatibility types in reviews

taxonomy. The reason for the creation of so many distinct groups was that users are discrete in describing the unique issues that they encounter and thus each of the different issues can be attributed to individual categories. So to build a taxonomy that can be generalized, we have further grouped the different types of incompatibilities to broader categories. After due deliberation and multiple iterations, we have finalized 12 categories into which the different types of incompatibilities could be mapped. These 12 categories form the low-level taxonomy that describes the different aspects of the compatibility issues. These low-level taxonomies have been mapped to five broader categories that represent the high-level taxonomy. These five broad categories represent the primary sources for the compatibility issue. Table 7.5 describes our two-level taxonomy describing the compatibility issue types extracted from user reviews.

It is clear from the table that each of the first four high-level categories contains two subcategories; the first subcategory represents the cases where the apps crash or suffers terribly as a result of the change discussed under the main category, and the second category represents those types where the app works with some imperfection and fault as a result of the changes described under the main category. To maintain uniformity in the terms, we have associated the term *Crash* with the first subcategory, and the term *Behavioral* with the second subcategory. Again, the different types of behavioral incompatibilities have been described under Table 7.6. This ensured that we can keep track of the different variants of the incompatibility types discussed in the user reviews and at the same time keep the taxonomy manageable.

	Behavioral			
Compatibility Types	**Android**	**Platform**	**Version**	**Device**
SD Card & External device	✓	✓	✓	✓
Sync & Connectivity	✓	✓	✓	✓
Finger print scanner		✓	✓	✓
Display & GUI	✓		✓	✓
Access permissions	✓			✓
Audio & Mic	✓			✓
Unsupported Features	✓			✓
Notifications	✓			
Others				✓

Table 7.6: Types of behavioral issues in reviews

From Table 7.5, we can see that percentage of cards associated with the five broad categories are as follows: 28.73% associated to Android version, 5.44% are for Platform update, 19.54% belong to App version, 31.98%

are for Devices, and the remaining 14.30% are classified under Others. Contrary to the taxonomy generated from commit messages where the majority of the cards belonged to a single category, in this case, the cards are more uniformly distributed among the five categories.

7.3.3 Classifying commits based on compatibility types

We have classified all the compatibility-related commits (8,153 commit messages - those identified by running the SVM classifier on 308 apps, Section 7.1) into the different categories based on the taxonomy generated under Section 7.3.1. The performance of the classifiers suffered when the classification was done using the low-level taxonomy. This was because the number of classes was too many for the machine learning classifiers to deal with. As such, to achieve better results, we decided to classify the commits based on the high-level taxonomy only.

As described under Section 7.3.1, we had identified three broad categories into which the compatibility fixes could be mapped. All the commits were classified into these three categories. To choose the best model for this multi-label classification, we again evaluated the four ML classifiers (Naive Bayes, Logistic Regression, Support Vector Machine, and Random Forest). The only difference, in this case, was that while earlier the classifier was used for binary classification (related vs non-related), this time the classifier was multi-label (Android versions, Devices, and Others).

Commit Messages				
Classifier	Precision	Recall	Accuracy	F1-score
SVM	88.62	89.75	89.75	88.54
LR	87.53	88.85	88.85	86.08
RF	72.87	85.36	85.36	78.62
NB	80.31	86.12	86.12	80.86

Table 7.7: Performance of classifiers for identifying compatibility types

To evaluate the performance of the classifiers, we have used 10-fold cross-validation (as described under Section 3.4.5) and calculated the four performance metrics (precision, recall, accuracy, and f1-score). In Table 7.7, the average values of these metrics for the four classifiers have been listed. The performance of the Support Vector Machine (SVM) classifier was again the best among all the four classifiers. Other than SVM, the performance of the Logistic Regression (LR) classifier was also very good on all the measures.

The performance of the Random Forest (RF) classifier was again the lowest among the four classifier models. Therefore, we chose SVM for classifying the 8,153 commit messages into the different compatibility types.

Category	# of commits	% of commits
Android version	6,868	84.24%
Devices	800	9.81%
Others	485	5.95%
Total	8,153	100%

Table 7.8: Distribution of compatibility types in commits

The result of the compatibility type classification is shown under Table 7.8. As evident from the table, the distribution of the commits into the three categories is similar to what we had observed during the card sorting exercise; almost 85% of the related commits fell under Android versions compatibility type, while the other two types had a much smaller share.

Therefore, we can conclude that a majority of the compatibility fixes done by app developers are for Android versions. All incompatibilities arising out of the discrepancies in the Android versions are of utmost importance to Android app developers as the impact of these issues is much wide; on the contrary, the incompatibilities with particular devices assume a much lower priority.

7.3.4 Classifying reviews based on compatibility types

We have also classified all the compatibility-related reviews (17,845 reviews - those identified by running the SVM classifier in Section 7.2) into the different categories based on the compatibility types identified under Section 7.3.2. Again, as the low-level taxonomy had a large number of categories, we have used the high-level taxonomy categories for this classification.

In Section 7.3.2, we had identified five broad compatibility types from the user reviews. For classifying the reviews into these five categories, we had again evaluated the four ML classifiers. The process was exactly similar as we had performed under Section 7.3.3. Table 7.9 describes the performance of the four classifiers for classifying user reviews messages into the five compatibility types. We observed that Logistic Regression performed the best on all the four performance metrics; SVM's performance was very close to LR. However, the overall performance of the classifiers was significantly lower as compared to all the earlier cases.

We have examined some of the reviews and the reason for this apparent drop in classifier performances.

Review Messages				
Classifier	Precision	Recall	Accuracy	F1-score
SVM	69.72	71.79	71.79	69.74
LR	70.98	74.38	74.38	69.91
NB	53.99	60.35	60.35	54.09
RF	48.43	48.94	48.94	39.83

Table 7.9: Performance of classifiers for identifying compatibility types from user reviews

This can be attributed to the same reason as to why there were a larger number of conflicts while building the training sets. Users often provide all the configuration specifications while reporting a problem - as such it becomes difficult to pinpoint the actual origin of the issue. We chose LR for classifying the remaining user reviews into the five compatibility types.

Category	# of reviews	% of reviews
Android version	3,854	21.60%
Platform update	818	4.58%
App version	3,119	17.48%
Devices	8,197	45.93%
Others	1,857	10.41%
Total	17,845	100%

Table 7.10: Distribution of compatibility types in user reviews

The result of the compatibility type classification for user reviews is shown under Table 7.10. The top three categories into which the reviews had been classified were *Devices*, *Android version*, and *App version*. As such we can infer that the majority of the compatibility issues reported by users in their reviews correspond to these three categories. We have observed that almost half of the compatibility-related reviews have been classified as Devices compatibility type; we can infer that users mostly feel that the incompatibility in the app is related to their particular device and not for any other reason.

7.4   Evaluate RQ4: Responsiveness of developers to user requests

To measure the degree of alignment between reviews and commits, we have performed two types of analysis as described in Section 3.7. In this section, we shall discuss the results of our analysis.

- **Linking individual reviews to commits** To measure the responsiveness of the developers to the reviews, we wanted to determine if a particular compatibility fix is a result of a compatibility requirement or issue articulated in the reviews. As such we have identified all the compatibility fixes (commits) that have been made after a compatibility-related review has been posted and within a specified time frame (for our thesis, we have considered the time frame as 90 days). Next for each of the reviews, we have calculated the similarity with all the identified commits.

  Textual Similarity - Out of 17,845 reviews that were identified as compatibility related, 12,878 reviews either did not have a corresponding commit or the textual similarity was zero. For the remaining 4,967 reviews, the textual similarity was non-zero; i.e., there is at least one commit with a minimum of a one-word match. However, this approach did not provide any promising results as we could not identify any threshold value for textual similarity to consider the commit linked to the review.

  On manual investigation, we have identified several cases where a commit is linked to the review despite textual similarity being zero; also there are cases where there is no apparent link between a review and the commit although the textual similarity is non-zero. Therefore, it was evident that textual similarity based on word matching is not a suitable approach for linking reviews to commits.

  Semantic Similarity - Before calculating the semantic similarity on the entire data set, we evaluated the approach on a small set of reviews and commits that were manually selected. The semantic similarity between reviews and commits was calculated using both the approaches – cosine similarity, and word mover's distance. However, just like textual similarity, the results were not comprehensible. Let us take the following examples -

  Case 1:

  Review : The app is incompatible with samsung phone (Date: 2017-09-22)

  Commit : Fixed crash while loading (Date: 2017-11-17)

  There is no word match between these two sentences. As such the textual similarity is zero. However, the cosine similarity (using the GloVe embedding) between these two texts is 0.85 and the WMD is 1.35. This implies a high semantic similarity. But no manual inspection it is obvious that there is no apparent connection between these two sentences. The fix in the commit is not associated with the issue raised in the review.

  Case 2:

Review: The autofill function has stopped working, and uninstall/reinstall doesn't work. I'm wondering if it's an OS incompatibility because I've seen it work on newer phones. I'm running Android 4.1.2 on the Samsung Galaxy Stellar. (Date: 2015-03-16)

Commit : Fixes for autocomplete crashing (Date: 2015-03-22)

Again, in this case, there is no word match; so textual similarity is zero. The GloVe embeddings cosine similarity is 0.64 and WMD is 1.24. These values are similar to Case 1, but the commit is probably related to the issue raised in the review. Also, the dates of the review and the commit suggest that the fix has been applied six days after the issue has been reported.

As explained in the above two cases, it was obvious that even semantic similarity between review and commit may not be an ideal way to establish the linkage between reviews and commits. Therefore, we resorted to the second option to measure the responsiveness of the developers to user reviews.

- **Linking compatibility types of reviews to those in commits**

  Tables 7.8 and 7.10 show the distribution of the commits and reviews respectively across the different compatibility types. In Figure 7.2, we have plotted the data in one graph for easy comparison.
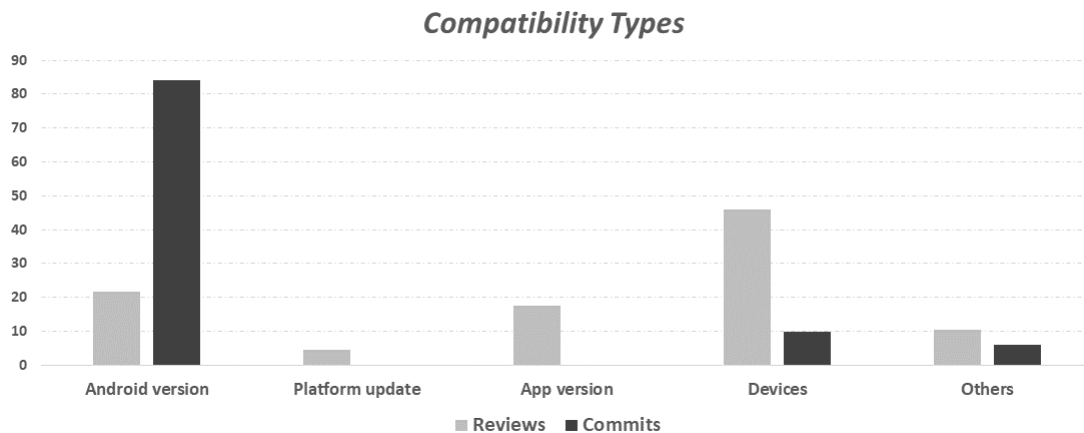
Figure 7.2: Distribution of compatibility types for commits and reviews

In the case of commits, the majority of the fixes are related to the Android version. We have observed that $\approx 85\%$ of the commits fall under this category; less than 10% of the commits were related to Devices, and only about 6% were of the Others types.

On the other hand, in the case of reviews, the distribution of the reviews across the compatibility types are different. First of all, the compatibility types for the reviews do not completely match those of the

commits; therefore, we can not perform a point-by-point comparison. However, we have found that the largest share of reviews ($\approx 46\%$) has been associated with the Devices compatibility types, while 21.59% discuss the Android version, and 17.48% are related to the App version incompatibility. The remaining two categories have around 10.41% and 4.58% of the reviews.

From these two distributions, it is evident that while users mostly focus on Device specific compatibility issues, developers mostly target Android versions specific issues. Thus, we can only conclude that the percentage of commits across the different compatibility types does not align with those for the reviews. As such, that there is not proper alignment between user reviews and developer commits when respect to compatibility requests.

It might be possible that while users perceive a compatibility issue as device-specific, the actual reason might be different (as expressed by developers in the survey). As such while the user reports the issue as device type, the developer fix might be of a different type. We believe that the developers address the compatibility fixes based on what they deem necessary (developer-driven development).

7.5 Key findings

In this section, we have discussed some of our important findings.

- The percentage of reviews discussing about apps' incompatibility issues is higher than the percentage of commits related to compatibility fixes.

- The percentage of compatibility fixes related to Android versions is the highest. The number of fixes in the other categories is low.

- The compatibility requirements from reviews are more uniformly distributed across the different categories.

- The existing textual similarity measures for identifying compatibility requirements are not effective if the nature of the texts are different (reviews vs commit messages).

- App developers often follow a developer-driven development approach when addressing compatibility requirements.

7.6   Threats to validity

We proposed an approach to perform compatibility analysis in mobile applications and evaluated our approach to open-source Android apps. However, it is pivotal to any research to identify and evaluate any threats to validity to further reinforce the robustness and applicability of the approach for other non-functional requirements and other data types.

We identified several instances of the existence of threats to validity in the evaluation of our approach. In this section, we define the threats to validity for the studies conducted and our efforts to mitigate them. Following Wright et al. [82], Siegmund et al. [68], and Wohlin et al. [81], we define the following classifications of threats to validity:

- **Internal validity threats**. It relates to the threats that refer specifically to whether an experimental treatment/condition makes a difference to the outcome or not. It relates to how well the experiments and analyses have been conducted.

  Here we shall discuss some of the internal threats to our work.

  Dependence on ML and NLP techniques: The ACOCUR approach for identifying compatibility fixes from commit messages and requirements from reviews is dependent on a pipeline of natural language processing and machine learning techniques. The outcome of the process and the results might vary based on the extensiveness of data sanitization and the process of lemmatization. Also, the choice of hyper-parameters for the classifiers is dependent on the current data set. While these have produced good results for the selected apps, the outcome might change on a different data set. Moreover, the efficiency of any ML technique depends on a lot of factors. Testing all of them in conjunction was beyond the scope of this work.

- **External validity threats**. It relates to the validity concerns caused by the generalization and replication of the results of an experiment to other scenarios. We have identified two types of external validity threats that have been listed below.

  Origin of Datasets: Our evaluation is based on 784 open-source Android applications publicly hosted on GitHub. Therefore, our results may not be valid for applications on other platforms. Also, the total number of available apps in the Android play store is very large and a majority of them are not open

source which we could not evaluate. Also, we have not evaluated any app from the Apple App Store. As such, we can not claim the general applicability of our results for all the available mobile apps.

Analysis of these apps may lead to different results. Nonetheless, our approach is still relevant to all mobile apps for which we intend to analyze the compatibility requirements.

Applicability of the approach for other non-functional requirements: We have proposed an approach for analyzing non-functional for mobile apps. However, in this thesis, we have worked with only one NFR (compatibility) and analyzed it in detail. We have not dealt with any other NFRs in this thesis, nor have we evaluated the approach for other types of non-functional requirements. As such, though, the concept is theoretically applicable to other cases as well, we have not empirically evaluated the same. As such, we can not claim for external validity for this approach with other non-functional requirements.

- **Construct validity threats**. It relates to the influence of generalizing the result of the experiment on the concept or theory behind the experiment. It covers issues that relate to the design of the experiment.

Here we have listed two construct validity threats that have been identified.

Training set misclassification: The proposed ACOCUR model is dependent on supervised machine learning classifiers; as such it is very important to build good training sets to train the models to produce desirable results. The performance of the models and the applicability of the results relies on the training set. Although the training sets for our proposed models have been labeled by two independent annotators to mitigate the problem of biased labeling, we can not completely eliminate the risk of misclassification. The results might vary if the training set was built by different annotators.

Commit messages are developer dependent: The ACOCUR methodology relies on commit messages for identifying compatibility related fixes. The commit messages are supposed to describe the nature of change done by the developer in that particular commit. As per the standard good development practices, it is recommended that commit messages should clearly describe the changes done in that commit such that anyone can comprehend the change in the commit from the commit message itself. However, in reality, we have seen that developers often fail to include good commit messages; the

messages are rather vague or cryptic. In such a scenario, it might not be possible to comprehend the type of changes done in a particular commit by referring to the commit messages alone. Our proposed approach is solely dependent on commit messages to identify the non-functional requirement fixes using the assumption that app developers will provide meaningful descriptions in the commit messages.

- **Conclusion validity threats**. It relates to the extent to which the conclusions made about the relationship in our observations are correct and scientifically sound. It corresponds to the relationship between the independent and dependent variables.

Developer-Driven development: Using the ACOCUR methodology, we have empirically evaluated a large number of apps. Our attempt to measure the dependence of developers on reviews to identify compatibility requirements resulted in making a comparative analysis of the different compatibility types. We have found that while users report mostly based on devices incompatibility type, developer fixes are mostly based on Android versions.

Although this analogy makes us believe that app developers follow a developer-driven development approach for compatibility requirements and don't rely on reviews, it might be possible that developers rely on reviews on certain other aspects. While majority of the fixes related to compatibility are Android versions specific, the compatibility requirements might have been traced from the reviews which described some other type of compatibility issue.

7.7 Summary

In this chapter, we have discussed the empirical evaluation for the first four research questions. The answers to the other two questions (RQ5 and RQ6) have already been discussed in the previous chapters. We have identified the percentage of commits that are related to compatibility fixes; using that analysis, we can estimate the effort dedicated by developers to address the compatibility related requirements. Also, we have identified the different types of compatibility fixes performed by the developers and how the effort dedicated to the different types vary.

Simultaneously, we have also identified the importance given to the compatibility requirements by users by calculating the percentage of reviews that are related to the app's incompatibility issues. We have also extracted the different types of compatibility issues raised by users.

Comparing the different statistics, we have analyzed how the distribution of data among the different

compatibility types vary for commits and reviews. We have also discussed our findings on measuring the dependence of developers on reviews.

Finally, we have listed our key findings and observations from these evaluations; also discussed the different threats to validity that we have identified in our work.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

It is clear beyond doubt that non-functional requirements are a key component of any system; failing to address these requirements severely affects the overall execution of the software system. This ongoing research is part of a larger effort to study NFRs for mobile apps. Nayebi et al. [53] performed a study with participants of the 4th International Workshop on Software Release Engineering (RELENG 2016) to evaluate the impact of "the market" on release decisions for mobile apps. Asked to compare mobile with non-mobile apps, 20 out of the accepting 22 participants (90.9%) believed that customer feedback has the highest importance for evaluating the success and failure of mobile apps. The current work was following this agenda and tried to figure out how customer feedback is related to actual changes of developers.

In the study conducted by Jha et al. [33], it has been identified that 40% of user reviews depict at least one type of non-functional requirements; of this 8.22% are related to *supportability*. Zou et al. [86] and Ahmad et al. [10] have evaluated stack overflow posts from iOS developers to identify which non-functional requirements developers focus on; it has been found that more than 90% of the posts discuss at least one type of NFR and developers mostly focus on *usability* and *reliability*; however, none of these studies have analyzed *compatibility* in particular.

In this thesis, we have studied the compatibility aspect in great detail; however, the study can be extended to other non-functional requirements as well. We have analyzed a large number of open-source Android apps and studied their commit messages to ascertain how much effort the app developers provide to deal with mobile app incompatibilities. In the course of this research, we have explored different machine learning algorithms and various word embedding techniques for identifying and classifying compatibility

fixes from commit messages. We observed that two of the ML classifiers (Support Vector Machine and Logistic Regression) have performed consistently well in this regard. We also built a two-level taxonomy for describing the different types of compatibility issues fixed in the commits. With these results, we have successfully answered the research questions that dealt with analyzing compatibility from commit messages.

Simultaneously, we have also evaluated the user reviews from the apps that were selected for this study. We have been interested in identifying how much the users are concerned about the app's incompatibilities and what types of incompatibilities they report in the reviews. Just like the commits, we have explored different ML classifiers and several word embedding techniques for identifying and classifying compatibility requirements from user reviews. We have also built another taxonomy on the different types of compatibility requirements described in reviews. These studies have enabled us to answer the research questions related to analyzing compatibility from user reviews.

As a part of this thesis, we have also evaluated the degree of alignment between user reviews and developers' commits related to the app's compatibility. We have also investigated the limitations of some of the available techniques for analyzing the degree of responsiveness of developers to user reviews.

We have proposed an approach, ACOCUR, for systematically analyzing compatibility requirements from reviews and their corresponding implementations from commits. To facilitate the process, we have also proposed a tool that implements the methodology of ACOCUR. Based on the user's choice, the tool would automatically mine user reviews and commits from their respective data sources and systematically analyze compatibility requirements and fixes from them.

As part of this thesis, we have also surveyed with mobile app developers and discussed different aspects related to compatibility for mobile apps. The developers have stressed the importance of identifying and fixing app incompatibilities reported in user reviews, but have also expressed their inability to effectively do so for lack of automated tools. They have also unanimously conveyed the need for an automated tool that would facilitate this process.

While we have discussed certain aspects related to the mobile app's compatibility in our work, there are other aspects that also needs to be investigated to form a comprehensive knowledge on this subject. In the following section, we shall discuss some of these aspects that we intend to study in the future.

8.2   Future work

As part of our future work, we aim to extend, formalize, and generalize our approach to other non-functional requirements. Although theoretically, the approach should apply to other non-functional requirements as well, we can not claim for external validity as those have not been tested.

Also, in this thesis, we have concentrated only on open source Android applications. However, a major chunk of the Android applications is not open source and we could not empirically evaluate those apps in this study. As such, we would like to study other Android apps to evaluate the behavior of the developers for those apps. At the same time, this study has not referred to other non-android platforms. Since iOS applications constitute the second-biggest share of mobile apps, it will be interesting to evaluate compatibility and other non-functional requirements for iOS apps as well.

A major part of the work in this thesis is related to identifying compatibility fixes in commit messages. But as described under the threats to the validity section, commit messages may not reflect the complete details of changes done in the commit. As such, the analysis performed using commit messages only may miss certain critical information. As part of our future work, it would be interesting to include other aspects of the data apart from the commit messages only to identify the actual nature of the commit fixes.

And lastly, we have noticed that the existing techniques for linking individual reviews to commits did not provide meaningful results for non-functional requirements. As such, we would also like to explore other options to be able to map individual review to commits and explicitly check which reviews have been acted upon and which all have been ignored.

Bibliography

[1] gensim: Topic modelling for humans. https://radimrehurek.com/gensim/index.html. [Online; accessed 15-May-2020].

[2] SciPy.org – SciPy.org. https://www.scipy.org/. [Online; accessed 15-May-2020].

[3] Software incompatibility - Wikipedia. https://en.wikipedia.org/wiki/Software_incompatibility. [Online; accessed 15-April-2020].

[4] AppACTS: Mobile app automated compatibility testing service, author=Huang, Jun-fei. In *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pages 85–90. IEEE, 2014.

[5] A Study of Non-Functional Requirements in Apps for Mobile Devices, author=Corbalán, Leonardo and Thomas, Pablo and Delía, Lisandro and Cáseres, Germán and Sosa, Juan Fernández and Tesone, Fernando and Pesado, Patricia. In *Conference on Cloud Computing and Big Data*, pages 125–136. Springer, 2019.

[6] A Gentle Introduction to k-fold Cross-Validation. https://machinelearningmastery.com/k-fold-cross-validation/, 2020. [Online; accessed 05-May-2020].

[7] Statista - The Statistics Portal for Market Data, Market Research and Market Studies. https://www.statista.com/, 2020. [Online; accessed 03-September-2020].

[8] Text Similarities : Estimate the degree of similarity between two texts. https://medium.com/@adriensieg/text-similarities-da019229c894, 2020. [Online; accessed 15-April-2020].

[9] Zahra Shakeri Hossein Abad, Oliver Karras, Parisa Ghazi, Martin Glinz, Guenther Ruhe, and Kurt Schneider. What works better? a study of classifying requirements. In *2017 IEEE 25th RE Conf.*, pages 496–501. IEEE, 2017.

[10] Arshad Ahmad, Chong Feng, Kan Li, Syed Mohammad Asim, and Tingting Sun. Toward Empirically Investigating Non-Functional Requirements of iOS Developers on Stack Overflow. *IEEE Access*, 7:61145–61169, 2019.

[11] Ron Artstein and Massimo Poesio. Inter-coder agreement for computational linguistics. *Computational Linguistics*, 34(4):555–596, 2008.

[12] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[13] Andrew Begel and Thomas Zimmermann. Analyze this! 145 questions for data scientists in software engineering. In *Proceedings of the 36th International Conference on Software Engineering*, pages 12–23, 2014.

[14] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[15] Raymond PL Buse and Thomas Zimmermann. Information needs for software development analytics. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 987–996. IEEE, 2012.

[16] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[17] Ning Chen, Jialiu Lin, Steven CH 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*, pages 767–778, 2014.

[18] Adelina Ciurumelea, Andreas Schaufelbühl, Sebastiano Panichella, and Harald C Gall. Analyzing reviews and code of mobile apps for better release planning. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 91–102. IEEE, 2017.

[19] Jane Cleland-Huang, Raffaella Settimi, Xuchang Zou, and Peter Solc. The detection and classification of non-functional requirements with application to early aspects. In *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 39–48. IEEE, 2006.

[20] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[21] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[22] Thomas M Cover and Joy A Thomas. Elements of information theory second edition solutions to problems. *Internet Access*, 2006.

[23] D Méndez Fernández, Stefan Wagner, Marcos Kalinowski, Michael Felderer, Priscilla Mafra, Antonio Vetrò, Tayana Conte, M-T Christiansson, Desmond Greer, Casper Lassenius, et al. Naming the pain in requirements engineering. *Empirical software engineering*, 22(5):2298–2338, 2017.

[24] Salisu Garba, Babangida Isyaku, and Mujahid Abdullahi. DATA-DRIVEN MODEL FOR NON-FUNCTIONAL REQUIREMENTS IN MOBILE APPLICATION DEVELOPMENT.

[25] Martin Glinz. On non-functional requirements. In *15th IEEE Requirements Engineering Conf. (RE 2007)*, pages 21–26. IEEE, 2007.

[26] John A Gosden. Software compatibility: what was promised, what we have, what we need. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 81–87, 1968.

[27] Xiaodong Gu and Sunghun Kim. What Parts of Your Apps are Loved by Users?(T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 760–770. IEEE, 2015.

[28] Emitza Guzman, Mohamed Ibrahim, and Martin Glinz. A little bird told me: Mining tweets for requirements and software evolution. In *2017 IEEE 25th International Requirements Engineering Conference (RE)*, pages 11–20. IEEE, 2017.

[29] Philipp Haindl and Reinhold Plösch. Towards Continuous Quality: Measuring and Evaluating Feature-Dependent Non-Functional Requirements in DevOps. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 91–94. IEEE, 2019.

[30] Dan Han, Chenlei Zhang, Xiaochao Fan, Abram Hindle, Kenny Wong, and Eleni Stroulia. Understanding android fragmentation with topic analysis of vendor-specific bugs. In *2012 19th Working Conference on Reverse Engineering*, pages 83–92. IEEE, 2012.

[31] Mark Harman, Yue Jia, and Yuanyuan Zhang. App store mining and analysis: Msr for app stores. In *2012 9th IEEE working conference on mining software repositories (MSR)*, pages 108–111. IEEE, 2012.

[32] Abram Hindle, Neil A Ernst, Michael W Godfrey, and John Mylopoulos. Automated topic naming. *Empirical Software Engineering*, 18(6):1125–1155, 2013.

[33] Nishant Jha and Anas Mahmoud. Mining non-functional requirements from App store reviews. *Empirical Software Engineering*, 24(6):3659–3695, 2019.

[34] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 1972.

[35] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E Hassan. What do mobile app users complain about? *IEEE software*, 32(3):70–77, 2014.

[36] Barbara Kitchenham and Shari Lawrence Pfleeger. Principles of survey research: part 5: populations and samples. *ACM SIGSOFT Software Engineering Notes*, 27(5):17–20, 2002.

[37] David G Kleinbaum, K Dietz, M Gail, Mitchel Klein, and Mitchell Klein. *Logistic regression*. Springer, 2002.

[38] Eric Knauss, Daniela Damian, Germán Poo-Caamaño, and Jane Cleland-Huang. Detecting and classifying patterns of requirements clarifications. In *2012 20th IEEE RE Conf.*, pages 251–260. IEEE, 2012.

[39] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. From word embeddings to document distances. In *International conference on machine learning*, pages 957–966, 2015.

[40] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.

[41] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning. *Journal of Machine Learning Research*, 18(17):1–5, 2017.

[42] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. Cid: Automating the detection of api-related compatibility issues in android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 153–163, 2018.

[43] Tong Li and Zhishuai Chen. An Ontology-Based Learning Approach for Automatically Classifying Security Requirements. *Journal of Systems and Software*, page 110566, 2020.

[44] Edward Loper and Steven Bird. Nltk: The natural language toolkit. In *Proceedings of the ACL-02 Workshop on Effective tools and methodologies for teaching natural language processing and computational linguistics-Volume 1*, pages 63–70. Association for Computational Linguistics, 2002.

[45] 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*, pages 344–353, 2017.

[46] Xuan Lu, Xuanzhe Liu, Huoran Li, Tao Xie, Qiaozhu Mei, Dan Hao, Gang Huang, and Feng Feng. PRADA: Prioritizing android devices for apps by mining large-scale usage data. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2016.

[47] Walid Maalej, Zijad Kurtanović, Hadeer Nabil, and Christoph Stanik. On the automatic classification of app reviews. *Requirements Engineering*, 21(3):311–331, 2016.

[48] Walid Maalej, Maleknaz Nayebi, Timo Johann, and Guenther Ruhe. Toward data-driven requirements engineering. *IEEE Software*, 33(1):48–54, 2015.

[49] Mary L McHugh. Interrater reliability: the kappa statistic. *Biochemia medica: Biochemia medica*, 22(3):276–282, 2012.

[50] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[51] Meiyappan Nagappan and Emad Shihab. Future trends in software engineering research for mobile apps. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 21–32. IEEE, 2016.

[52] Qamar Naith and Fabio Ciravegna. Hybrid crowd-powered approach for compatibility testing of mobile devices and applications. In *Proc. of the 3rd Conf. on Crowd Science and Engineering*, pages 1–8, 2018.

[53] Maleknaz Nayebi, Homayoon Farahi, and Guenther 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. IEEE, 2017.

[54] Nan Niu, Sjaak Brinkkemper, Xavier Franch, Jari Partanen, and Juha Savolainen. Requirements engineering and continuous deployment. *IEEE software*, 35(2):86–90, 2018.

[55] Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. Using of Jaccard coefficient for keywords similarity. In *Proceedings of the international multiconference of engineers and computer scientists*, volume 1, pages 380–384, 2013.

[56] Dennis Pagano and Walid Maalej. User feedback in the appstore: An empirical study. In *2013 21st IEEE international requirements engineering conference (RE)*, pages 125–134. IEEE, 2013.

[57] Klérisson VR Paixão, Crícia Z Felício, Fernanda M Delfim, and Marcelo de A Maia. On the interplay between non-functional requirements and builds on continuous integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 479–482. IEEE, 2017.

[58] Fabio Palomba, Mario Linares-Vasquez, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. User reviews matter! tracking crowdsourced reviews to support evolution of successful apps. In *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pages 291–300. IEEE, 2015.

[59] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A Visaggio, Gerardo Canfora, and Harald C Gall. How can i improve my app? classifying user reviews for software maintenance and evolution. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 281–290. IEEE, 2015.

[60] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python, 2011.

[61] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[62] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[63] Yu Ping, Kostas Kontogiannis, and Terence C Lau. Transforming legacy Web applications to the MVC architecture. In *Eleventh Annual International Workshop on Software Technology and Engineering Practice*, pages 133–142. IEEE, 2003.

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

[65] Gordon Rugg and Peter McGeorge. The sorting techniques: a tutorial paper on card sorts, picture sorts and item sorts. *Expert systems*, 14(2):80–93, 1997.

[66] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza, and Rocco Oliveto. Data-driven solutions to detect API compatibility issues in Android: an empirical study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 288–298. IEEE, 2019.

[67] Shaked Zychlinski. The search for categorical correlation – towards data science. https://towardsdatascience.com/the-search-for-categorical-correlation-a1cf7f1888c9. [Online; accessed 11-September-2020].

[68] Janet Siegmund, Norbert Siegmund, and Sven Apel. Views on internal and external validity in empirical software engineering. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 9–19. IEEE, 2015.

[69] Yuan Tian, Meiyappan Nagappan, David Lo, and Ahmed E Hassan. What are the characteristics of high-rated apps? a case study on free android applications. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 301–310. IEEE, 2015.

[70] László Tóth and László Vidács. Study of various classifiers for identification and classification of non-functional requirements. In *International Conference on Computational Science and Its Applications*, pages 492–503. Springer, 2018.

[71] Lorenzo Villarroel, Gabriele Bavota, Barbara Russo, Rocco Oliveto, and Massimiliano Di Penta. Release planning of mobile apps based on user reviews. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 14–24. IEEE, 2016.

[72] Tianlu Wang, Peng Liang, and Mengmeng Lu. What Aspects Do Non-Functional Requirements in App User Reviews Describe? An Exploratory and Comparative Study. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 494–503. IEEE, 2018.

[73] Lili Wei, Yepang Liu, Shing-Chi Cheung, Huaxun Huang, Xuan Lu, and Xuanzhe Liu. Understanding and detecting fragmentation-induced compatibility issues for android apps. *IEEE Transactions on Software Engineering*, 2018.

[74] Wikipedia contributors. Correlation ratio — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Correlation_ratiooldid=850116842, 2018. [Online; accessed 11-September-2020].

[75] Wikipedia contributors. Chi-square distribution — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Chi-square_distributionoldid=979585636, 2020. [Online; accessed 11-September-2020].

[76] Wikipedia contributors. Cosine similarity — Wikipedia, the free encyclopedia, 2020. [Online; accessed 2-May-2020].

[77] Wikipedia contributors. Kruskal–wallis one-way analysis of variance — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Kruskal%E2%80%93Wallis_one-way_analysis_of_varianceoldid=981441008, 2020. [Online; accessed 11-September-2020].

[78] Wikipedia contributors. Logistic regression — Wikipedia, the free encyclopedia, 2020. [Online; accessed 19-September-2020].

[79] Wikipedia contributors. Tf–idf — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Tf%E2%80%93idfoldid=979969361, 2020. [Online; accessed 7-October-2020].

[80] Wikipedia contributors. Wikipedia — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Wikipediaoldid=981810697, 2020. [Online; accessed 4-October-2020].

[81] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering.* Springer Science & Business Media, 2012.

[82] Hyrum K Wright, Miryung Kim, and Dewayne E Perry. Validity concerns in software engineering research. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 411–414, 2010.

[83] Liang Yu, Emil Alégroth, Panagiota Chatzipetrou, and Tony Gorschek. Utilising CI environment for efficient and effective testing of NFRs. *Information and Software Technology*, 117:106199, 2020.

[84] Fatima Zahra, Azham Hussain, and Haslina Mohd. Usability evaluation of mobile applications; where do we stand? In *AIP Conference Proceedings*, volume 1891, page 020056. AIP Publishing LLC, 2017.

[85] Thomas Zimmermann. Card-sorting: From text to themes. In *Perspectives on Data Science for Software Engineering*, pages 137–141. Elsevier, 2016.

[86] Jie Zou, Ling Xu, Weikang Guo, Meng Yan, Dan Yang, and Xiaohong Zhang. Which non-functional requirements do developers focus on? an empirical study on stack overflow using topic analysis. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 446–449. IEEE, 2015.

Appendix A

Categories of mobile apps

App Category	Count of apps
Tools	90
Productivity	39
Communication	31
Games	22
Books & Reference	19
Social	13
Finance; Music & Audio; Personalization	12 * 3
Video Players & Editors	9
News & Magazines	8
Maps & Navigation; Travel & Local	7 * 2
Health & Fitness	6
Education; Photography	5 * 2
Entertainment; Shopping	3 * 2
Business; Comics; Lifestyle; Sports; Weather	1 * 5

Table A.1: App Categories

Appendix B

List of mobile apps

Table B.1: List of 308 apps analyzed in our study

App Name	Category	Rating	# Commits	# Reviews
And Bible	Books & Reference	4.6	1619	5044
APV PDF Viewer	Books & Reference	-	1200	293
Atarashii!	Books & Reference	-	410	1983
BookWorm	Books & Reference	-	100	510
Cool Reader	Books & Reference	4.3	7483	4356
FBReader: Favorite Book Reader	Books & Reference	4.4	7814	7673
Google I/O 2019	Books & Reference	4.4	1100	1764
Kiwix, Wikipedia offline	Books & Reference	4.4	2120	3516
Librera PRO - eBook and PDF Reader (no Ads!)	Books & Reference	4.5	456	3426
MHGU Database	Books & Reference	4.8	1291	585
Offline Survival Manual	Books & Reference	4.6	6725	386
Pathfinder Open Reference	Books & Reference	4.8	1838	220
Poet Assistant (English)	Books & Reference	4.7	787	802
QuickDic restored	Books & Reference	4.6	129	688
Simple Dilbert	Books & Reference	-	132	480

App Name	Category	Rating	# Commits	# Reviews
Sky Map	Books & Reference	4.2	18893	330
Wikipedia	Books & Reference	4.6	19977	7952
WikipOff	Books & Reference	-	150	416
Wiktionary	Books & Reference	-	351	1218
Wire - Secure Messenger	Business	3.5	4145	3943
Easy xkcd	Comics	4.7	194	730
/u/app	Communication	4.7	183	226
Riot.im	Communication	3.8	245	6827
APG	Communication	-	499	3775
ConnectBot	Communication	4.4	5009	1488
FairEmail - open source, privacy oriented email	Communication	4.7	674	12421
Firefox Browser: fast, private & safe web browser	Communication	4.4	19999	6727
Firefox Klar: The privacy browser	Communication	4.1	131	2596
GTalkSMS	Communication	-	111	1118
Jitsi Meet	Communication	3.3	1062	6523
K-9 Mail	Communication	3.9	15483	8071
Lightning Browser Plus - Web Browser	Communication	3.9	464	1852
OpenConnect	Communication	-	434	987
OpenKeychain: Easy PGP	Communication	4.2	583	6216
OpenVPN for Android	Communication	4.1	4078	1600
Plumble - Mumble VOIP	Communication	4	268	378
QKSMS	Communication	4.3	4343	2113
RDP Remote Desktop aFreeRDP	Communication	3.9	123	10912
Rocket.Chat	Communication	4.3	769	3154
Signal Private Messenger	Communication	4.5	20000	5368
Sipdroid	Communication	3.1	1826	302

App Name	Category	Rating	# Commits	# Reviews
SMS Popup	Communication	-	10884	297
Telegram	Communication	4.4	20000	331
Telegram	Communication	4.4	20000	370
Tint Browser	Communication	-	210	233
Tutanota - Free Secure Email & Calendar App	Communication	4.2	989	4667
VoIP.ms SMS	Communication	4.4	232	386
VX ConnectBot	Communication	3.9	252	746
Weechat Android	Communication	3.9	133	929
Xabber	Communication	4.1	1254	4023
Yaaic	Communication	-	463	1056
yaxim - XMPP/Jabber client	Communication	4	118	1490
AnkiDroid Flashcards	Education	4.5	6524	7551
AnyMemo: Flash Card Study	Education	4.4	248	1817
phyphox	Education	4.6	307	654
PSLab	Education	4.7	288	911
Stepik: best online courses	Education	4.8	1388	7726
Moonlight Game Streaming	Entertainment	4.3	1798	2183
Reicast - Dreamcast emulator	Entertainment	3.2	5317	3291
SUSI.AI	Entertainment	4.1	115	970
Bankdroid	Finance	-	433	1054
Bitcoin Wallet	Finance	4	3319	3495
Dash Wallet	Finance	4	206	3297
GnuCash	Finance	4.5	1338	1354
Green: Bitcoin Wallet	Finance	4.2	151	786
GreenAddress.It	Finance	-	114	345
Mileage	Finance	3.4	2037	442
Ministocks - Stocks Widget	Finance	4.1	1082	259
Money Manager Ex	Finance	-	617	4091
My Expenses	Finance	4.6	523	7749
Smart Receipts Plus	Finance	4.7	430	2166

App Name	Category	Rating	# Commits	# Reviews
Stocks Tracker Widget (open-source)	Finance	4.6	565	456
Andor's Trail	Games	4.2	3365	3420
BoardGameGeek	Games	-	606	4957
Chess	Games	4.1	552	289
Client for Pretend You're Xyzzy (open source)	Games	4.3	182	722
Dolphin Emulator	Games	3.9	9364	24740
DroidFish Chess	Games	4.6	2362	964
Freebloks 3D	Games	4.5	704	1219
Hex	Games	3.9	145	344
MH4U Database	Games	-	1141	465
Mindustry	Games	4.5	5715	6478
Minetest	Games	3.5	786	8491
Nounours and friends	Games	-	103	401
OpenMicroWave (OMW) Nightly	Games	-	319	812
OpenSudoku	Games	4.3	1392	239
PPSSPP - PSP emulator	Games	4.3	17880	21201
Scid on the go	Games	4.4	158	327
ScummVM	Games	4.1	1019	96692
Shattered Pixel Dungeon: Roguelike Dungeon Crawler	Games	4.8	3828	3740
Simon Tatham's Puzzles	Games	4.7	1570	3337
Simple Solitaire Collection	Games	4.5	206	766
SuperTuxKart	Games	4.2	537	18855
Unciv	Games	4.5	2445	3506
Dr. Greger's Daily Dozen	Health & Fitness	4.9	2244	553
openScale	Health & Fitness	-	126	1602
Pedometer	Health & Fitness	3.8	459	404
Plees Tracker	Health & Fitness	-	440	217

App Name	Category	Rating	# Commits	# Reviews
RunnerUp	Health & Fitness	-	129	2278
Zen!	Health & Fitness	4.7	128	240
Aelf - Bible et lectures du jour	Lifestyle	4.7	127	668
A2DP Volume	Maps & Navigation	4	274	330
Androzic	Maps & Navigation	-	127	723
BART Runner	Maps & Navigation	4.4	474	210
CycleStreets journey planner	Maps & Navigation	3	288	1513
GPSTest	Maps & Navigation	-	189	660
Navit	Maps & Navigation	2.8	235	7547
RMaps	Maps & Navigation	-	604	638
AntennaPod	Music & Audio	4.7	5074	4903
Car Cast Podcast Player	Music & Audio	4.4	474	246
Clementine Remote	Music & Audio	4	404	718
DSub for Subsonic	Music & Audio	3.9	624	3559
Phonograph Music Player	Music & Audio	4.1	7423	1458
QuickLyric - Instant Lyrics	Music & Audio	4.3	6409	955
RadioDroid 2	Music & Audio	4.6	110	833
Simple Scrobbler	Music & Audio	3.3	2513	653
Squeezer	Music & Audio	4.2	493	1519
Vanilla Music	Music & Audio	4.3	653	1251
Vinyl Music Player	Music & Audio	4.2	168	1479
Voice Audiobook Player	Music & Audio	4.3	1551	3825
Diode for Reddit	News & Magazines	4	291	935
Flym News Reader	News & Magazines	4.4	132	1281
HN - Hacker News Reader	News & Magazines	4.3	247	254
Materialistic - Hacker News	News & Magazines	4.3	631	1178
NewsBlur	News & Magazines	3.7	488	11105
Reddinator Widget for Reddit	News & Magazines	4.3	139	362
RedReader	News & Magazines	4.6	911	1265
Slide for Reddit	News & Magazines	4.2	1767	3089
AcDisplay	Personalization	3.6	8301	1060

App Name	Category	Rating	# Commits	# Reviews
ADW Launcher 2	Personalization	4.3	18782	923
DashClock Widget	Personalization	-	6688	239
Default Dark Theme for Substratum	Personalization	2.7	306	302
KISS Launcher	Personalization	4.4	756	3101
Lawnchair 2	Personalization	4.4	4812	8686
Lawnchair 2	Personalization	4.4	4812	6391
MultiPicture Live Wallpaper dn	Personalization	3.9	177	250
Muzei Live Wallpaper	Personalization	4.2	5248	2036
Rootless Launcher	Personalization	4.3	4059	6095
Status	Personalization	4	3452	570
Theia Icon Theme	Personalization	-	20000	238
Camera Roll - Gallery	Photography	4.1	742	367
Focal	Photography	-	2655	463
FreeDCam	Photography	3.8	297	5718
Simple Camera - Capture photos & videos easily	Photography	3.9	400	1010
Wikimedia Commons	Photography	4	125	4023
BiglyBT, Torrent Downloader & Remote Control	Productivity	4.4	178	1477
Book Catalogue	Productivity	4.1	1245	1084
CUPS Printing	Productivity	3.7	102	238
Diary	Productivity	-	19507	552
DigitalOcean Swimmer Android	Productivity	4.3	193	249
Document Viewer: PDF, DjVu,...	Productivity	4.1	715	1665
Event Sync for Facebook	Productivity	2.1	356	384
ForkHub for GitHub	Productivity	4.2	160	2803
GitHub	Productivity	-	984	3121

App Name	Category	Rating	# Commits	# Reviews
Habitica: Gamify Your Tasks	Productivity	4.3	4095	2821
Hacker's Keyboard	Productivity	4.2	7982	1244
LibreOffice and OpenOffice document viewer	Productivity	3.9	2521	4740
Loop Habit Tracker	Productivity	4.7	7296	1284
Markor: Markdown Editor - todo.txt - Notes Offline	Productivity	4.7	503	1292
Minimalist Pomodoro Timer (Goodtime)	Productivity	4.6	1168	856
Nextcloud dev	Productivity	-	729	11851
NoNonsense Notes	Productivity	-	523	1385
OctoDroid for GitHub	Productivity	4.5	354	2578
Omni Notes FOSS	Productivity	-	629	2808
Open Explorer Beta	Productivity	3.9	291	1441
OpenTasks	Productivity	4.4	284	671
Orgzly: Notes & To-Do Lists	Productivity	4.7	483	1821
ownCloud	Productivity	4.1	1493	6922
Password Store (legacy)	Productivity	4.4	124	1136
PDF CONVERTER: Files to PDF	Productivity	4.3	271	582
Persian Calendar	Productivity	4.5	4645	2852
PinDroid	Productivity	3.8	158	900
Scarlet Notes	Productivity	4.2	177	626
Seafile	Productivity	4	192	1974
SealNote Secure Encrypted Note	Productivity	4.2	970	283
SGit	Productivity	-	221	302
Simple Alarm Clock Free No Ads	Productivity	4.1	1010	1039
Slim Launcher - Fewer distractions, more life	Productivity	4.1	460	297

App Name	Category	Rating	# Commits	# Reviews
Standard Notes	Productivity	4	482	598
Syncthing	Productivity	4.5	790	1673
Tasks.org: Open-source To-Do Lists & Reminders	Productivity	4.7	1266	8573
Termux:API	Productivity	4.4	498	262
WordPress	Productivity	4.4	20000	35032
World Scribe	Productivity	4.2	253	529
Barcode Scanner	Shopping	4.1	20000	3443
Loyalty Card Keychain	Shopping	4.6	103	393
OI Shopping list	Shopping	4.1	2345	469
BombusMod	Social	3.9	108	1366
Chanu	Social	-	556	1243
Clover	Social	-	1131	1218
Frost for Facebook	Social	-	167	815
nan	Social	-	391	3117
surespot encrypted messenger	Social	4.3	317	2258
surespot encrypted messenger	Social	4.3	317	1778
Talon for Twitter	Social	4.3	4762	480
Tinfoil for Facebook	Social	-	2085	295
Tusky for Mastodon	Social	3.8	703	2096
Tweet Lanes	Social	2.9	1491	612
Twidere for Twitter	Social	4	2449	2957
Twidere for Twitter	Social	4	2448	2957
XCSoar	Sports	4.8	190	34633
AdAway	Tools	-	2774	1885
AdGuard: Content Blocker for Samsung and Yandex	Tools	4.4	3169	345
Aegis Authenticator - Two Factor (2FA) app	Tools	4.5	135	582
AFWall+ (Android Firewall +)	Tools	4.1	1171	1522
Alarm Klock	Tools	3.8	1254	622

App Name	Category	Rating	# Commits	# Reviews
Amaze File Manager	Tools	3.8	1767	2957
Andlytics	Tools	-	465	947
andOTP - Android OTP Authenticator	Tools	4.6	144	956
AnySoftKeyboard	Tools	4	3469	4960
Autostarts	Tools	3.3	609	431
Battery Charge Limit [ROOT]	Tools	4.2	378	223
BatteryBot Battery Indicator	Tools	4.3	6387	200
BatteryBot Pro	Tools	4.4	1457	954
Better Wifi On/Off	Tools	4.1	124	273
Cache Cleaner	Tools	3.8	668	349
Calculator	Tools	4.3	2814	959
Calculator ++	Tools	4.8	2236	2190
Calendar Notifications	Tools	-	255	921
CatLog - Logcat Reader!	Tools	3.7	606	336
Counter	Tools	4.5	478	268
cSploit	Tools	-	8737	1210
CurrentWidget: Battery Monitor	Tools	3.6	1178	218
Device Frame Generator	Tools	3.9	292	366
DigiLux: Fingerprint Gestures for Phone Brightness	Tools	3.4	144	354
dreamDroid	Tools	4.2	435	1416
DuckDuckGo Privacy Browser	Tools	4.7	20000	784
Equate	Tools	4.8	141	402
EtchDroid [NO ROOT] - Write ISOs and DMGs to USB	Tools	4.2	688	274
FareBot	Tools	3.5	547	256
Hangar - Smart app shortcuts	Tools	4.1	306	267
J2ME Loader	Tools	4.3	1793	1212

App Name	Category	Rating	# Commits	# Reviews
Keepass2Android Password Safe	Tools	4.6	3779	2443
KeePassDroid	Tools	4.1	4963	1085
KeepScore - Score Keeper	Tools	4	336	499
Kernel Adiutor	Tools	-	4702	1208
Keyboard/Button Mapper	Tools	4.1	162	1067
MIFARE Classic Tool - MCT	Tools	4.1	109	640
ML Manager: APK Extractor	Tools	4.2	166	295
Moscow Wi-Fi autologin	Tools	-	313	1090
Mozilla Stumbler	Tools	3.9	181	1799
MTG Familiar	Tools	4.5	2836	1591
NetGuard - no-root firewall	Tools	4.2	2898	3347
Network Discovery	Tools	-	387	439
Network Log	Tools	3.7	167	469
OI File Manager	Tools	-	4228	449
OI Safe	Tools	4	818	233
OONI Probe	Tools	4.2	221	1179
Open Link With	Tools	-	410	829
OS Monitor	Tools	4.5	1402	333
Overchan (fork)	Tools	-	160	1113
Pocket Paint: draw and edit!	Tools	3.7	500	2386
Port Authority	Tools	-	138	904
Prayer Times (Namaz Vakti)	Tools	4.2	130	422
Primitive FTPd	Tools	4.5	120	661
qBittorrent Controller Pro	Tools	4.2	139	825
RasPi Check	Tools	4.4	202	551
Recurrence	Tools	-	152	231
SAI	Tools	-	519	608
Screen Notifications	Tools	-	642	241
Screen Stream over HTTP	Tools	3.9	813	347

App Name	Category	Rating	# Commits	# Reviews
SecondScreen - better screen mirroring for Android	Tools	3.9	267	326
Shader Editor	Tools	4.6	110	546
Simple App Launcher - Launch apps easily & quickly	Tools	4.3	108	378
Simple Calendar Pro - Events & Reminders Manager	Tools	4.8	1054	3487
Simple Contacts Pro - Manage your contacts easily	Tools	4.3	171	1658
Simple File Manager Pro - Manage files easy & fast	Tools	4.5	248	1232
Simple Flashlight - Bright display & stroboscope	Tools	4.5	131	424
Simple Gallery Pro - Photo Manager & Editor	Tools	4.8	7729	5302
Simple Music Player - Play audio files easily	Tools	4.2	586	1048
Simple Notes Pro: To-do list organizer and planner	Tools	4.8	186	1035
Simple Thank You - Thanks for supporting us :)	Tools	4.7	453	223
SMS Backup+	Tools	2.8	15529	1642
Superuser	Tools	-	123	570
Superuser	Tools	-	3471	482
Taskbar - PC-style productivity for Android	Tools	4.1	857	1235
Terminal Emulator for Android	Tools	4.3	7408	879
Termux	Tools	4.4	9071	651
Todo Agenda for Android 4 - 7.0	Tools	-	1634	645
Traccar Client	Tools	4.3	111	337

App Name	Category	Rating	# Commits	# Reviews
UniPatcher	Tools	3.9	962	217
Unit Converter Ultimate	Tools	4.5	3351	332
UserLAnd	Tools	4.3	613	876
Vespucci - an OSM Editor	Tools	4.1	117	4246
VIMTouch	Tools	-	313	344
VPN Hotspot - tethering/Wi-Fi repeater	Tools	4.3	334	955
WiFi Analyzer (open-source)	Tools	4.2	1548	1327
WiFi Automatic	Tools	4.2	1195	252
Wifi Fixer	Tools	3.9	2506	1255
WiGLE WiFi Wardriving	Tools	4.4	499	1410
Yubico Authenticator	Tools	3.3	181	394
FixMyStreet	Travel & Local	3.5	135	1039
GPS Logger for Android	Travel & Local	4.2	693	1478
OsmAnd+ - Offline Maps, Travel & Navigation	Travel & Local	4.8	1843	53347
PassAndroid Passbook viewer	Travel & Local	4.4	885	1642
Tram Hunter	Travel & Local	4.4	515	275
WarmShowers	Travel & Local	2.8	142	682
World Clock & Weather Widget	Travel & Local	4.4	494	502
Kodi	Video Players & Editors	4.2	20000	42981
Kore, Official Remote for Kodi	Video Players & Editors	4.3	2177	688
Markers	Video Players & Editors	4.1	847	213
MPDroid	Video Players & Editors	4.2	509	2412
Mythmote	Video Players & Editors	4.1	229	330
ObscuraCam	Video Players & Editors	2	322	571
Transdroid	Video Players & Editors	-	705	617
VLC for Android	Video Players & Editors	4.4	20000	13691
XBMC Remote	Video Players & Editors	-	2681	897
Blitzortung Lightning Monitor	Weather	3.5	423	927