Improving the Quality of User Feedback for Continuous Software Evolution

Dissertation with the aim of achieving a doctoral degree (Dr. rer. nat.) at the Faculty of Mathematics, Informatics and Natural Sciences
Department of Informatics
Applied Software Technology
of the Universität Hamburg

submitted by

Daniel Martens

from Geesthacht

Hamburg, 2020
Day of the oral defense: May 20, 2020

Head of the examination commission: Prof. Dr. Chris Biemann

Deputy of the examination commission: Prof. Dr. Mathias Fischer

First evaluator of the dissertation: Prof. Dr. Walid Maalej

Second evaluator of the dissertation: Prof. Dr. Stefan Wagner
Declaration on Oath:
Eidesstattliche Versicherung:

I hereby declare, on oath, that I have written the present dissertation by my own and have not used other than the acknowledged resources and aids.

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Dissertationsschrift selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

__________________________________  __________________________________________
Date, Place                                                     Signature
This thesis is dedicated to my grandfather.
I am deeply grateful to Prof. Dr. Walid Maalej for giving me the opportunity to work with him. Walid, thank you for your time and effort, for continuously motivating me, and for always having an open ear. From you, I learned much more than software engineering. I traveled to conferences around the world and met so many exciting persons from different nations and of different cultures.

It has been my pleasure to work with my colleagues at the Applied Software Technology Group. Thanks to all of you for accompanying me on this journey and for any kind of support. I especially want to thank Timo Johann, who introduced me to the academic world and co-authored my first accepted research paper with me. I thank Christoph Stanik with whom I have taken the last steps of this journey together. Also, I thank all of my students who have given me their trust to supervise their theses.

I am thankful to everyone reviewing my submitted papers. Taking into account your valuable comments, I was able to improve my research significantly.

Finally, I thank my family, my parents and my parents-in-law. This thesis is for my grandfather, whom I promised to complete this challenge. I especially thank my wife, Jovana, for her unconditional love and devotion. I could not have done all this without you. I enjoy every moment with you, and I am so grateful for our daughter Karlotta.
Abstract

With the emergence of app stores and social media, also non-technical users began to frequently and informally communicate with developers. User feedback contains requirements-related information, such as bug reports or feature requests. This information is of great potential to developers as it allows them to develop software systems according to users’ needs. We identified two central problems regarding the quality of user feedback that prevent its automated analysis and, therefore, integration into continuous software evolution processes, such as DevOps. First, in an explorative study, we found that user feedback includes fake information, which might lead to wrong insights about real users’ needs. Second, in an empirical study, we showed that user feedback often misses basic context information, such as the concerned platform, and is therefore likely to be non-actionable to developers.

This dissertation has two main contributions. First, we develop automated approaches to check and increase the quality, i.e., authenticity and actionability, of user feedback. To support developers in understanding real users’ needs, we compare fake to regular app reviews. Based on the identified differences, we develop, optimize, and evaluate a machine learning classifier to detect fake reviews. To assist developers in understanding and reproducing reported user feedback, we propose two complementary solutions to augment the feedback with context information automatically. The first approach augments newly created user feedback with context information implicitly captured during app executions. The second approach is based on a chatbot that mines user-developer conversations to identify and explicitly request missing context items. We describe the implementation of the context extraction approach and evaluate it on a manually labelled truthset.

The second main contribution integrates the quality-improved user feedback into continuous software evolution processes. To support developers in understanding the provided context information, we introduce a crowdsourcing approach. If an anomaly is not immediately visible, such as specific steps within long interaction traces, our approach isolates, i.e., highlights relevant and hides irrelevant, context items. To assist developers in monitoring users’ satisfaction
with evolutionary app changes, we discuss the use of sentiment analysis tools. We identify recurring emotional patterns by applying sentiment analysis tools to user feedback. By comparing these with corresponding app updates, we derive appropriate app release strategies.

We evaluate the identification of fake reviews in an in-the-wild experiment. Given a proportional distribution of fake and regular reviews as reported in other domains, our classifier to detect fake reviews achieved a recall of 91% and AUC/ROC value of 98%. Further, we evaluate the usefulness of context information in real user feedback with 16 experienced iOS developers. In experiments, developers reproduced four real bugs of open-source apps, as described in app reviews. We found that given context information, developers have an improved understanding of and ability to reproduce bugs. Developers require, on average, 46% less time and 78% fewer interactions considering context information. Simulations confirmed that our approach to capture context information automatically does not introduce observable runtime overhead. In interviews, we collected developers’ concerns and possible improvements.
Kurzfassung


Diese Dissertation enthält zwei Hauptbeiträge. Zum einen entwickeln wir automatisierte Ansätze, um die Qualität, d.h. die Authentizität und Verständlichkeit von Nutzerfeedback zu überprüfen und zu erhöhen. Um die Entwickler dabei zu unterstützen, die Bedürfnisse realer Nutzer zu verstehen, vergleichen wir gefälschte mit echten App-Reviews. Basierend auf den identifizierten quantitativen Unterschieden entwickeln, optimieren und evaluieren wir einen maschinell lernenden Klassifikator, der gefälschte Reviews erkennt. Um Entwickler dabei zu unterstützen, das berichtete Nutzerfeedback zu verstehen und zu reproduzieren, schlagen wir zwei sich ergänzende Lösungen vor, um das Feedback automatisch mit Kontextinformationen anzureichern. Der erste Ansatz versieht neu erstelltes Nutzerfeedback mit Kontextinformationen, die implizit während der Anwendungsanfertigung erfasst werden. Der zweite Ansatz basiert auf einem Chatbot, der Benutzer-Entwickler Konversationen analysiert, um fehlende Kontextinformationen zu identifizieren und explizit anzufordern. Wir beschreiben die Implementierung des Ansatzes zur Extraktion von Kontextinformationen und evaluieren diesen auf einem manuell beschriften Datensatz.

# Contents

1 Introduction  
1.1 Problem Statement .................................................. 1  
1.2 Thesis Objectives and Contributions ................................ 4  
1.3 Thesis Scope ............................................................. 7  
1.4 Thesis Outline ............................................................ 7  

I Problem  

2 Foundation  
2.1 Software Evolution ................................................... 12  
2.1.1 Definitions ............................................................... 12  
2.1.2 Issue Tracking Systems ............................................ 14  
2.1.3 Issue Context Information ........................................... 18  
2.1.4 Continuous Software Evolution (DevOps) ........................ 21  
2.2 User Involvement ......................................................... 26  
2.2.1 Definitions ............................................................... 26  
2.2.2 User Involvement in Software Evolution .......................... 27  
2.2.3 Explicit User Feedback ............................................. 29  
2.2.4 Implicit User Feedback ............................................. 32  
2.3 User Feedback and App Store Analysis .............................. 35  
2.3.1 Feedback Classification .......................................... 35  
2.3.2 Sentiment Analysis ............................................... 37  
2.3.3 Release Engineering ................................................. 40  
2.4 Summary ................................................................. 41  

3 A Study of Fake User Feedback ....................................... 43  
3.1 Research Setting ......................................................... 44  
3.1.1 Research Questions ............................................... 44  
3.1.2 Research Method and Data ....................................... 45  
3.2 Fake Review Market .................................................... 46
## Contents

3.2.1 Review Providers and Market Strategies 46  
3.2.2 Offers and Pricing Models 48  
3.2.3 Pretended Fake Review Characteristics 50  
3.3 Discussion 53  
3.3.1 Implications 53  
3.3.2 Limitations and Threats to Validity 55  
3.4 Related Work 56  
3.5 Summary 58  

4 A Study of Context Information in User Feedback 59  
4.1 Research Setting 60  
4.1.1 Research Questions 60  
4.1.2 Research Method and Data 61  
4.2 Context Information in User Feedback 68  
4.2.1 Requirements-Related Feedback 69  
4.2.2 Exchanged Context Items 69  
4.3 Context Extraction Approach 71  
4.3.1 Platform and Device 72  
4.3.2 App and System Version 76  
4.3.3 Classification Results 78  
4.4 Clarification of Missing Context Information 82  
4.5 Discussion 86  
4.5.1 Implications 86  
4.5.2 Limitations and Threats to Validity 87  
4.6 Related Work 88  
4.7 Summary 89  

II Solution 91  
5 Detection of Fake User Feedback 93  
5.1 Research Setting 94  
5.1.1 Research Questions 94  
5.1.2 Research Method and Data 94  
5.2 Fake Review Characteristics 99  
5.2.1 Apps 99  
5.2.2 Reviewers 103  
5.2.3 Reviews 104  
5.3 Fake Review Detection Approach 108  
5.3.1 Feature Extraction 109  
5.3.2 Classification 110  
5.3.3 Optimization 111
### 6 Context Augmentation of User Feedback

6.1 Implicit Context Augmentation
   - 6.1.1 Approach
   - 6.1.2 Implementation Challenges

6.2 Explicit Context Augmentation
   - 6.2.1 Approach
   - 6.2.2 Implementation Challenges

6.3 Discussion
   - 6.3.1 Implications
   - 6.3.2 Limitations

6.4 Related Work
6.5 Summary

### 7 Crowdsourced Isolation of Context Information

7.1 Context Isolation Approach
   - 7.1.1 Issue Validation Phase
   - 7.1.2 Issue Isolation Phase

7.2 Framework Architecture
   - 7.2.1 Client Library
   - 7.2.2 Server Component

7.3 Discussion of Limitations
7.4 Related Work
7.5 Summary

### 8 Monitoring of Evolutionary Software Changes

8.1 Research Setting
   - 8.1.1 Research Questions
   - 8.1.2 Research Method and Data

8.2 User Emotion in App Reviews
   - 8.2.1 Emotion and the Rating, Price, and Content
   - 8.2.2 Emotional Patterns
   - 8.2.3 Release Strategies

8.3 Discussion
   - 8.3.1 Implications
   - 8.3.2 Limitations and Threats to Validity

8.4 Related Work
Contents

8.5 Summary ........................................... 171

III Synopsis ........................................... 173

9 Evaluation ........................................... 175
  9.1 Evaluation Setting ................................ 175
  9.1.1 Evaluation Question ................................ 175
  9.1.2 Evaluation Methods and Data .................. 177
  9.2 Evaluation Results ................................ 182
  9.2.1 Authenticity of User Feedback ................. 182
  9.2.2 Actionability of User Feedback ............... 187
  9.3 Limitations and Threats to Validity .............. 192
  9.4 Summary ........................................ 193

10 Discussion ......................................... 195
  10.1 Limitations ..................................... 195
  10.2 Untampered Feedback on the Blockchain ....... 197
    10.2.1 Ethereum Blockchain ......................... 198
    10.2.2 Decentralized Reviewing Approach .......... 198
    10.2.3 Implementation Challenges .................. 199
    10.2.4 Blockchain Design Trade-Offs ............. 205

11 Conclusion ......................................... 207
  11.1 Summary of Contributions ....................... 208
    11.1.1 Authenticity of User Feedback ............ 208
    11.1.2 Actionability of User Feedback ............ 211
    11.1.3 Isolation of Reported Issues .............. 212
    11.1.4 Monitoring Users’ Satisfaction with App Evolution .... 213
  11.2 Future Work .................................... 214
    11.2.1 Research .................................. 214
    11.2.2 Implementation ............................. 215

Appendices

Appendix A A Study of Fake User Feedback ........ 219
  A.1 Disguised Questionnaire ......................... 219

Appendix B Evaluation ................................ 221
  B.1 Interview ....................................... 221

List of Figures ......................................... 225

List of Tables ......................................... 229
Contents

List of Publications 231
Bibliography 233
Chapter 1

Introduction

This chapter introduces the problems that motivate this thesis, summarizes its contributions, and describes its scope. Finally, it presents the thesis outline.

1.1 Problem Statement

More than four decades ago, Lehman began to formulate eight laws of software evolution. These state that any actively used software system must continually be adapted to remain satisfactory to their users [170]. Lehman describes software evolution as a continual learning experience that is driven by feedback, for example, from the results of the software system under execution as perceived by its users [171]. This feedback is typically managed in issue tracking systems. Issues can be of different types, such as bug reports describing misbehavior of the existing software system or feature requests proposing new functionality. Each issue type requires specific context information and is created using structured templates. Bug reports ask for, e.g., steps to reproduce to help developers understand and fix issues. Issues that contain relevant and correct context information are actionable to developers and can be implemented and scheduled for software updates [312]. After their release, users proceed to provide feedback based on which the adapted software system can continually be improved.

While Lehman’s laws still apply for current software systems [171], the process of eliciting requirements has fundamentally changed in the last decade [181]. Traditionally, bugs were reported by company-internal software testers, and feature requests gathered, e.g., in workshops limited to a small number of users. Later, some software vendors allowed every user to submit feedback by making their issue trackers publicly accessible. This process was tailored towards technically experienced users. With the emergence of app stores and social media, also non-technical users began to frequently communicate with developers.
Chapter 1. Introduction

Nowadays, software vendors are facing masses of user feedback. The popular music app Spotify, for example, received 1.4 million app reviews within the Apple App Store [10], 4.6 million app reviews within Google Play [107], and 1.3 million tweets via its official support channel on Twitter [262] since the initial app release. This corresponds to about 3,500 issues reported daily. Research found that the manual analysis of user feedback is unfeasible [118]. On the other side, meeting the requirements users express in their feedback has become even more critical within today’s highly competitive markets that offer several alternative apps for a single use case [291]. Adapting too slowly to users’ needs has already led to the downfall of apps [303].

Software vendors are aware of this challenge and started utilizing development practices, such as DevOps, to support continuous software evolution. DevOps aims at providing frequent, incremental, and reliable software releases by integrating the development and operation of software systems through a high degree of automation [185, 308]. It applies tools to automate, e.g., software development or deployment [72], and to monitor the operation of the software system. The collected operational measures include, for example, performance metrics (e.g., [136, 177, 213]) or code quality metrics (e.g., [17, 54, 74]). User feedback, although often being considered more important than operational measures [58, 301], is rarely considered in DevOps, possibly due to its informal nature, insufficient quality, and missing solutions for its automated analysis.

To improve this circumstance, a significant recent amount of research studied user feedback on software systems [96, 192, 310]. The results help software vendors, e.g., to automatically classify requirements-related information, such as bug reports or feature requests [224], or to automate release planning by clustering and prioritizing reported issues [291]. For example, researchers found that about one-third of the reviews in app stores include requirements-related information [224] and that more than 30% of the bugs in issue trackers can be discovered earlier by analyzing tweets [200]. Martin et al. provide a comprehensive survey of the research area on app store analysis [192]. Despite these recent advances, we identified several problems that we address in this thesis:

1. **Fake user feedback** leads to wrong insights about real users’ needs and requirements. Apps are targeted by fake user feedback that manipulates ratings and rankings in app stores to gain competitive advantage [88, 190]. App store operators are aware of this problem and prohibit “paid, incentivized, filtered, or fake feedback” in their policies [12, 233]. Despite the significant recent research, none of the works on app store analysis considers fake reviews and their implications. We found that fake reviews include bug reports and feature requests. These might influence the results of existing approaches, e.g., to automate release planning by prioritizing bugs based on their reporting frequency [190].
1.1. Problem Statement

2. **Incomplete context information** results in issues that are non-actionable, i.e., not understandable and reproducible, to developers. The quality of requirements-related user feedback widely differs [119]. Users submit incomplete or incorrect context information, such as wrong steps to reproduce, hindering developers from understanding reported issues [75, 181, 224]. Research has shown that the more details users encounter, the more unlikely they will include such information in their feedback [34, 202]. Especially for non-crashing bugs context information in user feedback is highly relevant, since these can not be automatically captured using crash reporting tools. By analyzing about three million tweets from official Twitter support accounts of popular apps, we found that context information is exchanged in every tenth conversation. More than half of all context items are provided only after the engagement of support teams, possibly to clarify missing information in effortful conversations with users [188].

3. **Unfiltered context information** likewise leads to issues that are non-actionable by developers. Even with complete and correct context information available, developers might be unable to determine single or combinations of multiple items, such as specific interactions within long steps to reproduce, that trigger the reported bug. Research has shown that developers fail to identify erroneous configurations already for a low number of features [198]. To automatically make inferences about the causes of bugs, developers should gather a bit of information from every, i.e., both successful and unsuccessful, software execution [172].

4. **Lack of integrated overview** hinders developers from quickly assessing the impact of evolutionary software changes. Users express their satisfaction with released changes via different channels that were not specifically designed for feedback on software systems. On channels where no ratings are available, such as social media, developers are unable to quickly assess the impact of released changes in order to, e.g., react to unforeseen issues or to derive release strategies. Research has shown that users begin to explore alternative apps when releasing unexpected app changes, such as the removal of app features. Inappropriate release strategies have already led to the downfall of apps [303].

To summarize, user feedback provided via app stores, social media, and user forums is of potentially great value for software evolution. However, the quality and integration of user feedback need to be improved. To be an authentic source, fake reviews need to be identified and removed. Depending on the issue reported, the provided unstructured, informal user feedback needs to be augmented with context information to be actionable, i.e., understandable and reproducible, to developers. Further, the collected context information, for example, in the
Chapter 1. Introduction

form of long user interaction traces, possibly needs to be filtered. Finally, to determine if the released evolutionary software changes meet users’ expectations, developers need integrated possibilities to assess how these are perceived by software users, as expressed across diverse feedback channels.

1.2 Thesis Objectives and Contributions

The goal of this thesis is to qualitatively and quantitatively study user feedback to support continuous software evolution. We aim at improving the quality of user feedback and its integration into software development practices, such as DevOps. While we focus on single users when enhancing its quality, through integrating user feedback, we aim at benefiting from multiple software users. We summarize the contributions of this thesis in the following paragraphs:

1. Automated improvement of user feedback quality.

1.1 Automated detection of fake user feedback. We studied the market for fake app reviews, including providers, their offers, and strategies, through online research and a disguised questionnaire. We created a gold-standard truthset of about 8,000 fake reviews that have been published within the Apple App Store. We compared the fake reviews to all 62 million reviews within the Apple App Store and identified empirical differences between the apps, reviewers, and reviews itself. Based on these differences, we developed a supervised classifier to detect fake reviews automatically. We compared the results of different classification algorithms, optimized the algorithms by feature selection and hyperparameter tuning, and conducted an in-the-wild experiment to evaluate how our classifier performs on imbalanced data. Our experiments show that fake reviews can be detected with a recall of 91% and AUC/ROC value of 98%, for proportional distributions of fake and regular reviews as reported within other domains [178, 203, 277]. Moreover, we discuss the use of blockchain technology as one future direction to avoid fake user feedback.

1.2 Automated augmentation of context information. We aim at supporting both users and developers in exchanging precise context information with the least possible effort. We introduced two complementary approaches for newly created, as well as existing user feedback. First, we developed an in-situ approach to report issues with automatically attached context information, which is implicitly captured during app executions. We highlighted the implementation challenges encountered when realizing our approach on the iOS platform. Second, we developed an automated approach to extract basic context information from explicitly provided unstructured, informal user feedback on mobile apps, including
1.2. Thesis Objectives and Contributions

The platform, device, app version, and system version. Evaluated against a manually labelled truthset of about 3,000 tweets, our approach achieved precisions from 81% to 99% and recalls from 86% to 98% for the different context item types. Combined with a chatbot that automatically requests missing context items from reporting users, our approach aims at auto-populating issue trackers with structured bug reports. We describe the approach implementation as a web app.

2. Integration of user feedback into continuous software evolution.

2.1 Automated isolation of non-crashing bugs. We applied crowdsourcing to isolate, i.e., highlight relevant and remove irrelevant, context items and user interactions to help developers understand and reproduce reported bugs. Our approach identifies recurrent patterns within captured context information used to isolate the occurrence of reported issues. In experiments with professional iOS developers, we showed that these need 30% to 70% less time and fewer interactions with the user interface to understand and reproduce non-crashing bugs when context information is given. In simulations, we confirmed that our crowdsourcing approach does not introduce observable runtime overhead to the application.

2.2 Automated monitoring of users’ satisfaction with evolutionary software changes. To automatically assess users’ satisfaction with software systems as expressed through diverse feedback channels, such as app stores and social media, we applied sentiment analysis tools. These represent users’ emotions as a numeric value and allows software vendors to quickly understand if the introduced evolutionary software changes meet users’ expectations, especially in channels where no ratings are available. Moreover, we applied a sentiment analysis tool to about seven million app reviews. By analyzing users’ emotions expressed in the reviews, we found emotional patterns corresponding to app releases. We derived five release lessons by comparing emotional patterns to the release history, content of user reviews, official vendor presentations, and technical blogs of several apps corresponding to each pattern. We provide actionable recommendations that should encourage and inspire practitioners to consider users’ emotions when fine-tuning release processes.

Figure 11.1 integrates the thesis contributions into continuous software evolution (DevOps) phases. The first two contributions improve the quality, i.e., authenticity and actionability, of user feedback. App developers are enabled to get insights about real users’ needs (contribution A in Figure) and to make use of their feedback by extracting understandable and reproducible issues, including relevant and correct context information (contribution B). The remaining
Chapter 1. Introduction

Figure 1.1: Integration of thesis contributions into continuous software evolution (DevOps) phases.

CONTINUOUS SOFTWARE EVOLUTION (DEVOPS) PHASES

(1) Plan

(2) Code

(3) Monitor

Develop

Software System

Software System

Feature Requests

Bug Reports

User Feedback

Feature Requests

Bug Reports

User

User Feedback

Explicit Feedback

Implicit Feedback

Isolation

Augmentation

of context

information

User Classify

user feedback

Remove

fake feedback

Monitoring of evolutionary changes

SOFTWARE SYSTEM ISSUE TRACKING SYSTEM

Users’ Satisfaction

CONTINUOUS SOFTWARE EVOLUTION (DEVOPS) PHASES

(1) Plan

(2) Code

(3) Monitor

Developer

Bug Reports

Feature Requests

Software System

Software System

Feature Requests

Bug Reports

User Feedback

Feature Requests

Bug Reports

User

User Feedback

Explicit Feedback

Implicit Feedback

Isolation

Augmentation

of context

information

User Classify

user feedback

Remove

fake feedback

Monitoring of evolutionary changes

SOFTWARE SYSTEM ISSUE TRACKING SYSTEM

Users’ Satisfaction

CONTINUOUS SOFTWARE EVOLUTION (DEVOPS) PHASES

(1) Plan

(2) Code

(3) Monitor

Developer

Bug Reports

Feature Requests

Software System

Software System

Feature Requests

Bug Reports

User Feedback

Feature Requests

Bug Reports

User

User Feedback

Explicit Feedback

Implicit Feedback

Isolation

Augmentation

of context

information

User Classify

user feedback

Remove

fake feedback

Monitoring of evolutionary changes

SOFTWARE SYSTEM ISSUE TRACKING SYSTEM

Users’ Satisfaction
two contributions integrate user feedback into the continuous software evolution processes. App users are monitored to isolate the occurrence of reported issues using crowdsourcing (contribution C) as well as to assess their satisfaction with introduced evolutionary software changes (contribution D).

1.3 Thesis Scope

Data Type. This thesis analyzes feedback on software systems, which is mainly provided by non-technically experienced users via app stores, in the form of app reviews, social media, in the form of tweets, or user forums, in the form of posts. We do not focus on feedback within issue tracking systems or similar, created by professional developers and voluntary contributors. Also, we do not analyze other text artifacts, such as software documentation. We focus on analyzing and making use of informal user feedback from the perspective of software developers.

Software Product. Software systems can be of different product types, such as databases that run in the background, or specialized applications, such as software compilers, that are only being used by experts. In this thesis, we focus on user feedback for mobile apps that are being used by a wide range of persons, such as music streaming apps. We expect that these persons have different technical experiences and knowledge about what information is relevant to developers, e.g., to fix reported issues.

Software Lifecycle Phase. Feedback on software systems can be provided in all phases of the software lifecycle [205]. We focus on user feedback that is provided during the software evolution phase [234], i.e., after a software system’s initial release, including updates modifying the software functionality.

1.4 Thesis Outline

The remainder of this thesis is organized in three parts: Part I presents a detailed analysis and discussion of the identified problems. Part II introduces our concepts to solve the problems. Part III presents the synopsis of the thesis.

Part I contains three chapters. Chapter 2 introduces the foundations of software evolution, describes the organization of evolutionary changes in issue tracking systems, highlights the relevance of issue context information, and discusses recent continuous software evolution practices. Further, it describes user involvement in software evolution and separates and introduces explicit as well as implicit user feedback. Finally, it summarizes existing research and approaches on user feedback and app store analysis. The following two chapters
Chapter 1. Introduction

study quality aspects of user feedback. Chapter 3 presents a study on fake reviews in app stores that identifies fake review providers, including their market strategies and offers, as well as pretended fake review characteristics. Chapter 4 introduces and evaluates a search approach that extracts basic context information from unstructured, informal user feedback. It studies the exchange of context information in user-support conversations on Twitter and reports on the effort support teams invest to clarify missing information.

Part II contains four chapters. The first two chapters focus on improving the quality of user feedback. Chapter 5 empirically analyzes the characteristics of fake reviews. It describes the development of a classifier to detect and remove fake reviews automatically. Further, it optimizes and evaluates the performance of different machine learning algorithms on balanced datasets. Chapter 6 presents two complementary automated approaches to augment user feedback with context information. The first approach augments newly created user feedback with context information implicitly captured during app executions. We describe the implementation challenges encountered when realizing the approach on the iOS platform. The second approach augments already existing user feedback using a chatbot that identifies and explicitly requests missing context items. We describe the approach implementation as a web-based app.

The following two chapters focus on integrating the quality-improved user feedback within continuous software evolution processes. Chapter 7 introduces a crowdsourcing approach to isolate the occurrence of reported issues, especially non-crashing bugs. The approach identifies recurrent patterns within the collected information to highlight relevant and hide irrelevant context information. Chapter 8 applies sentiment analysis tools to extract users' emotions from explicitly provided user feedback. It identifies recurrent emotional patterns. From these and corresponding software updates, it derives strategies to release evolutionary software changes.

Part III contains three chapters. Chapter 9 reports on the evaluation of real and context augmented user feedback. It introduces an evaluation framework, including the evaluation questions and methods. Afterwards, it describes the results of the evaluation, including an in-the-wild experiment on imbalanced datasets of fake and regular reviews, experiments on real non-crashing bugs with professional iOS developers, as well as simulations and interviews. Chapter 10 summarizes the limitations of this thesis. Further, it discusses the use of blockchain technologies as one future direction to avoid fake user feedback. Chapter 11 summarizes the contributions of this thesis. Finally, it separates and describes issues for future work by research and implementation.
Part I

Problem
Chapter 2

Foundation

To remain satisfactory to their users, software systems must continually be adapted [170]. Software evolution is driven by feedback, for example, from the results of the software system under execution as perceived by its users [171]. User involvement in software development has been attracting the focus of researchers (e.g., [114, 162]), tool vendors (e.g., [89, 240]), and standardization institutions (e.g., [134, 143]) for about four decades [313].

Studies highlight that most research focuses on the early phases of user involvement in software development while little is known about evolutionary post-deployment user feedback, e.g., provided in the form of app reviews or tweets [73, 223]. Understanding and reacting to this feedback is increasingly becoming important to meet users’ expectations and thereby remain successful in highly competitive markets, such as app stores [160, 291, 303]. At present, developers most often analyze user feedback manually, which requires high effort [223]. Developers highlight the need for tool support to consolidate, structure, analyze, and track user feedback [194, 312]. Its automated analysis is essential, as modern software development practices, such as DevOps, consider the continuous involvement of users as their key capability [256].

This chapter surveys related work of the corresponding research areas. The remainder is organized as follows. Section 2.1 explores different definitions of software evolution, introduces and describes the organization of evolutionary changes using issue tracking systems, and highlights the relevance of context information for developers to understand and reproduce reported issues. Further, it introduces continuous software evolution practices. Section 2.2 describes user involvement in software evolution, in the form of explicitly and implicitly provided user feedback. It emphasizes the need for explicit user feedback to capture non-crashing bugs. Section 2.3 describes existing user feedback and app store analysis approaches. Finally, Section 2.4 summarizes the chapter.
2.1 Software Evolution

In the following, we explore definitions for software evolution and describe the organization of evolutionary changes in issue tracking systems. We separate evolutionary changes by their types and highlight the relevance of context information for developers to understand and reproduce reported issues. Finally, we introduce continuous software evolution practices.

2.1.1 Definitions

Software evolution and software maintenance are often used as synonyms [101]. Both terms refer to efforts invested into a software system after its initial release [45]. Traditionally, any change to a software system following the initial development has been considered as maintenance [173].

In 1973, Boehm reported that about 40% of the development effort is invested in software maintenance [32], which demanded a separation of such efforts through more fine-grained definitions.

In 1976, Swanson defined and divided software maintenance by its types of activities into corrective, adaptive, and perfective maintenance [272]. This terminology has been widely adopted, although it is often defined differently [45]. Swanson defines corrective maintenance as a response to processing, performance, and implementation failures, such as software crashes. Adaptive maintenance responds to change in the data and the processing environments, for example, the logical restructuring of a database. Perfective maintenance eliminates processing inefficiencies and improves maintainability, e.g., using code comments.

In 1992, the definition of the Eureka Software Factory’s European Platform for Software Maintenance (ESF/EPSON) was published [122]. This definition originated from the need for a wider range of definitions for post-deployment activities. It extends the terminology of Swanson, inter alia, by adding evolutive maintenance, which it defines as:

“the activities aimed at adding new functionalities to the software (or modifying existing ones), in response to new or changed functional requirements [...] to meet the evolving and/or expanding needs of the users”

Further, it states that evolutive maintenance is usually triggered by explicit change requests and corrective maintenance by bug reports.

In 1995, the International Organization for Standards/International Electrotechnical Commission (ISO/IEC) 12207 standard [143] for software life cycle processes was published. It highlights maintenance as one of its primary processes. This definition was repetitively revised in subsequent years. In 1998,
it was followed by the Institute of Electrical and Electronics Engineers (IEEE) 1219 standard [134] for software maintenance. In 1999, the ISO/IEC 14746 standard replaced the IEEE 1219 standard, which also describes the software maintenance process of the ISO/IEC 12207 standard in greater detail. The current ISO/IEC 14746:2006 [142] standard includes definitions for the terms corrective, adaptive, perfective, and preventive maintenance.

In 2000, Rajlich and Bennett divided the life cycle of a software system into five distinct stages [234], as shown in Figure 2.1. The stages are initial development, evolution, servicing, phaseout, and closedown. The authors state that evolution in the context of software engineering refers to changes of software systems after the development of their initial version [234]. While in the stage of evolution, developers introduce, possibly significant, changes to software systems to meet user needs, in the servicing stage, only minor defect repairs and simple functional changes are conducted. In the phaseout stage, no changes are made to the software system. When no more revenue is generated, the closedown stage is reached where the software system is withdrawn from the market. The authors state that evolution can be triggered by changing customer demands, competitive pressure, legislative actions, changing business practices, or operating environments.

In 2002, Lehman and Ramil highlighted that software systems that are regularly used can not be specified entirely and implemented at once [171]. Their initial implementation is inevitably followed by further evolution of the software. The authors describe this evolution as a continual learning experience that is driven by feedback. The feedback might contain results from the usage of the software, such as errors as perceived by users.

**Conclusion 1.** In the last four decades, the definitions for software evolution became more fine-grained, indicating the increased effort invested by developers after a software system’s initial release. Software systems can not be specified entirely at once. Instead, it is becoming increasingly important to meet users’ evolving needs to remain successful in highly competitive markets.
Taking into account the previous definitions, we consider a software system in its evolutionary stage [234], as long as users frequently provide feedback. Therefore, in oppose to the ESF/EPSOM definition [122], we consider both the implementation of changes as well as the fixing of bugs, reported within user feedback, as software evolution. We define software evolution as follows.

**Definition 1.** Software evolution refers to changes of a software system after its initial deployment in response to feedback provided by users, including the fixing of bugs and the modification of existing or addition of new functionality.

### 2.1.2 Issue Tracking Systems

Developers typically organize evolutionary software changes using issue tracking systems. These are software systems that help reporting, assigning, tracking, resolving, and archiving issues [28]. An issue usually corresponds to a unit of work to accomplish an improvement in a software system.

Issue tracking systems have extensively been studied. For example, many researchers focused on improving the included information [235, 312]. Other researchers focused on the social aspects and highlighted that issue tracking systems are, instead of solely maintaining lists of issues, a focal point of communication and coordination for many stakeholders [28]. Besides being broadly used in industry, a recent study shows that issue tracking systems are the most frequently used communication channel within open source projects [155].

### Issue Types

Issues can be of different types [224], as shown in Figure 2.2. Change requests can be separated into four types. Feature requests propose new functionality, while improvement requests describe the enhancement of existing functionality. Content requests ask for additional, e.g., music to be added to the app with-
out requiring to add or modify existing functionality. Shortcomings describe a particular aspect the user is dissatisfied with, without proposing a solution [224].

Bug reports inform developers about problems encountered while using the software system [312]. These can be separated into reports about crashing and non-crashing bugs. A sample crashing bug is the unexpected termination of the software system after the user interacts with a user interface element, such as a button. Software systems throw exceptions when crashing bugs occur. These are used by crash reporters to automatically capture context information, such as the affected system version and error logs [89, 274, 276]. The collected information constitutes a crash report. On app crash, the report is sent to a central repository where it can be accessed by developers. Crash reports can be used to auto-populate issue trackers with structured bug reports that include complete and correct context information. These bugs are actionable, i.e., understandable and reproducible, to developers [312]. The information included, such as stack traces, can also be used to identify and cluster reports that relate to the same bug. This allows developers, e.g., to prioritize the importance of bugs based on their frequency reported or to identify the range of affected device models.

Non-crashing bugs, such as incorrectly displayed images or inaccessible user interface elements, do not interrupt the software execution. In contrast to crashing bugs, no exception is thrown to trigger the automated submission of structured reports that include context information.

**Conclusion 2.** For non-crashing bugs, developers mainly rely on manually provided feedback, e.g., by software testers or users. The quality of the provided information, in terms of completeness, i.e., the amount of information included, and correctness, e.g., whether the given information is correct, highly depends on the motivation and experience of the reporter [75, 181, 224, 312].

**Issue Organization**

When creating an issue of a specific type, issue trackers use structured templates that request specific information to be provided by the reporter. Bug reports require context information, such as the affected app version or steps to reproduce, while feature requests require a description of the desired feature.

Figure 2.3 shows an example bug report within the public issue tracker of Mozilla [37]. The issue is separated into several sections. The first section includes the issue’s unique identifier, which is generated by the issue tracking system, a title summarizing the issue, and a status reflecting the progress of the issue. The status *Unconfirmed* (Needinfo) implies that the developers are unable to understand and reproduce the bug, as more information from the reporting user is required. The following sections, titled *Status, People, Tracking, Firefox*
Chapter 2. Foundation

Figure 2.3: Example of a structured bug report regarding the Firefox app for iOS within the issue tracking system of Mozilla [37].
2.1. Software Evolution

Figure 2.4: Typical issue states and workflow within an issue tracking system.

*TrackingFlags*, and *Details*, include more structured information regarding the issue, including the affected product, component, reporting date and user. These sections are followed by the issue comments, which include information in text form. The first comment is written by the reporter when creating the issue. It includes the browser’s user agent, steps to reproduce, actual results, and expected results. The second comment is created by a software developer or tester of the Firefox project asking for a screenshot, as the provided information is insufficient to understand and reproduce the reported issue, which is also reflected by the issue’s current status.

The status of the issue changes, for example, when additional information is provided. Figure 2.4 shows a sample issue lifecycle as used by Mozilla [61]. In the beginning, the issue is reported by a manager, tester, or developer itself. Open source projects, as well as single commercial projects, also allow volunteers to create issues by making their issue trackers publicly accessible. Once the issue is created, its status is *Unconfirmed*. At this point, the issue is centrally accessible to developers and everyone else involved in the project. If a developer or tester is able to understand and reproduce the issue, its status is changed to *New*. If the provided information is insufficient to understand and probably reproduce the issue, the developer requests additional information from the reporter, such as the affected device or system version, and sets the issue state to *Unconfirmed (NeedInfo)*. When the issue could be reproduced, and a developer starts working on it, the status is set to *Assigned*. Developers can either select issues they want to work on by themselves or these are assigned to them, e.g., by project managers. After the developer fixed the bug or implemented the requested changes, the issue status is set to *Resolved*. Resolved issues are verified by testers. If the issue has been solved as expected, its state is set to *Verified*. Otherwise, it is set to *Reopened*, and the solution needs to be reconsidered. After releasing the verified changes, the issue status is set to *Closed*. 
2.1.3 Issue Context Information

Context information supports developers in understanding and reproducing reported issues [312]. Even if developers can guess the user's interactions, an issue might only occur on specific combinations of device model and system version, or might depend on preferences and sensor states, e.g., if the GPS sensor is turned on [105]. Research found that developers fail to identify erroneous configurations already for a low number of features [198].

The presence of context information feedback manually provided by, e.g., software testers or users, is especially relevant for non-crashing bugs, where the automated reporting of structured information using crash reporters is not possible (cf. Section 2.1.2).

In the following, we propose and describe a model to separate context information by the execution context and interaction context, as shown in Figure 2.5.

**Execution Context**

The execution context includes data related to the application-, device-, sensor-, and system-states, see Table 2.1. The application context includes the app’s name, unique identifier, and version numbers. The short version denotes app releases, while the internal version is used to distinguish internal from production builds. The device context includes the device model and orientation. Also, the disk and memory usage are attached as functionality, such as downloads, require sufficient disk space. The battery level discloses bugs that, for example, occur in low power mode. The sensor context reflects if the device sensors, such as Bluetooth, are enabled and active. The amount of sensors is dependent on the device used. The system context captures the operating system name and version. A flag indicates if the system is jailbroken, as some bugs only emerge under this condition [175]. The UI orientation reveals issues that, e.g., only occur in landscape mode.
### 2.1. Software Evolution

Table 2.1: Example of execution context captured from the Spotify iOS app.

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>App</td>
<td>Name</td>
<td>Spotify</td>
</tr>
<tr>
<td></td>
<td>Identifier</td>
<td>com.spotify.client</td>
</tr>
<tr>
<td></td>
<td>Short Version</td>
<td>8.4.17</td>
</tr>
<tr>
<td></td>
<td>Internal Version</td>
<td>686</td>
</tr>
<tr>
<td>Device</td>
<td>Model</td>
<td>iPhone 6</td>
</tr>
<tr>
<td></td>
<td>Orientation</td>
<td>Portrait</td>
</tr>
<tr>
<td></td>
<td>Disk Usage</td>
<td>43/64 GB (67%)</td>
</tr>
<tr>
<td></td>
<td>Memory Usage</td>
<td>0.7/1.0 GB (70%)</td>
</tr>
<tr>
<td></td>
<td>Battery Level</td>
<td>100% (charging)</td>
</tr>
<tr>
<td>Sensor</td>
<td>Bluetooth</td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td>GPS</td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td>Internet (e.g., Wi-Fi)</td>
<td>On (connected)</td>
</tr>
<tr>
<td>System</td>
<td>Name</td>
<td>iOS</td>
</tr>
<tr>
<td></td>
<td>Version</td>
<td>10.3.3 (14G60)</td>
</tr>
<tr>
<td></td>
<td>Jailbreak</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>UI Orientation</td>
<td>Portrait</td>
</tr>
</tbody>
</table>

### Conclusion 3.

In this thesis, we consider the platform, device, app version, and system version as basic context items. We found these to be most frequently exchanged between users and support teams during our manual data exploration of samples of app reviews, tweets, and posts in user forums in Chapter 3.

**Interaction Context**

The interaction context captures the user’s interactions with the application user interface. It is an alternative to error logs, that only exist for crashing bugs and allow to map reported issues to specific areas of the source code.

The interactions are represented as a trace of elements. These elements are differentiated into two types. View elements denote the appearance of app views, such as the settings screen. Event elements represent single user interactions, such as tapping a button. Figure 2.6 shows an example of a user interaction trace within the Spotify app. The trace includes three app view elements. On the first app view $v_1$, the user performs a single event $e_1$ by tapping on the search button. The event results in the appearance of the search app view $v_2$. On this view, the user performs two events, an edit event $e_2$ by typing a string into a text field and a tap event $e_3$ on a table view cell that displays an artist. The artist app view $v_3$ opens, where the user notices a non-crashing bug, i.e., that the font size of the section captions is too large.
Figure 2.6: Example user interaction trace including three app views \(v_i\) and three events \(e_i\) captured from the Spotify iOS app.

The interactions are associated with the captured execution context. To the first view \(v_1\) the entire execution context is attached. The following views \(v_{n+1}\) only include the context information delta, such as changes of sensor states. Each event is associated with two app views, the container app view (e.g., \(v_1\)) and detail app view (e.g., search button) where it was performed.

Table 2.2 lists the captured event types and their specific values. The chosen abstraction level corresponds to the “single widget interaction” [246]. Its monitoring overhead is lower than, e.g., the “UI hardware interaction” level, which monitors every keyboard action.

### Table 2.2: Overview of captured events, as part of the interaction context.

<table>
<thead>
<tr>
<th>Events</th>
<th>Values</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Long) Tap</td>
<td>tap count</td>
<td>User taps a button to open another app view.</td>
</tr>
<tr>
<td>Scroll</td>
<td>content offset ((x, y))</td>
<td>User scrolls through a table view to read single cells.</td>
</tr>
<tr>
<td>Type</td>
<td>text before editing; text after editing</td>
<td>User writes/edits/deletes a text (except for passwords).</td>
</tr>
<tr>
<td>Swipe</td>
<td>swipe direction; number of fingers used</td>
<td>User swipes left to return to the previous app view.</td>
</tr>
<tr>
<td>Pinch</td>
<td>view scale before gesture; view scale after</td>
<td>User zooms into a map view to reveal details.</td>
</tr>
</tbody>
</table>
Definition 2. Issue context information refers to data that helps developers understand and reproduce reported issues, such as non-crashing bugs. It can be separated by the execution context, which includes information related to the application-, device-, sensor-, and system-states, as well as the interaction context, which reflects the user’s interactions with the application user interface.

2.1.4 Continuous Software Evolution (DevOps)

DevOps is a set of practices to integrate the development and operation of software systems through automation, with the aim of providing frequent, incremental, and reliable software releases [72, 185, 308].

The distribution of software systems and their evolution have significantly changed in line with each other [293]. About a decade ago, software systems were distributed to users, for example, on compact discs that were sold within retail markets. The software release cycles were relatively long, with durations of six months to one year [293]. With the widespread availability of broadband Internet access, software vendors began to offer their products as downloads on the Internet. This also enabled them to release changes to their software systems, in the form of software updates, at any time. Previously, software vendors were often only able to update their systems once a year, and new versions possibly introduced fundamental changes at once.

Developers realized this benefit and began to release small, incremental software changes. This allowed them to collect feedback on changes more frequently. Nowadays, updates are distributed through app stores almost weekly [197]. Web-based software systems, such as Facebook, Amazon, or Google, stated to release changes to their users within cycle times of minutes [72, 85].

With shorter release cycles, tool vendors started offering solutions to automate tasks developers have to perform with every release, such as build or continuous integration tools. Moreover, tools emerged that allow the automated monitoring of software systems in the form of operational metrics, e.g., to assess the software performance [51], or to report software crashes [89]. DevOps aims to reduce the manual efforts for developers by applying such tools within the development and operation of software systems.

Definitions

The term ‘DevOps’ was first introduced by Patrick Debois during the DevOps-Days in 2009 [67]. A unique definition does not exist in research or practice [71, 76, 145, 179, 244]. Several researchers proposed definitions or conducted literature reviews to extract a shared definition.

In 2014, Erich et al. [77] described DevOps as a collection of principles and practices that force a close collaboration between persons working in devel-
opment and operations to improve the software deployment and maintenance phase. The authors separate DevOps into the concepts culture, automation, measurement, sharing, services, quality assurance, structures, and standards.

In 2015, Dyck et al. [71] defined DevOps as:

"an organizational approach that stresses empathy and cross-functional collaboration within and between teams – especially development and IT operations – in software development organizations, in order to operate resilient systems and accelerate delivery of changes"

The authors state that the abbreviation ‘DevOps’ refers to the terms ‘development’ and ‘IT operations’. As it is not limited to these areas, the authors suggest refining the term, e.g., into ‘DevSecOps’ or ‘DevNetOps’.

In 2015, Bass et al. [24] proposed defining DevOps as:

"a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality"

In 2015, Smeds et al. [256] conducted a literature review. The authors selected 27 publications to extract a definition of the term ‘DevOps’. The authors state that while the purpose of DevOps is clear, i.e., “bridging the gap between development and operations” as described by Wettinger et al. [300], there exist many interpretations of its actual meaning. The authors highlight that “providing a complete and clear definition of the term is challenging”.

In 2015, Lwakatare et al. [179] conducted another literature review. The authors propose a conceptual framework to characterize DevOps by its problems addressed, elements, and outcomes. According to the authors, DevOps addresses, for example, the problem of poor communication or the manual efforts that need to be performed, e.g., during the deployment of software systems. The authors separate DevOps by the four elements collaboration, automation, measurement, and monitoring. The outcomes of DevOps are summarized as shared responsibilities, i.e., a single team is responsible for an entire service or product, the continuous deployment of functionality and infrastructure, as well as the collection of technical metrics to assess the software system.

In 2016, Jabbari et al. [145] conducted a systematic mapping study on the definitions and practices of DevOps. The authors identified 44 studies that define the term ‘DevOps’. The majority of studies agree that the term has been coined by a combination of ‘Development’ and ‘Operations’. Of the analyzed studies, 15 state DevOps practices. The authors propose the definition:

“DevOps is a development methodology aimed at bridging the gap between Development and Operations, emphasizing communication
2.1. Software Evolution

and collaboration, continuous integration, quality assurance and delivery with automated deployment utilizing a set of development practices.

In 2016, Ebert et al. [72] stated that "quality deliveries with short cycle time need a high degree of automation". The authors describe DevOps as "a culture shift toward collaboration between development, quality assurance, and operations". Instead of having distributed groups that perform functions separately, cross-functional teams work on "continuous operational feature deliveries".

Comparison to Agile Methods. In a literature survey, Jabbari et al. [145] highlighted the differences between DevOps and Agile methods. When planning a software project, agile methods, such as Scrum, utilize a sprint planning to define the work that needs to be performed within the sprint, i.e., a fixed amount of time. During the sprint, communication solely happens within the team [163]. Questions that occur during this period of time are answered through the Scrum master. In contrast, DevOps focuses on continuous planning. It aims to enable direct feedback loops between developers and operators. Similar is the elicitation of requirements, where DevOps suggests the active participation of stakeholders, while Agile methods rely on on-site customer communication and discuss requirements internally amongst developers within sprint review meetings.

Conclusion 4. In the short period of a decade, continuous software evolution practices, such as DevOps, gained the large interest of researchers and practitioners, as frequent, incremental, and reliable software delivery is increasingly becoming important. Several publications and systematic literature reviews attempted to provide a shared definition or tried to describe DevOps by its problems addressed, concepts, or outcomes. While most publications agree that DevOps aims to integrate the development and operation of software systems, the term DevOps, similar to software maintenance, no longer seems to be fine-grained enough. Researchers suggest to further refine the term into ‘DevSecOps’, ‘DevNetOps’, or similar.

Taking into account the previous definitions and focusing on software evolution in general, rather than security- or network-specific aspects, we define DevOps as follows.

Definition 3. DevOps integrates the development and operations of software systems intending to provide frequent, incremental, and reliable software releases. Therefore, DevOps provides a set of practices and principles that stress automation, as well as cross-functional communication and collaboration.
Chapter 2. Foundation

Figure 2.7: Continuous delivery through development and operations loop, i.e., DevOps (source: [221]).

Phases and Automation Tools

DevOps separates the development and operations of software systems into different stages that iteratively repeat each other. Figure 2.7 visualizes a typical DevOps loop, which consists of the eight phases plan, code, build, test, release, deploy, operate, and monitor.

Within each of the phases, DevOps applies tools to automate specific tasks. In addition, the tools continuously gather operational metrics to help developers make data-driven decisions or to adapt the software system automatically. Ebert et al. state that “tools are mandatory in automating DevOps” [72]. Table 2.3 lists several existing automation tools, such as build tools or continuous integration tools, that are being used within different DevOps phases [72]. We summarize several of these tools in the following.

Build tools automate the generation of various artifacts, such as compiling software releases from source code, updating the software documentation, or managing dependencies. Maven [196], for example, is used to build applications from source code. It utilizes XML files to define the build process.

Continuous integration tools are being used within the release phase of the DevOps loop. These aim to prevent integration problems. When a developer submits source code to the version control system, the tool executes pre-defined tasks, such as performing unit tests. Jenkins [146] is a sample continuous integration tool that offers many plug-ins for different use cases.

Monitoring tools are used to observe the software system. These collect operational metrics, which originate from the operation of the software system itself, such as information related to the performance or stability. According to Ebert et al. monitoring tools can further be separated into tools for logging and monitoring [72]. Logging tools collect data of the software systems,


## 2.1 Software Evolution

Table 2.3: Example of automation tools used in DevOps.

<table>
<thead>
<tr>
<th>DevOps Phase</th>
<th>Tool</th>
<th>Language</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plan</td>
<td>Draw.io</td>
<td>JavaScript</td>
<td>Apache</td>
</tr>
<tr>
<td></td>
<td>BugZilla</td>
<td>Perl</td>
<td>MIT</td>
</tr>
<tr>
<td>Code</td>
<td>Git</td>
<td>C</td>
<td>GPL</td>
</tr>
<tr>
<td></td>
<td>GitLab</td>
<td>Ruby</td>
<td>MIT</td>
</tr>
<tr>
<td>Build</td>
<td>Maven</td>
<td>Java</td>
<td>Apache</td>
</tr>
<tr>
<td></td>
<td>Gradle</td>
<td>Java</td>
<td>Apache</td>
</tr>
<tr>
<td>Test</td>
<td>JUnit</td>
<td>Java</td>
<td>EPL</td>
</tr>
<tr>
<td></td>
<td>JMeter</td>
<td>Java</td>
<td>Apache</td>
</tr>
<tr>
<td>Release</td>
<td>Jenkins</td>
<td>Java</td>
<td>MIT</td>
</tr>
<tr>
<td></td>
<td>Bamboo</td>
<td>Java</td>
<td>Commercial</td>
</tr>
<tr>
<td>Deploy</td>
<td>Puppet</td>
<td>Ruby</td>
<td>Apache</td>
</tr>
<tr>
<td></td>
<td>Docker</td>
<td>Go</td>
<td>Apache</td>
</tr>
<tr>
<td>Operate</td>
<td>DataDog</td>
<td>Go</td>
<td>Apache</td>
</tr>
<tr>
<td></td>
<td>Kubernetes</td>
<td>Go</td>
<td>Apache</td>
</tr>
<tr>
<td>Monitor</td>
<td>Crashlytics</td>
<td>-</td>
<td>Commercial</td>
</tr>
<tr>
<td></td>
<td>Nagios</td>
<td>C</td>
<td>GPL</td>
</tr>
</tbody>
</table>

such as error logs, while monitoring tools observe infrastructure aspects, such as the CPU load or RAM allocation. Crash reporters, such as Firebase Crashlytics [89], are considered as logging tools according to the definition. These capture information about software crashes, e.g., in the form of stack traces or context information related to the system or the device.

**Conclusion 5.** Operational metrics are used, e.g., to assess the system performance. Unfortunately, these do not mirror users’ satisfaction with the software and introduced evolutionary changes. Furthermore, specific information, such as non-crashing bugs, can not be automatically captured by operational metrics. Several recent blog posts highlighted the importance of user feedback, such as app reviews or tweets, to DevOps, instead of only considering operational metrics [58, 301]. While user feedback is of potentially great value, only few tools for its automated analysis exist in comparison to operational metrics. This might depend on the problems we identified and address in this thesis, such as low-quality user feedback, i.e., user feedback containing fake information, as well as users missing to provide context information resulting in issues that are non-actionable to developers. This thesis aims to fill the existing gap by providing automated approaches to increase the quality of user feedback and integrating the information contained into DevOps processes.
2.2 User Involvement

In the following, we explore definitions for user involvement and describe how users are involved in software evolution.

2.2.1 Definitions

For the development of software systems, practitioners and researchers first began to consider user involvement as critical at the beginning of the 1960s [23]. In the software engineering and information systems literature, the terms user involvement and user participation are often used as synonyms [22, 43].

In 1980, Olson and Ives highlighted several problems with the existing definitions of user involvement [219]. The authors stated that both the stage and form of involvement is rarely considered. Users can be involved in many stages of the software development life cycle, such as during the system definition or its actual implementation (cf. Figure 2.8). Also, the form of involvement can vary, the authors separate symbolic involvement, where the user input is ignored, and substantive involvement, where the user influences the system design.

In 1989, Barki and Hartwick suggested to unambiguously separate the terms user involvement and user participation [22]. The authors describe user involvement as “a subjective psychological state reflecting the importance and personal relevance of a system to the user” [255]. User participation refers to “a set of behaviors or activities performed by users in the system development process”.

In 1994, Barki and Hartwick refined their definitions and added the term user attitude, which is defined as the affective evaluation of the system by a user [23]. The authors refer to additional studies that separate the forms of involvement into direct or indirect, formal or informal, as well as alone or in groups [176, 295]. For example, open-source projects allow users to make direct changes to the documentation or source code, while closed source projects ask users for their feedback based on which the software system is modified indirectly.
In 1996, Damodaran described the form of user involvement as a “continuum from informative, through consultative to participative” [62] (cf. Figure 2.8). The author describes informative involvement as “users providing and/or receiving information”, while consultative involvement is characterized by “users commenting on a predefined service or range of facilities”. Participative involvement refers to “users influencing decisions relating to the whole system”.

In 2015, Kujala et al. state that, although user involvement has extensively been studied, a clear definition does not exist [161]. The authors highlight that the term has been used “synonymously with focus on users, consulting end-users, contacting with system users and participation of users”.

### 2.2.2 User Involvement in Software Evolution

In this thesis, we focus on user involvement in software evolution, which aims at improving the usefulness and usability of a software system by better understanding users’ needs and expectations [161, 223]. A few years ago, requirements-related feedback was elicited, for example, in field studies, such as interviews or observations, that only involved a limited number of persons [31, 305]. Nowadays, every user can provide post-deployment feedback, e.g., in the form of automatically captured crash reports or manually provided app reviews. As the success of a software system depends on meeting the users’ expectations, this feedback is increasingly important to software practitioners [6, 291, 303].

Several researchers highlighted the benefits of user involvement [162]. Brown stated that the information collected from users is invaluable for decision making [25]. Through user involvement, developers were able to understand the needs of users more precisely [247], and to develop a long term vision for product development [25]. The better understanding resulted in improved levels of acceptance and higher customer satisfaction [62, 245]. Also, users were able to understand the system better, which resulted in a more effective use [62].

Researchers also underlined several challenges developers face when involving users. Bano and Zowghi [21] systematically reviewed the relationship between user involvement and system success. The authors found that user involvement can equally create problems and benefits. For example, the amount of raw data collected can be overwhelming [247]. The analysis of the data is time and labour-intensive [25]. Also, it is challenging to compare subjective data, such as the satisfaction with a specific app feature, across users [25].

Studies showed that most of the research focuses on user involvement in the early phases of software development [73], as this was considered as sufficient to capture users’ needs [73, 162, 313]. While this claim was refuted [20], still little is known about post-deployment user feedback, e.g., provided in the form of app reviews or tweets [223].
Figure 2.9: Analysis model of user feedback, separated by explicitly and implicitly provided feedback.

**Conclusion 6.** Users can be involved in different stages of the software development life cycle, such as during the system definition or its actual implementation, and in many forms, for example, by allowing direct modifications to the source code. While user involvement has extensively been studied within the last four decades, most of the work focuses on the early phases of software development. Post-deployment feedback, e.g., in the form of app reviews and tweets, remains understudied. However, this feedback is increasingly becoming important to meet users’ evolving needs and expectations in highly competitive markets offering several alternative apps for a single use case.

In this thesis, we study user involvement in the form of explicitly and implicitly provided user feedback within the evolutionary phase of a software system. Figure 2.9 gives an overview of the information considered. Explicit feedback is manually provided by users through different channels, such as social media, in the form of reviews, for example, containing information as a title, a description, or a rating. Implicit feedback is automatically captured during app executions, e.g., using crash reporters. It consists of the execution context, containing information related to the application-, device, sensor-, and system-states, as well as the interaction context, reflecting the user’s interactions with the application user interface (cf. Section 2.1.3). Implicit feedback improves the understandability and reproducibility of explicitly reported issues to developers [312]. We describe both explicit and implicit feedback in the following.
2.2. User Involvement

2.2.3 Explicit User Feedback

Users provide app-related explicit feedback through various channels, such as app stores, social media, and user forums in an unstructured manner. This feedback typically consists of a title, description, and a star rating. Studies show that user feedback is of potentially great value to software practitioners. About one-third of the app reviews provided within the Apple App Store include requirements-related information, such as bug reports [224]. We introduce popular feedback channels. For each channel, we depict the requirements to provide feedback, its limitations, and the ability to submit issue context information.

App Stores

In app stores, such as the Apple App Store [10] or Google Play [107], users provide feedback in the form of app reviews, see Figure 2.10. Reviews can directly be provided since the users already registered to download the app. In the Apple App Store, app reviews consist of a 1 to 5-stars rating, a title, and an optional description with a maximum of 5,000 characters. In Google Play, app reviews consist of a 1 to 5-stars rating and a description with a maximum of 500 characters. Both stores do not allow reviewers to attach files.
Chapter 2. Foundation

Figure 2.11: Example of explicit user feedback provided via Spotify’s official Twitter support account.

such as screenshots depicting the reported issue. Also, the descriptions cannot include links to reference external resources. Developers can respond to reviews, enabling the clarification of reported issues [236, 289].

The Apple App Store was launched in July 2008 [11] and is available in 157 countries [18]. The store offers about 2.2 million apps for iPhone, iPad, Apple Watch, and Apple TV as of February 2019 [46]. Until 2016, more than 130 billion apps have been downloaded [15]. Google Play was launched in October 2008 [108] and is available in 145 countries [225]. The store offers about 2.6 million apps as of March 2019 [109]. Until 2016, more than 82 billion apps have been downloaded [304]. The overall number of app ratings and reviews has not been published for any of the stores. Only considering the storefront of the United States, we found that the Apple App Store received 208 million ratings, of which 68 million included a review until March 2017 [190].

Social Media

On social media, such as Twitter, users can address app-related feedback to official app support accounts, as shown in Figure 2.11. These general-purpose accounts are used for questions both regarding desktop and mobile apps, as
2.2. User Involvement

Figure 2.12: Example of explicit user feedback provided via Spotify’s official user forum.

Twitter was launched in 2006, since then, many popular apps created official support accounts, such as Netflix, Snapchat, and Spotify. The Netflix account (@Netflixhelps [212]) was created the earliest in 2009 and received about 1.6 million tweets. The Snapchat account (@Snapchatsupport [257]) was created the latest in 2014 and includes 1.2 million tweets. The Spotify account (@SpotifyCares [262]) was created in 2012 and is the most frequented account with 2.4 million tweets.

User Forums

In user forums, such as the Spotify Community Forums [278], users create topics in different sub-forums, as shown in Figure 2.12. Each topic can contain several posts. For iOS app-related feedback, a single sub-forum exists. Before submitting a post, users need to register. A post is a textual description of an issue offering the ability to attach any kind of file. External resources can be referenced via links. Developers, as well as users, can reply to posts.
Conclusion 7. Context information, such as the affected app version or device model, supports developers in understanding and reproducing reported issues. In case users’ feedback misses relevant context information, these issues might become hard to reproduce [75, 182]. Official app support accounts on Twitter prominently highlight the need for context information when reporting technical issues, such as bugs. For instance, Spotify’s profile includes “for tech queries, let us know your device/operating system” (cf. Figure 2.11), while the Netflix account states “for tech issues, please include device & error”. The presence of context information in user feedback is especially relevant for non-crashing bugs, where the automated reporting of structured information using crash reporters is not possible. Unfortunately, we found that user feedback, including basic context information, such as the concerned platform, device, app version, and system version, is rare [188].

2.2.4 Implicit User Feedback

Implicit feedback is automatically captured using tools that can be integrated into software systems to log structured context information. We introduce crash reporters, record-and-replay approaches, and general logging frameworks.

Crash Reporting Tools

Crash reporting is a conventional research line in software engineering [309]. For example, the Apple Crash Reporter [274] is used to report and analyze app crashes, e.g., on the iOS platform. Figure 2.13 shows the Apple Crash Reporter summarizing captured bugs to developers. On the left side, developers first select one of their apps published within the app store (part A of the figure). Then, developers select the specific app version (part B). Below, a list of crashes appears sorted by their reporting frequency (part C). The selected crash, e.g., affected 782 devices. In the middle (part D), the tool displays the stack traces of the selected crash. On the right, more details are provided (part E), such as the device of the initial reporter, the crash frequency during the last days, and the distribution of occurrences over different devices (iPhone/iPad). Similar tools exist for Android, such as Crashlytics [89].

Both crash reporters do not capture users’ interactions with the application user interface to reproduce their behavior, as for crashing bugs error logs exist that point developers to defective parts of the source code.

Record-and-Replay Tools

Record-and-replay approaches are well known in software engineering research and practice, too. These capture the users’ interactions. Moran et al. [201] present FUSION for Android. The approach allows users to report issues in a
Figure 2.13: Screenshot of a crash reports as displayed within the Apple Crash Reporter (adapted from: [13]).
web-based user interface outside the application. In a guided process, users can manually annotate the automatically captured context data, such as steps to reproduce that are being recorded in the form of screenshots. Narayanasamy et al. [207] propose BugNet, an architecture to continuously record information on production runs. Information captured before the crash can be used to deterministically replay the last instructions. Gomez et al. present Reran [102] for Android, an approach and tool to capture the low-level event stream on the phone, which includes both UI events and sensor events. It is able to replay sophisticated UI gestures and inputs from a variety of device sensors. Using their tool, the authors were able to replay 86 out of the top 100 Android apps on Google Play. Further, they could reproduce bugs in popular apps, e.g., Firefox and Facebook.

Logging Tools

General logging frameworks, such as Google Analytics [106], can be used to track user interactions within apps. By programmatically calling an application programming interface (API), these can be used to track events whenever specific user interface elements are selected or app views appear to create a user interaction trace. However, these tools do not allow to associate captured traces with explicitly provided user feedback containing, e.g., the issue description.

Recently commercial tools have been published that associate explicitly provided user feedback with implicitly captured information. For example, InstaBug [57] allows users to write a textual description of the experienced issue and attach an annotated screenshot, see Figure 2.14. This information is automatically supplemented by, e.g., the device model and system version used.
2.3 User Feedback and App Store Analysis

Researchers and tool vendors have suggested approaches to automatically analyze, filter, and aggregate the large amount of user feedback into actionable insights for developers. Martin et al. provide a comprehensive overview of the research area of app store analysis [192]. In this section, we summarize approaches to classify requirements-related feedback, such as bug reports, to analyze the sentiment expressed in user feedback, i.e., users’ emotions corresponding to evolutionary app changes, and to mine reported issues for release planning.

2.3.1 Feedback Classification

Several studies focused on classifying requirements-related information within explicit user feedback, provided in the form of app reviews. Table 2.4 provides a chronologically ordered list of the works we present.

Iacob and Harrison [131] present an automated tool to extract feature requests from app reviews. The authors manually trained the tool on 3,279 app reviews from 161 apps. When evaluating their tool on 136,998 app reviews, the authors found that 23.3% contained feature requests. The authors extended their approach to also extract bug reports from app reviews by creating linguistic rules [132]. The bug reports are further classified into the sub-categories major, medium, and minor bugs. Major bugs make it impossible to use the app (e.g., “app is not working”). Medium bugs only hinder the user from using one specific function of the app (e.g., “miles do not add right”). Minor bugs to not prevent the user from using any of the apps’ features (e.g., “text overlaps for lower percentages”).

Oh et al. [218] automatically classified app reviews into the categories of bug reports, functional requests, and non-functional requests. Further, the authors...
produced a digest to present the most informative reviews per category.

Pagano and Maalej [224] let two researchers manually annotate 1,100 app reviews with multiple topics. The results show that 13.27% of the app reviews include shortcomings, 10.00% bug reports, 6.91% feature requests, 2.91% content requests, and 1.18% improvement requests. Overall, the authors state that requirements-related feedback of these categories is predominant in about one-third of all user feedback.

Chen et al. [49] present AR-Miner, a framework that can be used to remove noisy and irrelevant app reviews, leaving reviews informative to developers. The authors consider as non-informative reviews four different types, app reviews that describe “pure user emotional expression”, “description of (apps, features, actions, etc.)”, “too general/unclear expression of failures and requests”, as well as “questions and inquiries”. The authors evaluated their framework on a dataset with reviews of four different apps, which overall include 35.1% (24.6% to 55.4%) informative reviews and achieved F-measures from 0.76 to 0.88.

Panichella et al. [227] developed a tool to automatically classify user reviews on a predetermined taxonomy to support software maintenance and evolution. The tool was evaluated on a labelled truthset (created using AR-Miner [49]), including language structure, content, and sentiment features of 1,421 sentences extracted from app reviews. Of these sentences, 218 (15.34%) describe feature requests and 488 (34.34%) problem discoveries. Their tool achieved a precision of 0.85 and a recall of 0.85.

The authors describe feature requests as “sentences related to ideas/suggestions/needs for improving or enhancing the product/service or its functionalities (e.g., we should add a button)”. Problem discoveries are described as “sentences related to issue definitions and unexpected behaviours (e.g., the problem occurs when I try to access to database)”.

Maalej and Nabil [180] develop a tool to classify app reviews into the four categories bug reports, feature requests, user experiences, and ratings (i.e., simple text reflections of the numeric star rating). The authors evaluated different classification techniques on a manually labelled dataset of about 4,000 app reviews from the Apple App Store and Google Play. These reviews include 378 bug reports (9.14%), 299 feature requests (7.23%), 737 user experiences (17.82%), and 2721 ratings (65.80%). To classify bug reports, the authors achieved the best precision of 0.71, recall of 0.72, and F1-score of 0.72 using document classification and natural language processing (i.e., bag of words and stop-word removal). To classify features request, the best scores, i.e., precision of 0.71, recall of 0.79, and F1-score of 0.75, were achieved using a combination of text and metadata (bag of words, stop-word removal, lemmatization, rating, sentiment analysis, and tensiment analysis).
Conclusion 8. Despite the significant recent research on feedback classification in app store analysis, none of the works aims to classify fake reviews or considers their implications. We found that fake reviews include bug reports and feature requests [190]. These might influence the results of existing approaches, e.g., to prioritize bugs based on their reporting frequency, and mislead developers in understanding real users’ needs.

2.3.2 Sentiment Analysis

Sentiment analysis tools use natural language processing (NLP) to extract emotions from text messages [186]. Initially developed for marketing and political opinion mining, sentiment analysis became popular in many domains, including software engineering. Sentiment analysis is applied to user feedback, for example, to extract users’ opinions on app features [119] or to guide release decisions [189]. In the following, we introduce current sentiment analysis tools and research applying these tools to user feedback.

Table 2.5 lists several state-of-the-art sentiment analysis tools, including their approaches, scores, languages supported, technology used, and licenses.

SentiStrength [280] is commonly used as the baseline for emotion classification [216]. It is designed for short informal texts. SentiStrength uses a simple lexicon-based approach to match each text token to dictionaries containing negative and positive words. The dictionaries define sentiment scores for specific tokens, such as “I hate [-4] that u need wifi but overall the app is great [+3]”. SentiStrength also handles abbreviations, intensifiers (e.g., ‘!!!’ or ‘very’), and emoticons (e.g., ‘:-(' or ‘:-)’). For each given input, SentiStrength outputs a negative and a positive sentiment score to be able to reflect on mixed emotions. The positive emotion is on the range from +1 (absence of positive emotion) to +5 (extremely positive emotion), similar the negative score ranges from -1 (absence of negative emotion) to -5 (very negative emotion).

SentiStrengthSE [141] uses a manually adjusted version of the SentiStrength lexicon. Since specific terms, such as ‘bug’, correlate with other emotions in software engineering, the authors aim to improve the classification results by creating a domain-specific lexicon. Further, the authors implement ad-hoc heuristics to correct misclassifications [220].

Senti4SD [40] is a polarity classifier trained on a dataset of manually labelled questions, answers, and comments from StackOverflow. Hence, the training data includes more software engineering specific terms. Senti4SD leverages a suite of features based on n-grams, sentiment lexicons, and semantic features based on word embeddings.

Vader [130] is a lexicon and rule-based sentiment analysis tool which is part of the natural language toolkit (NLTK) [208]. It is fine-tuned for sentiments of microblog-like contexts.
<table>
<thead>
<tr>
<th>Tool</th>
<th>Approach</th>
<th>License</th>
<th>Technologies</th>
</tr>
</thead>
<tbody>
<tr>
<td>SentiStrength</td>
<td>Lexicon-based, ad-hoc heuristics</td>
<td>Command-line (Java)</td>
<td>Lexicon-based</td>
</tr>
<tr>
<td></td>
<td>Estimates strength of positive and negative</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sentiment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SentiStrengthSE</td>
<td>Lexicon-based</td>
<td>Command-line (Java)</td>
<td>Lexicon-based</td>
</tr>
<tr>
<td></td>
<td>Estimates strength of positive and negative</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sentiment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Senti4SD</td>
<td>Lexicon-based, keyword-based, semantic</td>
<td>Command-line (R)</td>
<td>Python package</td>
</tr>
<tr>
<td></td>
<td>features (machine learning, word vectors)</td>
<td></td>
<td>Open source (Apache license)</td>
</tr>
<tr>
<td></td>
<td>Estimates strength of positive and negative</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sentiment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vader NLTK</td>
<td>Lexicon-based</td>
<td>Command-line (R)</td>
<td>Python package</td>
</tr>
<tr>
<td></td>
<td>Estimates strength of positive and negative</td>
<td></td>
<td>Open source (Apache license)</td>
</tr>
<tr>
<td></td>
<td>sentiment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Google Natural Language API</td>
<td>Machine-learning (not further specified)</td>
<td>Remote API</td>
<td>Commercial, closed source</td>
</tr>
<tr>
<td></td>
<td>Classifies emotion into positive, neutral,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>and negative</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM Watson Tone</td>
<td>Machine-learning (not further specified)</td>
<td>Remote API</td>
<td>Commercial, closed source</td>
</tr>
<tr>
<td>Analyzer</td>
<td>Estimates presence and affect of</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>positive, neutral, and negative tones</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.5: Overview of sentiment analysis tools.
Table 2.6: Selected research applying sentiment analysis tools to explicit user feedback.

<table>
<thead>
<tr>
<th>Year</th>
<th>Research</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012</td>
<td>Goul et al.: Managing the Enterprise Business Intelligence App Store: Sentiment Analysis Supported Requirements Engineering [110]</td>
</tr>
<tr>
<td>2013</td>
<td>Hoon et al.: Awesome!: conveying satisfaction on the app store [127]</td>
</tr>
<tr>
<td>2013</td>
<td>Pagano and Maalej: User feedback in the appstore: An empirical study [224]</td>
</tr>
<tr>
<td>2014</td>
<td>Guzman and Maalej: How do users like this feature? a fine grained sentiment analysis of app reviews [119]</td>
</tr>
<tr>
<td>2015</td>
<td>Guzman et al.: Retrieving Diverse Opinions from App Reviews [116]</td>
</tr>
<tr>
<td>2015</td>
<td>Maalej and Nabil: Bug report, feature request, or simply praise? On automatically classifying app reviews [180]</td>
</tr>
</tbody>
</table>

Google Natural Language API [52] and IBM Watson Tone Analyzer [297] are examples for commercial closed-source sentiment analysis tools that can be accessed via a remote API.

Table 2.6 lists recent works that apply sentiment analysis tools to user feedback. Goul et al. [110] applied sentiment analysis tools to app reviews to identify bottlenecks in requirements. Hoon et al. [127] analyzed the sentiment of short reviews and found that it matches the star rating closely. We confirmed this finding in an empirical study on about 7 million app reviews and found a moderate correlation between the sentiment and the star rating [186]. As a result, the sentiment can be used to quickly assess the opinion of users in channels where no ratings are available, such as Twitter. Pagano and Maalej [224] analyzed about one million app reviews. The authors found that positive user feedback is mostly associated with frequently downloaded apps and negative feedback with less downloaded apps. Guzman and Maalej [119] mined app reviews that include app features. The authors grouped reviews discussing the same feature and used the reviews’ sentiments to determine the users’ satisfaction with the specific feature. Chen et al. [49] utilize the sentiment to prioritize app reviews by their informativeness. The authors prioritize app reviews with negative sentiments as more informative. Guzman et al. [116] apply sentiment analysis to group similar app reviews, i.e., those with a similar sentiment score. Maalej and Nabil [180] used sentiments when training a classifier that categorizes app reviews into, e.g., bug reports or feature requests.

**Conclusion 9.** Sentiments are often used only as a supportive feature, e.g., for machine learning approaches to classify requirements-related information. Its integration into continuous software evolution processes, similar to the collection of operational metrics, to understand users’ satisfaction with software systems and evolutionary changes is rarely taken into account.
Table 2.7: Selected research on release engineering based on information extracted from explicit user feedback.

<table>
<thead>
<tr>
<th>Year</th>
<th>Study Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>Henze and Boll: Release your app on Sunday eve: Finding the best time to deploy apps [125]</td>
</tr>
<tr>
<td>2013</td>
<td>Datta and Kajanan: Do app launch times impact their subsequent commercial success? an analytical approach [63]</td>
</tr>
<tr>
<td>2014</td>
<td>Lee and Raghu: Determinants of mobile apps’ success: Evidence from the app store market [167]</td>
</tr>
<tr>
<td>2016</td>
<td>McIlroy et al.: Fresh apps: An empirical study of frequently-updated mobile apps in the google play store [197]</td>
</tr>
<tr>
<td>2016</td>
<td>Martin et al.: Causal impact for app store analysis [193]</td>
</tr>
<tr>
<td>2016</td>
<td>Martin et al.: Causal impact analysis for app releases in Google Play [194]</td>
</tr>
</tbody>
</table>

2.3.3 Release Engineering

Release engineering is the subject of several studies in the area of app store analysis. Table 2.7 lists the studies we present in the following.

Henze and Boll [125] showed that the release date, i.e., day of the week, has an impact on the success of app updates. The authors further showed that version updates were an effective strategy to increase app rankings within the charts of app stores. Datta and Kajanan [63] showed that the release day has an impact on the number of reviews an app receives after its update. Lee and Raghu [167] highlighted that frequent feature updates are the most important factor in increasing the likelihood of staying in the top charts of the Apple App Store. Further, the authors showed that high volumes of positive reviews improve this likelihood. Guerrouj et al. [115] found that high code churn in releases correlates with lower ratings. McIlroy et al. [197] studied the update frequencies of 10,713 apps from Google Play. The authors found that only 1% of the apps were updated weekly, and only 14% were updated in a two-week period. Martin et al. [193] conducted a longitudinal study on 1,033 apps over a time period of 12 months. The apps were both from Google Play and Windows Phone Store. Based on the apps’ release notes, the authors found that those discussing features and not bug fixes led to releases that improve the app ratings more significantly. Martin et al. [194] extended their study by developing a tool and applying it to 38,858 apps from Google Play. Further, the authors conducted interviews with developers of the apps. In the interviews, 33% of the developers responded that they would change their release strategy based on the results identified in the paper, highlighting the need for tool support in release planning.
Conclusion 10. Works focusing on release engineering in the area of app store analysis mostly take an isolated look at the software system and related artifacts, such as their release notes. Only few studies consider the input of users, such as in interviews. None of the works relates explicit user feedback and corresponding information, such as app reviews and their sentiments, to software changes in order to facilitate release engineering.

2.4 Summary

User involvement in the form of post-deployment user feedback is essential for software evolution. Users frequently provide evolutionary feedback via app stores and social media. We summarize our main observations regarding the evolution of software systems and the involvement of users in the following:

- In the last four decades, the definitions for software evolution became more fine-grained, indicating the increased effort invested by developers after a software system’s initial release.

- Continuous software evolution practices, such as DevOps, gained the large interest of researchers and practitioners within the last decade. These intend to provide frequent, incremental, and reliable software releases. Therefore, DevOps integrates the development and operation of software systems through automation using tools and the collection of operational metrics to allow data-driven decisions.

- Operational metrics are, e.g., used to assess the system performance. Unfortunately, these do not mirror users’ satisfaction with software systems and corresponding evolutionary changes, as user feedback does.

- Although the importance of user feedback is recently being highlighted, only few tools for its automated analysis exist that could be used within continuous software evolution practices [58, 301].

- Post-deployment feedback, e.g., in the form of app reviews and tweets, remains understudied [20, 223]. However, this feedback is increasingly becoming important to meet users’ evolving needs and expectations in highly competitive markets offering several alternative apps for a single use case.

- Context information, such as the affected app version or device model, supports developers in understanding and reproducing reported issues. In case users’ feedback misses relevant context information, these issues might become hard to reproduce [75, 182]. The quality of the provided information, in terms of completeness, i.e., amount of information included,
and correctness, e.g., whether given information is correct, highly depends on the motivation and experience of the reporter [75, 181, 224, 312].

- Official app support accounts on Twitter prominently highlight the need for context information when reporting technical issues, such as bugs. For instance, Spotify’s profile includes “for tech queries, let us know your device/operating system” (cf. Figure 2.11), while the Netflix account states “for tech issues, please include device & error”.

- The presence of context information in user feedback is especially relevant for non-crashing bugs, where the automated reporting of structured information using crash reporters is not possible. Unfortunately, we found that user feedback, including basic context information, such as the concerned platform, device, app version, and system version, is rare. This results in low-quality feedback non-actionable to developers [188].

- Another quality aspect is the authenticity of user feedback. As app stores are highly competitive markets and sales and download ranks correlate with app ratings and reviews, the provided user feedback contains fake, e.g., incentivized, information. We found that app stores are targeted by fake reviews that include bug reports and feature requests. Although this might impact the results of existing approaches to mine user feedback, none of the existing works considered fake app reviews and their implications [190].

- User feedback lacks of integration into continuous software evolution practices. Most studies only take an isolated look at the software system and related artifacts, such as their release notes. Mining user feedback and corresponding information, such as the sentiment, to understand users’ satisfaction with software systems and evolutionary changes, as well as to guide release decisions, is rarely considered.

In the following chapters, we focus on the quality, i.e., authenticity and actionability, of user feedback. We perform studies on fake user feedback and on context information exchanged within explicit user feedback. Afterwards, we propose automated solutions to increase the quality of user feedback. Further, we integrate user feedback into continuous software evolution practices, e.g., by using crowdsourcing to isolate the occurrence of reported issues or by determining users’ satisfaction with evolutionary changes to derive release strategies.
Chapter 3

A Study of Fake User Feedback

This chapter presents a study that investigates how and by whom app ratings and reviews are manipulated. The study is based on the first research question of the paper “Towards understanding and detecting fake reviews in app stores” [190] by Martens and Maalej, published in the Empirical Software Engineering Journal in 2019. My contribution to this publication is conducting all research, implementation, and analysis work, as well as leading the writing of the paper. My co-author supervised the work, inspired the research questions and goals, and made final edits to the paper.

Users read through reviews before downloading an app, similar to other online stores. Research found that ratings and reviews correlate with sales and download ranks [88, 123, 192, 224, 271]. Stable numerous ratings lead to higher downloads and sales numbers. As a side effect, an illegal market for fake reviews has emerged, offering services for app vendors to improve their ratings and ranking in app stores.

Fake reviews negatively impact the quality, i.e., authenticity, of user feedback and are prohibited in popular app stores such as in Google Play [233] or Apple App Store [12]. According to app store operators, in regular app reviews, real users are supposed to be triggered by their satisfaction or dissatisfaction of using the app to provide feedback. Fake reviewers, however, get paid or similarly rewarded to submit reviews. They might or might not be real users of the app. Their review might or might not be correct and reflect their opinion. We refer to this type of non-spontaneous, requested, and rewarded reviews as fake reviews.

Recently, Google highlighted the adverse effects of fake reviews in an official statement. Although their effort invested in filtering fake reviews is high, they explicitly ask developers not to buy and users to not accept payments to
provide fake reviews, as not every fake review can be identified [135]. Even governmental competition authorities started taking actions against companies using fake reviews to embellish their apps. For instance, the Canadian telecommunication provider Bell was fined $1.25 million [26] for faking positive reviews to their apps. Vice versa, the CNN app was affected by thousands of negative fake reviews to decrease its rating and ranking within the Apple App Store [41].

The remainder of this chapter is organized as follows. Section 3.1 describes our research setting including the research questions, method, and data. Section 3.2 reports on the results along the research questions. Section 3.3 discusses the finding’s implications as well as the study’s limitations and threats to validity. Section 3.4 lists related work. Finally, Section 3.5 summarizes the chapter.

3.1 Research Setting

In the following, we first introduce the research questions. Then, we describe our research method and data.

3.1.1 Research Questions

We aim to qualitatively and quantitatively understand the market for fake app reviews to reveal how app sales and downloads are manipulated and to which conditions. In our study, we focus on the following research questions:

RQ3.1 By whom are fake reviews offered, and what strategies do fake review providers follow? We identified 43 providers offering fake reviews. We show that developers buy reviews or deal with reviews in exchange portals.

RQ3.2 What specific services do fake review providers offer, and under which conditions? We found that fake review providers offer three services to manipulate app ratings and reviews. Among all services, incentivized installs are the cheapest option to manipulate an app’s number of downloads. Indirectly, these installs should also result in more ratings and reviews. Fake reviews are the most expensive, these include a star rating and review message. Fake ratings do not include a textual description.

RQ3.3 What are providers’ policies for submitting fake reviews, and do these reveal indicators to detect fake reviews? The policies submitted fake reviews must comply with reveal that these do not mean short, low-quality reviews. Fake reviews are written to sound authentic, i.e., including custom keywords or predefined texts, and are submitted by humans.
3.1. Research Setting

3.1.2 Research Method and Data

Figure 3.1 shows our research method, separated into three phases which respectively answer our research questions.

In the first phase, we identified 43 fake review providers by performing a structured manual Google web search. To identify relevant search terms, we initially searched for the phrase ‘buy app reviews’. We extracted search terms suggested by the search engine. For those we repeated the previous step, resulting in 39 unique search terms. Afterwards, we crawled the results of the first ten pages for each search term. We removed duplicate results and classified each result as fake review provider, relevant discussion about fake reviews (e.g., in forums), or irrelevant result. From relevant discussions we extracted additional fake review providers by reading through all sub-pages of the discussions.

In the second phase, we manually extracted the providers’ offers from their websites.

In the third phase, we conducted a disguised questionnaire to collect initial indicators for fake reviews, such as the minimum star-rating and length. The questionnaire was presented as a request for buying fake reviews and sent to all providers per email on 26/04/2017\(^1\). For providers offering users to sign-up as fake reviewers to exchange or get paid for providing fake reviews, we created accounts and extracted the policies submitted fake reviews must comply with.

\(^1\)Conducted with permission of the Ethics Committee of the University of Hamburg.
Chapter 3. A Study of Fake User Feedback

3.2 Fake Review Market

This section describes fake review providers and their market strategies, as well as offers and pricing models. Afterwards, pretended characteristics of fake reviews are summarized based on the results of the disguised questionnaire and analysis of the reviewing policies.

3.2.1 Review Providers and Market Strategies

We identified 43 providers offering fake reviews. These can be separated into two groups by their strategies used to supply reviews.

**Paid review providers** (PRP) accept payments to provide fake reviews. This applies to 34 out of 43 (79%) providers. Users can select a package of, e.g., 50 reviews, specify their app name and identifier, and purchase it via Paypal or similar services. Afterwards, the fake reviews are submitted to the app store.

**Review exchange portals** (REP) allow app developers to sign-up and exchange reviews, as shown in Figure 3.2. The remaining nine providers (21%) belong to this group. After sign-up, developers browse through a list of apps requesting fake reviews. Figure 3.3 shows a sample request for fake reviews. Depending on their policies, review exchange portals ask users to submit fake reviews, e.g., with predefined ratings and review messages. For each fake review the developer submits, one credit is given as a reward. Developers with credits can add their app to the list. Then, the credits are redeemed into reviews written by other developers. In some cases, review exchange portals allow developers...
3.2. Fake Review Market

to buy credits and non-developers to sign-up and submit fake reviews. Non-
developers are rewarded using micropayments, about $0.20 to $1.50 per review.

Figure 3.4 summarizes the strategies fake review providers apply. These can be separated into four phases. First, after deciding to buy fake reviews at a paid review provider or to exchange (or buy) reviews at a review exchange portal, the developer provides basic information, such as the application identifier and whether the reviews should be positive or negative. Optionally, further information, such as keywords to be included within the reviews or predefined review messages, can be submitted. Using this information, the provider creates a review request (cf. Figure 3.3).

Figure 3.4: Reviewing strategies of fake review providers.

In the second phase, review exchange portals publish these requests on internal platforms to recruit fake reviewers. For paid review providers the publishing process is not transparent. Using social investigation and by offering our service as fake reviewer, we identified that at least five providers publish their review requests on invite-only Slack or Telegram channels. By observing the communication within these channels, we found that paid review providers occasionally cross-post review requests on exchange portals while offering micropayments.

In the third phase, fake reviewers can browse through and select review requests. They are presented a review policy that regulates what information or rating the review should include. The fake reviewer submits an appropriate fake review to the app store. As a proof, the fake reviewer uploads a screenshot of the review edit screen showing their rating and review to the provider.

Last, the provider compares if the provided review meets the reviewing policy. If this applies, and the review has been published within the app store, the fake reviewer is rewarded. Reviewers providing reviews that do not meet the policies are excluded from the channels or portals and are not rewarded.

### 3.2.2 Offers and Pricing Models

To increase app downloads and sales, paid review providers offer fake reviews, ratings, and installs. Table 3.1 shows the prices of these offers for both the Android and iOS platforms per identified paid review provider. The table lists the offers’ minimum and maximum price, which varies, e.g., depending on the number of reviews bought.

**Paid fake reviews** are offered by 17 of 34 (50%) providers for iOS and by 32 of 34 (94.1%) for Android. Reviews always include a rating. Among all offers, reviews are the most expensive. The price of a review for iOS is, on average, between $3.41 and $4.24 with a standard deviation from 1.85 to 2.21. The price of a review for Android is less expensive, on average, between $1.73 and $2.14 with a standard deviation from 1.73 to 2.14. The price for iOS is about 97 to 98% higher.

**Paid fake ratings** are offered by 2 of 34 (5.9%) providers for iOS and by 10 of 34 (29.4%) for Android. With this offer fake reviewers rate the app without submitting a written review. The price of a rating for iOS is on average between $1.50 and $2.00 with a standard deviation from 0.01 to 0.71. The price of a rating for Android is on average between $0.76 to $0.83 with a standard deviation of 0.76 to 0.83. Compared to Android the price for iOS ratings is about 97 to 140% higher.

**Paid installs** are generated, e.g., by advertising the app on blogs. Also, users can be paid to install the app. The acquired new app users decide by themselves to rate and review the app. According to our definition these reviews are not considered as fake, as these are not directly requested or paid for. Installs
### Table 3.1: Offers and their prices (in US$) of paid review providers.

<table>
<thead>
<tr>
<th>PRP Co.</th>
<th>Review Price</th>
<th>Rating Price</th>
<th>Install Price</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>iOS Min</td>
<td>Max</td>
<td>Android Min</td>
</tr>
<tr>
<td>IN</td>
<td>1.35</td>
<td>1.50</td>
<td>0.25</td>
</tr>
<tr>
<td>DK</td>
<td>4.63</td>
<td>4.90</td>
<td>0.20</td>
</tr>
<tr>
<td>IN</td>
<td>1.50</td>
<td>1.98</td>
<td></td>
</tr>
<tr>
<td>GB</td>
<td>1.11</td>
<td>1.50</td>
<td>0.10</td>
</tr>
<tr>
<td>US</td>
<td>2.90</td>
<td>2.95</td>
<td>1.28</td>
</tr>
<tr>
<td>RU</td>
<td>0.25</td>
<td>0.25</td>
<td>0.20</td>
</tr>
<tr>
<td>US</td>
<td>6.00</td>
<td>9.00</td>
<td>6.00</td>
</tr>
<tr>
<td>US</td>
<td>3.33</td>
<td>4.17</td>
<td>1.00</td>
</tr>
<tr>
<td>NL</td>
<td>1.55</td>
<td>1.55</td>
<td>0.65</td>
</tr>
<tr>
<td>US</td>
<td>2.50</td>
<td>4.00</td>
<td>0.49</td>
</tr>
<tr>
<td>CA</td>
<td>1.00</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>US</td>
<td>2.15</td>
<td>2.50</td>
<td>0.34</td>
</tr>
<tr>
<td>US</td>
<td>4.30</td>
<td>5.00</td>
<td>0.35</td>
</tr>
<tr>
<td>IN</td>
<td>0.15</td>
<td>0.15</td>
<td>0.08</td>
</tr>
<tr>
<td>RU</td>
<td>2.09</td>
<td>2.99</td>
<td>2.99</td>
</tr>
<tr>
<td>US</td>
<td>5.02</td>
<td>5.20</td>
<td>0.40</td>
</tr>
<tr>
<td>DE</td>
<td>2.50</td>
<td>2.50</td>
<td></td>
</tr>
<tr>
<td>US</td>
<td>8.69</td>
<td>10.00</td>
<td>1.28</td>
</tr>
<tr>
<td>VN</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>US</td>
<td>2.00</td>
<td>2.00</td>
<td>1.40</td>
</tr>
<tr>
<td>US</td>
<td>1.45</td>
<td>2.00</td>
<td>0.29</td>
</tr>
<tr>
<td>RU</td>
<td>3.40</td>
<td>4.00</td>
<td>2.75</td>
</tr>
<tr>
<td>US</td>
<td>1.00</td>
<td>1.00</td>
<td>0.80</td>
</tr>
<tr>
<td>NL</td>
<td>1.78</td>
<td>3.30</td>
<td>0.50</td>
</tr>
<tr>
<td>RU</td>
<td>3.00</td>
<td>3.00</td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>1.99</td>
<td>2.40</td>
<td>0.39</td>
</tr>
<tr>
<td>CN</td>
<td>2.09</td>
<td>2.99</td>
<td>2.39</td>
</tr>
<tr>
<td>SG</td>
<td>3.00</td>
<td>3.00</td>
<td>3.00</td>
</tr>
<tr>
<td>DE</td>
<td>1.93</td>
<td>4.00</td>
<td>0.45</td>
</tr>
<tr>
<td>US</td>
<td>1.00</td>
<td>2.00</td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>0.50</td>
<td>0.50</td>
<td></td>
</tr>
<tr>
<td>AE</td>
<td>0.90</td>
<td>1.00</td>
<td>0.75</td>
</tr>
<tr>
<td>IN</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
</tbody>
</table>

| NUM | 17 | 17 | 32 | 32 | 2 | 2 | 10 | 10 | 12 | 12 | 23 | 23 |
| AVG | 3.41 | 4.24 | 1.73 | 2.14 | 1.50 | 2.00 | 0.76 | 0.83 | 0.48 | 0.55 | 0.33 | 0.40 |
| SD | 1.85 | 2.21 | 1.70 | 0.71 | 0.01 | 0.64 | 0.71 | 0.38 | 0.40 | 0.45 | 0.50 |
are offered by 12 of 34 (35.4%) providers for iOS and by 23 of 34 (67.6%) for Android. Among all offers, installs are the least expensive. The price of an iOS app install is between $0.48 to $0.55 with a standard deviation of 0.38 to 0.40. For Android the price is between $0.33 to $0.40 with a standard deviation of 0.45 to 0.50. Comparing both platforms the price difference is about 37 to 45%.

Overall, the offers are rather expensive, e.g., compared to an average crowdsourcing task or buying followers on Twitter. Ten thousand Twitter followers can be bought for a price of $4 [268]. This might depend on the fact that fake app ratings and reviews are generated manually, e.g., due to a strict moderation by app store operators, while Twitter followers can be generated automatically.

We further tried to identify the popularity of the three different types of offers. Unfortunately, we were only able to extract usage numbers from paid review provider 10 (PRP10) and thus cannot provide generalizable information. PRP10 sold 354,000 offers, of which 20,750 (5.9%) were fake reviews, 29,150 (8.2%) fake ratings, and 304,100 (85.9%) paid installs. We cannot give any numbers on how many paid installs result into a rating or review, and if these are comparable to fake ratings and reviews.

3.2.3 Pretended Fake Review Characteristics

To understand the rules and conditions of providing fake reviews, we conducted a disguised questionnaire with the paid review providers. We also extracted the policies submitted reviews must comply with in review exchange portals.

Disguised Questionnaire

The disguised questionnaire consisted of eleven questions and was presented to the providers in the form of a request to buy fake reviews. A sample question is: “We have several competitors which gain more and more market share. For this reason we are looking for both positive and negative reviews, positive for our apps and negative for our competitors’ apps. [...]”. We decided against open questions as we noticed during a pre-run of the questionnaire, conducted using different identities, that providers returned incomplete answers. The questionnaire is included in the Appendix A.

Eleven out of 34 paid review providers (32.3%) answered our questionnaire. Table 3.2 summarizes their answers. Even upon request, not all providers answered all of our questions. Therefore, the total answers refer to the number of providers that explicitly answered the specific question.

While all eleven providers offer positive ratings and reviews, six also offer negative, e.g., to lower the reputation of competing apps. Regarding the content, six of eight providers reported accepting keywords to be included in their
reviews. Seven of eight providers accept predefined reviews to be submitted by their reviewers. All providers state their reviews are written by humans and not generated using algorithms. Five of ten providers guarantee to replace deleted fake reviews.

Regarding the geographical origin of fake reviews, PRP10 and PRP15 provide reviews from the US. PRP23 and PRP26 additionally provide reviews from Russia. P25 also provides reviews from India. PRP28 specified 13 countries from which the reviews are submitted, these are Austria, Canada, China, France, Germany, India, Italy, Japan, Russia, Taiwan, United Kingdom, United States, and Vietnam. Four providers (PRP9, PRP12, PRP16, and PRP29) reported submitting reviews from all over the world. According to the results, the top three countries are United States (54.5%), Russia (36.4%), and India (18.2%).

Regarding the language, PRP9 and PRP29 reported providing reviews in all languages. PRP10, PRP12, PRP15, PRP25, and PRP28 only provide reviews in English. PRP23 and PRP26 also provide reviews in Russian language. By analyzing all 60,431 fake reviews, initially collected, using LangID, we found that these are written in 70 languages. The five most common are English (87.7%), French (2.3%), German (2.2%), Italian (1.3%), and Spanish (1%).

### Review Policies

For all nine review exchange portals, we were able to extract policies that state the requirements submitted fake reviews must comply with, see Table 3.3. The policies have different levels of detail. Thus, not every requirement is stated by each policy. With the total number, we refer to the policies that explicitly state a requirement.
Table 3.3: Review characteristics extracted from review exchange portal policies.

<table>
<thead>
<tr>
<th>REP</th>
<th>Co.</th>
<th>Real</th>
<th>Dev.</th>
<th>Install</th>
<th>Use App</th>
<th>Keep App</th>
<th>Honest</th>
<th>Rating</th>
<th>Length</th>
<th>Copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IN</td>
<td>Yes</td>
<td>No</td>
<td></td>
<td></td>
<td>1-5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ES</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
<td>5 days</td>
<td>Yes</td>
<td>3-5</td>
<td>&gt;10 words</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>US</td>
<td>Yes</td>
<td>Yes</td>
<td>1-2 days</td>
<td></td>
<td>Yes</td>
<td>2-3 sentences</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>US</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td>1-2 sentences</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>GB</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
<td>1 day</td>
<td>Yes</td>
<td>4-5</td>
<td>1-2 sentences</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>CN</td>
<td>Yes</td>
<td>Yes</td>
<td>4 min</td>
<td></td>
<td>5 days</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>GB</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
<td>1-2 sentences</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>SE</td>
<td>Yes</td>
<td>Yes</td>
<td>2 days</td>
<td></td>
<td>Yes</td>
<td>3-5</td>
<td>&gt;10 words</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>RU</td>
<td>Yes</td>
<td>Yes</td>
<td>10 min</td>
<td></td>
<td>7 days</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Five portals require to use a real device to submit a review. The installation of the app is explicitly required by seven portals. Only two portals request the reviewers to use the app before submitting a review. REP6 requires the reviewer to use the app for at least 4 minutes and REP9 for at least 10 minutes. REP1 explicitly states that the usage of the app is not required. Six portals state that the app should remain installed for a specific period after leaving the review. The minimum amount of time is one (REP3, REP5) up to 7 days (REP9).

Regarding the rating, four providers specify a range the rating should follow. REP2 and REP8 request 3-5 stars, and REP5 4-5 stars. REP1 is the only provider explicitly allowing positive and negative (1-5 stars) ratings. However, reviews from REP1 are not included in our fake reviews dataset. Five providers state that the review should be honest, although three of those allow only positive ratings with at least 3-4 stars. These providers state that reviewers should skip apps if they are unable to submit a positive review.

Regarding the review length, six portals make a statement: two require at least ten words, three portals 1-2 sentences, and one portal 2-3 sentences. Three providers explicitly state the review should not contain content copied from the app description. REP5 and REP9 additionally require sufficiently detailed reviews, e.g., “describe app features instead of providing only praise”.

REP2 allows reviewers to only rate up to ten apps per day. Further, the app should not be immediately reviewed after its installation. Reviewers should randomly download apps from the app store without leaving a review. Before leaving another review, the reviewer should wait a few minutes. Last, reviewers should not only provide 5-star ratings but vary between 3-5 star ratings. REP7 requires the ratings to match review content. Finally, REP9 requires to launch reviewed apps periodically within the next 7 days. A possible reason for this might be to hide the suspicious behavior of quitting to use an app after providing a positive review from app store operators.
3.3 Discussion

Initial Fake Review Indicators

Considering the questionnaire results, the review policies, and taking into account the efforts of providers to disguise fake reviews, we can hypothesize that fake reviews are highly diverse. For example, the rating of fake reviews can be positive or negative. The length of a review can also vary. In addition, the quality of the content may strongly differ. Overall, fake reviews do not mean short, low-quality reviews, as our initial results reveal. These reviews could be either written by paid reviewers (whether or not they have to use specific keywords), or they can be predefined reviews that are written by the app developers and that have to be published by the reviewers.

3.3 Discussion

In modern app stores, developers are for the first time able to publicly retrieve users’ opinions about their software and to compare its popularity in the form of rank or number of downloads. Although app reviews provide a rich source of information, they may not be fully reliable, as customers may leave reviews that do not reflect their true impressions [88].

Our work sheds light on one of these cases: fake reviews. Fake reviews negatively affect the authenticity and, therefore, quality of user feedback. The automated analysis of user feedback and integration into continuous software development practices to improve software evolution is only possible after identifying and removing fake reviews. Otherwise, developers might be mislead when trying to understand real users’ needs.

We discuss the implication of fake reviews on software engineering, as well as from the perspective of app users and app store operators.

3.3.1 Implications

Fake reviews, i.e., paid, incentivized reviews, which can be provided either directly or via fake review providers, are prohibited by official app store reviewing policies. The main reason is to preserve the integrity of app stores [12, 233]. Users that do not trust app stores and their reviews will most likely refrain from providing app reviews themselves. This would harm one of the most important advantages of app stores: collecting real, spontaneous feedback on software in a channel used by both developers and users.

App developers and analysts might be affected by the negative implications of fake reviews. Numerous software and requirements engineering researchers studied app reviews, e.g., to derive useful development information such as bug reports [157, 183] or to understand and steer the dialogue between users and developers [131, 149, 218, 291]. Further, researchers [42, 49, 181] and
more recently tool vendors [8] suggested tools that derive useful information for software teams from reviews, such as release priorities. However, none of these works considers fake reviews and their implications. Negative fake reviews, e.g., by competitors reporting false issues, can lead to confusion and waste of developers’ time. Positive fake reviews might also lead to wrong insights about real users’ needs and requirements.

App users might, by using positive or negative fake reviews, get misled to either downloading an app or not. As shown by Ott et al. [222] fake reviews sound authentic and are hard to detect by humans. In an experiment, humans at most scored an accuracy of 61% identifying fake reviews, even as the word distribution of the used fake reviews differed from regular reviews. We think that this also applies to apps. Users and developers might not be able to identify fake reviews only based on their text.

Measures and tools should enable users (and developers) to identify fake reviews and affected apps. Such tools already exist for products sold on Amazon, e.g., Fakespot [84]. Users enter the name of a product to determine if its reviews are trustworthy. Fakespot also takes a step towards analyzing fake reviews in the app store. The features used to classify reviews as fake also related to the review context, such as if a large number of positive reviews is provided within a short period of time. However, the selected criteria to classify reviews as fake are not transparent nor empirically validated. Also, we assume that no gold-standard dataset has been used for fake reviews. For example, for the Instagram app the site classifies 50% (about 600,000) of the reviews as fake, which raises accuracy concerns for this approach.

App store operators try to prevent fake reviews by providing review policies. The Apple App Store policy states that apps will be removed and that the developers may be expelled from the app store’s developer program “if we find that you have attempted to manipulate reviews, inflate your chart rankings with paid, incentivized, filtered, or fake feedback” [12]. This is the case when app developers buy fake reviews. However, the concrete actions taken to identify fake reviews are non-transparent. We also noticed that larger, popular apps try to prevent negative feedback from being submitted to the app store. These apps ask users to submit feedback within the app in the form of star ratings. The rating is not directly forwarded to the app store. In case of a one-star rating, a mail form appears asking the user to submit the review directly to the app developer instead of forwarding it to the app store, where the review is publicly visible. Such actions might also manipulate the app ratings and reviews as well.

We think that researchers should carefully sample apps and perform data cleanings before studying and mining app reviews. For example, apps with an unusual distribution of ratings might be affected by fake reviews. Similar, reviews of users with an amount or frequency above average might have to
be removed or considered separately during data cleaning. Otherwise, wrong assumptions for the future development of an app could be drawn.

In addition, collecting and analyzing context and usage information can help substantiate decisions and check the quality of reviews [181]. App developers can, e.g., utilize the number of users or the average time users spend with a specific feature to decide which parts of their apps to improve and which suggestions should be taken into consideration in the next release. App store operators can take measures, such as the number of times a user opened an app or the daily app usage time to decide the trustworthiness and weight of reviews within an app’s overall rating.

Instead of limiting the number of users who can participate in the reviewing process, an alternative is to weight or consider incentivized reviews differently and in a transparent manner. Since the overall app store ecosystem is designed that more positive reviews lead to more downloads and thus increase the app’s success [123], developers will likely continue to ask, e.g., friends to rate their apps. Even if incentivized and not independent, such reviews can also include useful information. Instead of excluding the reviews and their reviewers, a possible alternative might be to highlight these reviews with badges (e.g., friend, expert, or crowd testers).

3.3.2 Limitations and Threats to Validity

The structured Google search we conducted was limited to crawling the first ten pages of the identified suggested search terms. By crawling additional pages, the number of fake review providers might increase. Further, service portals, e.g., Fivrr, could have been crawled to identify additional providers.

Also, more fake review providers might be identified using other search engines. Different languages for search results could reveal providers only advertising their services, e.g., in Russian or Indian languages, as these countries are most frequently providing fake reviews according to our collected indicators.

Further, our results are limited to fake reviews provided via the Apple App Store and Google Play. The indicators for fake user feedback might vary on other platforms. Although not focusing on feedback for software products, several researchers studied fake reviews targeting other platforms, such as Twitter (e.g., [68, 87, 269]) or Amazon (e.g., [86, 148]).

The questionnaire we conducted with the paid review providers was hidden as a request for buying app reviews. We cannot assure that the responses only contain true statements. Therefore, we contacted several providers again after a few weeks using a different identity and communication channel, such as Skype. For providers we contacted again, their responses did not change.
Chapter 3. A Study of Fake User Feedback

3.4 Related Work

User feedback is a valuable source for decision making, both to users and developers. Similar to other online stores, before downloading an app, users often read through the reviews. Research found that ratings and reviews correlate with sales and download ranks [88, 271]. Stable numerous ratings lead to higher download and sales numbers. As a consequence, an illegal market that offers fake reviews emerged, to mislead users either download an app or not. Fake reviews are prohibited in popular app stores such as Google Play [233] or Apple App Store [12].

While the phenomena of fake participation, e.g., in the form of commenting, reporting, or reviewing, is well-known in domains such as online journalism [68, 87, 168, 269] or on business and travel portals [86, 148, 204, 222], it remains understudied in software engineering. Despite the recent research on app store analysis, none of the works considers fake app reviews and their implications.

Gyongyi and Garcia-Molina [121] studied a phenomenon similar to fake reviews in app stores, called web spamming. Web spamming is defined as the act of misleading search engines to rank pages higher than they deserve. Therefore, website operators edit their pages, e.g., by repeatedly adding specific terms that improve their ranking in search results. This is comparable to apps competing within app stores today.

Jindal and Liu [148] first analyzed opinion spam, which relates to user-generated content. Its definition is based on web spamming. The authors divide opinion spam into three categories, of which the first category describes untruthful opinions. These mislead readers and opinion mining systems by giving unjust, either positive reviews to promote or negative reviews to damage the reputation of a target object. Untruthful opinions are commonly known as fake reviews. The authors analyze 5.8 million reviews and 2.14 million reviewers from Amazon to detect spam activities and present techniques to detect those. Due to the difficulty to create a fake review dataset, the authors used duplicate and near-duplicate reviews written by the same reviewers on different products.

Ott et al. [222] state that an increasing amount of user reviews is provided. Due to their value, platforms containing user reviews are becoming targets of opinion spam for potential monetary gain – our work confirms this for app stores and provides further insight on the fake review offers and policies. The authors focus on analyzing deceptive fake reviews, which are reviews that have been written to sound authentic, instead of disruptive fake reviews. The authors highlighted that there are few sources for deceptive fake reviews. To overcome the issue, they hired 400 humans using a crowd-sourcing platform to write fake hotel reviews. Their classifier integrates work from psychology and computational linguistics. It has an accuracy of nearly 90% on the crowdsourced dataset.
3.4. Related Work

The authors showed that classifiers are better in recognizing deceptive fake reviews compared to humans, which scored an accuracy of 61% at most.

Feng et al. [86] analyzed fake reviews on TripAdvisor and Amazon. They identified fake reviews based on the hypothesis that for a given domain a representative distribution of review rating scores exists, which is distorted by fake reviews. The authors used an unsupervised learning approach to create a review dataset that is labeled automatically based on rating distributions. Using a statistical classifier trained on that dataset the authors were able to detect fake reviews with an accuracy of 72%.

Mukherjee et al. [204], compared to existing studies, used real fake reviews instead of pseudo-fake reviews, e.g., generated using crowdsourcing platforms. Their dataset consists of fake reviews published on Yelp, filtered and marked by the platform itself. The authors used the supervised approach of Ott et al. [222] on their dataset an achieved a significantly lower accuracy of 67.8%, compared to 89.6%. The authors found that the word distribution of pseudo-fake reviews is different from the word distribution of real reviews. However, this does not apply to fake reviews within their dataset. Instead of using linguistic features, the authors suggest behavioral features. These include numeric values, such as the maximum number of reviews of a reviewer per day, or the review length.

Dickerson et al. [68] present a sentiment-aware architecture to identify Twitter bots. The authors use machine learning features related to the tweet syntax, tweet semantics, user behavior, and network-centric user properties. Using a dataset of the Indian election in 2014, including about 8 million tweets and half a million users, the authors show that the tweet sentiment is key to identify bots. By using the sentiment, the AUC/ROC value improved by about 50%.

Ferrara et al. [87] identified fake user feedback by only using non-textual features. These are related to the user, such as the account creation time or the total number of followers. By using such features, their classifier to detect bots in social networks that, e.g., influence political discussions, achieved better results compared to Mukherjee et al. [204] who mainly use textual machine learning features. The results underline that the metadata of the user feedback, such as information related to the reviewer, is essential to identify fake reviews.

Fake reviews have also been frequently discussed within the media. Streitfeld [267] reported that every fifth review submitted to Yelp is detected as dubious by its internal filters. Instead of removing dubious reviews, these are moved to the second page, where they are read by fewer users. As fake reviews further increase, Yelp began a ‘sting’ campaign to expose businesses buying fake reviews publicly.
Chapter 3. A Study of Fake User Feedback

3.5 Summary

App reviews can be a valuable source of information for software engineers reflecting the opinions and needs of actual users. Potential users read through the reviews before deciding to download an app, similar to buying other products on the Internet. Our work shows that part of reviews in app stores are fake, i.e., they are incentivized and might not reflect spontaneous unbiased user opinions. Fake reviews should mislead users to download an app or not.

In this chapter, we analyzed the market of fake review providers and their fake reviewing strategies. We found that developers buy reviews to relatively expensive prices of a few dollars or deal with reviews in exchange portals. Our initial indicators for fake reviews, that we collected through a disguised questionnaire and by extracting the policies submitted fake reviews must comply with, show that providers invest efforts to disguise fake reviews. Moreover, these show that fake reviews do not mean short reviews. Fake reviews are written to sound authentic, i.e., including custom keywords or predefined texts that discuss actual app features, and are submitted by humans.

Fake reviews negatively impact the quality, i.e., authenticity, of user feedback. These might influence the results of existing app store analysis approaches, e.g., to prioritize feature requests based on their frequency reported. The results might deceive developers trying to understand real users’ needs. Before integrating approaches for the automated analysis of user feedback into continuous software evolution practices, fake reviews need to be identified and removed.
This chapter presents a study that empirically analyzes the user feedback of three popular apps exchanged via their official Twitter support accounts. The study is based on and extends the paper “Extracting and Analyzing Context Information in User-Support Conversations on Twitter” [188] by Martens and Maalej, published at the 27th IEEE International Requirements Engineering Conference in 2019. My contribution to this publication is conducting all research, implementation, and analysis work, as well as leading the writing of the paper. My co-author supervised the work, inspired the research questions and goals, and made final edits to the paper.

Modern apps include options to support users in providing relevant, complete, and correct context information when reporting bugs or requesting features. For example, Facebook attaches more than 30 context items to bug reports submitted via their apps, including the app version installed and the device in use [83]. Despite the presence of such options, an increasing amount of users still reports their issues via, e.g., social media, such as Twitter. A possible reason might be to increase the pressure on software vendors through the public visibility of reported issues. Research has shown, that mining tweets allows additional features and bugs to be extracted, that are not reported in official channels as app stores [211]. Mezouar et al. found that one-third of the bugs reported in issue trackers can be discovered earlier by analyzing tweets [200]. Many app vendors are aware of these benefits and have thus created Twitter support accounts as @Netflizhelps, @Snapchatsupport, or @SpotifyCares.
Chapter 4. A Study of Context Information in User Feedback

Compared to structured reports in issue trackers that usually include context information [29, 312], feedback on, e.g., Twitter is primarily provided by non-technical users in a less structured way [200].

Tweets that miss basic context items, such as the concerned platform, are considered of low quality, since these are likely to be non-actionable to developers [198]. Hence, several support accounts prominently highlight the importance of this information in their Twitter bio. For instance, Spotify’s profile includes “for tech queries, let us know your device/operating system”, while Netflix states “for tech issues, please include device & error”. However, tweets, such as

“I can’t open playlists shared via WhatsApp on my iPhone XR, iOS 12.1.4, Spotify 8.4.61”

that include all basic context items, i.e., the concerned platform, device, app version, and system version, are rare. In contrast, support teams engage in recurring, effortful conversations with users to obtain missing information.

The remainder of this chapter is organized as follows. Section 4.1 describes our research setting including the research questions, method, and data. Section 4.2 compares the exchange of requirements-related feedback within different feedback channels for a sample app release. Further, it studies user-support conversations to identify the context items most relevant to developers. Section 4.3 introduces an approach to extract basic context information from unstructured, informal user feedback on mobile apps, including the platform, device, app version, and system version. Section 4.4 empirically analyzes the context information exchanged in user feedback via official Twitter support accounts of three popular apps by applying the context extraction approach. It studies the effort support teams invest in clarifying missing context items. Section 4.5 discusses the implications of our findings as well as the study’s limitations and threats to validity. Section 4.6 lists related work. Finally, Section 4.7 summarizes the chapter.

4.1 Research Setting

In the following, we first introduce the research questions. Then, we describe our research method and data.

4.1.1 Research Questions

Our study aims at understanding the effort support teams invest in clarifying missing context items in informal conversations with users, specifically for issues reported via social media. We focus in particular on three research questions:
4.1. Research Setting

RQ4.1 What information is exchanged within app stores, social media, and user forums? We analyze the user feedback of a sample Spotify app release, provided via the Apple App Store, Twitter, and the app’s official user forum. We found that users provide most feedback via app stores, since the effort to write an app review is the lowest. While channels that required higher efforts receive less feedback, their density of requirements-related issues is higher. In our sample, 15% of the user feedback in the Apple App Store reports bugs, compared to 58% in the official user forum. By analyzing these, we found that 98% of the reported bugs are non-crashing. We found that the context items platform, system version, app version, and the device model are most often exchanged for non-crashing bugs, however often only after the request of developers.

RQ4.2 How well can basic context items automatically be extracted from tweets? We introduce a simple unsupervised approach that uses pre-defined keyword lists, word vector representations, and text patterns to extract basic context items from tweets, including the platform, device, app version, and system version. The results allow to identify issues potentially actionable to developers or requiring further clarification. Evaluated against a manually labelled truthset of about 3,000 tweets, the approach achieved precisions from 81% to 99% and recalls from 86% to 98% for the different context item types.

RQ4.3 How much effort do support teams invest in clarifying missing context items in user-support conversations on Twitter? We apply our approach to about three million tweets from support accounts of the three popular apps Netflix, Snapchat, and Spotify. By analyzing the results, we found that users and support teams exchange context information related to iOS or Android in about every tenth conversation. More than half of all extracted context items are provided only after the engagement of support teams. Support teams participate in about 40% of the conversations including context items until relevant items are present.

4.1.2 Research Method and Data

We describe our research method including the data collection, data preparation, truthset creation, and data analysis phase, as shown in Figure 4.1.
Chapter 4. A Study of Context Information in User Feedback

Data Collection Phase

In the data collection phase, we crawled tweets using the Twitter Search API [284] in January 2019. We refer to this data as crawled dataset.

For our study, we collected tweets of the official Netflix, Snapchat, and Spotify support accounts. For each account, we used the search query ‘q=account-name&f=tweets&max_position=’ to crawl the tweets. The query parameter q is set to a combination of the @-symbol and the account name {Netflixhelps, Snapchatsupport, SpotifyCares}. Thereby, we only consider tweets directly addressed to the support accounts (cf. Figure 4.2). We do not crawl tweets that solely use related hashtags (e.g., “Listen to my #spotify playlist [...]” or “Today, relaxed #netflix sunday!”). The type parameter f is set to ‘tweets’ to receive all tweets addressed to the support accounts in temporal order, instead of only the top tweets as per default. The pagination parameter max_position is set to the identifier of the last tweet received, as the API returns a fixed amount of 20 tweets per request. For each tweet, we extracted the identifier (id), text, creation date, conversation id, reply flag, as well as the author’s name and id.

Each tweet can result in a conversation which possibly contains responses written by the support team, by users facing similar issues, or by the reporting user (cf. Figure 4.2). To extract these responses, we additionally crawl each of
4.1. Research Setting

Figure 4.2: Example of a conversation between user and support team on Twitter to obtain missing basic context items (i.e., the device and platform).

Table 4.1 summarizes the crawled dataset by the support accounts. The Netflix account (@Netflixhelps) [212] was created the earliest in February 2009 and exists for about ten years. For this account, we crawled 1,643,281 tweets by 385,935 users. These tweets result in 686,488 conversations (about 2.4 tweets per conversation). The Snapchat account (@Snapchatsupport) [257] was created the latest in March 2014 and exists for about five years. We crawled 1,164,824 tweets by 422,643 users. These result in 612,645 conversations with about 1.9 tweets per conversation. The Spotify account (@SpotifyCares) [262] was created in February 2012 and exists for about seven years. For this account, we crawled 2,446,864 tweets by 491,282 users, resulting in 892,441 conversations (about 2.7 tweets per conversation).

The most frequented support account is Spotify with about 30 tweets and 11 conversations created per hour. Netflix and Snapchat are comparable with 19 tweets and 8-10 conversations per hour. The most active support team is Spotify with 1,256,465 (51.35%) of the crawled tweets in all conversations created. Netflix created 752,951 (45.82%) of the tweets, while Snapchat is least active with only 303,087 (26.02%) tweets.

Overall, the crawled dataset includes 5,245,969 tweets within 2,191,574 conversations, written by 1,299,860 users.
Table 4.1: Key figures of the crawled and cleaned datasets.

<table>
<thead>
<tr>
<th></th>
<th>Netflix</th>
<th>Snapchat</th>
<th>Spotify</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Crawled dataset</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Tweets</td>
<td>1,643,281</td>
<td>1,164,824</td>
<td>2,446,864</td>
<td>5,254,969</td>
</tr>
<tr>
<td># Users</td>
<td>385,935</td>
<td>422,643</td>
<td>491,282</td>
<td>1,299,860</td>
</tr>
<tr>
<td># Conversations</td>
<td>686,488</td>
<td>612,645</td>
<td>892,441</td>
<td>2,191,574</td>
</tr>
<tr>
<td>Account created</td>
<td>02/2009</td>
<td>03/2014</td>
<td>02/2012</td>
<td>n/a</td>
</tr>
<tr>
<td>Tweets per hour</td>
<td>18.71</td>
<td>18.73</td>
<td>29.53</td>
<td>22.32</td>
</tr>
<tr>
<td>Conversations per hour</td>
<td>7.82</td>
<td>9.85</td>
<td>10.77</td>
<td>9.48</td>
</tr>
<tr>
<td>Tweets by support</td>
<td>45.82%</td>
<td>26.02%</td>
<td>51.35%</td>
<td>41.06%</td>
</tr>
<tr>
<td><strong>Cleaned</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Tweets</td>
<td>708,729</td>
<td>350,550</td>
<td>1,585,456</td>
<td>2,644,735</td>
</tr>
<tr>
<td># Users</td>
<td>196,609</td>
<td>92,955</td>
<td>384,802</td>
<td>674,366</td>
</tr>
<tr>
<td># Conversations</td>
<td>219,643</td>
<td>101,665</td>
<td>477,668</td>
<td>798,976</td>
</tr>
</tbody>
</table>

To answer the first research question, we additionally extracted user feedback for the Spotify iOS app provided via the Apple App Store and the app’s official user forum. We chose the release 8.4.17, which was available in the Apple App Store for seven days, from 05/09/2017 until 11/09/2017. Within this time frame, we extracted the user feedback submitted via both channels manually. From the Apple App Store, we collected 3,025 app reviews from all available store fronts, i.e., different languages. From the official user forum, we only extracted 178 posts as we specifically focused on the iOS sub-forum.

**Data Preparation Phase**

In the data preparation phase, we removed incomplete conversations and conversations including multiple users. Then we re-created the conversations. Further, we removed non-English conversations and preprocessed the tweet texts. We refer to this data as *cleaned* dataset.

By manually exploring a random sample of 100 conversations from the crawled dataset, we observed that a large number of conversations were crawled incompletely. We were unable to crawl the missing data using the Twitter Search API. Even when manually accessing the conversations on the Twitter website, these were displayed incompletely. Possible reasons we identified are users that set their Twitter account to private or permanently deleted it. Tweets of these users are replaced by the message ‘This tweet is unavailable.’, however the corresponding conversation remains on Twitter, including only the tweets of the support account. To improve this circumstance, we excluded conversations from the dataset which begin with a tweet written by the support account or are flagged as a reply, indicating that the conversation is incomplete.

During the manual exploration, we further observed that conversations in-
including multiple users often request content, i.e., songs or movies, or are official announcements, such as ‘Spotify Premium, now with Hulu on us.’ Therefore, we decided to remove multi-user conversations from our dataset.

Also, we found that a huge amount of conversations between users and the support account are split into parts. A reason we identified are inexperienced users that do not use the reply functionality but create a new tweet outside the conversation context. Also, we found responses not provided within a threshold of about one day to be handled as a new conversation by Twitter. For this reason, per user we selected all tweets of conversations the user participates in, and sorted these in temporal order. To re-create conversations, we set a threshold of 7 days and grouped tweets successively provided within the threshold into separate conversations.

Afterwards, we determined the language of the tweets using the LangID library [264] and removed conversations including non-English tweets. Then, we preprocessed the tweet texts by converting it into lowercase. Further, we removed linebreaks, double whitespaces, and mentions of support account names.

Table 4.1 describes the resulting cleaned dataset. The Netflix dataset includes 708,729 (43.13%) of the crawled tweets. Of the initial tweets, 40.06% were removed while filtering conversations that are incomplete or include multiple users, and further 16.81% by removing non-English conversations. These tweets were provided by 196,609 (50.94%) of the initial users in 219,643 conversations (about 3.2 tweets per conversation). The Snapchat dataset includes 350,550 (30.09%) of the crawled tweets, 69.63% were removed as part of incomplete and multi-user conversations, and further 0.28% due to non-English conversations. These tweets were provided by 92,955 (21.99%) of the initial users in 101,665 conversations (about 3.5 tweets per conversation). The Spotify dataset includes 1,585,456 (64.80%) of the crawled tweets, 32.93% were removed while filtering incomplete conversations or those including multiple users, and further 2.27% by removing non-English conversations. These tweets were provided by 384,802 (78.33%) of the initial users in 477,668 conversations (about 3.3 tweets per conversation).

Overall, the cleaned dataset includes 2,644,735 tweets (50.33% of the crawled dataset) in 798,976 (36.46%) conversations, written by 674,366 (51.88%) users.

Truthset Creation Phase

To be able to evaluate how well our approach extracts basic context items from tweets, we created a truthset including labelled tweets of the Netflix, Snapchat, and Spotify support accounts from the cleaned dataset.

To create the truthset, we use the tool doccano [44], an open-source text annotation tool that can be used for, e.g., named entity recognition or sentiment analysis tasks. It can be deployed on a local or remote machine and offers rich
Chapter 4. A Study of Context Information in User Feedback

Figure 4.3: Doccano tool used to manually annotate basic context information within tweets of the truthset.

functionality, such as user management. Using the tool, two human annotators performed a sequence labelling task by assigning the labels ‘Platform’, ‘Device’, ‘App Version’, and ‘System Version’ to sequences within the tweets, as shown in Figure 4.3. We decided for the labels, as we identified these context items as most often exchanged on Twitter when analyzing user-support conversations of a sample app release within our first research question.

We started from a random sample of conversations which resulted in truthsets nearly including no context items, being unusable to measure the performance of our approach. Thus, we changed the sampling strategy and searched for conversations including the keyword ‘app’. The labelled context items were often referring to platforms such as desktops or smart TVs, which we do not consider in this study. To select tweets including relevant context items, we only consider conversations containing the words ‘iOS’ or ‘Android’ in at least one of their tweets, even though this introduces the bias of more platforms being mentioned within the truthset. From the extracted conversations, we randomly selected as much for each account to contain about 1,000 user tweets. We removed tweets written by the support teams as our approach is designed to extract context items from users’ feedback. Further, user tweets include more context items and are needed to determine how our approach performs on informal language, e.g., referencing the device ‘iPhone 6 Plus’ by the alternative spelling ‘iphone6+’.

In case of disagreements between the two coders, a third annotator resolved the conflicts which resulted mainly from different sequence lengths due to including additional information, such as the device manufacturer or system architecture (e.g., ‘8.4.17’ vs. ‘8.4.17arm7’). We calculated the inter-coder reliability using Cohen’s Kappa on a scale of 0-1 [55]. Per tweet of the truthset, we compared if the two coders agree or disagree that it includes context items. As
4.1. Research Setting

Table 4.2: Key figures of the truthset.

<table>
<thead>
<tr>
<th></th>
<th>Netflix</th>
<th>Snapchat</th>
<th>Spotify</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td># Conversations</td>
<td>410</td>
<td>410</td>
<td>200</td>
<td>1,020</td>
</tr>
<tr>
<td># User Tweets</td>
<td>1,005</td>
<td>1,004</td>
<td>1,005</td>
<td>3,014</td>
</tr>
<tr>
<td>(incl. Context)</td>
<td>379 (37.71%)</td>
<td>453 (45.12%)</td>
<td>284 (28.26%)</td>
<td>1,116</td>
</tr>
<tr>
<td># Context Items</td>
<td>546</td>
<td>736</td>
<td>558</td>
<td>1,840</td>
</tr>
<tr>
<td># Platform</td>
<td>311</td>
<td>416</td>
<td>204</td>
<td>931 (50.60%)</td>
</tr>
<tr>
<td># Device</td>
<td>168</td>
<td>164</td>
<td>156</td>
<td>488 (26.52%)</td>
</tr>
<tr>
<td># System Version</td>
<td>56</td>
<td>130</td>
<td>109</td>
<td>295 (16.03%)</td>
</tr>
<tr>
<td># App Version</td>
<td>11</td>
<td>26</td>
<td>89</td>
<td>126 (6.85%)</td>
</tr>
</tbody>
</table>

suggested by Landis and Koch, we considered the ranges 0.61-0.80 as ‘substantial’ and 0.81-1.00 as ‘almost perfect’ [164]. The kappa agreement among the two coders is 0.933.

Table 4.2 summarizes the truthset. It includes 3,014 tweets, of which 1,005 are tweets from the Netflix support account, 1,004 tweets from Snapchat, and 1,005 tweets from Spotify. Of these, 1,116 (37.03%) tweets include context information. The tweets include an overall amount of 1,840 context items (about 1.65 items per tweet), of which 931 (50.60%) mention the platform, 488 (26.52%) refer to the device, 295 (16.03%) indicate the system version, and 126 (6.85%) the app version.

Data Analysis Phase

The data analysis phase consists of four steps. To answer what information is exchanged in app stores, social media, and user forums (RQ4.1), we study the user feedback of a sample Spotify app release. We found that the context items most often exchanged are the platform, device model, app version, and system version. We consider these as basic context items within our study.

To answer how well basic context items can be automatically extracted from tweets (RQ4.2), we apply our approach to the truthset including labelled tweets of the Netflix, Snapchat, and Spotify support accounts. We measure the approach performance by comparing its output to the results of the human annotators. Considering all support accounts, the approach achieved precisions from 81% to 99% and recalls from 86% to 98% for the different context item types. This allows us to use the approach for answering the follow-up research question while assuming a moderate error margin in the data analysis.

To determine the effort invested by support teams to clarify missing context information in user-support conversations (RQ4.3), we apply our approach to the cleaned dataset and run a summative analysis on the extracted information.
Chapter 4. A Study of Context Information in User Feedback

4.2 Context Information in User Feedback

To understand the exchange of context information within different feedback channels, we empirically analyze the user feedback of a popular sample app. We chose the Spotify iOS app (release 8.4.17), which was available in the Apple App Store for seven days, from 05/09/2017 until 11/09/2017. For this app release, we analyzed the user feedback provided via the Apple App Store, as well as the app’s official Twitter support account and user forum, more specifically the iOS sub-forum.

Per feedback channel, we classified the requirements-related feedback to identify the number of bugs reported. We separated those bugs into crashing and non-crashing bugs. Last, we analyzed the dialogue between users and support teams within non-crashing bug reports. We manually studied the context items exchanged to show the time and effort developers have to invest to receive complete and useful information.

In the Apple App Store [10], users provided 3,025 app reviews in 56 different languages within the period of the chosen app release. 1,686 reviews (55.74%) are written in English language. None of the reviews received a response. On the official Twitter support account of the Spotify app [262], 8,058 tweets were exchanged of which 4,345 were written by users and 3,713 by developers. These tweets constitute 2,856 conversations, with an average of 2.82 tweets per conversation. More than 99% of the tweets are written in English language. For further analysis, we removed conversations not including at least one of the keywords ‘ios’, ‘iphone’, ‘ipad’, or ‘apple’, resulting in 132 conversations with 568 tweets (4.3 tweets per conversation). In the Spotify Community Forums [278] for the selected release 178 posts were created in the iOS sub-forum, of which 162 were written by users and 16 by developers. These posts constitute 67 conversations (2.66 posts per topic). All posts are written in English language.

Comparing all channels, the Apple App Store received the highest number of feedback. However, developers did not reply to reviews. The most interactive channel in terms of responses per feedback is Twitter, with 4.3 tweets per conversation. This may be influenced by notifications Twitter sends to users when receiving replies. In contrast, users of the Apple App Store need to manually check their comments for replies and therefore possibly browse through several pages of app reviews since those appear in fixed temporal order. In the forum least feedback is provided. This might depend on the need to solely register in order to provide feedback, while Twitter accounts can be used for other occasions. From the results, we assume that the lower a channel’s barrier to entry the more feedback users provide.
4.2. Context Information in User Feedback

Figure 4.4: Distribution of requirements-related issues within different feedback channels.

4.2.1 Requirements-Related Feedback

To determine the amount of requirements-related feedback we classified each issue, i.e., app review in the Apple App Store, conversation on Twitter, and topic in the Spotify Community Forums, into one of the categories bug report, content request, feature request, improvement request, shortcoming, and other [224]. Two human annotators coded the user feedback. The annotators were provided with a coding guide. In case of mismatches, which applied for about 4% of all cases, a third annotator resolved the conflicts.

Figure 4.4 shows an overview of the requirements-related issues per channel. Most issues were reported in the Apple App Store in the form of 476 app reviews. On Twitter, 131 requirements-related conversations, consisting of 564 tweets, were published. Requirements-related conversations are more active with 4.3 tweets, compared to regular with 2.82 tweets. This may indicate the need for additional clarification, when developers try to reproduce bug reports that miss context information. Least issues were reported in the iOS sub-forum of the Spotify Community Forums, in the form of 67 topics with 178 posts.

Overall, 165 (24.48%) of the 674 requirements-related issues provided through all three channels report bugs. Nearly all (98%) of these bugs are non-crashing (cf. Figure 4.5). By channels, 72 non-crashing bugs were reported in the Apple App Store, 52 on Twitter, and 39 in the Spotify Community Forums. Comparing the relative numbers per channel, most bugs (58.21%) were reported in the user forum, followed by Twitter (40.46%) and the Apple App Store (15.36%). This indicates that the more effort users invest in setting up an account to submit feedback, the higher the feedback quality and the less praise and non requirements-related feedback is provided. However, the results also show that fewer app users sign up for additional services to provide app-related feedback.

4.2.2 Exchanged Context Items

To understand the amount and types of context items provided, we analyze all 52 Twitter conversations that report non-crashing bugs. We decided for Twitter as these conversations are the most active amongst our channels considered, e.g.,
app reviews did not receive a reply. For each conversation, we list the context items initially provided by users, provided after developer request, and missing although requested by developers.

Figure 4.5 shows an exemplary conversation. It consists of six tweets, of which each three are written by the app user (obfuscated by the author) and developer (@SpotifyCares). The user complains about Spotify stops playing music when running the app in background. In the first tweet, the user initially provides a context item, namely the relevant step to reproduce the issue, i.e., “when I leave the app”. As the issue remains unclear, the developer requests the device model and app version. The user provides both versions after request. The developer still being unable to reproduce the issue, asks the user if the low power mode is activated which the user denies. Finally, the developer asks the user to perform specific users interactions, i.e., steps to reproduce. However, the user does not answer the developers request and misses to confirm the provided context item.
Figure 4.6: Overview of context items within issues reported on Twitter.

Figure 4.6 shows an overview of the context items exchanged in all analyzed conversations. Overall, 118 context items were exchanged. This equals around two context items per conversation. Of the exchanged items, 17 (14.41%) were initially provided by app users and 101 (85.59%) after developer request. Additional 24 items were requested by developers but not submitted by app users. This highlights the effort developers have to invest in gathering context items in order to be able to understand and reproduce non-crashing bugs. The item most often initially provided by users is the platform. User interactions, for example, are never initially submitted by app users. The most requested items by developers, and the most often exchanged, are the app version, system version, device model, and platform. These are also most often missing, since developers repeatedly request several of these items at once and users forget to provide one of the items. Within this thesis, we consider these four items as basic context items.

The results underline the existing communication gap between users and developers in explicitly provided user feedback through channels such as app stores, social media, and user forums. Further, these are a first hint towards the effort support teams invest to make requirements-related user feedback actionable, i.e., understandable and reproducible, to developers by clarifying missing context information.

4.3 Context Extraction Approach

We describe a simple approach that accurately extracts basic context information from unstructured, informal user feedback on mobile apps. The approach allows us to identify user feedback potentially actionable to developers or requiring further clarification. We decided to consider the context items platform, device, app version, and system version, as we identified these four types to be
frequently requested by support teams during our manual data exploration of conversations that report non-crashing bugs on Twitter (cf. Figure 4.6). Moreover, researchers highlighted their importance for understanding and reproducing issue reports [312].

Our approach focuses on the Android and iOS platform. Both platforms cover 99.9% of the mobile operating system market [95]. The approach is designed to work with other platforms as well (e.g., desktop apps, smart TV apps), by exchanging its configuration files, i.e., the pre-defined keyword lists, without modifying the actual implementation.

We separate the description by the context item types and their strategies used for extraction.

4.3.1 Platform and Device

We crawl pre-defined keyword lists, including platform and device names, and generate word vector representations to handle informal writing frequently used in social media. Word vector similarities allow spelling mistakes and abbreviations of items included within the pre-defined lists to be determined. The lists and alternative spellings are used to create regular expressions that are applied to user feedback in order to extract context information. Figure 4.7 summarizes our approach.

Pre-Defined Keyword Lists

We crawled pre-defined lists of code names for the Android platform, as well as lists including device names for iOS and Android. These lists are maintained by app store operators or user communities and updated regularly, e.g., with the release of new devices.
4.3. Context Extraction Approach

For the Android platform, 15 alternative code names exist, such as ‘Cupcake’, which we extracted from a public list [7]. For the iOS platform, no such alternative names exist.

For iOS devices we extracted 51 names, such as ‘iPhone 8 Plus’ [174]. Since several users only refer to the product line, e.g., “[…] the error appears on my iPhone.”, we extend the device list by the five product lines iPhone, iPad, iPod Touch, Apple TV, and Apple Watch, resulting in 56 iOS devices.

For Android devices the diversity is much higher. We crawled an official list from Google Play containing all 23,387 supported devices [270] (as of January 2019). The list includes four columns, listing the retail branding (e.g., ‘Samsung’), marketing name (e.g., ‘Galaxy S9’), device (e.g., ‘star2qlteue’), and model (e.g., ‘SM-G965U1’). We pre-process the list in five steps: We create a unique list of marketing names, as these possibly occur several times due to the same device being manufactured for different markets (e.g., European or Asian). The resulting list includes 15,392 devices. Then, we remove all marketing names shorter than five characters, such as ‘V’ or ‘Q7’, resulting in 13,259 devices. Further, we remove marketing names that are not mentioned within the collected tweets. We removed these, as word-vector models perform better on extracting similar words when a given input is included in the training data, while extracting alternative spellings for unseen words could negatively influence the results [33]. It significantly reduced the number of devices to 1,324. This step needs to be repeated in fixed periods of time when new tweets are addressed to the support accounts. As the list of marketing names also includes common words (e.g., ‘five’, ‘go’, or ‘plus’), we used the natural language processing library spaCy [258] to remove words that appear in the vocabulary of the included en_core_web_sm model, trained on the CommonCrawl dataset. Thereby, we reduced the number of devices to 1,133.

Until this point the processing of the keyword lists is fully automated. We decided to manually fine-tune the Android device list by removing remaining common names not included within the vocabulary of the CommonCrawl dataset (e.g., ‘horizon’), while preserving more specific names (e.g., ‘galaxy s8’), resulting in 896 Android devices. This step could possibly be automated with datasets of larger vocabulary sizes.

Word Vector Representations

User feedback written in informal language might include alternative spellings of platform and device names, i.e., abbreviations or misspellings. For example, several users reference the Android code name ‘Lollipop’ as ‘lolipop’ or ‘lollypop’.

To enable our approach to also identify these cases, we create word vector representations using the fastText library [154]. Comparing vector distances allows to automatically identify similar words that frequently appear in the same
context. A subset of these similar words are alternative spellings of the platform and device names included in our lists. We decided to use fastText over simpler methods, such as the Levenshtein distance, to also identify alternative spellings that vary significantly. For example, users often reference the ‘iPhone 6 Plus’ as ‘iphone6+’, where the Levenshtein distance is 7. High edit distances would negatively impact the results by detecting, e.g., ‘one’ as alternative spelling to ‘iPhone 4’, where the edit distance is 5.

To train the fastText model, we pre-process all 5,254,969 crawled tweet texts according to the truthset (i.e., we convert the tweet texts into lowercase, remove line breaks, double whitespaces, and mentions of support account names).

We extract similar spellings for given keywords in the pre-defined lists of, e.g., iOS devices names, using word vector representations in four steps. First, we tokenize the tweets and remove non-informative tokens, then we train the word vector model using the tweets. Afterwards, we obtain alternative spellings for each given keyword from the word vector model. Finally, we generate a regular expression of the original keywords and their alternative spellings. In the following, we explain each step separately:

1. **Tokenize Tweets.** We begin by tokenizing each tweet. We remove non-informative tokens including punctuation and spaces using spaCy’s [307] built-in functionality.

2. **Train Word Vector Model.** For the actual training of the model, we use Gensim [239] as suggested by spaCy. We use the default configuration and set the word vector size to 300, the minimum occurrences of words to 5, the window size to 5, and perform the training in 10 epochs. Our trained model has a vocabulary size of 149,889 words.

3. **Extract Alternative Spellings.** We extracted similar words per platform and device name included in our lists. Figure 4.8 shows the ten most similar
4.3. Context Extraction Approach

words and their distances for the device ‘iPhone 8 Plus’, clustered in a dendrogram. The most similar word is ‘iphone8plus’ (cosine distance of 0.047), followed by ‘iphone6plus’ (0.051). Other device names, such as ‘zenphone’, are also extracted but have a higher cosine distance, in this case 0.328. These can be automatically filtered by setting a fixed threshold for the similarity, we used a threshold of 0.2. For the Android code names we extracted 14 unique alternative spellings. For iOS devices we extracted 44 alternative spellings and 392 for Android devices.

(4) Generate Regular Expression. Per list, we combine the given keywords (e.g., devices) and their alternative spellings into a single regular expression using the ‘OR’ operator, e.g., ‘iPhone XR|iPhone 7’. Later, we apply the Python functionality `re.search(pattern, string)` [238] to the user feedback. As users also include multiple devices, such as “[...] the error occurs on my iPhone 6 and iPad Mini.”, we modify the function to return the locations of all matches within a given input.

Manual Fine-Tuning of Results

The proposed approach to extract context items using pre-defined keyword lists and word vector representations can be run completely automated. Whenever, e.g., new devices are released, the keyword lists are updated by the app store operators or user communities. These, as well as the updated tweets dataset, including the most recent tweets of an app support account which possibly contain alternative spellings of new device names, need to be regularly provided as input to train the word vector model, in order to extract alternative spellings and to update the regular expression used to extract the context items. To fine-tune the results, we invested manual effort at two points.

First, when pre-processing the keyword lists to extract alternative spellings using word vectors, we manually removed device names solely consisting of common words (e.g., ‘horizon’) that could not be automatically removed. From the original 1,133 devices, we thereby removed 237 devices. This manual effort lasted for about two hours. It needs to be repeated regularly, e.g., when new devices are added to the pre-defined keyword lists. However, in these cases the effort is significantly lower since only single device names need to be processed instead of all supported devices since the release of Google Play ten years ago.

Second, we decided to manually filter alternative spellings for Android code names. We found that these include words (such as ‘bake’) that are unrelated to the inputs in the context of software engineering (such as the Android code name ‘pie’). Thereby, we removed 6 out of the 14 alternative spellings. Similar, we processed the alternative spellings for Android and iOS devices. For Android, we removed 326 spellings out of 392 alternative spellings. For iOS, we removed
22 of the 44 alternative spellings. We assume that more Android devices names have been removed, since the device names are very diverse and not as often included within the tweets which causes word vector models to suggest similar words that are not as closely related, compared to the names of iOS devices. This manual step lasted under an hour and is – as for the first step – significantly faster for future updates.

4.3.2 App and System Version

To extract app and system versions from user feedback, we crawled pre-defined keyword lists and created text patterns. The keyword lists include the released app and system versions. We collected 107 system versions for iOS and 59 for Android. Concerning the app versions [8, 9], we extracted 224 iOS and 133 Android versions for Netflix, 248 iOS and 346 Android versions for Snapchat, as well as 169 iOS and 165 Android versions for Spotify.

We tokenize the user feedback with spaCy. Then, we pre-process all tokens by removing leading characters before digits, such as ‘v8.4.17’. If the leading characters equal a platform (e.g., ‘iOS12’), we split the token to keep the platform. We also remove trailing characters often referring to system architectures, as such as ‘8.1.13arm7’. This might be a limitation that has to be adapted for other platforms. Versions for the platforms considered in our study cannot be named with leading or trailing characters (e.g., ‘A1.0’ or ‘1.1a’).

By manually comparing the collected versions to those mentioned in the user feedback, we identified two challenges. First, the collected versions have
intersections, as shown in Figure 4.9. For example, version 7.1.2 exists for the Netflix iOS app, as well as for both the iOS and Android operating system. Therefore, we cannot directly associate it with, e.g., the Android operating system. The intersections highly vary, the Netflix iOS app shares only 8 (3.57%) out of 224 versions with its Android app. In contrast, for Snapchat the app versions are much more similar with an intersection of 32.66% between iOS and Android. A relatively large overlap also exists for Android system versions and versions of the Netflix iOS app (27.12%), as well as the iOS operating system (42.37%). The second challenge are users reporting more detailed app versions (e.g., ‘8.0.1.785’) than included in the public lists. In this example, the user refers to the Snapchat Android app where the list only includes the version ‘8.0.1’, missing the subversion ‘.785’.

We implement a version matcher to handle these challenges. As input, it takes a conversation consisting of multiple tweets or single tweets, as well as the version lists. Further, the previously extracted platform and device can be provided as input, if exists. The version matcher processes the input in three steps: First, it generates a version tree. Then, it processes conversations or single tweets to extract included versions. Optionally, it resolves existing conflicts. In the following, we explain these steps separately:

(1) **Generate Version Tree.** Each version list and respectively their included versions are processed to create a version tree. If the list of app versions for iOS, e.g., includes the version ‘8.0.1’, this is split by its subversions and three leaves are added to the tree (i.e., ‘8’, ‘8.0’, and ‘8.0.1’). Each leave is marked as an iOS app version. If the list of system version for Android includes the version 8.0, no leaves are added to the version tree. Instead, the existing leaves ‘8’ and ‘8.0’ are marked as versions of both the iOS app and Android system.
(2) Process Conversation or Tweet. The matcher takes each token including a number and respectively its previous token as input. Figure 4.10 shows the matcher traversing the version tree on separate levels (L1-L4) to process the input ‘version 8.0.1.785’. The subversion ‘.785’ (L4) is not included in our crawled lists. Therefore, the closest version, i.e., ‘8.0.1’, of the previous level (L3) is selected. This version exists for both the Snapchat iOS and Android app, as well as the iOS system. As noted previously, not all app versions were included in the pre-defined lists, however we know that the collected list of system versions is complete. For this reason, the iOS operating system is removed as potential match (cf. Figure 4.10). If multiple system versions would remain, the matcher would process the previous token. If this token equals ‘iOS’ or ‘Android’, the matcher flags this respectively as iOS or Android system version. This is especially relevant for shorter versions, such as ‘8’ or ‘8.0’, where more potential matches exist. Since several possible matches remain, i.e., the version could refer to the iOS or Android app, it is conflicted and will be processed in the next phase.

(3) Resolve Conflicts (optional). If conflicts remain, potential version matches are assessed in their overall conversation context. Another feedback in the conversation might include additional context items, e.g., as a device either for Android or iOS, which helps to determine which platform the version is referring to. In the example conversation, the user previously wrote “The error occurs on my HTC One with Android installed”. As this feedback includes the Android platform and device, both context items are provided as input to the version matcher. If one of these relates to Android and none to iOS, the conflict is resolved by marking the version as Android app. A limitation of our approach are tweets, such as “It worked with my Galaxy S5, but is not working with my new Galaxy S6”. In this case, the conflict would not be resolved and both devices would be extracted. We consider this as beneficial, as knowing that the error occurs on one device but not the other might help developers. However, automatically highlighting on which of the devices the reported issue does not occur requires more complex natural language processing approaches, which we do not consider as focus of our study.

4.3.3 Classification Results

We evaluated the performance of our approach to extract basic context items (including the platform, device, app version, and system version) by comparing its results to the manually labelled truthset. Our truthset contains 3,014 tweets of the Netflix, Snapchat, and Spotify support accounts. Of these, 1,116 (37.03%) tweets include an overall amount of 1,840 context items.
### 4.3. Context Extraction Approach

Table 4.3: Performance of the approach compared to the truthset (T/FP = True/False Positive, F/TN = False/True Negative).

<table>
<thead>
<tr>
<th>Type</th>
<th>Account</th>
<th># Items</th>
<th>TP</th>
<th>FP</th>
<th>FN</th>
<th>TN</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>Netflix</td>
<td>311</td>
<td>303</td>
<td>0</td>
<td>8</td>
<td>701</td>
<td>n/a</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>Snapchat</td>
<td>416</td>
<td>403</td>
<td>0</td>
<td>13</td>
<td>615</td>
<td>n/a</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>Spotify</td>
<td>204</td>
<td>204</td>
<td>0</td>
<td>0</td>
<td>801</td>
<td>n/a</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>Combined</td>
<td>931</td>
<td>910</td>
<td>0</td>
<td>21</td>
<td>2,117</td>
<td>n/a</td>
<td>0.98</td>
</tr>
<tr>
<td>Device</td>
<td>Netflix</td>
<td>168</td>
<td>140</td>
<td>8</td>
<td>20</td>
<td>845</td>
<td>0.95</td>
<td>0.88</td>
</tr>
<tr>
<td></td>
<td>Snapchat</td>
<td>164</td>
<td>130</td>
<td>12</td>
<td>22</td>
<td>840</td>
<td>0.92</td>
<td>0.86</td>
</tr>
<tr>
<td></td>
<td>Spotify</td>
<td>156</td>
<td>116</td>
<td>18</td>
<td>22</td>
<td>859</td>
<td>0.87</td>
<td>0.84</td>
</tr>
<tr>
<td></td>
<td>Combined</td>
<td>488</td>
<td>386</td>
<td>38</td>
<td>64</td>
<td>2,544</td>
<td>0.91</td>
<td>0.86</td>
</tr>
<tr>
<td>App Ver.</td>
<td>Netflix</td>
<td>11</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>994</td>
<td>0.70</td>
<td>0.88</td>
</tr>
<tr>
<td></td>
<td>Snapchat</td>
<td>26</td>
<td>17</td>
<td>9</td>
<td>0</td>
<td>977</td>
<td>0.65</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>Spotify</td>
<td>89</td>
<td>74</td>
<td>11</td>
<td>4</td>
<td>918</td>
<td>0.87</td>
<td>0.95</td>
</tr>
<tr>
<td></td>
<td>Combined</td>
<td>126</td>
<td>98</td>
<td>23</td>
<td>5</td>
<td>2,889</td>
<td>0.81</td>
<td>0.95</td>
</tr>
<tr>
<td>System Ver.</td>
<td>Netflix</td>
<td>56</td>
<td>47</td>
<td>1</td>
<td>8</td>
<td>948</td>
<td>0.98</td>
<td>0.85</td>
</tr>
<tr>
<td></td>
<td>Snapchat</td>
<td>130</td>
<td>116</td>
<td>0</td>
<td>14</td>
<td>876</td>
<td>1.00</td>
<td>0.89</td>
</tr>
<tr>
<td></td>
<td>Spotify</td>
<td>109</td>
<td>88</td>
<td>2</td>
<td>19</td>
<td>898</td>
<td>0.98</td>
<td>0.82</td>
</tr>
<tr>
<td></td>
<td>Combined</td>
<td>295</td>
<td>251</td>
<td>3</td>
<td>41</td>
<td>2,722</td>
<td>0.99</td>
<td>0.86</td>
</tr>
</tbody>
</table>
Table 4.3 summarizes the results per context item and support account. The table shows the number of corresponding items in the truthset, the number of true positives (i.e., correctly identified context items), false positives (incorrectly identified items, such as ‘galaxy s8’ instead of ‘galaxy s8 plus’), false negatives (no items extracted although present in tweet), and true negatives (no items detected, where no items are present).

Based on these, the approach’s precision and recall is calculated. The precision indicates how many of the extracted items are correctly identified. The recall summarizes how many of all items included in the truthset were extracted. The table further combines the results per context item type for different apps by calculating their average. For different types, the precision varies from 81% to 99%, and the recall from 86% to 98%.

**Platform**

The platform is most frequently provided within the truthset, with 931 (50.60%) out of 1,840 context items. Of all platform mentions, our approach extracted 910 correctly (true positives) and missed to extract the remaining 21 (false negatives). The absence of the platform was correctly detected in 2,117 tweets (true negatives). For this context type, we refrain from reporting the precision. The truthset is biased, since we sampled for conversations that include the words ‘Android’ or ‘iOS’ in one of the tweets, to increase the amount of labelled context items. This ratio is not representative for the whole dataset. However, this is the least complex context item to extract and alternative platform code names, such as ‘Gingerbread’, were successfully extracted.

The recall for the platform is 98%. False negatives result from alternative spellings of Android code names that are not used frequently. For these, additional tweets need to be collected to train the fastText model or its minimum occurrences of words has to be tuned to increase the vocabulary size.

**Device**

Users within the truthset report 488 (26.52%) context items referencing a device. Our approach identified 386 true positives, 38 false positives, 64 false negatives, and 2,544 true negatives. For this type, the approach achieved a precision of 91% and a recall of 86%.

The detected false positives, e.g., include the device ‘Galaxy S8’ within the tweet “[…] android version 8.0.0 galaxy s8 plus for t-mobile”. Here the user used the device name ‘Galaxy S8 Plus’ instead of ‘Galaxy S8+’ included in the list. False negatives primarily consist of shortened device names, such as the ‘Samsung Galaxy S5’ mentioned as ‘s5’ within the tweet “[…] worked fine on my iphone and laptop, just not on my s5.”. Other short device names include ‘g4’
4.3. Context Extraction Approach

Part of these devices has been removed while pre-processing the device lists by filtering short device names, such as the ‘1610’. The approach results might improve by adding short devices names in combinations with their manufacturer (e.g., ‘Vivo 1610’). Short names have been previously excluded to reduce false positives.

App Version

The truthset includes 126 items (6.85%) reporting the app version. Our approach detected 98 true positives, 23 false positives, 5 false negatives, and 2,889 true negatives. The approach precision is 81% and recall 95%.

The detected false positives, e.g., include the version ‘0.9.0.133’ appearing in the tweet “version 0.6.2.64 on the phone and think its 0.9.0.133 on the desktop” from the Spotify dataset. In the tweet the user also refers to the desktop version, however version ‘0.9.0’ also exists for the Spotify iOS app. Other false positives include the version ‘3.0’ and ‘4.0’ which are detected as app version while referring to system versions, e.g., in the tweets “[...] cant connect with spotify on android since 3.0 update [...]” or “[...] i try to sync my ipod shuffle 4.0. any help? [...]”. The false positives mainly result from intersections between app and system versions of different platforms (cf. Figure 4.9). The intersections are the highest for Snapchat, resulting in a precision of 65%. False negatives are rare with only 5 occurrences and include versions not correctly separated by dots, such as ‘2.07’ instead of ‘2.0.7’.

System Version

Users report 295 context items (16.03%) including a system version. The approach identifies 251 true positives, 3 false positives, 41 false negatives, and 2,722 true negatives. For this context type, our approach achieved a precision of 99% and recall of 86%.

False negatives mainly result from the version matcher only taking into consideration a potential version’s previous token to decide if this refers to the system. This applies, e.g., for the tweet “on android (8.1 pixel xl) [...]” where the previous token is ‘(', as well as for the tweet “[...] android v 7.0 on spotify 8.4.39.673 armv7” where ‘v’ is the previous token. For these matching if one of the two previous tokens equals ‘iOS’ or ‘Android’ would improve the results. Other, false positives result from misspellings of users, such as ‘iso7’ instead of ‘ios7’. Also, more complex patterns which one user applies to report the system and app versions of multiple devices are not considered by the version matcher, as in the tweet “it also happens on my ipad (ios 10.1.1, spotify 6.8.0) and my
wife’s ipad (10.1.1/6.8.0) and iphone (10.1.1/6.8.0)."

Only 15 tweets within the truthset were marked as conflicted. An example tweet reports a Spotify app version, that exists for both iOS and Android “i’m on 8.4.74. doesn’t bother me too much... just thought i’d report it”. For conflict resolution, other tweets within the conversation are analyzed, such as “just so you know... the toast keeps going out of sync with what’s actually playing at the moment. using a pixel 2 on android 9.”. In this tweet the user reports both the Android platform and Android device. The conflict is resolved by marking the version as Android app version. Using this strategy, all of the 15 conflicts were resolved.

4.4 Clarification of Missing Context Information

In this section, we aim to understand how context items are exchanged in user-support conversations on Twitter and the effort support teams invest in clarifying missing information. Figure 4.11 shows the percentage of conversations per number of tweets included. Two-third of all conversations contain more than two tweets. This might also indicate that additional tweets need to be exchanged to clarify reported issues, possibly including basic context items.

To analyze the effort, we apply our context extraction approach on the cleaned dataset including about 3 million tweets from the Netflix, Snapchat, and Spotify support accounts. In the following, we answer three questions:

1. How many context items are exchanged via Twitter support accounts?
2. When are context items provided by users?
3. How active are support teams in conversations until context items are provided?

Figure 4.11: Percentage of conversations per number of tweets included.
4.4. Clarification of Missing Context Information

Table 4.4: Overview of context items extracted from the cleaned dataset.

<table>
<thead>
<tr>
<th></th>
<th>Netflix</th>
<th>Snapchat</th>
<th>Spotify</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td># Context Items</td>
<td>31,315</td>
<td>13,836</td>
<td>116,518</td>
<td>161,669</td>
</tr>
<tr>
<td>(all Types)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Context Items</td>
<td>25,142</td>
<td>11,870</td>
<td>101,153</td>
<td>138,165</td>
</tr>
<tr>
<td>(provided by Users)</td>
<td>80.29%</td>
<td>85.79%</td>
<td>86.81%</td>
<td>85.46%</td>
</tr>
<tr>
<td># Tweets</td>
<td>28,860</td>
<td>11,467</td>
<td>83,214</td>
<td>123,541</td>
</tr>
<tr>
<td>(including Context Items)</td>
<td>4.07%</td>
<td>3.27%</td>
<td>5.25%</td>
<td>2.35%</td>
</tr>
<tr>
<td># Conversations</td>
<td>22,663</td>
<td>9,319</td>
<td>60,625</td>
<td>92,607</td>
</tr>
<tr>
<td>(including Context Items)</td>
<td>10.32%</td>
<td>9.17%</td>
<td>12.69%</td>
<td>11.59%</td>
</tr>
<tr>
<td># Platform</td>
<td>5,646</td>
<td>4,497</td>
<td>39,476</td>
<td>49,619</td>
</tr>
<tr>
<td></td>
<td>18.03%</td>
<td>32.50%</td>
<td>33.88%</td>
<td>30.69%</td>
</tr>
<tr>
<td># Device</td>
<td>23,510</td>
<td>7,493</td>
<td>49,664</td>
<td>80,667</td>
</tr>
<tr>
<td></td>
<td>75.08%</td>
<td>54.16%</td>
<td>42.62%</td>
<td>49.90%</td>
</tr>
<tr>
<td># App Version</td>
<td>1,383</td>
<td>648</td>
<td>14,964</td>
<td>16,995</td>
</tr>
<tr>
<td></td>
<td>4.42%</td>
<td>4.68%</td>
<td>12.84%</td>
<td>10.51%</td>
</tr>
<tr>
<td># System Version</td>
<td>776</td>
<td>1,198</td>
<td>12,414</td>
<td>14,388</td>
</tr>
<tr>
<td></td>
<td>2.48%</td>
<td>8.66%</td>
<td>10.65%</td>
<td>8.90%</td>
</tr>
</tbody>
</table>

How many context items are exchanged via Twitter support accounts?

Table 4.4 lists the overall number of context items for mobile apps, exchanged per official Twitter support account. Further, it reports the number of conversations and tweets including basic context items. Last, it details context items by their types.

Within tweets of the Netflix support account, our approach identified 31,315 context items of which 25,142 (80.29%) are provided by users. Support teams also include context items in their tweets, such as the platform in “Are you using an iOS device?”. The items are exchanged within 28,860 (4.07% of all) distinct tweets as part of 22,663 (10.32% of all) conversations. This equals about 1.38 context items per conversation on average. Of all context items, 23,510 (75.08%) reflect the device, 5,646 (18.03%) the platform, 1,383 (4.42%) the app version, and 776 (2.48%) the system version.

For Snapchat 13,836 context items are exchanged of which 11,870 (85.79%) are provided by users. The items are included within 11,467 (3.27%) tweets belonging to 9,319 (9.17%) conversations. This equals about 1.48 context items per conversation on average. Of all items, 7,493 (54.16%) report the device, 4,497 (32.50%) the platform, 1,198 (8.66%) the system version, and 648 (4.68%) the app version.

Tweets of the Spotify support account include 116,518 context items of which 101,153 (86.81%) are provided by users. The items are exchanged in 83,214 individual tweets as part of 60,625 (12.69%) conversations (about 1.92 context items per conversation). Of all items, 49,664 (42.62%) reference the device, 39,476 (33.88%) the platform, 14,964 (12.84%) the app version, and 12,414 (10.65%) the system version.

Overall, the amount of context items is comparable among support accounts. Context items are exchanged within 11.59% of all conversations (about 1.75 con-
text items per conversation). The context items are included within 2.35% of all tweets. The context item most frequently provided by users is the affected device (49.90%), followed by the platform (30.69%), app version (10.51%), and system version (8.90%). For support teams, the context item type most frequently provided is the device (50.96%), followed by platform (44.06%), app version (2.73%), and system version (2.25%).

The results indicate that support teams are rather unable to deduce concerned platforms and devices, due to the variety of potential devices and the large number of context items referencing platform and device, that users provide for clarification.

When are context items provided by users?

We found that context items are not immediately provided by users. Conversations might take up to ten clarification tweets by users to get the context sought by the support teams. Figure 4.12 shows the percentage of context items and their order of appearance within conversations, only considering the tweets of users. Within the first tweet of users, less than half of the context items (42.35%) are provided. Further, 38.42% of the context items are provided within the second tweet of users, i.e., after a tweet of developers possibly requesting missing context items, such as of the Netflix support team “what device are you watching from that you’re having issues on?”. Another 10.92% of the context information is provided within the third user tweet and 4.3% within the fourth user tweet.

With the order of user tweets in conversations, the types of provided context items change. In the first tweet, 56.30% of the context items refer to the device, 33.21% to the platform, 5.66% to the system version, and 4.83% to the
4.4. Clarification of Missing Context Information

app version. In the second tweet, the distribution changes to 43.92% context items referring to the device, 24.65% the platform, 17.57% the app version, and 13.86% the system version. Within these tweets, users provide more app and system versions which support accounts might not immediately request at the beginning. The distribution remains comparable within tweets of following orders.

How active are support teams in conversations until context items are provided?

To understand the effort support teams invest in clarifying missing information, we further analyzed the conversations including context items. We only considered conversations in which users provide context items. In each conversation, we removed tweets that were created after all context items were provided. These tweets are likely irrelevant to clarify missing context. For example, support teams often end conversations with tweets as “You’re welcome! If you need us again, you know where to find us”.

Table 4.5 summarizes the results. For Netflix, 20,583 (90.82%) of all 22,663 conversations with context items, include context items provided by users. Support teams clarify on missing context items in 9,051 (43.97%) of these conversations. Until the context items included within the conversations are present, support teams create 12,055 tweets (1.33 tweets per conversation). The median response time of the Netflix support is the lowest compared to all support accounts with 4m 17s, same applies for users with 3m 57s.

For Snapchat, 8,515 (91.37%) of all 9,319 conversations with context items include context items provided by users. Support teams are active in 2,496 (29.31%) conversations providing 3,661 tweets (1.47 tweets per conversation) to clarify missing context items. The median response time of the support is 11m 25s and of the users 7m 50s.

For Spotify, 56,628 (93.41%) of all 60,625 conversations with context items include context items provided by users. The support team participates in 28,122 (49.64%) of these conversations, providing 38,256 tweets to clarify on missing context information. The median response time is the highest, compared to all three support accounts, with 56m 54s for the support and 20m 44s for users.

Overall, about half (46.27%) of the conversations including context items provided by users required effort of the support team until all context items are present. The median response time for the support is 32m 48s and for the users 13m 10s. The rather short response times of users indicate that these are willing to quickly provide additional missing information, such as basic context items, when asked for clarification.
Table 4.5: Effort invested by support teams until context items in conversations are present.

<table>
<thead>
<tr>
<th></th>
<th>Netflix</th>
<th>Snapchat</th>
<th>Spotify</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td># Conversations</td>
<td>22,663</td>
<td>9,319</td>
<td>60,625</td>
<td>92,607</td>
</tr>
<tr>
<td>with Context by User</td>
<td>20,583</td>
<td>8,515</td>
<td>56,628</td>
<td>85,726</td>
</tr>
<tr>
<td>requiring Clarification</td>
<td>9,051</td>
<td>2,496</td>
<td>28,122</td>
<td>39,669</td>
</tr>
<tr>
<td></td>
<td>(43.97%)</td>
<td>(29.31%)</td>
<td>(49.64%)</td>
<td>(46.27%)</td>
</tr>
<tr>
<td># Tweets by Support</td>
<td>12,055</td>
<td>3,661</td>
<td>38,256</td>
<td>53,972</td>
</tr>
<tr>
<td>Response Time Support</td>
<td>4m 17s</td>
<td>11m 25s</td>
<td>56m 54s</td>
<td>32m 48s</td>
</tr>
<tr>
<td>Response Time Users</td>
<td>3m 57s</td>
<td>7m 50s</td>
<td>20m 44s</td>
<td>13m 10s</td>
</tr>
</tbody>
</table>

4.5 Discussion

We discuss the implications of missing context information on software engineering and list potential limitations and threats to validity of our study.

4.5.1 Implications

Many software vendors, including Netflix, Snapchat, and Spotify, have recognized the advantages of gathering, analyzing, and reacting to user feedback provided via social media. About one-third of the bugs reported in issue trackers can be discovered earlier by analyzing tweets [200]. Speed is certainly a major advantage of social media-like feedback channels. Compared to reviews in app stores, the conversational nature of Twitter allows additional features and bugs to be identified. However, for the reported issues to be actionable to developers, basic context information, such as the utilized app version or device, needs to be included.

Our research shows that support teams themselves are very active, providing about 40% of the tweets within the crawled dataset, in many cases to clarify missing context items. Spotify and other vendors also initiated local support teams, such as @SpotifyCaresSE for Swedish users, with multiple involved persons [263]. Smaller teams receiving a large amount of feedback might not be able to afford such a large investment.

This chapter introduced a simple unsupervised approach that identifies the presence of and extracts context items from tweets. The results of our approach can be primarily used to filter actionable issues, i.e., conversations in which basic context items are present. When present, tweet texts and included context items can be used to auto-populate issue trackers with structured information [312].

Other conversations including only part of the basic context items might be non-actionable to developers. These can be automatically identified using our approach, as well as the exact information missing. When continuously applied
4.5. Discussion

to tweets, the output of our approach can be used, e.g., by a chatbot to immediately request missing context items from users by responding to conversations, e.g., “Can you tell us the device you are using?”. Both measures help reduce the manual efforts of support teams on social media.

Besides support teams and developers, our approach can also assist users. Users – often lacking software engineering and issue tracking knowledge – might be unaware of the importance of context information and therefore simply exclude them from the feedback. In a first step, users can be made aware about the importance of context items. While composing a tweet, this can continuously be analyzed to detect if a bug is reported, a feature is requested, or if the user might simply provide praise [180]. When reporting a bug, a message can be shown to the user that the issue reported might only be actionable to developers when including context information. The context items that the user already included while writing the tweet, can be identified using our approach, and missing context items can even be suggested in-situ [182].

For non-technical users who might simply not know how to gather the requested context items, these can be automatically extracted, if the device from which the issue is reported equals the device on which the issue occurred, and attached to the tweet.

4.5.2 Limitations and Threats to Validity

The support accounts on Twitter which we selected for our study are all of popular apps that appear within the top 25 charts of the Apple App Store. To improve the generalizability of our results, further support accounts for apps of different popularity (i.e., receiving different amounts of feedback) should be considered in future studies. Also, further studies need to be carried out to determine if the type of selected apps might correlate with the amount of non-technical users and possibly the amount of context items exchanged.

We filtered the crawled dataset by removing incomplete conversations that, e.g., begin with a tweet of the support account. Also, we focus on studying conversations involving single users, removing conversations with multiple users. We decided so, as we found during a manual analysis of the tweets, that a large number of conversations with multiple users request missing content, i.e., songs or movies. Therefore, conversations in which third users get involved, as they are facing similar issues, are excluded from our results. These users could possibly provide additional context items which can help developers to, e.g., determine the range of system versions affected by an issue. Further studies need to be carried out to analyze how context items are exchanged in conversations with multiple users.

To create the truthset, we extracted only conversations including at least one of the keywords ‘Android’ or ‘iOS’. Without this step, the number of con-
text items in the truthset would have been too small. As these keywords are also detected as platform context, the percentage of context items reporting the platform might be not representative for the whole dataset. We also tried more general keywords, such as ‘app’ or no keywords at all but the extracted tweets included much less context items or context information related to platforms that we do not consider in our study, such as Windows, Mac, or Linux. Further studies need to determine how our approach performs when considering all platforms supported by an app. Nevertheless, other identifiers for the platform, such as code names for Android versions (e.g., ‘Froyo’) could successfully be extracted by our approach.

The pre-defined keyword lists extracted for the platform, device, app version, and system version certainly influence our results. These need to be updated regularly for the platforms and the apps our approach is applied to. The results for the app and system version are negatively affected if the app versions are equal or similar to the system versions. To improve this circumstance, we consider the previous token to detect if a potential version refers to the Android or iOS system version. Further studies should not only consider the previous token but use different window sizes of tokens before and possibly after a potential version to increase the accuracy of the approach.

To improve our results, we trained a fastText model on all collected tweets. We extracted similar spellings for the platform and device names of our pre-defined lists. We manually identified relevant alternative spellings from the extracted similar words, such as ‘iphone6+’ for the input ‘iPhone 6 Plus’. This might introduce errors. Further tweets need to be collected to train the fastText model and determine if it provides more similar words to given inputs, such as device names. Then, this manual step can possibly be automated by only using a fixed threshold for the cosine distance.

4.6 Related Work

Users provide an increasing amount of feedback on software products, e.g., in the form of app reviews. Studies repeatedly showed that a significant amount of app reviews include information which is potentially useful to developers, such as shortcomings of app features, improvement requests, and bug reports [123, 180, 223, 224]. However, studies also found that that negative feedback often misses useful details as context information [192, 224]. To solve this issue, Maalej et al. [181] suggested to improve feedback quality by automatically collecting context information. Many software vendors offer such built-in options and automatically attach relevant context information to reported issues. Our approach complements this direction by focusing on users who only exchange text information.
Research found that especially non-technical end-users are more likely to express their opinions on social networks, such as Twitter [200]. Several studies have identified Twitter as an important source for crowd-based requirements engineering and software evolution [113, 200, 211]. Similar to app reviews, tweets contain important information, such as feature requests or bug reports. By performing a survey with software engineering practitioners and researchers, Guzman et al. [118] underlined the need for automatic analysis techniques to, e.g., summarize, classify, and prioritize tweets. The authors highlight that a manual analysis of the tweets is unfeasible due to its quantity, unstructured nature, and varying quality. Nayebi et al. [211] found that tweets provide additional requirements-related information. Compared to app reviews, by mining tweets the authors extracted about 22% additional feature requests and about 13% additional bug reports. Mezouar et al. [200] found that by considering tweets, at least one-third of the bugs for the browsers Firefox and Chrome created by technical end-users in issue trackers can potentially be discovered earlier. Other authors have used tweets to crowdsource app features [210], to support release decisions [303], to categorize and summarize technical information included in tweets [302], or to rank the reported issues [117]. These studies enforce the relevance of our approach.

Hassan et al. [124] studied the dialogue between users and developers in Google Play. Within their dataset of 4.5 million app reviews, the authors identified and analyzed 126,686 responses by developers. The authors did not focus on context items but on conversations in general, to show that reviews in app stores are not static. Compared to our results, developers in the dataset of Hassan et al. (i.e., on Google Play) are much less active with about 3% of all reviews being from developers. Within our dataset developers provided 26% to 51% of all tweets. Similarly, Bailey et al. [19] studied the dialogue between users and developers. The authors focused on the Apple App Store and found that developers reply to reviews for about one-fifth of the analyzed apps. The most discussed topics by developers are log-in issues, feature requests, and app crashes, possibly also by clarifying missing context information.

4.7 Summary

Despite built-in options to report issues in a structured manner, users continue to share a large amount of unstructured, informal feedback on software products via social media. This feedback contains information of relevance to development teams, such as bug reports or feature requests. Unfortunately, reported issues are often of low quality, as these miss basic context information, and, therefore, might therefore be non-actionable to developers. Software practitioners face several key challenges when working with bug reports included in user
feedback provided via app stores and social media:

- **Missing Information.** Compared to reports in issue trackers, feedback in app stores and social media is primarily provided by non-technical users in a less structured way [200]. Unfortunately, users often miss to provide context items needed by developers, such as the app version [34, 75, 202, 224, 312]. This is compounded by review processes that are purposefully unguided [254] and lack quality checks, to allow as many users as possible to participate.

- **Irrelevant Information.** Even when users provide context items, research has shown that there exists a mismatch between the information users provide and developers consider helpful [312]. Tool support should guide reporters to provide the information developers require [312]. However, in spite of including such options in apps, users continue to provide feedback via app stores and social media [262].

- **Unreproducible Issues.** In case user feedback that reports bugs misses relevant context information, these bugs might become hard to reproduce [75, 182]. Even if developers are able to guess the user’s interactions, an issue might only occur on specific combinations of device model and system version [105]. Research found that developers fail to identify erroneous configurations already for a low number of features [198].

We introduce a simple unsupervised approach to identify and extract basic context items from user feedback, including the affected platform, device, app version, and system version. Evaluated against a manually labelled truthset of 3,014 tweets, our approach achieved precisions from 81% to 99% and recalls from 86% to 98% for the different context item types. Our approach can assist support teams to identify and separate reported issues into non-/actionable. Actionable issues can be used to auto-populate issue trackers with structured information. Non-actionable issues can potentially be automatically clarified, e.g., by chatbots requesting missing context items.

To understand how context items are exchanged within conversations on Twitter, we applied our approach to about 3 million tweets from official support accounts of the three popular apps Netflix, Snapchat, and Spotify. We found that nearly half of the user feedback is not directly actionable to developers, as users miss to provide context information, such as the device used when experiencing an issue. Our results show that more than half of all context items are only provided after the engagement of support teams, sometimes requiring multiple clarifications. Support teams engage in effortful conversations with users to clarify missing context information — for popular apps, such as Spotify or Netflix, in about ten parallel conversations per hour.
Part II

Solution
Chapter 5

Detection of Fake User Feedback

This chapter is based on the second and third research question of the paper “Towards understanding and detecting fake reviews in app stores” [190] by Martens and Maalej, published in the Empirical Software Engineering Journal in 2019. My contribution to this publication is conducting all research, implementation, and analysis work, as well as leading the writing of the paper. My co-author supervised the work, inspired the research questions and goals, and made final edits to the paper.

In Chapter 3, we showed that app stores are targeted by fake user feedback to mislead users to either download an app or not. Fake reviews negatively impact the quality, i.e., authenticity, of user feedback and are prohibited in popular app stores. Moreover, fake reviews influence the results of existing app store analysis approaches, e.g., to prioritize feature requests based on their frequency reported. The results might deceive developers trying to understand real users’ needs. Fake reviews need to be identified and removed before integrating approaches for the automated analysis of user feedback into continuous software evolution practices. This chapter presents a study that extracts and investigates empirical differences between fake and regular app reviews. Based on these, we develop a classifier to identify and remove fake reviews automatically. In this chapter, we describe the development of the classifier and its classification results.

The remainder is organized as follows. Section 5.1 describes our research setting including the research questions, method, and data. We then report on the results along the research questions. Section 5.2 describes the characteristics of fake reviews. Section 5.3 reports on our approach to detect fake reviews. We discuss the implications and limitations of our findings in Section 5.4. Section 5.5 lists related work. Finally, Section 5.6 summarizes the chapter.
5.1 Research Setting

In the following, we first introduce the research questions. Then, we describe our research method and data.

5.1.1 Research Questions

In this chapter, we aim to quantitatively understand fake app reviews, including their characteristics, as well as their possible automated detection. We focus on two major research questions, which can be separated into several sub-questions:

RQ5.1 How do fake reviews differ from regular app reviews?
We crawled about 60,000 fake reviews, empirically analyzed and compared them with about 62 million official app reviews from the Apple App Store, including corresponding apps and reviewers.

RQ5.1.1 Apps. Which apps are typically affected by fake reviews? What are their categories, prices, and deletion ratio?

RQ5.1.2 Reviewers. What is a typical fake reviewer, e.g., in terms of the number of reviews provided and the reviewing frequency?

RQ5.1.3 Reviews. How do official and fake reviews differ, e.g., concerning the rating, length, votes, submission date, and content?

RQ5.2 How accurate can fake reviews be automatically detected?
We developed a supervised classifier to detect fake reviews. We evaluated the performance of multiple classification algorithms, configurations, and classification features.

RQ5.2.1 Features. Which machine learning features are useful to detect fake reviews automatically?

RQ5.2.2 Classification. Which machine learning algorithms perform best to classify reviews as fake/non-fake?

RQ5.2.3 Optimization. How can classifiers further be optimized? What is the relative importance of the classification features?

5.1.2 Research Method and Data

Our research method consists of a data collection, preparation, and analysis phase, as depicted in Figure 5.1. We detail each of the three phases in the following.
5.1. Research Setting

Data Collection Phase

For this study we collected two datasets: an official reviews dataset including app metadata and reviews from the Apple App Store, as well as a fake reviews dataset including metadata of apps affected by fake reviews and fake reviews.

The official reviews dataset created in March 2017 consists of 1,430,091 apps, their metadata, and reviews. To collect the data, we implemented a distributed crawling tool using GoLang, which we deployed on hundreds of cloud servers. The data collection included three steps. First, we crawled a list of all app identifiers available on the Apple App Store of the United States, as we focus on English reviews. Second, we obtained the metadata for each app, including the category, price, and number of reviews, using the iTunes Search API [144]. Last, we retrieved the app reviews using an internal iTunes API.

Overall, the Apple App Store included 207,782,199 ratings, of which 67,727,914 (24.6%) include a review. We were able to crawl 62,617,037 (92.4%) of these reviews, as iTunes does not allow to receive more than 30,000 reviews per app. The size of the dataset is 36.58 GB. The crawled reviews were written by 25,333,786 distinct reviewers, i.e., users with different Apple IDs [14]. On average, every reviewer submits around 2.47 reviews. The oldest app review was entered on 10/07/2008, therefore our dataset spans for nearly nine years.

Figure 5.1: Research method including the data collection, preparation, and analysis phase.
Table 5.1: Overview of collected fake reviews and apps requesting fake reviews.

<table>
<thead>
<tr>
<th>Provider Id</th>
<th>Provider Type</th>
<th># Apps</th>
<th># Reviews</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRP10</td>
<td>Paid Review Provider</td>
<td>77</td>
<td>-</td>
<td>Crawl</td>
</tr>
<tr>
<td>PRP16</td>
<td>Paid Review Provider</td>
<td>19</td>
<td>4</td>
<td>Crawl, Social</td>
</tr>
<tr>
<td>PRP21</td>
<td>Paid Review Provider</td>
<td>3</td>
<td>-</td>
<td>Social</td>
</tr>
<tr>
<td>PRP25</td>
<td>Paid Review Provider</td>
<td>-</td>
<td>3</td>
<td>Social</td>
</tr>
<tr>
<td>PRP26</td>
<td>Paid Review Provider</td>
<td>-</td>
<td>10</td>
<td>Social</td>
</tr>
<tr>
<td>PRP28</td>
<td>Paid Review Provider</td>
<td>-</td>
<td>3</td>
<td>Social</td>
</tr>
<tr>
<td>REP1</td>
<td>Review Exchange Portal</td>
<td>268</td>
<td>-</td>
<td>Crawl</td>
</tr>
<tr>
<td>REP2</td>
<td>Review Exchange Portal</td>
<td>277</td>
<td>-</td>
<td>Crawl</td>
</tr>
<tr>
<td>REP3</td>
<td>Review Exchange Portal</td>
<td>2,007</td>
<td>60,411</td>
<td>API, Crawl</td>
</tr>
<tr>
<td>REP5</td>
<td>Review Exchange Portal</td>
<td>7</td>
<td>-</td>
<td>Crawl</td>
</tr>
<tr>
<td>REP6</td>
<td>Review Exchange Portal</td>
<td>9</td>
<td>-</td>
<td>Crawl</td>
</tr>
<tr>
<td>REP8</td>
<td>Review Exchange Portal</td>
<td>182</td>
<td>-</td>
<td>Crawl</td>
</tr>
<tr>
<td>REP9</td>
<td>Review Exchange Portal</td>
<td>4</td>
<td>-</td>
<td>Crawl</td>
</tr>
</tbody>
</table>

∑ = 2,853  \sum = 60,431

The fake reviews were collected in April 2017. As described in Chapter 3, we identified 43 fake review providers. For each provider, we tried to collect fake review data, i.e., lists of apps requesting fake reviews and fake reviews itself. For this, we used three approaches:

1. **Social investigation**, i.e., asking the providers for fake review examples, while pretending to be interested in their services. We contacted the providers via email or live-chats on their websites. Using this strategy, we received 3 apps and 20 fake reviews from 5 providers.

2. **Crawling**, for providers offering to sign-up as fake reviewers, we checked if the lists of apps requesting fake reviews are available. To extract the apps, we implemented crawlers. A sample crawler is included in the replication package. Overall, we collected 2,850 apps from 9 providers.

3. **APIs**, we found that providers require reviewers to upload screenshots of their reviews as proof. We searched for publicly accessible screenshots and downloaded them. Based on this, we gathered 60,411 reviews from a single provider.

Overall we identified **60,431 fake reviews and 2,853 apps requesting fake reviews** from 13 providers, see Table 5.1. Per provider, the number of extracted apps and reviews is given, in case we could successfully apply at least one of the introduced approaches. The size of the collected data is 11.29 GB. We refer to this as **unfiltered fake data** (cf. Figure 5.1), as it needs to be prepared
5.1. Research Setting

Figure 5.2: Screenshots of fake reviews before submission to the app store used as proof for fake review providers, depicting a) fake review included in our study, b) cut-off fake review excluded from the study, and c) non-English fake review also excluded from the study.

for further analysis. For example, reviews within the dataset could have already been removed from the app store. For data preparation and analysis, all data, except the screenshots, are persisted as Parquet files and analyzed with Apache Zeppelin and Spark [311].

Data Preparation Phase

Most fake reviews were collected in the form of screenshots, as shown in Figure 5.2. We converted the screenshots into text using the Tesseract OCR engine [275]. We removed incomplete reviews that do not include a full readable title and body, e.g., if the title was outside the screen’s visible area. Then, we used the Language Identification (LangID) library [264] to retrieve fake reviews in English language only. We removed 7,445 reviews (12.32%), resulting in 52,986 fake reviews written in English language.

Since the screenshots show the review edit screens before submitting the reviews to the app store, we further filtered the collected fake reviews for three possible reasons. First, we cannot assure that the reviews were actually submitted to the app store. Second, the reviews could not have been unlocked by the app store operator. Third, the reviews could have been deleted. Therefore, we only considered fake reviews that have been published and still exist within the Apple App Store, i.e., which we could identify in the official reviews dataset.

For uniquely identifying (i.e., matching) reviews from the fake reviews dataset within the official reviews dataset, we removed duplicate reviews that consist of the same title and body within both datasets. Thereby, we removed 4,298 (8.11%) fake reviews leaving 48,688 items. The percentage of duplicate reviews within the official reviews dataset is with 16.08% nearly twice as high, which may be an indicator for the high diversity of fake reviews. We performed the match-
Chapter 5. Detection of Fake User Feedback

Table 5.2: Overview of the official and the fake reviews datasets.

<table>
<thead>
<tr>
<th></th>
<th>Official Reviews Dataset</th>
<th>Fake Reviews Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td># of reviews</td>
<td>62,617,037</td>
<td>8,607</td>
</tr>
<tr>
<td># of apps</td>
<td>1,430,091</td>
<td>1,929</td>
</tr>
<tr>
<td># of reviewers</td>
<td>25,333,786</td>
<td>721</td>
</tr>
</tbody>
</table>

ing using exact text comparison and by comparing the reviews’ Levenshtein distances. We used the Levenshtein distance as single characters on the screenshots were sometimes not parsed correctly by the OCR engine. We searched for all fake reviews within the official reviews dataset by using an edit distance of up to 10 characters. For possible matches identified using the Levenshtein distance, we manually verified if one of the suggested pairs is a match using two human annotators comparing the screenshot of the fake review and the possible matches. In case of disagreements (3% of all cases), a third annotator resolved the conflict. We matched 6,020 reviews by exact text comparison and 2,584 reviews by comparing the Levenshtein distance.

Overall, we were able to identify 8,607 of the 60,431 (14.2%) collected fake reviews within the Apple App Store. These reviews were extracted from 5 providers. We also matched apps affected by fake reviews against the official reviews dataset, as the apps might not be available in the US App Store or might have been deleted. Of the 2,853 collected apps, we found 2,174 apps (76.2%) in the official reviews dataset. Further, we identified 898 additional apps by extracting the app identifiers from previously matched fake reviews, resulting in 3,072 apps. We removed all apps that did not receive reviews within the app store, resulting in 1,929 of 3,072 (62.8%) apps provided by ten different providers. Finally, we identified 721 fake reviewers, i.e., accounts of persons submitting fake reviews to the app store, by extracting their user identifiers from fake reviews.

In summary, after data cleaning the fake reviews dataset consists of 43 providers and structural information about their offers and policies, as well as 8,607 fake reviews, 1,929 apps affected by fake reviews, and 721 fake reviewers. The dataset spans for nearly seven years, as the oldest fake review was entered on 16/10/2010. Table 5.2 summarizes the official and fake reviews datasets.

Data Analysis Phase

The data analysis phase consists of two steps, which respectively answer the research questions. To explore the fake review characteristics, we applied a statistical analysis of the reviews, apps, and reviewers. We compare the figures
5.2 Fake Review Characteristics

from the fake reviews dataset to those from the official reviews dataset and run statistical tests whenever applicable. For example, we found that most fake reviews are provided for games. While regular apps receive most reviews on the day of an app release, apps affected by fake reviews receive most reviews eleven days after the update. This time delay could indicate that apps affected by fake reviews do not have a real user basis that intrinsically provides reviews in reaction to changes introduced by app updates. Further, we found that fake reviewers provide twelve times as many reviews, with a four times higher frequency. Also, fake reviews have more positive ratings compared to official reviews. However, the most significant difference exists between the number of one-star ratings.

To detect fake reviews, we created a labelled and balanced dataset of fake reviews and official reviews. We used it to train and evaluate multiple classifiers based on machine learning features that we derived from the analysis of fake review characteristics. We conducted a hyperparameter tuning of the classifiers and evaluated the importance of the classification features.

We detail each of these analysis steps in the following chapters. To support replication, our dataset and the analyses source code as Zeppelin notebooks is publicly available [241].

5.2 Fake Review Characteristics

We investigate apps affected by fake reviews, reviewers providing fake reviews, and fake reviews themselves. In particular, we study the differences of fake reviews to the reviews from the official reviews dataset.

5.2.1 Apps

We identified 3,072 apps requesting fake reviews. As these apps could have, e.g., been entered on review exchange portals for testing purposes either or not by their developers, we only consider apps that received fake reviews to strengthen our results. Overall, we analyzed 1,929 (62.8%) of the identified apps.

Most apps with fake reviews fall into the category games, nearly twice as much as regular apps. Table 5.3 lists the 25 app categories of the Apple App Store. It compares apps from the fake reviews and the official reviews dataset per category. The table depicts each categories’ rank, number and percentage of apps, and percentage of reviews within the datasets. The highest rank is assigned to the category with the most apps included.

We found that more than half of the apps with fake reviews (53%) belong to the category “Games”, followed by the categories “Photo & Video” (5.8%),
Table 5.3: Category ranking, number of apps, and percentage of reviews per category within the fake reviews and official reviews dataset.

<table>
<thead>
<tr>
<th>Category</th>
<th>Fake Reviews Dataset</th>
<th>Official Reviews Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rank</td>
<td>Apps</td>
</tr>
<tr>
<td>Books</td>
<td>21</td>
<td>4 (0.21%)</td>
</tr>
<tr>
<td>Business</td>
<td>12</td>
<td>33 (1.71%)</td>
</tr>
<tr>
<td>Catalogs</td>
<td>23</td>
<td>3 (0.16%)</td>
</tr>
<tr>
<td>Education</td>
<td>3</td>
<td>92 (4.77%)</td>
</tr>
<tr>
<td>Entertainment</td>
<td>4</td>
<td>87 (4.51%)</td>
</tr>
<tr>
<td>Finance</td>
<td>17</td>
<td>17 (0.88%)</td>
</tr>
<tr>
<td>Food &amp; Drink</td>
<td>15</td>
<td>19 (0.98%)</td>
</tr>
<tr>
<td>Games</td>
<td>1</td>
<td>1023 (53.03%)</td>
</tr>
<tr>
<td>Health &amp; Fitn.</td>
<td>5</td>
<td>85 (4.41%)</td>
</tr>
<tr>
<td>Lifestyle</td>
<td>8</td>
<td>69 (3.58%)</td>
</tr>
<tr>
<td>Medical</td>
<td>20</td>
<td>13 (0.67%)</td>
</tr>
<tr>
<td>Music</td>
<td>17</td>
<td>36 (1.87%)</td>
</tr>
<tr>
<td>Navigation</td>
<td>20</td>
<td>13 (0.67%)</td>
</tr>
<tr>
<td>News</td>
<td>24</td>
<td>2 (0.10%)</td>
</tr>
<tr>
<td>Newsstand</td>
<td>23</td>
<td>3 (0.16%)</td>
</tr>
<tr>
<td>Photo &amp; Video</td>
<td>2</td>
<td>112 (5.81%)</td>
</tr>
<tr>
<td>Productivity</td>
<td>10</td>
<td>42 (2.18%)</td>
</tr>
<tr>
<td>Reference</td>
<td>18</td>
<td>14 (0.73%)</td>
</tr>
<tr>
<td>Shopping</td>
<td>13</td>
<td>25 (1.30%)</td>
</tr>
<tr>
<td>Social Netw.</td>
<td>6</td>
<td>82 (4.25%)</td>
</tr>
<tr>
<td>Sports</td>
<td>9</td>
<td>45 (2.33%)</td>
</tr>
<tr>
<td>Stickers</td>
<td>25</td>
<td>1 (0.05%)</td>
</tr>
<tr>
<td>Travel</td>
<td>14</td>
<td>22 (1.14%)</td>
</tr>
<tr>
<td>Utilities</td>
<td>8</td>
<td>69 (3.58%)</td>
</tr>
<tr>
<td>Weather</td>
<td>16</td>
<td>18 (0.93%)</td>
</tr>
</tbody>
</table>

\[ \sum = 1,929 \quad \sum = 1,430,091 \]
5.2. Fake Review Characteristics

“Education” (4.8%), “Entertainment” (4.5%), and “Health & Fitness” (4.4%). The categories with least apps with fake reviews are “Stickers” (0.05%), “News” (0.1%), “Catalogs” (0.16%), “Newsstand” (0.16%), and “Books” (0.21%).

Between both datasets, we found a strong, positive correlation for the distribution of apps over the categories. We compared the category ranks using the Spearman rank correlation coefficient \( r_s = 0.74 \), two-tailed p-value = 0.00002). The coefficient is used to measure the rank correlation, i.e., the statistical dependence between the rankings of two variables.

To identify categories with the highest difference between both datasets, we calculated the rank difference using the following formula.

\[
    r_{\text{diff}} = |\text{rank}_{\text{official}} - \text{rank}_{\text{fake}}|
\]  

(5.1)

The highest differences exist for the categories “Social Networking” \( r_{\text{diff}}=11 \), “Photo & Video” \( r_{\text{diff}}=10 \), “Business” \( r_{\text{diff}}=9 \), and “Shopping” \( r_{\text{diff}}=9 \). The third category “Business” within the app store is, for example, only ranked twelvth in the fake reviews dataset. Vice versa, the category “Photo & Video” contains more apps in the fake reviews dataset.

Apps with fake reviews are on average three times less offered as paid, compared to regular apps. Developers might invest much money buying fake reviews. Therefore, we analyzed the monetization of apps in the fake reviews dataset. We focused on the app price and in-app purchases.

Regarding the app price, we found that 1,799 apps (93.3%) are offered for free, while 130 apps (6.7%) are paid. In comparison, the app store includes 1,167,377 (81.6%) free and 262,714 (18.4%) paid apps. The mean price of an app is $2.16, with a standard deviation of 2.5. For the app store, the mean price is $4.07, with a standard deviation of 16.7. This difference between the mean prices is statistically significant (two-sample t-test, p<0.001, CI=0.99). However, the magnitude between the differences is slightly below small, found by calculating the effect size \( d=-0.160 \) [56]. Of the paid apps, 62 apps (47.7%, cf. 39.6% in app store) are offered for $0.99, 39 apps (30%, cf. 18.9% in app store) for $1.99, and 16 apps (12.3%, cf. 16.2% in app store) for $2.99. The remaining apps (10%) cost between $3.99-$24.99. In the app store, the price of the remaining apps (25.4%) is between $3.99-$999.99.

In-app purchases are offered by 759 fake-reviewed apps (39.4%). These apps contain 3,845 in-app offers, of which 3,186 are in-app purchases. On average, each app includes around 4.2 in-app purchases with an average price of $10.46. 26.33% of the in-app purchases are offered for $0.99, 26.9% are in the range of $1.99-$2.99, 26.6% in the range of $3.99-$9.99, and 20.2% in the range of $10.99-$399.99. We were unable to crawl in-app purchases automatically, therefore we cannot compare the figures from the fake reviews to the official reviews dataset.
Also, apps could be further monetized through advertisements. We were unable to study this aspect since no publicly available data on the number of advertisement impressions and revenue generated per impression exists.

Most apps targeted by fake reviews have 2-9 reviews, which is the case for 42.2% of all apps. We analyzed the total number of reviews for apps affected by fake reviews and apps from the official reviews dataset. Figure 5.3 groups both types of apps into given ranges of fake and official reviews. The result indicates that fake reviews are not necessarily limited to a small and specific group of apps, but could be distributed across the majority of apps.

Only less than 7% of apps affected by fake reviews were removed from the Apple App Store. We studied whether apps with a high percentage of fake reviews rather get removed from the app store, compared to apps with less fake reviews. Therefore, we crawled the apps affected by fake reviews again after three months in June 2017. Of the 1,929 apps, 131 (6.8%) were no longer available on the app store. Most of the deleted apps (68%) belong to the category “Games”, 5% to “Entertainment”, and 5% to “Utilities”. Since the app store operators do not justify the deletion of apps, there exist two possible reasons:
Either, the apps have been removed by their developers, or the app store operators have removed the app due to fake reviews or other compliance reasons, e.g., spam apps [251]. Figure 5.4 shows two plots of deleted and non-deleted apps and their percentage of fake reviews.

The upper plot considers all apps affected by fake reviews. We found that deleted apps received 27.5% fake (median: 20%) and 72.5% official reviews. Non-deleted apps received 38.1% fake (median: 30.8%) and 61.9% official reviews. By analyzing the median, we found that non-deleted apps receive twelve reviews (cf. 15 reviews for deleted apps), of which two are fake (cf. 2 reviews for deleted apps). A $\chi^2$-test showed that being no longer available on the app store and the percentage of fake reviews are independent ($\chi^2=2.0906$, $p=0.1482$).

As this gives the impression that the amount of fake reviews does not impact the removal of apps from the app store, we further analyzed apps with at least ten fake reviews only, see the lower plot. This applies to 181 apps, of which eleven were deleted. For these, the median of fake reviews for deleted apps is 63.5%. For non-deleted apps, the median is 37.1%. Based on medians, deleted apps receive 51 reviews, of which 22 are fake. Non-deleted apps receive 49.5 reviews, of which 15 are fake. For these apps, a $\chi^2$-test showed that both values are no longer independent ($\chi^2=6.8708$, $p=0.008762$), compared to considering all apps with at least one fake review.

### 5.2.2 Reviewers

Fake reviewers submit about 30 reviews on average — 12 times more than regular reviewers. We identified 721 users providing fake reviews. These fake reviewers provide 29.9 reviews per user, on average, compared to 2.5 reviews per reviewer in the official reviews dataset. This difference is statistically significant (two-sample t-test, $p<0.001$, CI=0.99), and the effect size is large ($d=0.802$). Overall, these users provided 21,581 reviews in total for 8,429 different apps. Surprisingly, fake reviewers do not seem to use several accounts to hide their activities.
More than 50% of the reviewers in the official dataset provide only a single review. The total number of reviews given per fake reviewer varies between 1 and 573. For reviewers within the official reviews dataset, this is between 1 and 913. Figure 5.5 groups both fake and regular reviewers according to their number of submitted reviews.

Exactly one review was given by 5.4% of the fake reviewers, compared to 53.1% for regular reviewers. 2-5 reviews were provided by 15.8% fake reviewers (cf. 35.6%), 6-10 reviews by 20.8% (cf. 9.3%), 11-50 reviews by 40.8% (cf. 1.9%), 51-100 reviews by 11.8% (cf. 0.02%), and more than 100 reviews by 5.4% (cf. 0.006%). Most fake reviewers (40.8%) provide 11-50 reviews, while most regular reviewers (53.1%) provide only a single review.

Fake reviewers review about four times more frequently than regular reviewers. Fake reviewers are more active compared to others. They have a frequency of one review per 78.8 days, compared to 328.9 days for regular reviewers. The difference is statistically significant with 250.1 days, i.e., 417.2% (two-sample t-test, $p<0.001$, CI=0.99). The effect size is large ($d=-0.955$).

The lifetime of fake reviewer accounts is nearly twice as long as regular users. The account lifetime, i.e., the time difference between the first and last review provided, is 622.3 days for fake reviewers, compared to 331.3 days for other app store users. The difference between fake and regular reviewers is 291 days (187.9%) and statistically significant using the previous test ($p<0.001$, CI=0.99). The effect size is near medium ($d=0.464$). This result shows that the accounts of fake reviewers remain undetected in app stores for several years.

5.2.3 Reviews

Although we found 60,431 fake reviews, in the following we only consider the 8,607 fake reviews that we identified and still exist in the Apple App Store. These reviews have not been filtered by the mechanism of the app store operators and could impact app developers and users.

The distribution between ratings of fake and official reviews varies most for one-star reviews. Figure 5.6 compares the distribution of ratings for reviews from the fake and official reviews dataset. 70% of the fake reviews are rated with five stars compared to 65% for official reviews. 23% of fake reviews are rated with 4 stars (cf. 16%), 5% with 3 stars (cf. 6%), 1% with 2 stars (cf. 4%), and 0.6% with 1 star (cf. 10%). Overall, ratings are very positive in both datasets. The greatest difference between fake and official reviews can be observed by the percentage of one-star ratings. We have evidence that fake review providers explicitly ask their reviewers within reviewing policies to provide
5.2. Fake Review Characteristics

not only five-stars reviews but also four-stars and even three-stars reviews. This might result in rather small differences between fake and official reviews regarding extremely positive ratings. Thereby, the suspicious behavior of writing and also receiving, only five-stars reviews should be hidden. Otherwise, this could result in the deletion of fake reviews by app store operators and in the worst case, the removal of the affected app from the app store (cf. Section 3.3.2).

Compared to official reviews, short reviews are rather uncommon in fake reviews. The length of a fake review (consisting of title and body) is, on average, 121.3 characters. Official reviews have a length of 110.8 characters, on average – resulting in a difference of 10.5 characters. Considering the median, fake reviews consist of 111 characters, while official reviews consist of 63 characters, see Figure 5.7. The difference regarding the median is 48 characters.

We further analyzed the number of words per review. Fake reviews have, on average, 22.9 words, with a median of 21 words. Official reviews have 21.3 words, with a median of 12 words. Regarding the number of average words, the difference is relatively small with 1.7 words. Considering the median, the difference is nine words. A typical fake review is given below.

Figure 5.6: Distribution of star ratings between official and fake reviews.

Figure 5.7: Distribution of review length (in characters) between regular and fake reviews.
Figure 5.8: Percentage of reviews received per day after app release (day 0 is release of app update).

Great for expense tracking ★★★★★
*Does a great job for expense tracking. Nice interface and color scheme. Definitely recommend!*

We found that rather short reviews, which constitute a major part of the official reviews, are uncommon for fake reviews (see example below).

Fantastic ★★★★★
*Great game, my son loves it. Lots of fun.*

We initially assumed fake reviews to be short. However, according to the dataset, fake reviews are significantly longer regarding the number of characters and words (Wilcoxon rank-sum test, p < 0.001, CI=0.99). The effect size [91] however is near zero (r=0.001).

**Fake reviews are rated more often helpful compared to official reviews.** In app stores, users can rate the helpfulness of reviews through votes. 132 of the 8,607 fake reviews (1.5%) received at least one vote, compared to 2.7% for reviews in the app store. Overall, the reviews received 270 votes. 245 are votes (90.7%) rating the reviews as helpful, and 25 votes (9.3%) rate the reviews not helpful. In the app store, less helpful votes (67.8%) exist. Both, the number of reviews with votes and the overall number of helpful votes are significantly different (two-sample t-test, p<0.001, CI=0.99). Also in this case, the effect size is near zero (d=-0.018).

**After releasing updates, apps affected by fake reviews do not immediately receive more reviews.** We analyzed the relative reviewing frequency by summing up all reviews per day after the apps' last releases. Figure 5.8 shows the percentage of received reviews per day for apps affected by fake reviews and regular apps for a period of three weeks. After three weeks, the number of
Fake Review Characteristics

received reviews stabilized. We choose only the apps’ last release as we were unable to crawl release dates automatically.

For regular apps, most reviews are given on the day of the app release [224]. For apps affected by fake reviews, there is only a small peak on the app release day. For these apps, the percentage of reviews provided increases daily until it decreases on the eleventh day after the app update. One reason might be that for apps affected by fake reviews no substantial user basis exists that could, intrinsically motivated, provide reviews. Developers of these apps have to buy fake reviews to promote their updates. The distribution of requests for providing fake reviews to the actual reviewers might take time. Compared to that, regular apps with a user basis that matches the number of reviews received, have enough users that spontaneously provide their feedback after installing the app update.

Fake reviews include more positives adjectives and less negative words related to software engineering such as “fix” or “crash”. We analyzed the review content by comparing the 100 most-common words of fake and official reviews. We extracted the lists of most-common words in five steps. First, we removed the punctuation. We transformed all words into lowercase writing. Then, we tokenized the words of each review and removed stopwords. Last, we counted the occurrences of each word.

Both lists have 63 words in common and 37 unique words. We sorted the lists descending by the occurrences of words. Afterwards, for each word the lists have in common, we calculated the difference between their positions in the lists. The word “simple” is, e.g., on position 14 for fake reviews and position 97 for official reviews, resulting in a rank of -83. Negative ranks denote words that are more common for fake reviews. We plotted word ranks in Figure 5.9.

Figure 5.9: Delta between occurrences of most-common words in official/fake reviews, negative rankings indicate that the specific word is more common for fake reviews while positive rankings denote that the word is less common.
Chapter 5. Detection of Fake User Feedback

The top five words that are more common for fake reviews are “simple”, “super”, “little”, “recommend”, and “well”. The top five words that are less common for fake reviews are “even”, “can’t”, “don’t”, “want”, and “free”.

Afterwards, we analyzed the distinct words in both lists. For official reviews the distinct five most common words are (in order): “update”, “ever”, “please”, “fix”, “every”. Also, words possibly related to the functionality of the apps, such as “doesn’t”, “crashes”, “wish”, and “bad” are included. The five most common distinct words for fake reviews are: “graphics”, “useful”, “idea”, “ads”, and “kids”. Also positive words, such as “interesting”, “perfect”, “helpful”, “recommended”, “funny”, and “learn” are popular.

Last, we compared the most common bi-grams for fake and official reviews. As for most common words, again 63 matches exist. We observed that bi-grams possibly pointing to bug reports, such as “please fix”, only exist in the official reviews dataset. Bi-grams indicating feature requests, such as “would like” or “wish could”, exist in both datasets. Negative bi-grams, such as “waste time” or “keeps crashing”, again only exist within the official reviews dataset.

5.3 Fake Review Detection Approach

We build a supervised binary classifier to classify reviews as fake or not. Figure 5.10 shows the three phases conducted after feature extraction. We begin by preprocessing the data. Then, we compare the results of different classification algorithms. We optimize the algorithms by feature selection and hyperparameter tuning. Last, we evaluate the importance of the classification features.
5.3. Fake Review Detection Approach

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Type</th>
<th>Null-Values</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reviewer</td>
<td># Reviews (Total)</td>
<td>Int</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>% Reviews (per Star-Rating)</td>
<td>[Float]</td>
<td>0</td>
<td>[0.7, 0.0, 0.0, 0.0, 0.3]</td>
</tr>
<tr>
<td></td>
<td>Review Frequency (in Seconds)</td>
<td>Int</td>
<td>1,734</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Account Usage (in Seconds)</td>
<td>Int</td>
<td>0</td>
<td>600</td>
</tr>
<tr>
<td>App</td>
<td># Reviews (Total)</td>
<td>Int</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td># Reviews (per Star-Rating)</td>
<td>[Float]</td>
<td>0</td>
<td>[0.2, 0.2, 0.2, 0.2, 0.2]</td>
</tr>
<tr>
<td>Review</td>
<td>Length (in Characters)</td>
<td>Int</td>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>

5.3.1 Feature Extraction

We extracted a balanced truthset of 16,000 reviews. Of these reviews, 8,000 are randomly selected fake and 8,000 are randomly selected official reviews. Per review, the truthset includes a vector containing 15 numerical features and a label, either ‘real’ or ‘fake’. The features were selected based on the differences we identified between fake and official reviews, and by experimentation.

We decided not to use the textual review itself, e.g., in the form of TF/IDF representation, for several reasons. Mukherjee et al. [204] analyzed fake reviews published on Yelp and found that the word distribution of fake reviews does not significantly differ from official reviews. As a result, their text-based classifier only achieved an accuracy of 67.8%. Vice versa, the word distribution within fake reviews could also highly differ. According to the questionnaire with paid review providers, custom keywords can be included or predefined reviews can be submitted by the fake reviewers. Finally, when using text, training data would be required for every language, which is challenging to collect.

In contrast, Ferrara et al. [87] use non-textual features related to the user, such as the account creation time or the total number of followers. By using such features their classifier to detect bots in social networks that, e.g., influence political discussions, achieved better results. Other researchers [68, 86, 168, 228] followed this approach and improved their classification results.

We therefore focused on features that relate to the context of the fake review, i.e., the reviewer and app. Table 5.4 lists all selected features. For the reviewer, we selected four features: the total number of reviews provided, the percentage of reviews per star rating (e.g., the reviewer could have provided 70% of all reviews with a 5-star rating and 30% with a 1-star rating), the review frequency (i.e., the average time in seconds between all reviews provided), and the account usage (which is the lifetime of the reviewers account, i.e., the timespan between the first and the last review provided in seconds). For the app, we selected
two features: the total number of reviews received for all app versions, and the percentage of reviews received per star rating. Finally, as feature for the review, we selected the length, i.e., the characters count.

Data Preprocessing

We preprocessed the data in three steps. We began by performing data cleaning, i.e., filling null values instead of removing affected columns. Of the selected features, only a single column includes null values, see Table 5.4. The review frequency is in 1,734 cases undefined because the reviewer provided only a single review. In this case, we set the frequency to the lifetime of the app store, which is nine years.

Then, we normalized the dataset so that individual samples to have unit norm. We used the `normalize()` method with standard parameters of the `preprocessing` module provided by scikit-learn [229].

Last, we standardized the dataset so that the individual features are standard normally distributed, i.e., gaussian with zero mean and unit variance. This is a common requirement for many classification algorithms, such as the Gaussian radial basis function (RBF) kernel of support vector machines. If not standardizing the data, features with a much higher variance compared to others might dominate the objective function. As a result, the classification algorithm is unable to learn from other features [249]. We used the scikit-learn `scale()` method with standard parameters of the `preprocessing` module.

5.3.2 Classification

We compare seven supervised machine learning approaches to classify reviews as fake or not. We use the implementations provided by the scikit-learn [229] library. Supervised approaches need to be trained using a labelled truthset before they can be applied. This truthset is split into a training and testing set. The training set is used by the algorithms to build a model on which unseen instances are classified. In the test phase, the classifier performs a binary classification and decides whether reviews within the test set are fake or not.

To get more reliable measures of the model quality, we apply cross-validation on our truthset. This is performed in several folds, i.e., splits of the data, called k-fold cross-validation. In this study, we perform ten folds. Per fold, a randomly selected amount of 1/k of the overall data is held out of the training as a test set for the evaluation. The final performance is the average of the scores computed in all folds. Using cross-validation, we also avoid bias, which would otherwise be introduced by using only a random train/test split. In addition, although our truthset is balanced, we apply stratification. Stratification ensures that each split contains a balanced amount of fake and official reviews. We repeat the
5.3. Fake Review Detection Approach

Table 5.5: Classifiers’ scores to detect fake reviews on balanced dataset.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
<th>AUC/ROC</th>
</tr>
</thead>
<tbody>
<tr>
<td>RandomForestClassifier</td>
<td>0.970</td>
<td>0.973</td>
<td>0.967</td>
<td>0.970</td>
<td>0.989</td>
</tr>
<tr>
<td>DecisionTreeClassifier</td>
<td>0.953</td>
<td>0.949</td>
<td>0.957</td>
<td>0.953</td>
<td>0.953</td>
</tr>
<tr>
<td>MLPClassifier</td>
<td>0.919</td>
<td>0.921</td>
<td>0.916</td>
<td>0.918</td>
<td>0.969</td>
</tr>
<tr>
<td>SVC(kernel='rbf')</td>
<td>0.901</td>
<td>0.879</td>
<td>0.930</td>
<td>0.904</td>
<td>0.959</td>
</tr>
<tr>
<td>SVC(kernel='linear')</td>
<td>0.899</td>
<td>0.878</td>
<td>0.926</td>
<td>0.902</td>
<td>0.960</td>
</tr>
<tr>
<td>LinearSVC</td>
<td>0.895</td>
<td>0.861</td>
<td>0.941</td>
<td>0.900</td>
<td>0.964</td>
</tr>
<tr>
<td>GaussianNB</td>
<td>0.765</td>
<td>0.731</td>
<td>0.889</td>
<td>0.755</td>
<td>0.955</td>
</tr>
</tbody>
</table>

cross-validation 30 times per classification algorithm with different seeds. We use the RepeatedStratifiedKFold method of the model_selection module.

The seven classification algorithms we compare are the following: Naive Bayes (GaussianNB) is a popular algorithm for binary classification [30], which is based on the Bayes theorem with strong independence assumptions between features. Compared to other classifiers, it does not require a large training set. Random Forest (RF) [126] is an ensemble learning method for classification and other tasks. It can build multiple trees in randomly selected subspaces of the feature space. Decision Tree (DT) [281] assumes that all features have finite discrete domains and that there is a single target feature representing the classification (i.e., the tree leaves). Support Vector Machine (SVM) [59] represents the training data as points in space. It creates support vectors for gaps between classes in the space. The test data is classified based on which side of the gap its instances fall. The Gaussian radial basis function (RBF) is used for non-linear classification by applying the kernel trick [3]. Linear support vector classification (LinearSVC) penalizes the intercept, in comparison to SVM. Multilayer perceptron (MLP) is an artificial neural network that consists of at least three layers of nodes. MLP utilizes the supervised learning technique backpropagation for training.

Table 5.5 shows the results of the seven classification algorithms, each using the default configuration. The results include accuracy, precision, recall, F1-score, and area under the ROC curve (AUC) value. Among all, the random forest algorithm achieved the best scores on a balanced dataset of fake and regular reviews.

5.3.3 Optimization

We optimize the classifiers by performing feature selection and hyperparameter tuning. We optimize for precision only. In comparison to fake reviews, for regular reviews we were unable to create a gold-standard dataset. To create
Chapter 5. Detection of Fake User Feedback

To select features, we apply recursive feature elimination with cross-validation. After the classification algorithm assigned a weight to each feature, these are eliminated recursively by considering smaller sets. Per iteration, the least important feature is removed to determine their optimal number. We use the RFECV method from the feature selection module. The cross-validation is performed as described in the previous phase.

We received the best result with the random forest algorithm using all features. Nearly similar accuracies are already possible with fewer features, e.g., the precision with three features is 0.969, compared to 0.973 using all features, see Figure 5.11. The three selected features are 1) the total number of reviews the app received and 2) the user provided, as well as 3) the frequency in which the user provides reviews.

To tune the hyperparameters, we apply the grid search method GridSearchCV from the model_selection module. This method performs a cross-validated, exhaustive search over a predefined grid of parameters for a classification algorithm. After finding the optimal combination of parameters within the grid, this is further manually tuned by adding more values around the current best.

We achieved the best result using the random forest algorithm and parameters {'criterion': 'gini', 'max_depth': 30, 'max_features': 'sqrt', 'n_estimators': 300}. The parameter criterion measures the quality of a split. We used Gini impurity, which is intended for continuous attributes and faster to compute compared to Entropy. It is recommended to minimize the number of misclassifi-
5.3. Fake Review Detection Approach

Figure 5.12: Relative importance of extracted features to detect fake reviews.

Feature Importance

Last, we analyzed the relative importance of the extracted features with respect to the predictability of whether a given review is fake or not, called feature importance. The feature importance is calculated on how often a feature is used in the split points of a tree. More frequently used features are considered more important.

Figure 5.12 shows that the three most important features are the total number of reviews an app received (30%), the total number of reviews a user provided (24%), and the frequency with that a user provides reviews (13%).

We assume the total number of reviews received by an app is the most important feature as apps with a specific amount of reviews, e.g., 2-9 reviews (cf. Section 5.2), are most often targeted by fake reviews. The total number of reviews a user provided as well as review frequency have a high importance as fake reviewers provide much more reviews than regular reviewers, with a higher frequency. The percentage of one- and two-star ratings an app received are important, as the difference between those star ratings provided are the highest when comparing apps with fake reviews and regular apps (cf. Figure 5.6).
Chapter 5. Detection of Fake User Feedback

5.4 Discussion

We discuss the implications of our findings on works from the research area of app store analysis, as well as potential limitations and threats to validity of our study.

5.4.1 Implications

App store analysis covers work to mine apps and their reviews and extract relevant information for software and requirements practitioners, e.g., to get inspirations about what should be developed and to guide the development process [42, 49, 119, 123, 131, 157, 223, 224, 291]. Martin et al. provide a comprehensive literature review of this area [192]. The majority of papers identified in their study (127 of 187 papers, 68%) analyze non-technical information, such as app reviews.

Review analysis itself is one of the largest sub-fields of app store analysis, which receives a significant and increasing number of publications each year. Work in this sub-field started by analyzing the content of app reviews (2012 – 2013), afterwards focusing on adding additional features such as sentiments (2013 – 2014). Then, app reviews were summarized to extract app requirements. Although information extracted from app reviews is increasingly getting integrated into the requirements engineering processes [181, 226], none of the papers discusses the impact of fake reviews. In the following, we discuss the potential impact of fake reviews on software engineering along with the main review analysis topics according to Martin et al. [192].

Fake reviews, similar to official reviews, include requirements-related information, such as feature requests. Oh et al. automatically categorize app reviews into bug reports and non-/functional requests to produce a digest for developers, including the most informative reviews [218]. Additional work focuses on extracting requirements-related information from app reviews [132, 133, 227]. Iacob and Harrison found that 23.3% of the app reviews include feature requests [131].

We applied the classifier of Maalej et al. [183] to extract bug reports and feature request from the 8,000 official and 8,000 fake reviews included in our truthset (see Figure 5.13). While fake and regular reviews are imbalanced within the overall app store dataset, when applying review analysis approaches on a subset of reviews, the distribution is unknown. For this reason, we did not set a specific distribution of fake and regular reviews. Within the official reviews, we identified 1,297 bug reports and 921 feature requests, while we found that the fake reviews contain 362 bug reports and 521 feature requests. We include an example feature request included in the fake review dataset below.
5.4. Discussion

Figure 5.13: Reviews within truthset classified as bug report and/or feature request.

Nice UI ★★★★

*Very clean and beautiful UI. I like the goal setting and the reminders. I would like to see some animation when scrolling the weekly progress bars.*

We assume that most fake reviewers did not use the reviewed app before and are unfamiliar with it. In addition, review policies ask fake reviewers to explicitly talk about features rather than providing praise only. For these reasons, it is unclear if those feature requests are really relevant or only thought up to make the review sound more authentic. Fake reviews might thus impact the results of existing classifiers. When not removing or at least flagging fake reviews with information relevant for developers, wrong assumptions for the future decisions might be drawn.

Researchers showed that nearly half of the feature requests included in app reviews are implemented. Hoon et al. highlight that user expectations are changing rapidly, as observable through app reviews. Developers must keep up with the demand to stay competitive [128]. Palomba et al. studied the reviews of 100 open-source apps. By linking reviews to code changes, the authors showed that 49% of the changes requested were implemented in app updates [226]. These results show that a significant amount of changes proposed by users are integrated into software systems. Some of these changes might be inspired and prioritized based on fake reviews.

Recent approaches summarize and extract requirements-related information from app reviews of related or competing apps. Fu et al. present WisCom to analyze app reviews per app/market level, e.g., to get an overview of competing apps [92]. Gao et al. present AR-Tracker to summarize app reviews to real issues and prioritize them by their frequency and importance [93]. Nayebi et al. mine app reviews and tweets of similar apps within a specific domain [209, 210]. These approaches monitor and extract information from app reviews of related or competing apps. While developers will probably know when their apps receive fake reviews, i.e., when they bought those instead of being affected
Chapter 5. Detection of Fake User Feedback

by negative fake reviews bought by competitors, developers cannot be fully sure if competing apps receive fake reviews. Wrong conclusions can also be drawn from fake reviews including the honest opinion of reviewers. Fake reviewers can just copy and modify a regular review they honestly agree with and resubmit those on review exchange portals. This way the frequency (and hence the priority) of, e.g., a feature request, might be fake (i.e., incentivized) and thus biased. Table 3.3 highlights that only three review exchange portals forbid fake reviewers to copy and modify existing reviews. Using those reviews as input for summarization approaches, “wrong” features could emerge as a result.

While the overall number of fake reviews in app stores is unknown, multiple indices indicate a non-trivial amount of fake reviews in app stores. To get a first understanding of the possible amount of fake reviews, we applied our fake review classifier to the full official Apple App Store dataset. As a result, 22,207,782 (35.5%) of all 62,617,037 reviews were classified as fake. This number seems very high at first and can only be used as a first indication. Further studies need to be carried out to give a precise approximation of the number of fake reviews in official app stores.

In our study, we identified about 60,000 reviews from only a single provider. Overall, we identified 43 providers while much more might exist or have existed before. If every provider provided the same number of reviews, the amount would sum up to 2.58 million fake reviews. We hypothesize that the majority of fake reviews is written by persons who get directly asked by developers. Although not generalizable, we repeatedly observed this phenomenon in our professional app development settings. When apps are developed privately, friends were asked to provide fake reviews. When programmed in a commercial environment, either employees of the developing company or of the ordering company (cf. Bell [26]) are asked to provide fake reviews. Given that 1.4 million apps exist within the dataset, the number of fake reviews does no longer seem unattainably high. Such amounts of fake reviews are also presumed in other domains. Streitfeld [267] report that every fifth review submitted to Yelp is detected as dubious by internal filters.

5.4.2 Limitations and Threats to Validity

The results of our study might have limitations that should be considered within the study context. The process of extracting actual fake reviews was challenging. However, we did not try to generate fake reviews using a crowdsourced approach, as we wanted to only rely on fake reviews that have been published to and still remain unidentified within the app store by its users and operators. For this reason, we decided only using a subset of our about 60,000 collected fake reviews.

Nearly all reviews were extracted from a single provider. This provider is a review exchange portal (see Table 5.1, REP3). On these portals reviewers could
submit their honest opinion. However, we consider the collected reviews as fake for the following reasons:

First, app store operators require that app reviews must be written by real users of the app and cannot be incentivized. Both conditions are not given in review exchange portals. Therefore, according to the definition and agreement of app store providers, our collected reviews are fake. Even if reviewers are allowed to submit their honest opinion according to the review policy of this exchange portal, rewarded, incentivized, or non-spontaneous reviews are prohibited by the official Google and Apple App Store Review Guidelines [12, 233].

Second, review exchange portals provide predefined ratings and review messages. These ratings do not necessarily correspond to the opinion of the reviewers. The providers’ review policies ensure that reviewers that post their honest opinion are not being rewarded and are excluded from reviewing portals. The review policy of REP3 does not include a general rating, e.g., three-stars or above (cf. Table 3.3). The provider uses individual policies per app (cf. Figure 3.3). We could not extract historical data to say if all individual policies included a predefined rating. However, for active review requests at the time of data collection, predefined ratings were included in all cases.

Third, paid review providers and review exchange portals share reviewers. As identified, paid review providers cross-post their review requests on review exchange portals (cf. Figure 3.4). This also applies for REP3. Paid review providers would not cross-post their review requests on these portals, if the app ratings would not change as desired by app developers.

Fourth, per app we compared the collected fake reviews to each other. We searched for apps that received fake reviews with a rating of 1-2 stars as well as fake reviews with a rating of 4-5 stars. These reviews are most likely written by reviewers that posted their honest opinion about an app, either positive or negative. This applies for only 32 of the 1,890 (1.69%) collected apps. For these, 41 one-star and two-star reviews out of 8,607 reviews (0.48%) were provided.

Another limitation is that although we filtered fake reviews for reviews in English language only and targeted the US storefront of the Apple App Store, reviews in English language could have been submitted to other storefronts. This could be a possible reason why we were only able to identify 8,607 of the initially collected 60,431 fake reviews within the app store, i.e., the official reviews dataset. In this case, the moderation of reviews by app store operators is less strict than observed.

Further, review exchange portals are also used by app developers, since the reward of providing a fake review is a credit which can be redeemed into another fake review for an app specified. As a result, the amount of requirements-related information included in fake reviews could be influenced since some users of the portals might be app developers.
Chapter 5. Detection of Fake User Feedback

For in-app purchases, we were unable to receive the offers programmatically. Therefore, we could not compare the manually collected in-app purchases for apps affected by fake reviews against other app store in-app purchases. Also, we could not identify statistics that we could have used alternatively or statistics on the monetization through ads.

Another limitation is the chosen app store. We decided to focus on the Apple App Store because of our prior experience with the technology. Further, this app store does not impose major API limitations to retrieve its data, e.g., compared to Google Play, which limits the number of accessible reviews to 2,000 per app and uses captchas. To have reliable results for the Apple App Store, we crawled the largest dataset of about 62 million app reviews which has been analyzed so far to our knowledge. Thereby, we also avoid the App Sampling Problem for app store mining, described by Martin et al. [191].

When manually labelling data, such as when finding agreements for potential matches between reviews within the fake reviews dataset and official reviews dataset, we used two human annotators that independently solved the task. In the case of mismatches (3%), we resolved the conflicts using a third human annotator. However, single reviews could have been mismatched.

For statistical tests, we calculated the effect size in addition to the p value. For t-tests we calculated the effect size using Cohen’s d, i.e., the means’ difference divided by the pooled standard deviation [56]. For Wilcoxon tests, we calculated the r value, dividing the z distribution by the square root of the number of samples [91]. We report the effect size considering the following values, for Cohen’s d (0.2 = small, 0.5 = medium, 0.8 = large) and for the correlation coefficient r (0.10 = small, 0.30 = medium, 0.50 = large). In two cases, although a statistical difference was observed, the effect size revealed that the magnitude between differences is near zero. For these, the tests need to be repeated with additional samples, i.e., fake reviews, to show a statistical difference.

We want to stress that our classifier is only a first attempt to automatically identify fake reviews and not the main contribution of our study. We wanted to verify if the features we identified are relevant for identifying fake reviews. The machine learning model could be overfitted. This may be due to the small number of fake reviews. More fake reviews need to be collected to improve the results. We tried to minimize overfitting by using k-fold cross-validation. To minimize the impact of randomly chosen data, we used another 8,000 randomly selected official reviews and were able to reproduce our results.

Moreover, we cannot ensure that all official reviews are non-fake reviews. As stated before, to provide a gold-standard dataset for regular reviews as well, all fake reviews must be known and removed, which is the problem we try to solve in this study. For this reason, we optimized the classifier for precision only.

We leave the development of an advanced classifier for future research. Ad-
ditional features, e.g., the emotion of users [39, 186], have to be analyzed to strengthen the results. For that, we publicly share our gold-standard fake reviews dataset within our replication package [241].

5.5 Related Work

Despite the recent research on app store analysis, none of the works considers fake app reviews and their implications. While the phenomena of fake participation remains understudied in software engineering, it is well-known in other domains such as online journalism [68, 87, 168, 269] or on business and travel portals [86, 148, 204, 222] (e.g., in form of commenting, reporting or reviewing).

In Chapter 3, we summarized related work studying fake user feedback within other domains. In the following, we compare our approach to identify fake reviews to the existing approaches. In comparison to Jindal and Liu [148], who used duplicate and near-duplicate reviews written by the same reviewer on different products as fake reviews, we were able to extract data from fake review providers. This allowed us to achieve more reliable results. Mukherjee et al. [204] applied the supervised approach of Ott et al. [222] to a dataset including crowdsourced pseudo-fake reviews and to a dataset including actual fake reviews. On the dataset of actual fake reviews the authors achieved a significantly lower accuracy of 67.8%, compared to 89.6%. The authors found that the word distribution of pseudo-fake reviews is different from the word distribution of real reviews, which does not apply for fake reviews. We followed the approach of Ferrara et al. [87] and use non-textual features only to identify fake reviews.

5.6 Summary

Fake reviews are written to look authentic and are hard to recognize by humans. In this chapter, we identified differences between fake and official reviews. We found that the properties of the corresponding app and reviewer are most useful to determine if a review is fake. Based on the identified differences, we developed, trained, fine-tuned, and compared multiple supervised machine learning approaches. We found that the Random Forest classifier identifies fake reviews, given a balanced distribution of fake and regular reviews, with a recall of 97% and AUC/ROC value of 99%.

Applying our classifier to identify and remove fake reviews, increases the authenticity and, therefore, quality of user feedback. Our work helps app store mining researchers to sample apps and perform data cleaning to achieve more reliable results. Tools for app users and store operators can be built based on our findings to detect if app reviews are trustworthy and to take further actions.
Chapter 5. Detection of Fake User Feedback

against fake reviewers. Further, the improvement in the quality contributes towards the integration of user feedback into continuous software evolution practices. We publicly share our gold-standard fake reviews dataset to enable the development of more accurate classifiers to identify fake reviews.
Chapter 6

Context Augmentation of User Feedback

The approach to augment explicit user feedback with context information is based on and extends the paper “Extracting and Analyzing Context Information in User-Support Conversations on Twitter” [188] by Martens and Maalej, published at the 27th IEEE International Requirements Engineering Conference in 2019. My contribution to this publication is conducting all research, implementation, and analysis work, as well as leading the writing of the paper. My co-author supervised the work, inspired the research questions and goals, and made final edits to the paper.

Users include development information, such as non-crashing bug reports, in their feedback explicitly provided via app stores, social media, and user forums. In case this feedback misses relevant context information, such as the app version or device concerned, developers might be unable to understand and reproduce the reported issue. In Chapter 4, we show that support teams try to improve the quality of user feedback by making reported issues actionable to developers. Therefore, support teams clarify missing context items in effortful conversations with reporting users – for popular apps, such as Spotify or Netflix, in about ten parallel conversations per hour. Developers have underlined that this manual effort is unfeasible and tool support is needed.

To support users in providing high-quality user feedback, that is actionable to developers, we introduce two complementary automated approaches to exchange context information with the least possible effort. The first approach implicitly collects relevant and correct context information during app executions. It supports users in providing actionable feedback by attaching the captured context information. The second approach analyzes already existing user feedback, e.g., provided via social media. It extracts the context information in-
Chapter 6. Context Augmentation of User Feedback

cluded and utilizes a chatbot to explicitly request missing items from reporting users. The second approach is complementary to the first and can be applied to feedback reported without implicitly captured context information. We use the collected information to auto-populate issue trackers with structured bug reports that are actionable, i.e., understandable and reproducible, to developers.

The remainder of this chapter is organized as follows. Section 6.1 introduces the implicit context augmentation approach and its implementation challenges encountered on the iOS platform. Section 6.2 describes the explicit context augmentation approach and discusses the chatbot implementation as a web-based application. Afterwards, we discuss the implications and limitations of our findings in Section 6.3. Finally, we survey related work in Section 6.4 and summarize the chapter in Section 6.5.

6.1 Implicit Context Augmentation

This section presents an automated in-situ approach that attaches implicitly captured context information to the feedback provided by users. The approach is relevant for newly created user feedback that has not been publishing, e.g., to app stores or social media, already. Further, this section summarizes the challenges encountered when realizing the approach on the iOS platform.

The approach's focus is to support developers in understanding and reproducing reported issues, especially non-crashing bugs (cf. Section 2.1.2). These bugs may appear in the form of unexpected app behavior, such as inaccessible buttons or incorrectly displayed app views. Non-crashing bugs cannot be reported automatically using state-of-the-art tools, such as Crashlytics [89] or Apple CrashReporter [274]. Consequently, users need to manually inform developers through app stores, social media, and user forums. Unfortunately, users often miss to provide context data, such as the app version or steps to reproduce, needed by developers [34, 75, 202, 224, 312].

6.1.1 Approach

The approach can be divided into the context capturing phase, where implicit feedback is automatically collected, and the report creation phase, where explicit feedback is manually provided. Both explicit and implicit feedback constitute a structured bug report. Figure 6.1 visualizes the approach.

Context Capturing Phase

In the first phase, the approach automatically captures implicit feedback during the app execution. The collected implicit feedback consists of the execution context and the interaction context (cf. Section 2.1.3).
6.1. Implicit Context Augmentation

The execution context includes data related to the application-, device-, sensor-, and system-states. The information is extracted using APIs of the operating system. The interaction context includes the steps required to reproduce the user behavior including visited app views, tap-, scroll-, and swipe-gestures. To collect this information, the approach observes the user while interacting with the app. In addition, a unique user identifier and a timestamp are attached.

The capturing process is reset with every screen un-/lock of the smart device. Studies showed that the average length of a usage session on smart devices lasts for about three to four minutes [64, 147].

Report Creation Phase

In the second phase, the user manually provides explicit feedback. The phase is activated in-situ by shaking the device when observing a non-crashing bug.

Figure 6.2 depicts the second phase. The left screen shows the app view on which the user observes the non-crashing bug. In our example, the user notices that the button to create a private message is not working.

The middle screen shows a report form that appears within the app after shaking the device. The form allows the user to enter a textual description of the observed behavior. Further, it shows a screenshot preview of the affected app view, where the process was triggered. The screenshot should help the developers to understand the reported behavior.
Chapter 6. Context Augmentation of User Feedback

Figure 6.2: Screenshot of the report creation form allowing the user to provide explicit feedback in the form of a textual description and annotated screenshot (e.g., to hide private data).

The right view appears after tapping the screenshot preview. It allows the user to visually annotate the screenshot, for example, by drawing a circle to highlight a particular part of the displayed view. The user can also hide parts, e.g., containing critical private data. In the example, the screenshot contains the user’s private message history. The user annotates the screenshot to hide this non-relevant area of the screen and highlights the non-functioning UI element.

6.1.2 Implementation Challenges

We realized our approach on the iOS platform. In the following, we present the main implementation challenges faced, as well as their enabling technologies. In particular, we use code injection to capture context information independently from the app implementation. To track distinct and reproducible user interactions, we generate unique identifiers for app views and events. Finally, we combine unique identifiers with the accessibility framework to allow the automated reproduction of user interactions.

App Independent Context Capturing

We use code injection to capture user interactions without requiring an additional modification of the monitored app’s and underlying system’s source
6.1. Implicit Context Augmentation

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UIViewController</td>
<td>-viewDidAppear:</td>
<td>Template method of the iOS view-lifecycle that is called after the view controller starts appearing.</td>
</tr>
<tr>
<td></td>
<td>-sendEvent: and -sendAction:to: from:forEvent:</td>
<td>Methods dispatch actions to given target objects. All user interactions with UIControl objects pass these methods.</td>
</tr>
<tr>
<td>UIGestureRecognizer</td>
<td>-initWithTarget:action:</td>
<td>Method is called when a gesture recognizer is initialized. Used to create a list of gesture recognizers to observe.</td>
</tr>
</tbody>
</table>

We focus on common UIKit-based apps. To capture interactions, four methods of the UIKit library need to be instrumented, shown in Table 6.1.

There exist two official techniques to instrument methods of iOS libraries without access to their source code. These are method swizzling and extensions. The combination of both allows invoking behavior extending methods whenever relevant methods of the UIKit library are executed. These techniques are allowed in the Apple App Store and used by popular iOS libraries such as Crashlytics or Google Analytics. Compared to Android, there exists no possibility to capture user interactions globally on the system level for all installed iOS applications.

Extensions are used to define additional methods of existing classes without subclassing. The source code of classes to be extended does not have to be available to developers.

Method swizzling is provided by the Objective-C 2.0 runtime and is used to change method invocations at runtime. It changes the mapping of selectors to underlying functions in a class dispatch table. Given two methods, executing `-methodB` whenever `-methodA` is invoked, can be achieved by mapping the selector of `-methodA` to the implementation of `-methodB`. This combination enables developers to modify the behavior of methods when source code is unavailable.

Figure 6.3 shows the implementation of method swizzling within an existing class’ `+initialize` method by using an extension. The Objective-C runtime automatically invokes this method before the app calls the first method of the corresponding class. By applying method swizzling, the `-trackView:` method is executed whenever the `-viewWillAppear:` method of the UIViewController is invoked, i.e., whenever an app view appears. As the mapping of both methods has been exchanged, line 15 calls the original `-viewWillAppear:` method and continues the regular program flow. Within the swizzled methods (line 16), we implement the tracking of user interactions.
extension UIViewController {
    open override class func initialize() {
        guard self === UIViewController.self else { return }

        let originalSelector = #selector(UIViewController.viewWillAppear)
        let swizzledSelector = #selector(UIViewController.trackView)

        let originalMethod = class_getInstanceMethod(self, originalSelector)
        let swizzledMethod = class_getInstanceMethod(self, swizzledSelector)

        method_exchangeImplementations(originalMethod, swizzledMethod)
    }

    func trackView(animated: Bool) {
        self.trackView(animated) // calls viewWillAppear(_:)
        print("Track view: \(self)")
    }
}

Figure 6.3: Combination of extension and method swizzling used to implement the app-independent tracking of context information.

Generation of Global View Identifiers

Each app view is assigned a unique global identifier. Thereby, we enable developers to precisely reproduce user interactions without guessing, e.g., which button to tap. While Android generates unique identifiers for static app views on compile-time, there exist no automatically generated identifiers on iOS.

Unique identifiers must fulfill several criteria. These must be equal after every application start, across different devices with various screen sizes and system versions. Thus, dynamic values such as the current time or absolute tap positions in the form of coordinates cannot be used. Instead, we use the UIKit view hierarchy. Figure 6.4, on its left side, shows an app view as it is displayed to the user. The right side reveals that this app view is composed of four single views. The views are arranged in three layers. Each view within a layer has an index. For example, the index of the green view is 2. Besides depicting content, views can also act as containers for other views. If a view contains another, a parent-child relationship exists. The child view is called subview, while the parent view is called superview. The blue view, e.g., is the superview of the green view. Vice versa, the green view is the subview of the blue view. The yellow view is a subview of the blue view as well.

We generate the unique identifier of an app view in four steps using its index and relations. The identifier of the red button, for example, is generated as follows:
6.1. Implicit Context Augmentation

Figure 6.4: Using the UIView hierarchy to generate unique view identifiers.

1. Create a string containing the view’s name: **RedButton**.

2. Append the view’s index: **RedButton1**.

3. Iteratively append the view’s superview name and its index. The superview of **RedButton** is **GreenView** with the index 2. The superview of **GreenView** is **BlueView** with the index 1. For **BlueView** there exists no superview, therefore the iteration ends. The view’s resulting identifier is: **RedButton1GreenView2BlueView1**.

4. Generate an MD5 hash of the string. This step is useful for apps with deeply nested views where the view identifiers can be particularly long.

On iOS, there exist several types of UI elements, such as tables, which can contain dynamic content. For tables, the generation of unique view identifiers is extended with additional parameters. For each view within a cell, the cell’s section and row number are appended to the identifier, to be able to differentiate interactions in case the same cell is used multiple times within a table. Additionally, we add a property to the NSObject class, which can be used by developers to set custom identifiers for regular and custom app views manually.
Automated Reproduction of User Interactions

Record-and-replay tools, such as ReplayKit [240] or AppSee [16], capture user interactions to help developers understand issues. These tools record interactions in the form of videos. Compared to our approach, this has a significant impact on the app’s performance. Also, videos are costly to transfer over the mobile network. When watching the videos, developers have to remember the user interactions to manually reproduce the behavior, for example, on different devices or system versions. Compared to our approach, current record-and-replay tools do not offer strategies to validate and isolate context information. Therefore, videos of duplicate or similar issues cannot be automatically summarized. These must be manually analyzed by developers, which – depending on the number of videos – can be time consuming and error-prone.

Our approach utilizes the UIAccessibility framework [285] to automatically replay user interactions across different devices and system versions. The framework is intended to be used by developers to provide accessibility information about UI elements. Developers can, e.g., provide a descriptive message for a small button that can be read to partially sighted users with the VoiceOver functionality. The descriptive message can be stored in various properties defined by the protocol, most likely in `accessibilityLabel`.

Functional testing frameworks, such as Keep It Functional [156], utilize the descriptive messages as identifiers for UI elements. By iterating a given view’s subviews, the library finds the desired UI element and interacts with it as defined in the replay script.

Our approach automatically sets the previously generated unique identifiers for each app view as descriptive message during reproduction using method swizzling. Since interactions, such as taps, are tracked, the approach can optionally generate replay scripts for functional testing frameworks to reproduce users’ behavior automatically. Alternatively, the identifiers can be used to highlight relevant UI elements during reproduction using colors.

6.2 Explicit Context Augmentation

This section presents an automated approach that explicitly requests missing context information from reporting users using a chatbot. The approach is relevant for existing user feedback, e.g., provided via social media, that has been submitted without the use of the implicit context augmentation approach. Further, this section discusses the approach implementation as a web-based app.
6.2. Explicit Context Augmentation

(1) Tweet Classification Phase
(2) Context Extraction Phase
(3) Issue Creation Phase

Figure 6.5: Explicit capturing approach setting to auto-populate issue trackers with structured bug reports including context items mined from user tweets.

6.2.1 Approach

In Chapter 4, we presented an approach that automatically extracts basic context items from tweets, including the platform, device model, app version, and system version. This approach is intended to be used in combination with a feedback classification and a chatbot approach to auto-populate issue trackers with structured bug reports mined from user feedback. The overall setting can continuously be applied, e.g., to an app’s Twitter support account to identify relevant development tweets and their included context items, as well as to automatically request missing information from reporting users.

The approach can be separated into four phases, as shown in Figure 6.5. The second phase is covered in Chapter 4. We briefly describe each of the phases.

Conversation Classification Phase

In the first phase, tweets addressed to the app’s support account are classified by their types of requirements-related information. Only tweets reporting bugs, i.e., issues that potentially require context items to be understandable and reproducible by developers, are passed to the next phase. Tweets including other types of information, such as praise (e.g., “This is the greatest app I’ve ever used.”), are excluded from further analysis. These do not require context items and a chatbot requesting such information would annoy app users.

Context Extraction Phase

In this phase, our context extraction approach is applied to single tweets or conversations consisting of multiple tweets, that report bugs. Each tweet is mined to extract the four basic context items, including the platform, device,
app version, and system version. For example, the tweet “The app widget has died and is now a rectangular black hole. Xperia xz3 running Android”, includes the device and platform. After processing a complete conversation, the approach verifies if all four items could be extracted.

**Context Clarification Phase**

If the four basic context items could not be extracted, a chatbot requests the missing information. In case of the example above, the chatbot would request the app version and system version by replying to the tweet: “Hey, help’s here! Can you let us know the app version you’re running, as well as the system version installed? We’ll see what we can suggest”. The conversations are periodically analyzed to see if the user provided the missing context items.

**Issue Creation Phase**

Once all context items are present, these are used to create a structured bug report within the app’s issue tracker. The comment section of the issue tracker remains connected with the conversation on Twitter so that developers can directly communicate with the reporting user to ask for further clarification or inform the user once the issue is fixed.

By automatically requesting missing context items, our approach reduces the manual effort for support teams and aids developers by addressing the challenges mentioned above to facilitate actionable bug reports.

### 6.2.2 Implementation Challenges

We present the realization of our approach as a web-based app. In the following, we describe the implementation challenges encountered, separated by the conversation classification, context clarification, and issue creation phase.

**Conversation Classification Phase**

In the first phase, the conversations or single tweets of the support account are continuously crawled and classified to identify relevant development information, such as bug reports. Only these tweets are passed to the next phase.

Tweets can be crawled using different strategies. For example, the Twitter Search API [284] is a paid service that offers to access data on Twitter via a REST API. Twitter offers several versions that differ regarding the rate at which tweets can be crawled and the amount of historical data that can be retrieved. The standard version of the Twitter Search API allows retrieving the tweets of the last seven days at a rate of 720 requests per hour, where each request can return a total of 100 tweets. This results in 72,000 tweets that can be crawled
6.2. Explicit Context Augmentation

Figure 6.6: Example response of tweets included in Twitter timeline.

per hour. The premium version allows retrieving all tweets since the release of
Twitter at a rate of about 2 million tweets per hour.

Alternatively, the tweets can be directly scraped from the Twitter website.
While this is not allowed for commercial projects, we use this strategy for our
research setting to avoid costs and be able to access all historical data without
a rate limit. We use the open-source tool TweetScraper [151], which is based
on Scrapy [250]. The tool allows parallelizing the scraping of the tweets. We
configure TweetScraper to extract the timeline of an official Twitter support
account by setting its input query to:

```
scrapy crawl TweetScraper -a query="to: SpotifyCares , from: SpotifyCares"
```

Internally, TweetScraper crawls the account timeline by accessing the URL
https://twitter.com/i/search/timeline?q=@[account]&f=tweets&max
position=.

An example response, including the paginated timeline, is shown in Fig-
ure 6.6. The tweets are wrapped into a JSON within the property `items_html`
(line 4). The JSON includes two additional properties. The first property `min_position` (line 2) indicates the position of the last tweet included the re-
sponse within the overall timeline of the crawled account. To paginate through
the timeline, this parameter is set as the `max_position` query parameter of the fol-
lowing request. The second property `has_more_items` (line 3) indicates whether
the timeline includes additional tweets that can be requested.

Figure 6.7 summarizes the information included per tweet. The `conversation_id` uniquely identifies and can be used to access the overall conversation the
tweet corresponds to. The unique `tweet_id` can be used to access the tweet it-
self. The property `date_created` includes the tweet’s date of creation. The actual
tweet is included in `tweet_text`. The `user_id` uniquely identifies the author of the
tweet, while `user_name` includes the author’s name. The properties `reply_count`
include the number of replies to the tweet, `retweet_count` how often it has been
retweeted, and `favorite_count` the number of users who favorited the tweet.
Chapter 6. Context Augmentation of User Feedback

Figure 6.7: Information included in a tweet.

TweetScraper paginates through the entire account timeline and persists all extracted tweets within a database. As the timeline does not include all tweets of a conversation, we modify TweetScraper to extract these by also crawling the URL https://twitter.com/[user_id]/status/[conversation_id] per conversation.

Afterwards the persisted tweets are grouped by their conversations. Each conversation’s first tweet is classified to determine whether it reports relevant development information in the form of bug reports. Conversations that report bugs are flagged and further processed within the next phase. Other conversations that include irrelevant information, such as praise, are ignored.

To classify the tweets several approaches exist [117, 265, 302] (cf. Section 2.3.1). We use the classifier of Stanik et al. [265], which combines traditional machine learning with deep learning. The classifier can be applied to app reviews and tweets written in English and Italian language. The classifier separates the tweets into the categories problem report, inquiry, and irrelevant. For the following phase, we only consider tweets of the category problem report.

Context Clarification Phase

In the third phase missing context information is requested from reporting users with a chatbot approach. This phase is optional and only applied to bug reporting conversations that miss basic context items, including the platform, device model, app version, and system version. To determine if this applies to a conversation, the chatbot takes the results of the previous two phases, i.e., the classification and context extraction phase, as input.

The chatbot itself is based on the official Twitter for Node.js package [283]. The package provides an interface to authenticate and communicate with the Twitter API. To access the API, the chatbot needs to be registered as an application on the Twitter Developer portal. After registering, Twitter provides a consumer API key and a consumer API secret key, as well as an access token.
6.2. Explicit Context Augmentation

```javascript
var Twitter = require('twitter');
var client = new Twitter({
  consumer_key: '',
  consumer_secret: '',
  access_token_key: '',
  access_token_secret: ''
});
var params = { screen_name: 'nodejs'};
client.get('statuses/user_timeline', params, function(error, tweets, response
) {
  if (!error) {
    console.log(tweets);
  }
});
```

Figure 6.8: Template code of the Twitter package for Node.js.

and an access token secret. Figure 6.8 lists the template code of the Twitter package, which utilizes the provided information.

If a bug reporting conversation misses one or several context items, a response to the reporting user is generated that requests the missing information. For example, if the user did not provide the device model that has been used when experiencing an issue, the chatbot generates a message as follows: "Hey, help’s here! Can you let us know the device model you are using? We’ll see what we can suggest”. Afterwards the tweet is published as a response to the existing conversation. Figure 6.9 lists the publishing of a response using the Twitter package and API.

This phase, as well as the previous two, is continuously repeated to extract additional conversations and tweets. These new tweets possibly include the responses of users, including the context items requested by the chatbot.

```javascript
bot.post('statuses/update', {
  status: '@User Hey, help’s here! Can you let us know the device model you are using? We’ll see what we can suggest.
}, function(error, tweet, response) {
  if (error) {
    console.log(error)
  } else {
    console.log('Successfully tweeted response: ' + tweet.text)
  }
});
```

Figure 6.9: Chatbot functionality to respond to a tweet in order to clarify missing context information.
Issue Creation Phase

Once all basic context items are present, the chatbot no longer requests additional information for the corresponding conversation. Using the information collected, the chatbot automatically creates an issue within the issue tracking system. We decided to use GitHub [38]. Figure 6.10 shows the issue as it is displayed to developers.

GitHub offers a REST API to list, create, and edit issues [99]. To access the API, there exist official libraries for different programming languages. We use octonode, which is the library for Node.js to create issues [231]. Figure 6.11 lists the creation of an issue. As for accessing the Twitter API, credentials in the form of an identifier and secret key (line 2-3) are required to authenticate the client. After authentication, the client accesses the specific repository (line 15) and creates the issue, including the basic context items (line 18-28).

If the issue remains unclear, although basic context information is available, the developer can comment on the issue (cf. Figure 6.10). This comment is automatically published as a tweet to the existing conversation on Twitter. Vice versa, the following tweets are added in the form of comments to the issue on GitHub, e.g., in case the developer requested additional context information.
6.2. Explicit Context Augmentation

```javascript
// Bug report and basic context items
var report = "I am unable to open the application settings via the settings icon in the top right corner of the app. Opening the settings via the main menu works for me!"
var reporter = "mrtnsd"
var platform = "iOS"
var device = "iPhone 6"
var appversion = "1.1.2"
var sysversion = "11.1"

// Authenticate access to GitHub API
var client = github.client({
  id: '',
  secret: ''
});

// Access the repository
var ghrepo = client.repo('mrtnsd/context-demo');

// Create an issue
ghrepo.issue(
  "title": "Bug reported by @" + reporter + " via Twitter",
  "body": report + "

**Context Information:**

Platform: " + platform + "
Device Model: " + device + "
App Version: " + appversion + "
System Version: " + sysversion,
"labels": ["bug"]
), callback);```

Figure 6.11: Creating issues using octonode and the GitHub API.
6.3 Discussion

We discuss the implications of our findings. Then, we list potential limitations of both the implicit and explicit context augmentation approach.

6.3.1 Implications

Using a chatbot approach to clarify missing context information in explicitly provided user feedback lowers the effort for support teams to increase the actionability (i.e., understandability and reproducability) of reported issues to developers. However, automatically requesting missing context information may not be required for every issue. This overuse of bots might lead to both information and interruption overload [166]. Studies highlight that the use of bots in software engineering should be carefully considered as these can both increase and decrease developers’ productivity [266].

Lee et al. report that the presence of bots impacts users’ behavior on communication platforms [169]. In their survey, more than half of the participants disagreed that bots improve social interaction. This might result in users that no longer provide requested context information when asked by a bot instead of a human. Murgia et al. [206] built a bot to answer simple questions on the StackOverflow platform that are being asked repetitively. In an experiment, the bot once impersonated a person and in the next iteration, was revealed as such. Despite equal functionality, users adapted better with the bot impersonating a person. For all available metrics to rate the provided answers, this bot achieved better scores, in the form of more accepted answers, more up-voted answers, and less down-voted answers. The authors reported that humans have a negative bias when reading answers provided by bots, as well as a lower error tolerance.

Studies show that the adoption of bots through users can be supported by giving the bot a personality. For example, the name and language influence how users interact with the bot. The bot’s language should be “casual, accessible, friendly, and fun” [166]. However, the best practices of, e.g., Slack underline that the personality should not be too artificial. For example, one suggestion states “Informality is good, but getting over-friendly is going to be charming to a very small number of people. […]” [294].

Bots have to continuously evaluate their success [169], i.e., the number of context items provided by users, with regard to their strategy applied. For example, it could be less daunting for users not to be asked for all context items at once. Moreover, the conversations between users and the chatbot should be optimized. For example, the bot might already suggest some context items the user can simply confirm instead of entering these manually. This can be supported by our context capturing approach, which we consider as complementary.
6.3.2 Limitations

Although context information is present, bugs might not be reproducible by developers. For example, this applies to bugs in apps with highly dynamic and user-specific content, such as Facebook. These bugs might depend on the information displayed at the moment the user submits the issue. To reproduce issues that depend on the exact content displayed to users, our context extraction approach must be extended also to capture the network traffic with respect to privacy regulations. By replaying the network traffic, these issues can be reproduced as users observed them.

Our context extraction approach has several technical limitations, of which we are aware. Its implementation is currently limited to UIKit-based apps. We assume the majority of apps in the Apple App Store to be UIKit-based, but were unable to identify numbers on the apps’ distribution. OpenGL games and web-based apps (e.g., created with phonegap, Sencha, etc.) are not supported. However, we created a prototypical implementation for web-based apps which was able to capture the required context information.

Further, we chose the abstraction level “single widget interaction” [246] to ensure the proportionality principle of the privacy, and do not track passwords to minimize privacy risks. However, this input is required for anonymization approaches to replay interactions that require a login and are dependent on the user’s information [50].

6.4 Related Work

App store analysis recently emerged as a new trend in software and requirements engineering research. Martin et al. [192] found that several researchers extract bug reports and feature requests from app reviews [131, 180, 292]. This confirms the assumption underlying our work, that app users are willing and able to report issues as well as ideas. The authors also report that negative feedback often misses user experience and contextual information.

To solve this issue, Maalej et al. [181] suggest improving the quality of reviews by automatically collecting execution data, logs, and user interaction traces. Our approach follows this approach by enriching app reviews with structured contextual information, while reducing the overhead for users to provide, and helping developers to understand issues.

Several researchers developed approaches to support crowd involvement in software requirements and evolution tasks. Seyff et al. present iRequire [253] and AppEcho [254], two manual approaches that enable users to report requirements in-situ. In a step by step review process, users can provide feedback in the form of pictures, audio recordings, and textual descriptions to help developers understand and transform user feedback into well-defined requirements.
Chapter 6. Context Augmentation of User Feedback

descriptions. Our approach adopts the in-situ feedback approach but is based on a simple yet popular review form as known from app stores. The approach mainly focuses on non-crashing bug reports rather than feature requests. Also, it automatically captures helpful context data in the background to lower the effort for users and to reduce the amount of erroneous information provided.

Crash reporting is a more conventional research line in software engineering. Liblit [172] highlights that current crash reporting systems only capture information about the crash itself. Information before the crash and about successful runs is excluded. The author proposes capturing a small amount of information about every application run to extract an accurate picture of how software is used in practice – as our approach does. Agarwal et al. [2] introduce MobiBug which monitors app executions within the crowd to detect crashes.

The underlying assumption behind the idea of our approach is similar but is specified to what happens before issues are reported by users. Our approach goes beyond crash reporting systems, such as Apple CrashReporter or Crashlytics, that send memory dumps to a central failure repository and mainly focus on error localization within the source code, by providing detailed context data and user interaction traces that help developers to understand and reproduce non-crashing issues.

Recently, commercial and open-source tools emerged, such as InstaBug [57], BugSee [35], or BugLife [36], that allow users to report non-crashing bugs in-situ, similar to our implicit context augmentation approach. All tools capture the execution context, such as information related to the app and system. InstaBug also captures the user interactions as a trace of elements, similar to our approach. BugSee captures the users’ interaction in form of videos. Capturing videos has a significant impact on the app’s performance and their transfer over the mobile network is costly. Further, these must be analyzed manually which can be time consuming and error-prone. BugLife does not capture user interactions. InstaBug, in addition, captures the network data, e.g., to understand and reproduce temporary bugs. Unfortunately, these tools cannot be applied to already existing user feedback, e.g., reported via app stores or social media.

Research [166] and industry [299] highlighted the need for bots to meet the increasing demands in software engineering. These bots are intended to be used in different ways, e.g., to automate tasks of humans or support humans in performing tasks that can not be entirely automated [166]. Especially the recent advances in data analysis and natural language processing [166, 195], e.g., in the form of automated user feedback mining, allow bots to support the work of development teams through automatically provided additional information.

Many bots exist and are already applied in practice, e.g., to manage issues [97, 98, 298], improve source code [53, 111, 232], or support continuous integration [66, 288]. Wessel et al. found that 26% of their analyzed GitHub
projects used bots to automate tasks [299].

Erlenkov et al. [78] summarize current and future bots in software development. The authors highlight that bots are often only a “little more than glorified scripts, with hardcoded functionality and little intelligence”. Future bots have to make increased use of artificial intelligence. Lebeuf et al. propose three dimensions to characterize the intelligence of a bot, these are adaption, reasoning, and autonomy [166]. Adaption refers to bots that apply context-awareness to decide how these interact with users. Reasoning divides bots into those that use simple logic rules and others that use advanced artificial intelligence. Last, autonomy describes if the bots act entirely autonomous or rely on human input before interacting.

According to these three dimensions [166] our bot is to be rated as intelligent, as it has a high adaption (i.e., is context-aware in the sense of which context items are requested), a high reasoning (i.e., combines simple approaches, such as text and search patterns, with more complex, such as word vector models that are trained using deep learning [154]), and is completely autonomous from the perspective of developers (i.e., does not require their input for training the word vector models, clarifying the missing information, or creating the issues).

6.5 Summary

Context information helps developers to understand and reproduce reported issues, such as non-crashing bugs. However, users often miss to provide relevant context items, which results in low-quality issues that are non-actionable to developers. Support teams engage in effortful conversations with users to extract missing context information.

To support both users and developers in exchanging context information with the least possible effort, we present two complementary automated approaches and their implementation challenges. The implicit context augmentation approach automatically collects relevant and correct context items during app executions. When users decide to report a new issue, this information is attached to their explicitly provided feedback. For issues that have been reported without automatically captured context information, the explicit context augmentation approach mines the user feedback provided via official app support channels. It automatically requests missing context items from reporting users by analyzing and replying to their feedback using a chatbot. The collected information is used to auto-populate issue trackers with structured bug reports that are actionable to developers.
Chapter 6. Context Augmentation of User Feedback
Chapter 7

Crowdsourced Isolation of Context Information

User feedback might remain unclear to developers, although structured context information is available. If an anomaly is not immediately visible, developers must configure their testing device exactly as reported to reproduce the observed behavior, which is time-intensive. To determine which context items, for example, the system version, are relevant to reproduce the observed behavior, we integrate the quality-improved authentic user feedback into continuous software evolution practices. Therefore, we validate the reported behavior via crowd-sourcing. In this chapter, we present a context isolation approach that identifies recurrent patterns to isolate the occurrence of reported non-crashing bugs, using context information of the initial reporter and additional validators.

The isolation of the provided context information might also be relevant, e.g., for bugs that are dependent on the events performed. In case the reporting user performs many interactions with the application user interface, the resulting long interaction trace might overwhelm developers. Moreover, without asking additional users to validate the occurrence of a reported bug, the range of affected devices might remain unclear to developers. This is especially relevant for prioritizing issues, to fix the bugs that affect most users as quickly as possible. As bugs remain in issue trackers for a certain time before being considered by developers, this time might also be used to remove false reports or temporary bugs through additional validations.

The remainder of the chapter is organized as follows. Section 7.1 presents our approach, as well as its ability to validate and isolate the occurrence of bugs by using crowd-sourcing. Section 7.2 introduces the framework architecture. Afterwards, we discuss potential limitations in Section 7.3. Finally, we survey related work in Section 7.4 and summarize the chapter in Section 7.5.
Chapter 7. Crowdsourced Isolation of Context Information

7.1 Context Isolation Approach

Figure 7.1 shows an overview of the context isolation approach. The approach uses a structured bug report as input. This report is created with the support of the context capturing approach presented in Chapter 6, and also includes a screenshot of the issue and the identifier of the affected app view. The context isolation approach itself consists of two main phases:

1) **Validate Issue.** If the submitted report is unclear to developers, although structured context information is available, or additional information is required to, e.g., prioritize reported issues, other app users are asked to validate it. When opening the affected app view, these users, i.e., validators, confirm whether they notice the reported bug. Their answer and implicitly captured context information is added to the central repository.

2) **Isolate Issue.** Afterwards, the implicit feedback of the reporter and all validators is used to isolate the occurrence of the reported bug by determining recurrent patterns, e.g., which system versions are affected or which user interactions facilitate the defective behavior.

In the following, we describe each phase in detail.

7.1.1 Issue Validation Phase

In the validation phase, additional users are asked to dis-/confirm whether being affected by the reported bug, when navigating to the corresponding app view. Depending on the developer’s preference, the validation process can be either triggered manually per report or as reports remain in issue trackers for a specific
time before being considered by developers [172], started automatically to gather additional information during this period. Reported bugs are validated in two steps, which we describe in the following.

Validation Sampling

First, a validation request is created, which includes the bug report’s identifier, textual description, screenshot, and the identifier of the affected view. Each validation request is assigned to several app users. To select these validators, each user is registered through the server by transferring the user identifier and execution context, when opening the app.

The execution context is used to select validators with different device models and system versions. Thereby, the occurrence of the reported bug is observed under different conditions. The execution context can also be used to select additional users, if specific sensor configurations are not present, such as GPS turned on and off. The unique user identifier is transferred to balance validation requests between users and to avoid asking the same users several times in a row to validate reports. An app flag can additionally be used to limit the validations to specific user groups, such as beta testers. To ensure the correctness of the answers, cross-validation can optionally be applied. The level of detail should be configured by developers, depending on criteria such as the overall number of app users, to avoid user distraction.

Bug Validation

The user interactions of the selected validators are observed during app runtime. When opening the app view, which is possibly affected by the reported bug, the user is presented a validation form, as shown in Figure 7.2. The form depicts the bug report’s textual description and annotated screenshot. The user can dis-/confirm the occurrence of the bug. The answer and the user’s implicit feedback is sent to the central repository. After collecting enough validations to match the level of detail configured by the developer, the report is marked as validated and no longer presented to users.

7.1.2 Issue Isolation Phase

The information submitted by the validators is continuously used to isolate the reported non-crashing bug. Research found that developers fail to identify erroneous configurations, already for a low number of features [198]. Therefore, reoccurring patterns within the execution and interaction context are identified. The patterns correlate with the report and help developers to understand and reproduce the bug. Moreover, the collected information can be used to prioritize issues by determining, e.g., the affected devices and corresponding users.
Chapter 7. Crowdsourced Isolation of Context Information

Figure 7.2: Screenshot of the validation request asking additional users if they are being affected by a reported bug.

Isolate Execution Context

When considering the execution context of only a single app user, it might remain unclear to developers which context items, e.g., sensor states, have an impact on the reported bug. If an anomaly is not immediately visible, developers must configure their testing device exactly as reported, which is time-intensive. Also, not only the single device model and system version of the reporter could be affected but a broader range of both. In the worst case, developers have to reproduce the reported bug on all supported devices, each with different system versions installed.

To support developers, our approach highlights important and hides irrelevant context items. Therefore, it compares the execution context of all validating users. Table 7.1 shows the execution context of five app users. These are separated into users affected by the bug (either reporting, or confirming), and users not affected by the bug. Each users’ execution context includes the device model and systems version as well as nine additional context items.

Device Model and System Version. First, the range of affected devices and system versions is determined. For users affected by the bug, the distribution of devices is iPhone 6 and 8, while the system version ranges from 10.3.3 to 11.1. For not affected users, the devices are iPhone 8 and X, and the system

144
Table 7.1: Example of isolated execution context reported by three confirming and two non-confirming app users.

<table>
<thead>
<tr>
<th>Context Item</th>
<th>State</th>
<th>Confirming Users</th>
<th>Non-Confirming Users</th>
<th>Priority</th>
<th>Suggested State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>User₁  User₂  User₃</td>
<td>Dist₀   User₄  User₅</td>
<td>Dist₁</td>
<td></td>
</tr>
<tr>
<td>Device Model</td>
<td>n/a</td>
<td>iPhone 6  iPhone 8  iPhone 8</td>
<td>iPhone 6  iPhone 8  iPhone 8</td>
<td>n/a</td>
<td>≤ iPhone 8</td>
</tr>
<tr>
<td>System Version</td>
<td>n/a</td>
<td>10.3.3   11.0  11.1</td>
<td>10.3.3 - 11.1</td>
<td>n/a</td>
<td>≤ 11.1</td>
</tr>
<tr>
<td>Bluetooth</td>
<td>Connected</td>
<td>True  True  True</td>
<td>False False</td>
<td>(100%)</td>
<td>False False</td>
</tr>
<tr>
<td>Disk Usage</td>
<td>≥ 90%</td>
<td>False  True  True</td>
<td>False  False</td>
<td>(66%)</td>
<td>False  False</td>
</tr>
<tr>
<td>Jailbreak</td>
<td>n/a</td>
<td>False  True  False</td>
<td>False  False</td>
<td>(33%)</td>
<td>False  False</td>
</tr>
<tr>
<td>Orientation</td>
<td>Portrait</td>
<td>True  False  False</td>
<td>False  False</td>
<td>(33%)</td>
<td>False  False</td>
</tr>
<tr>
<td>UI Orientation</td>
<td>Portrait</td>
<td>True  False  False</td>
<td>False  False</td>
<td>(33%)</td>
<td>False  False</td>
</tr>
<tr>
<td>Battery Level</td>
<td>≤ 10%</td>
<td>False  False  False</td>
<td>False  False</td>
<td>(0%)</td>
<td>False  False</td>
</tr>
<tr>
<td>GPS</td>
<td>Connected</td>
<td>False  False  False</td>
<td>False  False</td>
<td>(0%)</td>
<td>False  False</td>
</tr>
<tr>
<td>Internet</td>
<td>Connected</td>
<td>True  True  True</td>
<td>True  True</td>
<td>(100%)</td>
<td>True  True</td>
</tr>
<tr>
<td>Memory Usage</td>
<td>≥ 90%</td>
<td>False  False  False</td>
<td>False  False</td>
<td>(0%)</td>
<td>False  False</td>
</tr>
</tbody>
</table>
versions are 11.2 as well as 11.3. Comparing both distributions, the approach suggests to use an iPhone 8 or older, and removes the iPhone X as the bug does not occur on this device. Further, a system version equal or below 11.1 is recommended, as all users that are not affected have a higher version installed.

**Additional Context Items.** Second, the reported context items are isolated. Therefore, each item is assigned a *distribution* value between 0 and 100%. The Bluetooth item, for example, has a distribution value of 100% for users observing the issue, indicating that these were connected via Bluetooth. For not affected users, the value is 0%, as none of these activated the Bluetooth sensor. Then, a *priority* is set for every item, depending on its distribution values. The priority is the absolute difference of both distributions. In this case, the Bluetooth item has a priority of 100. For context items with a high priority, our approach assumes that it is highly relevant to activate those when reproducing the bug. To support developers, context items are sorted by their priority. Items with a priority above 75 are directly presented to developers, while the remaining can optionally be displayed.

**Isolate Interaction Context**

The interaction context reflects the user’s interactions with the application user interface. Studies found that an average user interaction trace, when solving a defined task within an app, consists of 13 user interactions [65]. However, interaction traces can be longer, for example, when a user searches for multiple songs of different artists to create a playlist.

If the occurrence of a bug is dependent on the events performed, long traces might overwhelm developers. Our context isolation approach filters the interaction context and removes irrelevant interactions. The resulting reproduction path includes potentially relevant steps that facilitate the erroneous behavior. As for the execution context, the interaction context of both confirming and non-confirming users is compared. The isolation is performed on app view and on event level (cf. Section 2.1.3).

**App View Level.** Figure 7.3 shows a directed graph reflecting the interaction context of the three confirming and two non-confirming users on app view level. Node $v_1$ represents the initial view shown after app start. Node $v_8$ represents the app view where the review process was activated, i.e., the view that is affected by the reported bug. The nodes $v_2$-$v_7$ represent further views visited by the app users. The edges represent all possible transitions between these app views.

Within each app view, several events are performed by the users, such as tapping buttons, before transitioning to another view. These events might influence the occurrence of the reported bug. For each app view, the events are compared
7.1. Context Isolation Approach

Figure 7.3: (a) Unfiltered user interaction trace (on app view level), including all visited app views, by confirming and non-confirming users, and their possible transitions; (b) Filtered user interaction trace including all visited app views by confirming users only.

Figure 7.4 shows the filtered user interaction trace on event level. For each app view, it lists the possible events and whether the users performed these. Event $e_2$ on app view $v_6$ is the only event exclusively performed by all confirming users and therefore marked as mandatory (highlighted in red). The remaining events are performed by a) none of the users ($e_2$ on $v_3$), by b)
both users groups (e\textsubscript{1} on v\textsubscript{1}), or by c) single users of one user group (e\textsubscript{1} on v\textsubscript{7}). Those events can be excluded from the filtered interaction trace, as these do not follow a pattern indicating an influence on the occurrence of the bug.

If several possible paths exist to include the mandatory events, and their app views, between the starting view v\textsubscript{1} and the view affected by the bug v\textsubscript{8}, the shortest path is suggested to developers (cf. Figure 7.3). In case of our example the shortest path to include event e\textsubscript{2} on app view v\textsubscript{6} is v\textsubscript{1} — v\textsubscript{2} — v\textsubscript{6} — v\textsubscript{8}.

### 7.2 Framework Architecture

The approach implementation is based on the client/server reference architecture proposed by Maalej and Pagano [182]. Figure 7.5 shows an overview of the framework architecture. It consists of a client library that is integrated into the target mobile application and a server component. These aim to integrate the application usage environment and the application engineering environment by exchanging captured information over the Internet. Both client library and server component consist of four subsystems each. In the following, we describe the client library and server component in detail.

#### 7.2.1 Client Library

The client library consists of the communication system, cache, context system, and report system. It is integrated into the target mobile app. We briefly summarize each system in the following.

**Communication System.** The communication system has three responsibilities, it registers the user for validations of issues reported by other users,
receives the corresponding validation requests, and transfers completed validations. On app start, the communication system transfers the user’s execution context, e.g., including the app version or system version installed, to the server component and, thereby, registers the user as a potential validator. Also, it downloads the validation requests of issues reported by other users. If a validation request has been completed, it transfers the response and implicit feedback of the user to the server component.

**Cache.** The cache stores the validation requests of issues reported by other users, including identifiers and screenshots of the affected app views, as well as the issue descriptions. Both description and screenshot are only downloaded in case the user is asked to validate the issue. If an issue has been validated by the user, the cache persists the user’s response and implicit feedback locally on the device. The cache is also useful in case the user is temporarily not connected to the Internet.

**Context System.** The context system captures implicit feedback during app executions. It uses sensors to monitor the execution context, including information related to the app-, device-, sensor-, and system-states, and the user interaction context reflecting the user’s interactions with the application user interface. We described our approach to capture implicit context information as well as its implementation challenges encountered on the iOS platform in Section 6.1. Further, the context system compare the currently active app view to app views potentially affected by bugs, i.e., whose app view identifiers are included in the cached validation requests.

**Report System.** The report system asks users to validate issues reported by other users. Therefore, it presents a validation dialogue to the user after a certain delay when opening an app view potentially affected by a bug. The dialogue displays the issue description and screenshot, possibly annotated by the reporter, to the user (cf. Section 7.2). After the validator completed the dialogue, by either confirming or disconfirming the reported behavior, the report system persists all collected information within the cache.

### 7.2.2 Server Component

The server component consists of the communication system, database, validation system, and isolation system. We briefly summarize each system in the following.

**Communication System.** On server side, the communication system acts as a counterpart to the client library. Both systems interact via a representational
Chapter 7. Crowdsourced Isolation of Context Information

state transfer (REST) API. The communication system receives the registrations of the clients including their execution contexts. Each issue should be validated under different conditions, such as multiple system versions installed. Depending on the execution context transferred during registration, the communication system returns appropriate validation requests, e.g., that still need to be validated for the specific system version.

**Database.** The database maintains a three lists. The first list includes reported issues that need to be validated, while the second list includes potential validators and their execution context. The third list includes the validations that have been completed. This list creates a mapping between the issues reported and validators, either confirming or disconfirming the reported behavior.

**Validation System.** The validation system receives reported issues, that need to be validated, from the database. Based on their executions contexts transferred during registration, the validation system selects potential validators and distributes validation requests to these users through the communication system. If an issue has been validated under different conditions, the validation system passes the issue as well as the collected responses and implicit feedback to the isolation system.

**Isolation System.** The isolation system uses the completed validations to identify recurrent patterns within the collected implicit feedback, i.e., the execution and interaction context. It visualized the isolated context to the developer in a report view, as shown in Figure 7.6.

7.3 Discussion of Limitations

In this section, we discuss the limitations of the context isolation approach. Our approach to isolate context information uses structured bug reports as input. For the current implementation, these must include a textual description and screenshot of the issue, as well as a unique identifier of the affected app view. This information is automatically collected and attached to explicitly provided user feedback by our context capturing approach (cf. Section 6.1).

User feedback provided via app stores and social media does not contain app view identifiers. To apply the context isolation approach to this feedback, the approach must be modified, for example, by making users aware of outstanding validations immediately on app start. Without knowing the app view that is potentially affected by the reported bug, users can no longer be asked for their feedback in-situ. As a result, unnecessary validation requests might be displayed to users that, e.g., do not utilize the affected app functionality.
When being asked to validate an issue, users might – either or not on purpose – give the wrong answer. This circumstance might be improved by limiting the validations to particular user groups, such as power users that are intrinsically motivated to provide correct validations. Our approach should be extended so that users providing incorrect validations can be excluded. Another limitation are temporary bugs, which might be difficult to distinguish from incorrect validations.

Moreover, users might be annoyed when receiving too many validation requests. We transfer a unique user identifier on app start to be able to balance the validations between users. In this case, special user groups, e.g., paid software testers, might be a solution.

To be able to prioritize issues based on the range of affected devices, which can be determined during the validation of an issue, the number of app users that use a specific device must be known. Therefore, the execution context of all users must be transferred on app start, which reduces users’ privacy. Alternatively, public lists of the popularity of devices can be used as approximation.

Our approach to validate context information is currently limited to iOS. However, we expect the validations to work on other platforms as well. The isolation is performed on server-side and not limited to a specific platform.
Another limitation is that the context capturing approach must continuously observe users to collect their interaction context. To preserve the privacy of users, this information is only being sent to the central repository when being asked to validate an issue. Our implementation for iOS stores this information within a container that is only accessible to the app itself. For other platforms, the collected information must be stored similarly so that it can not be used by other apps or third-party software.

7.4 Related Work

Researchers proposed several approaches for the isolation of context information during bug fixing. Melo et al. [198] performed a controlled experiment to measure the impact of variability on the time and accuracy of bug finding. While the impact on accuracy is low, the authors found that bug finding time increases linearly with the degree of variability. Developers fail to identify erroneous configurations already for a low number of features. To tackle this issue, Madsen et al. [184] present CROWDIE, a tool using a feedback-directed instrumentation technique for computing crash paths. These reveal information from the allocation of a selected object of interest to the crashing statement. Cui et al. [60] present RETRACER, a tool to triage crashes by using program semantics reconstructed from memory dumps. Pham et al. [230] introduce HERCULES a method for generating inputs that reach a given potentially crashing location. The authors use information gathered during a separate static analysis or provided by users as input for their method.

Our approach is complementary to those approaches. However, we aim at non-crashing bugs. We provide a validation approach for issues that remain unclear to developers, which collects additional crowd context to isolate the configurations under which an issue occurs.

Record-and-replay approaches capture the users’ interactions while using the application. These help developers understand and reproduce reported issues. While record-and-replay approaches (e.g., [102, 201, 207]) can be used to report non-crashing bugs, these take an isolated look at the reported behavior (cf. Section 2.2.4). The approaches do not monitor the interactions of other users to verify if the reported issues, e.g., only temporarily occurred or to determine specific configurations, for example, of the device used or system version installed, that trigger the reported behavior.

In contrast, our context isolation approach integrates the feedback into continuous software evolution practices by using the crowd of app users to validate the reported behavior. This is similar to automated DevOps tools that capture operational metrics, such as the system performance (cf. Section 2.1.4).

Automated approaches in the area of app store analysis that use crowd in-
7.5 Summary

Bug reports that include structured context information might still be hard to understand and reproduce by developers. When considering the execution context of only a single app user, it might remain unclear to developers which context items, e.g., sensor states, have an impact on the reported bug. If an anomaly is not immediately visible, developers must configure their testing device exactly as reported, which is time-intensive.

This chapter presents an approach to validate the occurrence of reported bugs and isolate the corresponding context information. Similar to automated DevOps tools collecting operational metrics, we integrate users' feedback into continuous software evolution practices. Our approach validates reported issues using crowd-sourcing and identifies recurrent patterns within the collected context information to isolate their occurrence. Important context items are highlighted to developers, while irrelevant items are removed from the report. Possibly long user interaction traces are filtered so that developers only perform events that are potentially relevant for the occurrence of reported bugs.
Chapter 8

Monitoring of Evolutionary Software Changes

This chapter is based on two papers, first “On the Emotion of Users in App Reviews” [186] by Martens and Johann, published at the 2nd International Workshop on Emotion Awareness in Software Engineering in 2017, and second “Release early, release often, and watch your users’ emotions” [189] by Martens and Maalej, published in the IEEE Software International Journal in 2019. My contribution to the first publication is leading the research, implementation, and analysis work, as well as writing the paper. My co-author inspired the research questions, contributed to the implementation and analysis, and edited the paper. My contribution to the second publication is conducting all research, implementation, and analysis work, as well as leading the writing of the paper. My co-author supervised the work, inspired the research questions and goals, and made final edits to the paper.

User reviews include useful information, such as bug reports. However, these are not mundane technical descriptions but also contain emotion. These emotions reflect users’ satisfaction with the app, such as “Good app!” or “I rather run 10 miles to a branch, barefoot on snow, than use this app.”. After improving the quality, i.e., authenticity, of user feedback, it can be used to monitor users’ satisfaction with evolutionary software changes. This is complementary to current automated tools within DevOps processes, that mainly focus on collecting operational measures corresponding to technical aspects of the software system, such as the performance.

This chapter presents a study that empirically analyzes the emotion of users, as expressed in about 7 million app reviews of popular apps by applying sentiment analysis tools. Sentiment analysis tools extract users’ emotions from written feedback. The sentiment allows developers to quickly assess the impact
Chapter 8. Monitoring of Evolutionary Software Changes

of implemented changes for channels where no ratings are available, such as Twitter. This supports developers, e.g., in reacting to unforeseen issues, which is especially relevant in highly competitive markets, such as app stores. Further, from the reviews, we identified recurrent emotional patterns. By comparing these to introduced app changes, we derive release strategies. These guide software practitioners in deciding how to release evolutionary changes without decreasing users’ experience and, therefore, satisfaction with the software system.

The remainder of this chapter is organized as follows. Section 8.1 describes our research setting including the research questions, method, and data. We then report on the results along the research questions in Section 8.2. Afterwards, we discuss the finding’s implications as well as the study’s limitations and threats to validity in Section 8.3. Finally, we survey related work in Section 8.4 and summarize the chapter in Section 8.5.

8.1 Research Setting

In the following, we first introduce the research questions. Then, we describe our research method and data.

8.1.1 Research Questions

We aim to empirically analyze the emotions of users in app reviews, extract emotional patterns, and derive release strategies by comparing the identified patterns to app changes. We focus on the following research questions:

RQ8.1 Do users’ emotions expressed in app reviews correlate with the app rating, price, or review content? By analyzing about 7 million app reviews from 245 popular apps, we found a medium correlation between the users’ emotion and the rating. To strengthen this correlation, sentiment analysis tools need to be adjusted to the software engineering domain and process special words, such as “bug” or “crash”. A correlation between users’ emotion and the app price does not exists. Further, by analyzing the sentiment of an existing labelled dataset, we showed that the content categories bug report and feature requests, on average, have a more negative sentiment.

RQ8.2 Do recurring emotional patterns exist within user feedback? To detect recurring emotional patterns we selected all apps that received more than 1,000 app reviews within the timeframe from January 4, 2016 to December 18, 2016. We analyzed the sentiment over
8.1. Research Setting

Figure 8.1: Research method separated into the data collection, data preparation, and data analysis phase.

time and revealed four recurring emotional patterns, these are consistent emotion, inconsistent emotion, steady decrease or increase, and emotion drop or jump.

RQ8.3 Do specific types of app changes correlate with emotional patterns? Can these correlations be used to derive release strategies? We derived five release lessons by comparing emotional patterns to the release history, content of user reviews, official vendor’s presentations, and technical blogs of several apps corresponding to each pattern. We provide actionable recommendations that should encourage and inspire practitioners to consider users’ emotions when fine-tuning release processes.

8.1.2 Research Method and Data

Our research method consists of a data collection phase, a data preparation phase, and a data analysis phase, as depicted in Figure 8.1. We detail each of the three phases in the following.
Chapter 8. Monitoring of Evolutionary Software Changes

Data Collection Phase

For our study, we selected the top five free and paid apps (by December 18, 2016) for each of the 25 categories of the Apple App Store. We chose the storefront of the United States to obtain reviews in English.

For each app, we gathered its details using the iTunes Search API [144]. An app detail consists of 44 values, such as the name, version, description, category, or price. The app reviews were programmatically scraped using a self-developed tool, which accesses an internal iTunes API. A review consists of 10 values, including the reviewer name, title, description, rating, or date. The update dates of apps were manually extracted from the AppStore for selected apps, as they are not available over the iTunes API. These indicate when a specific app version has been released. In addition, we collected the release notes for each version describing the introduced changes. For further analysis, the data was persisted inside a MongoDB database. To enable replication, our dataset is publicly available [241].

An overview of our dataset is depicted in Table 8.1. We collected 7,396,551 reviews from 245 applications (125 free, 120 paid) in total. The dataset consists of 23 distinct categories. The categories "Kids" and "Magazines & Newspapers" are not present in our dataset, as their top five free/paid apps were of another primary category, such as "News", "Entertainment", or "Education". Another five paid apps appeared in the top five apps of two categories, resulting in 120 paid apps in total. For each app, we collected all reviews beginning with the apps' first release. The oldest feedback was provided in July 2008. Overall, our dataset spans more than eight years.

In our dataset, most reviews for free apps were submitted in the category "Social Networks" (1,763,399 - 25.90%), least in the category "Catalogs" (5,586 - 0.08%). Most paid apps reviews were written in the category "Games" (320,296 - 54.43%), least in the category "Catalogs" (328 - 0.06%). We could calculate the sentiment for 7,371,701 reviews, 24,850 reviews could not be classified (0.003%). The average length of a review (combined of title and description) is 165 characters. The average rating is 3.819 stars. Only 214 reviews (0.003%) received a vote of another app user, indicating that the provided review is helpful.

Data Preparation Phase

We preprocess emojis since these are not considered by the sentiment analysis tool SentiStrength that we chose for our study (cf. Section 2.3.2) [280]. Emojis are a new form of emoticons and increasingly being used by reviewers. These are unicode graphic symbols, of which 1,851 different characters exist. In contrast to emoticons, their emotional content, due to their huge variety, in many cases remains unclear.
Table 8.1: User reviews by categories (N = 7,396,551).

<table>
<thead>
<tr>
<th>#</th>
<th>App Category</th>
<th># Reviews</th>
<th>Free Apps</th>
<th>Paid Apps</th>
<th># Reviews Sentiment</th>
<th>Mean</th>
<th>SD</th>
<th>Median</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Books</td>
<td>111,111</td>
<td>6,686</td>
<td>116,947</td>
<td>1.628</td>
<td>2.437</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Business</td>
<td>15,539</td>
<td>23,514</td>
<td>38,984</td>
<td>1.878</td>
<td>2.125</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Catalogs</td>
<td>5,586</td>
<td>328</td>
<td>5,908</td>
<td>2.137</td>
<td>1.726</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Education</td>
<td>86,858</td>
<td>9,046</td>
<td>95,849</td>
<td>2.449</td>
<td>1.681</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Entertainment</td>
<td>116,613</td>
<td>16,812</td>
<td>132,454</td>
<td>0.619</td>
<td>2.663</td>
<td>2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Finance</td>
<td>190,055</td>
<td>9,303</td>
<td>199,358</td>
<td>1.796</td>
<td>2.126</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Food &amp; Drink</td>
<td>297,732</td>
<td>3,784</td>
<td>301,516</td>
<td>2.047</td>
<td>1.947</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Games</td>
<td>189,102</td>
<td>320,296</td>
<td>506,947</td>
<td>1.515</td>
<td>2.288</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Health &amp; Fitness</td>
<td>357,377</td>
<td>16,033</td>
<td>373,410</td>
<td>2.359</td>
<td>1.769</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Lifestyle</td>
<td>196,733</td>
<td>4,807</td>
<td>201,540</td>
<td>1.683</td>
<td>2.252</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Medical</td>
<td>17,054</td>
<td>4,530</td>
<td>21,584</td>
<td>1.932</td>
<td>2.214</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Music</td>
<td>792,204</td>
<td>26,209</td>
<td>818,413</td>
<td>2.150</td>
<td>2.041</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Navigation</td>
<td>293,854</td>
<td>8,562</td>
<td>302,416</td>
<td>2.044</td>
<td>1.947</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>News</td>
<td>352,659</td>
<td>20,565</td>
<td>373,225</td>
<td>1.121</td>
<td>2.488</td>
<td>2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Photo &amp; Video</td>
<td>601,687</td>
<td>45,637</td>
<td>647,324</td>
<td>0.429</td>
<td>2.829</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Productivity</td>
<td>127,440</td>
<td>9,796</td>
<td>137,236</td>
<td>1.382</td>
<td>2.368</td>
<td>2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Reference</td>
<td>505,966</td>
<td>20,145</td>
<td>526,111</td>
<td>2.736</td>
<td>1.334</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Shopping</td>
<td>129,449</td>
<td>2,192</td>
<td>131,641</td>
<td>2.022</td>
<td>2.181</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Soc. Networking</td>
<td>1,763,399</td>
<td>1,970</td>
<td>1,755,369</td>
<td>0.993</td>
<td>2.634</td>
<td>2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Sports</td>
<td>174,046</td>
<td>3,483</td>
<td>177,529</td>
<td>0.959</td>
<td>2.617</td>
<td>2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Travel</td>
<td>139,285</td>
<td>3,373</td>
<td>142,658</td>
<td>1.442</td>
<td>2.360</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Utilities</td>
<td>106,584</td>
<td>9,522</td>
<td>116,106</td>
<td>1.305</td>
<td>2.426</td>
<td>2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Weather</td>
<td>237,739</td>
<td>21,916</td>
<td>259,655</td>
<td>1.273</td>
<td>2.463</td>
<td>2.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\sum 6,808,072 \quad \sum 588,479 \quad \sum 7,371,701 \quad \varnothing 1.845 \quad \varnothing 2.211 \quad \varnothing 2.522
\]
Novak et al. let 83 human annotators label over 1.6 million tweets in 13 European languages by the sentiment polarity (negative, neutral, or positive) [214]. While there exists no significant difference between the languages, there is a clear difference between tweets without emojis and tweets containing emojis. The authors provide an emoji sentiment lexicon, including 751 emojis giving each of them a score of either -1, 0, or 1.

From the list, we selected all emojis with above 100 occurrences in the tweets dataset, resulting in 214 emojis. Using the list, we replaced each of the 214 emojis with an emoticon (text representation) in order for sentiment analysis tools to be able to consider the emojis while calculating the sentiment.

To evaluate the emotion within user reviews, we measure the strength of the sentiment using SentiStrength 2.2 with its latest configuration files from September 21, 2011. SentiStrength is a lexical sentiment extraction tool with human-level accuracy that can be applied to short informal texts written in English [280]. It was trained for short social web texts such as Twitter. SentiStrength is commonly used as baseline for emotion classification [216]. Further, it has the highest average accuracy among 15 Twitter sentiment analysis tools [1]. Since user reviews in app stores have approximately an average length of a tweet (average length in our dataset is 165) and an informal style, we chose SentiStrength over comparable tools. Alternative tools might generate other results in detail, but we assume the overall results will correlate.

We provide each review (combination of title and description) as input to SentiStrength. To handle mixed emotions [27], SentiStrength returns the sentiment as two scores: a negative score from -1 (not negative) to -5 (extremely negative) and a positive score from 1 (not positive) to 5 (extremely positive). The review “I hate that u need wifi but it is great.”, e.g., has a positive strength of 3 and a negative strength of -4. The sentiment is calculated by matching each token within an input to a set of dictionaries, included in the configuration files. Each dictionary defines fixed scores for specific tokens: “I hate[-4] that u need wifi but it is great[+3].”. The overall score of an input is determined by using the maximum and minimum score of all annotated tokens, in this case [+3, -4]. The dictionaries also include scores for emoticons, booster words which can increase or decrease a token score (“extremely good[3] [+2 booster word]”), or lookup tables for slang words.

Finally, we computed the combined score per review using an approach of Thelwall et al. [279]. Considering both the negative (n) and positive score (p), we chose p as the combined sentiment if p + n > 0. If p + n < 0, we set n as the combined sentiment. In case p = -n and p < 4, we set the combined sentiment to 0 and assume the review to be neutral. If p = -n and p >= 4, we set the combined sentiment to undefined and removed the reviews from the dataset. In contrary to [279] we use a scale from -5 to 5 instead of a scale from -1 to 1. Our
example review with scores [+3, -4] received a combined score of -4.

The average sentiment scores of all analyzed reviews are [+2.535, -1.562]. The combined sentiment average is 1.544. Both sentiments clearly highlight a more positive than negative emotion within app reviews (cf. Table 8.1).

Data Analysis Phase

In the data analysis phase, we perform a statistical analysis of the app reviews and their sentiments to answer the first research question, i.e., if users’ emotions correlated with the app rating, price, or review content. To answer the second research question, we perform pattern mining. We try to identify recurring emotional patterns within all apps that received more than 1,000 reviews in a time period of a year. To answer the third research question, i.e., if the types of app changes correlate with the identified emotional patterns, we consider the release history of apps and the content of user reviews. To derive release strategies, we analyze official vendor presentations and technical blogs of several apps corresponding to each pattern.

8.2 User Emotion in App Reviews

This section describes correlations between users’ emotions as expressed in app reviews and the app rating, price, as well as review content. Afterwards, recurring emotional patterns are described. Finally, release strategies are introduced, which we derived from emotional patterns and corresponding app releases.

8.2.1 Emotion and the Rating, Price, and Content

Rating. Besides the review, a user can give a star rating on a scale from 1 to 5 stars. We assume that a higher star rating correlates with stronger positive emotions and vice versa, that a low star rating correlates with negative emotions. Figure 8.2 shows a box plot of the sentiment for each star rating. At first glance, the plot seems to confirm our assumption. The Pearson correlation coefficient of 0.5699 also draws attention towards a weak positive linear correlation. Although this finding is not surprising, since it follows the intuitive assumption, we will take a closer look at the outliers. Even though the Spearman rank correlation is influenced less by outliers, it also draws a weak correlation (0.5608). Especially for the four and five-star ratings, many reviews are classified as outliers. Out of 612,271 reviews rated with five stars, 199,194 have a combined score of -4 or -5 (32.53%). This can be explained by the nature of SentiStrength: "I would be very sad without it" has a rating of five stars, but the sentiscore has positive strength 1 and negative strength -5, as a consequence of the booster word "very" and the negative word "sad".
Chapter 8. Monitoring of Evolutionary Software Changes

Figure 8.2: Sentiment per star rating.

The results show that, on average, sentiment and the star rating are correlated but are not a sound measure to determine if a user is happy with the app. This affects studies that rely on the link between emotion and user satisfaction.

**Price.** We raise the question if users react emotionally stronger when money is involved. We aim to understand if a user reacts with more emotion to a bug or a newly added feature when the app was subject to a charge, and if the price makes a difference. Harman et al. [123] showed that there is no correlation between price and downloads, nor between price and rating. Pagano and Maalej [224] found that there is significant increase in feedback length between lower-price and higher-price applications, which we can confirm based on our sample. We can reject the hypothesis that there is a significant linear correlation between price and emotion (Pearson/Spearman: 0.021/0.026). A possible explanation might be that users, who are generally willing to provide feedback do not differentiate between free, low-price, and higher-priced apps and provide feedback intrinsically motivated. Their inclination of expression stays the same, independent of the pricing.

**Content.** Pagano and Maalej [224] described 17 different categories in user reviews. Especially for software engineers it is important to filter by the categories bug reports, feature requests, and user experience. Maalej and Nabil [180] compared the accuracy of different techniques for the classification into the categories: bug reports, feature requests, user experience, and rating. A rating here is a literal repetition of the star rating including praise, dispraise, a distractive critique, or a dissuasion.
The authors used bag of words with and without lemmatization and stop-
word removal, the star rating, tense, and the sentiment. They trained a classifier
with different combinations of these features. The authors found that there is
not a one-size-fits-all solution. The sentiment could be used to achieve higher
accuracies in some cases. At this, it did not make a difference if they used the
two-sided sentiment or the combined sentiment. However, not in every case
could the use of the sentiment as an additional feature could lead to better
results. Maalej et al. [183] showed that other features are more informative for
the classification. Only for the categories user experience and bug report, the
sentiment was ranked amongst the ten most important features.

We analyzed the sentiment based on their labeled dataset and found that, on
average, a bug report has a more negative sentiment than a user experience (cf.
Figure 8.3). A closer look on the dispersion measures reveals that the sentiments
are very scattered, especially for the categories bug report and feature request,
see Table 8.2. Ratings and user experience have a very high amount of outliers.

Overall, we think that the sentiment can be a much more important infor-
mative feature when the SentiStrenght tool is adjusted. Since SentiStrength was not intended for software engineering purposes, words that have a relative negative sentiment in the domain of software engineering are not necessarily negative by their nature. For example, the fictitious user review "This app is really buggy and crashes all the time on my phone" would receive a neutral sentiment \([-1, 1]\). In the software engineering domain, words like bug or crash have a very negative connotation. Several researchers highlighted the importance of adapting sentiment analysis tools to the software engineering domain \([141, 216]\).

### 8.2.2 Emotional Patterns

The emotionality of app reviews can vary over time. Reasons for this can be manifold, e.g., features are added, bugs occur, or feature requests are ignored by app developers. To understand the development of emotion, we selected all apps of our dataset that received more than 1,000 reviews within the timeframe from January 4, 2016 to December 18, 2016. For each app, we plotted the sentiment for the given timeframe.

Analyzing the sentiment for different apps over time reveals four recurring emotional patterns \([119]\), as shown in Figure 8.4. Each pattern can be related to specific issues or changes within the apps. For example, new app features might increase the average sentiment. The pattern \textit{consistent emotion} is characterized by a stable negative, neutral, or positive sentiment over time. It can be observed within 15% of the analyzed apps, such as Spotify or Duolingo. The pattern \textit{inconsistent emotion} is characterized neither by a constant nor by a clear positive or negative trend. This pattern can be observed for 62% of the analyzed apps, including WhatsApp. The pattern \textit{steady decrease/increase} is characterized by a constant negative or positive sentiment trend. A constant negative trend can be observed within 10% of the analyzed apps, such as CNN or Microsoft Outlook. A positive trend can only be observed within 3% of the apps, such as AccuWeather. The pattern \textit{emotion drop/jump} is characterized

![Figure 8.4: Four recurring emotional patterns in user feedback.](image-url)
8.2. User Emotion in App Reviews

by a sudden change of the sentiment from negative to positive or vice versa. A change from negative to positive can be observed within 15% of the analyzed apps. A change from positive to negative can be observed within 9% of the apps, such as Google Mail or OverDrive.

8.2.3 Release Strategies

Regularly watching users’ emotions and identifying corresponding patterns is a first step towards understanding an app’s health. We additionally present five release lessons that software practitioners can apply to improve users’ emotions and prevent general negative feedback, which can lead to the fall of apps [303].

We derived the lessons by looking at the release history, the content of user reviews, official vendor presentations, and technical blogs of several apps corresponding to each pattern. For each lesson, we observed at least two indications (e.g., two example apps). Recent studies also support some of the lessons. However, since our lessons are not the result of an in-depth empirical study, we refrain from claiming any generalizability or completeness of lessons. The lessons with their actionable recommendations should encourage and inspire practitioners to consider users’ emotions together with the release frequency and complexity when fine-tuning their release processes.

**Continuously Analyze User Feedback to Identify and React to Bug Reports and Feature Requests**

Software practitioners should analyze user feedback of released app versions and react to frequently mentioned bug reports and feature requests, especially when competitors already offer similar features.

The majority of apps follow the *inconsistent emotion* pattern. These apps are affected by temporary bugs that are quickly fixed by developers. Figure 8.4 shows that the sentiment of WhatsApp (similar to Pandora) strongly decreases and then restores for single periods of time. In the third month, relative from the start of our analysis, nearly all users report storage issues, e.g., “Major Storage Issues”. With the release of an update that fixes the bug after a week, the issue was reported less often. In month 6, users frequently mention crashes after the app start. Although these issues only appear temporarily, they impact the overall sentiment due to affecting the majority of users who installed the update.

Apps that do not react to issues reported by their users are associated with the pattern *steady decrease*. Microsoft Outlook’s reviews included diverse emotions until month 6, as shown in Figure 8.4. Reviews with positive sentiments such as “Best email app” exceeded negative reviews, leading to a stable average sentiment value around 2. The majority of negative reviews is related to issues
within the apps, e.g., ”Trash won’t empty” or “Bug when adding accounts”. As many competing apps exist, users apparently began to explore alternatives. A user wrote “In iOS mail, you can copy an attached excel spreadsheets within body of email but outlook doesn’t format correctly”. Similarly, another user reported: “Although Microsoft has addressed several issue, it’s still buggy at times. I just started looking for a replacement app”.

Recommendation #1. Software practitioners should use tools (e.g., OpenReq [242]) to classify and extract bug reports and feature requests from user feedback. Even when already using automated crash reporting tools, user feedback might include additional non-crashing bugs that cannot be automatically captured. These bugs should be clustered to determine their severity. Bugs that are frequently reported should be quickly fixed by developers before users explore alternative apps. Martin et al. [192] provide a broad overview of the research area app store analysis and existing approaches.

Frequently Release Small Changes

If possible, software practitioners should frequently introduce small changes to their apps instead of releasing fundamental changes at once, such as a significant redesign of the user interface or the removal of app features.

For apps introducing major changes at once, we observed emotion drops. For Google Mail users provided reviews with positive sentiments until the tenth month of our analysis, such as “Love it more than iPhone mail”. With an update that applied Android’s material design, that iOS users are unfamiliar with, the sentiment suddenly turned negative. Reviews including negative sentiments were often related to usability issues (e.g., “New update makes you click on each individual email to delete them.”) or to features removed (e.g., “Bring back Mark as Unread.”). A similar emotion drop can be observed as OverDrive introduced major changes within a single app update. The update included multiple bugs, as several users reported “Can’t download books to device”, “Buggy, buggy, annoying”, or “App crashes with last update”.

In the case of Google Mail, the vendor reacted with weekly updates integrating features requested by users in the reviews such as “Select multiple messages [...]” and “Mark as read/unread [...]”. With the release of those updates, the sentiment shows a positive trend. For OverDrive, to restore the sentiment, most bugs were fixed at once with a single app update after a longer period of time.

The frequency of app updates is controversially discussed in the literature. A recent study reports that frequently updated apps receive a significantly lower percentage of negative ratings [197]. Another study only found a weak correlation, considering negative and positive ratings [194]. However, the study shows that the types of released changes have a varying impact on app ratings. Terms
and topics around bug fixes and features frequently occur in the description of impactful releases [194].

We recommend considering the release of frequent and small updates to avoid surprising users with unexpected (i.e., too many or major) changes [115, 194, 197]. Further, we consider frequent releases beneficial, since bugs get fixed faster for apps with shorter release cycles [158]. Also, studies show that app releases lead to an increased amount of ratings and reviews [190, 194, 224], allowing developers to get more feedback and better understand user needs in highly competitive and dynamic markets [303].

**Recommendation #2.** High code churn in releases, i.e., the rate at which an app’s code evolves, correlates with lower ratings [115] and sentiment scores. Software practitioners should use issue trackers’ and version control systems’ built-in functionality or external tools to visualize the amount of change introduced (e.g., number of user stories resolved, number of bugs fixed, lines of code implemented). Based on these measures, the severity of changes can be determined to decide whether these should be introduced in separate smaller, more frequent releases, as suggested by continuous software evolution practices, such as DevOps.

**Pre-Release Changes to Subsets of App Users**

Changes should be pre-released to subsets of app users before making these available to everyone. Spotify and Duolingo apply this lesson and are able to maintain a consistent positive emotion among their users.

For initial tests, software practitioners should provide access to alpha and beta app versions to voluntary users, as Spotify does [260]. The alpha version is updated almost daily and may be affected by stability issues. The beta version is updated one week before official app releases to discover final issues. Feedback regarding these versions can not be provided in the form of app reviews. Instead, testers email their feedback directly to the development teams as indicated during sign up for the programs. This approach aims to decouple testing (i.e., identifying and reporting bugs) from actually using and assessing the app.

Further, software practitioners should select individual users to participate in A/B-tests (e.g. as both Duolingo and Spotify do [70, 243, 261]). One group of users temporarily receives access to new or modified app features. Duolingo states “Every week, we test at least 10 things on a portion of our users.” [138].

Last, whenever possible new features should be gradually rolled out to all users so that app vendors can assess the overall impact on the emotional trend and react to unforeseen issues, e.g., by deactivating the functionality until the next app update.
Chapter 8. Monitoring of Evolutionary Software Changes

Recommendation #3. Software practitioners should explore app stores’ functionality to distribute alpha and beta versions. The Apple App Stores allows distributing these versions using TestFlight via email invite or public links [276]. On Google Play, developers can similarly release their apps using the Play console [252]. Google Play allows advertising alpha and beta versions on the official app description page, visible to all users. After testing, changes should be gradually rolled out to assess their overall impact on users’ emotions and to be able to react to unforeseen issues.

Explain Changes to Users

Major changes, such as the increase of the minimum required system version or the removal of app features, should be announced and explained to users. Studies show that users do not pay too much attention to release notes [197]. Instead, software practitioners should engage in conversations with users [19] or directly explain the changes within the app itself, e.g., using tutorials and tooltips.

We observed that apps that did not follow this lesson were affected by the pattern of steady decrease. For example, for the CNN app, a user reported: “What happened to local news. I checked that every day [...] please bring it back”. For Microsoft Outlook, the sentiment decreased significantly when users were affected by incompatibilities with new and old iOS versions, such as “Update [...] broke the app on iOS 8. Went back to using the stock mail app on my iPhone. Uninstalled it.”.

Recommendation #4. App changes should be transparent and understandable to users. Therefore, major changes software practitioners do not want to release in smaller parts should be explained to users, e.g., using app built-in tutorials. Further, users with legacy devices and system versions should be redirected to alternatives (e.g., web version) using announcements before stopping the support.

Capture Implicit Feedback to Support Decisions

Software practitioners should capture implicit feedback to empirically determine whether experimental app features should be integrated. In the beginning, the taken measures should reflect a fundamental overall goal, e.g., maximize the number of tracks listened for music apps (cf. [259]). Then, more elaborate measures can be developed.

For example, Spotify performs A/B-tests even for unfinished features to decide whether these should be further developed. For testing, changes are split
into atomic parts. When changing, e.g., the navigation, one test looks into the UI while another test focuses on the content, i.e., order of menu items [165].

While implicit measures help software practitioners evaluate and optimize features against comparable criteria, explicit feedback provides additional information about why the taken measures change [94]. For explicit feedback where no ratings exist, the sentiment can be calculated to assess users' opinion quickly. Further, it offers a broader understanding of the impact of changes. Users of no longer supported devices are, e.g., able to express their opinion in explicit feedback.

**Recommendation #5.** Software practitioners should take further steps towards data-driven requirements engineering [181] by integrating logging frameworks, such as AppSee or Google Analytics, into their apps. Beginning with easy measures that relate to the app's overall goal, software practitioners should develop more complex ones by testing the impact of changes in atomic parts. The implicit measures complement explicit user feedback to support decisions which experimental features to integrate into apps.

### 8.3 Discussion

We discuss the implications of our findings and list potential limitations and threats to validity of our study.

#### 8.3.1 Implications

A large body of research, such as in the area of app store analysis, focused on analyzing user feedback. Most of the existing studies and approaches focus on aspects of the software system, such as the extraction bug reports. Only few studies (e.g., [186, 215, 216]) consider the perspective of users. We showed that user reviews are not mundane technical descriptions but also contain emotion. We found that emotion in user reviews can be seen as a meaningful additional meta-data. Especially in channels where no ratings are available, the sentiment can help developers to assess the impact of released changes quickly. This is especially relevant for highly competitive markets, such as app stores, that offer several alternative apps for single use cases. In case developers ignore users' feedback this can lead to the downfall of apps, as in the case of Yik Yak [303].

To monitor users' satisfaction with evolutionary software changes, user feedback should be integrated into continuous software evolution practices. Users' emotions as expressed in their feedback is complementary to automated DevOps tools that capture operational metrics. While operational metrics are related to the software system, such as the performance used to automatically scale...
the software system, these metrics do not reflect users’ satisfaction with the software system but solely are thresholds that were configured by developers. Users’ emotions can be used to automatically determine their satisfaction with evolutionary software changes and help developers identify if problems with the software system exist.

Another promising insight is the development of the sentiment over time and the reoccurring patterns that could be found in our analysis. Particularly mixed emotions attracted our interest since the manual analysis indicates that users actively discuss the pros and cons of a release. By comparing corresponding app changes to the identified patterns, we could derive release strategies for software vendors to update their apps. For example, the strategies can guide developers in deciding how to introduce fundamental updates to their apps without decreasing users’ satisfaction.

8.3.2 Limitations and Threats to Validity

The fact that we could only observe weak correlations with other indicators tempers the expectations of current approaches. This can be linked back to the general approach of tools like SentiStrength. In future work, it will be necessary to adjust tools and approaches to fit the contents in user reviews better.

The correlation between emotional sentiment and user ratings is only weakly defined. Especially for very positive ratings with four and five stars, the rating itself is a better indicator for the users’ satisfaction than the sentiment.

The sentiment becomes more relevant when user reviews are classified based on their category. We see the potential to increase the quality of classification techniques further when tools, such as SentiStrength, are adjusted to software topics by applying different sentiment weights to given buzzwords such as “bug” or “crash”. Our findings motivate to adjust sentiment analysis towards a better fit for the purpose of software reviews. In this study, we used SentiStrength knowingly without further modification as it is commonly used as a baseline in emotion classification.

8.4 Related Work

Natural language processing has been used for sentiment analysis for more than a decade [282]. With the rise of social media, it gained increasing importance. Initially intended for the use in marketing or political opinion mining, sentiment analysis became popular in many other social domains. This also applied to the social aspects of software engineering as the community recognized its potential. Most of the research has been conducted in the field of developers’ interactions within bug trackers, commit messages, or software related Q&A sites [120, 215].
With the growing interest in app store mining [49, 123, 131], emotional sentiment has been used in research related to user reviews in app stores. In this area, sentiment analysis mostly played a supportive role to detail findings or to increase the quality of results. Goul et al. used sentiment analysis on user reviews to address current bottlenecks in requirements [110]. Li et al. calculated the overall user satisfaction in the app stores [129]. A more fine-grained study by Guzman and Maalej used sentiment analysis to summarize the opinion on app features retrieved from user reviews [119]. Maalej et al. added sentiments as an informative feature to train a classifier used to categorize user reviews regarding different categories [180, 183].

However, Jongeling et al. reported on possible negative results when using sentiment analysis tools for software engineering research [153]. The use of sentiment analysis tools and APIs is highly dependent on the use case, and those have a significant impact on study results [152]. Fu et al. [92] built a regression model of words based on common user vocabulary and identified words with outstanding positive and negative sentiment based on their sample.

Our study differentiates from the earlier studies therein that we undertake an isolated look on sentiments of user reviews in order to better understand if and how the sentiment can be used as a supportive feature for app store analyses.

However, related work is not limited to natural language processing only. For example, Johanssen et al. [150] derive user emotions from facial expressions while using the app. The authors state that “users rarely report explicit feedback without being asked for it”. The authors utilize the recent improvement in consumer hardware, such as cameras being included within most smartphones, and develop the framework EmotionKit. Their framework processes the input of the front camera, directed towards the smartphone user while watching the screen, to capture the user’s emotion with the app in-situ. In a study with twelve participants, the authors find that users visibly react to usability issues, which is a first step towards their automated identification considering only implicit user feedback. This could also be an opportunity to automatically detect non-crashing bugs.

Other researchers follow this idea and use additional sensors, such as electroencephalogram (EEG) devices, to retrieve biological data and assess’ users’ satisfaction based on their implicit feedback [159].

8.5 Summary

App stores are highly competitive markets, sometimes offering dozens of apps for a single use case. Unexpected app changes, such as the removal of a feature, might incite even loyal users to explore alternative apps. We show that sentiment analysis tools can help monitor users’ emotions as expressed, e.g., in app reviews.
or tweets. By analyzing about 7 million app reviews from 245 popular apps within the Apple App Store, we studied correlations between the emotion and the apps’ rating, price, as well as the content of the user feedback. Further, we found that users’ emotions include four recurring patterns corresponding to the app releases. Based on these patterns and online reports about popular apps, we derived five release lessons to assist app vendors maintain positive emotions and gain competitive advantage.

We propose integrating users’ emotions, e.g., as expressed in their explicit feedback in the form of sentiment values, into continuous software evolution practices. User emotions are complementary to operational metrics captured by automated DevOps tools which focus on technical aspects of the software system, such as the performance. Continuously monitoring users’ emotions allows developers to assess their satisfaction with evolutionary software changes and to identify potential problems that originate from released changes.
Part III

Synopsis
Chapter 9

Evaluation

The first research question of this chapter is based on the paper “Towards understanding and detecting fake reviews in app stores” [190] by Martens and Maalej, published in the Empirical Software Engineering Journal in 2019. My contribution to this publication is conducting all research, implementation, and analysis work, as well as leading the writing of the paper. My co-author supervised the work, inspired the research questions and goals, and made final edits to the paper.

In the previous chapters, we underlined the problem of fake reviews in app stores and of issues in explicitly provided user feedback that miss basic context information. We presented solutions to detect fake reviews, augment new and existing user feedback with context information, as well as to isolate existing (i.e., highlight relevant and hide irrelevant) context information using crowdsourcing. This chapter reports on the evaluation of our concepts.

The remainder of the chapter is organized as follows. Section 9.1 describes our evaluation setting including the evaluation questions, method, and data. We then report on the results along the evaluation questions in Section 9.2. Afterwards, we discuss the limitations and potential threats to validity of our findings in Section 9.3. Finally, we summarize the chapter in Section 9.4.

9.1 Evaluation Setting

In the following, we introduce the evaluation questions, method, and data.

9.1.1 Evaluation Question

Our study aims at determining whether the solutions presented in this thesis contribute towards the authenticity (RQ9.1) and actionability (RQ9.2) of user feedback. We focus in particular on two main research questions:
RQ9.1 What is the performance of machine learning classifiers when detecting fake reviews in practice, i.e., on imbalanced datasets with different proportional distributions of fake and regular reviews? We performed an in-the-wild experiment to get more realistic results of how the classifiers perform in practice. Therefore, we used imbalanced datasets of fake and regular reviews. We varied the skewness of the datasets between 90% to 0.1% fake reviews and compared the classification results. We found that the Random Forest classifier identifies fake reviews, given a proportional distribution of fake and regular reviews as reported in other domains, with a recall of 91% and AUC/ROC value of 98%.

RQ9.2 Does complete and isolated context information support developers in understanding and reproducing non-crashing bugs? In experiments with 16 professional iOS developers and four real bugs from open source apps, we found that context information, implicitly captured during app executions and isolated using crowdsourcing, improves developers’ understanding of reported issues. Developers need 30% to 70% less time to understand and reproduce reported issues. The improved understanding is also reflected by the low number of interactions, 78% on average, developers perform during bug reproduction. Also, we study two sub-questions.

RQ9.2.1 How much runtime overhead does implicitly capturing context information during app executions introduce? In simulations with three closed-source apps we show that our approach to implicitly capture context information and isolate the occurrence of reported issues does not produce observable runtime overhead and memory usage.

RQ9.2.2 What are developers concerns when capturing and isolating context information as well as their suggested improvements? In interviews, developers highlighted the benefit of reproducing bugs when context information is given, as well as the easy integration of our presented approach. Their concerns were mainly related to privacy issues. Possible improvements stress the importance of integrating context information into the existing development workflow, such as an integration of the collected information into issue tracking systems.
9.1. Evaluation Setting

9.1.2 Evaluation Methods and Data

We evaluate the impact of our approaches on the quality of user feedback, separated by its authenticity and actionability. Figure 9.1 visualizes our evaluation method. To answer the first research question (RQ9.1), we performed an in-the-wild experiment. To answer the second research question (RQ9.2), we conducted a controlled experiment. For RQ9.2.1, we performed a simulation. To answer RQ9.2.2, we conducted semi-structured interviews. In the following, we describe the evaluation method and data of each question separately.

The study replication package, including the interview form, responses, experiment, and simulation results, is publicly available [241]. The questions of the semi-structured interview are included in Appendix B.

RQ9.1 In practice, fake and regular reviews are imbalanced. For app stores, no reliable estimate on the distribution exists. Other domains, such as social media, mark 10% to 15% of their reviews as fake [277]. The travel portal Yelp filters about 15% of their reviews as suspicious [178, 203]. This class imbalance can additionally be affected by numerous factors, such as the selected apps or time period. Free apps, for example, receive more fake reviews than paid apps. This reveals a skewed distribution of fake and regular reviews in app stores.

Figure 9.1: Overview of the evaluation method to evaluate the impact of our approaches on the quality, i.e., authenticity and actionability, of user feedback.
Table 9.1: Participants of the bug reproduction experiment and interview.

<table>
<thead>
<tr>
<th>Participant</th>
<th>Role</th>
<th>iOS Exp. (Yrs)</th>
<th>Focus Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Developer</td>
<td>6</td>
<td>Shopping apps</td>
</tr>
<tr>
<td>P2</td>
<td>Tester</td>
<td>6</td>
<td>News apps</td>
</tr>
<tr>
<td>P3</td>
<td>Developer</td>
<td>8</td>
<td>News apps</td>
</tr>
<tr>
<td>P4</td>
<td>Developer</td>
<td>5</td>
<td>Personal apps</td>
</tr>
<tr>
<td>P5</td>
<td>Developer</td>
<td>8</td>
<td>Sports apps</td>
</tr>
<tr>
<td>P6</td>
<td>Developer</td>
<td>8</td>
<td>Personal apps</td>
</tr>
<tr>
<td>P7</td>
<td>Developer</td>
<td>6</td>
<td>Reference apps</td>
</tr>
<tr>
<td>P8</td>
<td>Developer</td>
<td>7</td>
<td>Sports apps</td>
</tr>
<tr>
<td>P9</td>
<td>Tester</td>
<td>4</td>
<td>Media apps</td>
</tr>
<tr>
<td>P10</td>
<td>Developer</td>
<td>9</td>
<td>Reference apps</td>
</tr>
<tr>
<td>P11</td>
<td>Developer</td>
<td>10</td>
<td>Personal apps</td>
</tr>
<tr>
<td>P12</td>
<td>Developer</td>
<td>6</td>
<td>Business apps</td>
</tr>
<tr>
<td>P13</td>
<td>Developer</td>
<td>7</td>
<td>Sports apps</td>
</tr>
<tr>
<td>P14</td>
<td>Developer</td>
<td>5</td>
<td>Personal apps</td>
</tr>
<tr>
<td>P15</td>
<td>Developer</td>
<td>6</td>
<td>Transportation apps</td>
</tr>
<tr>
<td>P16</td>
<td>Developer</td>
<td>7</td>
<td>Photo editing apps</td>
</tr>
</tbody>
</table>

Research found that highly imbalanced data often results into poor performing classification models [48, 69, 248]. To have a more realistic setting of how our classifier can perform in practice, we conduct an in-the-wild experiment by varying the skewness of our dataset. We decided to vary the skewness on a logarithmic scale to depict the classification scores on finer granularities towards extremely imbalanced datasets with fake reviews as the minority class. We keep a fixed amount of 8,000 fake reviews and create 27 datasets including \(10^2 - 10^{-1}\) (90%) to \(10^{-1}\) (0.1%) fake reviews. For a skew of 90%, we used 889 regular reviews. With every change of the skewness, we added additional regular reviews. All of the about 8 million regular reviews were randomly selected at once from the official reviews dataset, so that the classification results are comparable.

We applied the machine learning classifiers to the datasets and compared the evaluation results. We highlighted the importance of the selected evaluation metrics to be able to compare the results for skewed datasets.

RQ9.2 We recruited 16 participants, of whom 14 are iOS developers, and 2 are software testers, shown in Table 9.1. Their professional experience regarding the iOS platform ranges from 5 to 11 years, with an average of 7 years. As iOS was initially released in September 2007, the maximum experience participants can have is 11 years. The participants work in diverse domains, with focuses on media, business, news, personal, photo editing, reference, shopping, sports, and transportation apps.
9.1. Evaluation Setting

Table 9.2: iOS apps used in the bug reproduction experiment.

<table>
<thead>
<tr>
<th></th>
<th>DuckDuckGo</th>
<th>Firefox</th>
<th>Wikipedia</th>
<th>Wordpress</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>6.0.6</td>
<td>5.0</td>
<td>5.0.5</td>
<td>6.3.1</td>
</tr>
<tr>
<td>Category</td>
<td>Reference</td>
<td>Utilities</td>
<td>Reference</td>
<td>Social</td>
</tr>
<tr>
<td>Avg. rating</td>
<td>4.5</td>
<td>3.5</td>
<td>2</td>
<td>4.5</td>
</tr>
<tr>
<td>Reviews</td>
<td>869</td>
<td>1984</td>
<td>54625</td>
<td>18786</td>
</tr>
<tr>
<td>Contributors</td>
<td>14</td>
<td>70</td>
<td>39</td>
<td>59</td>
</tr>
<tr>
<td>Stars</td>
<td>158</td>
<td>5948</td>
<td>850</td>
<td>1246</td>
</tr>
<tr>
<td>1st commit</td>
<td>2011</td>
<td>2014</td>
<td>2013</td>
<td>2008</td>
</tr>
<tr>
<td>KLOC</td>
<td>23</td>
<td>415</td>
<td>154</td>
<td>115</td>
</tr>
<tr>
<td>Commits</td>
<td>1467</td>
<td>5235</td>
<td>6950</td>
<td>22050</td>
</tr>
</tbody>
</table>

During the experiment, participants had to reproduce different non-crashing bugs that were described within app reviews. The app reviews are passed to different developers, each with and without context information given. We determine the helpfulness of our approach based on three criteria:

1. Are more developers able to reproduce a bug when complete and isolated context information is given?

2. Do these developers perform fewer interactions with the application user interface while reproducing the bug?

3. Do developers need less time for reproduction when context data is given?

With these criteria, we aim to understand whether developers, given context data, have a better understanding of the reported bug. Using the number of interactions, we determine if developers are able to reproduce a bug precisely or randomly browse through the app as the provided information is insufficient. Using the time for reproduction, we analyze if the presented data overwhelms developers.

We studied four open-source iOS apps, depicted in Table 9.2. For each app, we selected a single non-crashing bug. The selected bugs have different complexities. For three apps, we identified real bugs by reading through app reviews, browsing through issue trackers, and by looking for fixed bugs in unreleased commits of the apps. For bugs identified through unreleased commits and issue trackers, we searched for an appropriate app review within the Apple App Store. We implemented the fourth bug (Wordpress app) on our own. This bug has the highest complexity, as it only occurs on a specific combination of a device model and a system version. The app review describing the bug was written by an iOS user without development experience.
Table 9.3: App reviews shown to developers in the experiment.

<table>
<thead>
<tr>
<th>App</th>
<th>Review</th>
<th>Context</th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>DuckDuckGo</td>
<td>TEXT TERRIBLE: tried using but virtually impossible to read certain text</td>
<td>No restrictions</td>
<td>0. Start app</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1. Tap on <strong>SearchBar</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2. Type text &quot;Halloween&quot; into <strong>SearchBar</strong> (or other word with meanings section)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3. Tap on first cell in <strong>ResultsTableView</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4. Tap on <strong>MeaningsView</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>→ Text nearly impossible to read</td>
</tr>
<tr>
<td>Firefox</td>
<td>Can’t move tabs: Not able to move tabs while pressing, same functionality as changing app positions on iPhone</td>
<td>Appears on iOS versions below 9.0</td>
<td>0. Start app</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1. Tap on <strong>ShowTabsButton</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2. Tap on <strong>AddTabButton</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3. Tap on <strong>ShowTabsButton</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4. Long tap <strong>TabView</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>→ Tabs do not start moving and cannot be rearranged</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>Wrong language: Even though I have English set as my main language, all search results display in German...</td>
<td>Locale and region set to German</td>
<td>0. Start app</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1. Tap on <strong>SettingsButton</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2. Tap on third cell in <strong>SettingsTableView</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3. Tap on <strong>EditButton</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4. Tap on <strong>AddLanguageButton</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5. Type &quot;English&quot; into <strong>SearchBar</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6. Tap on first cell in <strong>ResultsTableView</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7. Tap on <strong>SetPrimaryLanguageButton</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8. Swipe left</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>→ App texts remain German</td>
</tr>
<tr>
<td>Wordpress</td>
<td>Can’t share logs: I am unable to share my activity logs. Please fix!</td>
<td>Appears on iPads with iOS 8.4 installed</td>
<td>0. Start app</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1. Tap on <strong>HelpButton</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2. Tap on <strong>ActivityButton</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3. Tap on first cell in <strong>ActivitiesTableView</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4. Tap on <strong>ShareButton</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>→ iOS Share dialogue does not appear</td>
</tr>
</tbody>
</table>
9.1. Evaluation Setting

Afterwards, we integrated our approach to implicitly capture context information into the apps and let ten students report the bug. Thereby, we simulated the isolation of context information. We did so, as we decided to exclude the influence of the number of validations provided by users from the experiment. This depends on several factors, such as the level of detail configured by developers, the willingness of users to validate reviews, and the overall number of app users. Table 9.3 depicts the bugs used in our experiment, their app review, as well as the captured context information.

We used a within-subjects design and randomly divided participants into an experiment and control group. We decided for this design to eliminate individual differences between participants. Participants of the control group were given a regular app review, while the experiment group additionally received the context information. Each participant had to reproduce one randomly assigned bug. Afterwards, participants of the control group were assigned to the experiment group and vice versa, to reproduce another bug. Thereby half of the participants began to reproduce a bug with context data given, while the other half began with a regular app review only. This is relevant to avoid biased interviews. Also, it ensures that each participant has solved bugs with both types of information. During the experiment, each bug – with and without context data given – was reproduced by four developers/testers.

We installed a lab setting to ensure that all participants use the same app versions and development environment. To limit the complexity, we restricted the available iOS simulators to three system versions (iOS 9.3, 9.0, and 8.4). Participants accessed the development environment either physically or using a remote-control tool. The reproduction of bugs was screen recorded to analyze the video for the time, and the steps participants needed to understand and reproduce the bugs. If developers needed more than 7 minutes, we aborted the reproduction not to exceed the 20 minute time limit of the experiment. As a warm-up, we presented developers a regular app review and an app review with context information of a bug they did not have to reproduce.

We normalized the time developers needed to reproduce a bug as there exist individual differences that do not depend on the participant. The time can be
separated into five steps, as shown in Figure 9.2. These are 1) Read the app review (and context data), 2) build the app, 3) start the simulator, 4) start the app, and 5) reproduce the bug. In cases participants needed to switch the device model and/or the system version, an additional step is 6) Reread app review (and context data) and switch device/system. The previous five steps are repeated afterwards. As the second, third, and fourth step are affected by differences, such as varying build times due to background tasks, we replaced these with their average values measured during the experiment.

RQ9.2.1 We conducted a simulation to study the runtime overhead introduced by our approach to capture context information and isolate the occurrence of bugs. Therefore, we instrumented closed-source apps we were given access to by our participants. We decided against the previously used apps to have more diverse results. The used apps are popular within the Apple App Store, each with a monthly user basis above 500,000 users. The simulation was performed on an iPhone 6 with iOS 9.1.4 installed.

To achieve comparable results, we use a monkey testing tool to perform 1,000 interactions with the application user interface per measurement. The tool mimics user interactions by randomly selecting elements of the app UI. Per app the simulation was repeated 30 times.

The measurement itself was performed using Xcode Instruments [137]. The tool was used to gather information about the running application, such as the CPU and memory usage, as well as the size of the captured context information.

RQ9.2.2 The interviews aim to understand developers’ concerns when collecting and isolating context information using our presented approaches. Further, these capture developers’ suggested improvements. We interviewed 16 participants (cf. Table 9.1). Each interview lasted for 20-30 minutes. The interviews were conducted either face to face or by telephone. We took notes during the interviews to analyze the answers of the participants. Using the notes, we identified important aspects, e.g., those reported by multiple users or those that might have a considerable impact on our approach.

9.2 Evaluation Results

In this section, we report on the study results along the evaluation questions.

9.2.1 Authenticity of User Feedback

Figure 9.3 shows an overview of the results of the seven supervised machine learning approaches studied in this work. Per classification algorithm and
9.2. Evaluation Results

![Classification scores](image.png)

Figure 9.3: Classification scores of machine learning algorithms on imbalanced datasets, including 90% to 0.1% fake reviews, plotted on a logarithmic scale.

<table>
<thead>
<tr>
<th>Predicted as fake review</th>
<th>Predicted as regular review</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual fake review</td>
<td>True positive (TP)</td>
</tr>
<tr>
<td>Actual regular review</td>
<td>False positive (FP)</td>
</tr>
</tbody>
</table>

Table 9.4: Confusion matrix of a two-class problem

Since identifying fake reviews is a two-class problem, the performance metrics can be derived from the confusion matrix that is generated with every classification, see Table 9.4. We explain the reported performance metrics in the following: Precision ($\frac{TP}{TP+FP}$) measures the exactness, i.e., the number of correctly classified fake reviews to the overall number of reviews classified as fake. Recall ($\frac{TP}{TP+FN}$) measures the completeness, i.e., the number of correctly classified fake reviews to the overall number of fake reviews. The F1-score ($\frac{2 \times \text{recall} \times \text{precision}}{\text{precision} + \text{recall}}$) is the harmonic mean between precision and recall. As improving precision and recall can be conflicting, it shows the trade-off between both. The AUC value measures the area under the ROC curve. It varies within the interval $[0, 1]$. The ROC curve itself depicts all possible trade-offs between TP rate, i.e., recall, and FP rate ($\frac{FP}{FN+FP}$). A better classifier produces a ROC.
As we are focusing on skewed datasets within our in-the-wild experiment, we need to select performance metrics that are insensitive to class imbalance. This applies to all measures that use values from only one row of the confusion matrix [296]. Precision and F1-score are sensitive to class imbalance and biased towards the majority class. Therefore, these metrics are inappropriate for our evaluation. Recall and AUC/ROC value are insensitive to class distribution. We use both measures to compare the performance of the classification algorithms within our in-the-wild experiment.

In the further evaluation, we include all classifiers that achieve a recall and AUC value higher than 0.5 for datasets including 90% to 1% fake reviews. This applies to the random forest (RF), decision tree (DT), and MLP algorithm. The remaining algorithms are excluded from the evaluation. The recall of the Gaussian naive bayes (GaussianNB) algorithm decreases to nearly 0 for datasets with fake reviews as the majority class. Also, its precision is extremely low for datasets including less than 10% fake reviews. Similarly, the recall of the SVC algorithms decreases to nearly 0 for datasets with less than 10% fake reviews. Also, the SVC(kernel='linear') implementation returns errors for skews below 5%, so do the remaining two SVC algorithms for skews below 0.1%.

**Classification Results with Imbalanced Data**

Figure 9.4 shows the performance measures of all three classification algorithms that remain within our evaluation. We chose to depict each measure as a single plot to more easily compare the algorithms. We report the precision and F1-score for reasons of completeness, although these are inappropriate measures for imbalanced data. Within the graphs, we highlight in gray the interval in which fake reviews typically occur in other domains. Further, we mark where the classes are equally distributed, i.e., there exist 50% fake reviews. At this point, the algorithms perform well with all measures above 0.9 (cf. Table 9.5).

In practice, there can exist imbalances towards fake or regular reviews being the majority class. From our results and research in other domains, it is more likely that the bias is towards regular reviews. For this reason, we choose more detailed results towards fake reviews being the minority class.

However, when fake reviews become the majority class (towards the right of the 50% mark), the recall improves for all algorithms, by up to 6.4% for the MLP algorithm. The best result is achieved by the RF algorithm with 0.986 recall. The AUC value decreases in all cases. For the RF and MLP algorithms, the value slightly decreases by up to 0.5%. The value of the DT algorithm decreases more strongly by 5.5%. The random forest algorithm achieves the best AUC value with 0.984.

When fake reviews become the minority class, most performance measures
9.2. Evaluation Results

Figure 9.4: Classification scores of appropriate machine learning algorithms for datasets with class imbalance, i.e., including 90% to 0.1% fake reviews, plotted on a logarithmic scale.
Table 9.5: Classification scores on imbalanced datasets with skews of 90% to 0.1% fake reviews (DT: DecisionTreeClassifier, MLP: MLPClassifier, RF: RandomForestClassifier).

<table>
<thead>
<tr>
<th>Skew</th>
<th>Recall</th>
<th>AUC/ROC</th>
<th>Precision</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DT</td>
<td>MLP</td>
<td>RF</td>
<td>DT</td>
</tr>
<tr>
<td>90.0</td>
<td>0.982</td>
<td>0.978</td>
<td>0.986</td>
<td>0.987</td>
</tr>
<tr>
<td>80.0</td>
<td>0.972</td>
<td>0.963</td>
<td>0.980</td>
<td>0.973</td>
</tr>
<tr>
<td>70.0</td>
<td>0.966</td>
<td>0.943</td>
<td>0.976</td>
<td>0.964</td>
</tr>
<tr>
<td>60.0</td>
<td>0.964</td>
<td>0.940</td>
<td>0.970</td>
<td>0.946</td>
</tr>
<tr>
<td>50.0</td>
<td>0.953</td>
<td>0.920</td>
<td>0.962</td>
<td>0.950</td>
</tr>
<tr>
<td>40.0</td>
<td>0.945</td>
<td>0.902</td>
<td>0.956</td>
<td>0.953</td>
</tr>
<tr>
<td>30.0</td>
<td>0.942</td>
<td>0.872</td>
<td>0.947</td>
<td>0.956</td>
</tr>
<tr>
<td>20.0</td>
<td>0.924</td>
<td>0.842</td>
<td>0.937</td>
<td>0.950</td>
</tr>
<tr>
<td>10.0</td>
<td>0.896</td>
<td>0.758</td>
<td>0.912</td>
<td>0.941</td>
</tr>
<tr>
<td>9.0</td>
<td>0.894</td>
<td>0.824</td>
<td>0.907</td>
<td>0.940</td>
</tr>
<tr>
<td>8.0</td>
<td>0.885</td>
<td>0.809</td>
<td>0.896</td>
<td>0.936</td>
</tr>
<tr>
<td>7.0</td>
<td>0.884</td>
<td>0.785</td>
<td>0.892</td>
<td>0.936</td>
</tr>
<tr>
<td>6.0</td>
<td>0.857</td>
<td>0.804</td>
<td>0.882</td>
<td>0.924</td>
</tr>
<tr>
<td>5.0</td>
<td>0.860</td>
<td>0.694</td>
<td>0.876</td>
<td>0.925</td>
</tr>
<tr>
<td>4.0</td>
<td>0.857</td>
<td>0.714</td>
<td>0.869</td>
<td>0.925</td>
</tr>
<tr>
<td>3.0</td>
<td>0.830</td>
<td>0.681</td>
<td>0.846</td>
<td>0.912</td>
</tr>
<tr>
<td>2.0</td>
<td>0.806</td>
<td>0.567</td>
<td>0.814</td>
<td>0.901</td>
</tr>
<tr>
<td>1.0</td>
<td>0.775</td>
<td>0.395</td>
<td>0.756</td>
<td>0.886</td>
</tr>
<tr>
<td>0.9</td>
<td>0.776</td>
<td>0.255</td>
<td>0.755</td>
<td>0.886</td>
</tr>
<tr>
<td>0.8</td>
<td>0.764</td>
<td>0.300</td>
<td>0.735</td>
<td>0.881</td>
</tr>
<tr>
<td>0.7</td>
<td>0.765</td>
<td>0.252</td>
<td>0.731</td>
<td>0.881</td>
</tr>
<tr>
<td>0.6</td>
<td>0.756</td>
<td>0.166</td>
<td>0.710</td>
<td>0.877</td>
</tr>
<tr>
<td>0.5</td>
<td>0.747</td>
<td>0.065</td>
<td>0.703</td>
<td>0.873</td>
</tr>
<tr>
<td>0.4</td>
<td>0.735</td>
<td>0.021</td>
<td>0.661</td>
<td>0.867</td>
</tr>
<tr>
<td>0.3</td>
<td>0.712</td>
<td>0.001</td>
<td>0.634</td>
<td>0.855</td>
</tr>
<tr>
<td>0.2</td>
<td>0.692</td>
<td>0.001</td>
<td>0.608</td>
<td>0.846</td>
</tr>
<tr>
<td>0.1</td>
<td>0.680</td>
<td>0.000</td>
<td>0.560</td>
<td>0.840</td>
</tr>
</tbody>
</table>
9.2. Evaluation Results

decrease. With an amount of 10% fake reviews \(10^1\), as reported to be typical for other domains [178, 203, 277], the recall of the RF and DT algorithm are nearly identical (RF: 0.912, DT: 0.896). Compared to the result of the balanced dataset, the recall decreased by 5.3% for the RF algorithm and by 5.9% for the DT algorithm. The recall of the MLP algorithm is significantly less with 0.758 (-17.6%). The AUC value is the highest for the RF algorithm (0.986, +0.3%), followed by the MLP (0.977, +0.6%) and DLT (0.941, -0.9%) algorithms.

For 1% fake reviews \(10^0\), the recall of the RF and DT algorithms is still nearly identical (RF: 0.756, -21.4%; DT: 0.775, -18.7%), followed by the MLP algorithm (0.395, -57.1%). The AUC value is the highest for the MLP algorithm (0.978, +0.7%), followed by the RF (0.971, -1.8%) and DT (0.886, -6.7%) algorithms.

For an amount of 0.1% fake reviews \(10^{-1}\) the recall is the highest for the DT algorithm (0.680, -28.6%), followed by the RF algorithm (0.560, -41.8%). The recall of the MLP algorithm dropped to 0 at about 0.3% fake reviews and below within the dataset. The AUC value is the highest for the MLP algorithm (0.973, +0.2%), followed by the RF algorithm (0.945, -4.4%). Last, the AUC value of the DT algorithm significantly decreased to 0.840 (-11.4%).

Comparing all three algorithms using their recall and AUC/ROC value, the random forest algorithm performs best for imbalanced datasets. Although the decision tree algorithm achieves a better recall when the dataset is extremely skewed (less than 1% fake reviews), its AUC/ROC value is significantly lower for all datasets. Similar, the MLP algorithm achieves better AUC/ROC values for datasets with less than 1% fake reviews. However, the recall of the MLP algorithm drops to 0 for extremely skewed datasets. With skews typical for other domains, the random forest algorithm performs best with a recall of 0.912 and AUC/ROC value of 0.986.

9.2.2 Actionability of User Feedback

To determine the actionability of user feedback, with and without context information included, we perform experiments with professional iOS developers. To determine the overhead of implicitly capturing context information during app executions, we perform simulations. Finally, we conduct interviews with developers to understand their concerns regarding our presented approaches, as well as possible improvements.

**Experiments**

In experiments, we aim to understand if context information collected and isolated by our approaches supports developers reproduce non-crashing bugs.

We measured the time developers needed to reproduce non-crashing bugs
and counted their interactions performed. We included four bugs with different complexities in our experiment. Each bug was reproduced by eight developers, of which four received an app review as bug report (cf. Table 9.3), while the remaining four additionally received the collected and isolated context information. Overall, the developers performed 32 bug reproduction sessions. Table 9.6 lists the experiment results.

We found that the context information captured and isolated by our approach supports participants in understanding and reproducing non-crashing bugs. This applies to all bugs included in our experiment, independently from their complexity. With context data, participants were able to reproduce the reported behavior in all 16 cases. Using regular app reviews, participants were only able to reproduce the behavior in 10 of the 16 cases (63%). Further, developers were 30% to 71% faster in reproducing bugs when context data was provided (46% on average). Also, developers needed 67% to 89% fewer interactions with the application user interface to reproduce bugs when considering context data (78% on average). We used an unpaired t-test (α < 0.05) and found that both differences are statistically significant (p=0.0002).

The first bug, in the DuckDuckGo app, was successfully reproduced by 4 of 4 members of the experiment group using the app review and context information, as well as 3 of 4 members of the control group only using the review. The average time needed by members of the experiment group was 1:18 minutes with 4 interactions. For members of the control group, the time needed was 4:32 minutes with 35 interactions. We conclude that isolated steps to reproduce help developers to reproduce bugs. For this bug, the second reproduction step, i.e., to enter a specific search term, is relevant, as the too small text only appears for search results with a meanings section.

The second bug, in the Firefox app, was successfully reproduced by 4 of 4 members of the experiment group and 4 of 4 members of the control group. The average time needed by the experiment group was 3:58 minutes, with 5 interactions. The control group needed 5:40 minutes with 21 interactions. We conclude that context data related to the system, in this case, the system version, helps developers to reproduce bugs faster. With context data, developers were able to reproduce the bug fast and precisely with fewer interactions, while developers without context data needed more interactions to find the affected view and had to switch system versions as they chose it wrong initially.

The third bug, in the Wikipedia app, was successfully reproduced by 4 of 4 members of the experiment group and 3 of 4 members of the control group. The time needed by the experiment group was 3:19 with 9 interactions on average. The control group needed 5:18 minutes with 28 interactions. We conclude that the context related to the system, in this case the region (language and locale) and steps to reproduce, helps developers to solve the bug faster. The experiment
### 9.2. Evaluation Results

Table 9.6: Quantitative experiment results of bug reproduction sessions using both app review and context data or using the app review only as bug report.

<table>
<thead>
<tr>
<th>App</th>
<th>Type</th>
<th>Participant</th>
<th>Time to Reproduce (in min.)</th>
<th># of Steps to Reproduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>DuckDuckGo</td>
<td>App Review &amp; Context Data</td>
<td>P1</td>
<td>1.23</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P4</td>
<td>1.19</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P5</td>
<td>1.03</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P10</td>
<td>1.28</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>(Only) App Review</td>
<td>P2</td>
<td>1.39</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P6</td>
<td>4.08</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P13</td>
<td>1.32</td>
<td>4</td>
</tr>
<tr>
<td>Firefox</td>
<td>Participant</td>
<td>P2</td>
<td>4.42</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Time to Reproduce (in min.)</td>
<td>P3</td>
<td>4.48</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td># of Steps to Reproduce</td>
<td>P10</td>
<td>4.32</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P7</td>
<td>4.18</td>
<td>5</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>Participant</td>
<td>P6</td>
<td>3.28</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Time to Reproduce (in min.)</td>
<td>P8</td>
<td>3.36</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td># of Steps to Reproduce</td>
<td>P14</td>
<td>3.31</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P9</td>
<td>3.31</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P15</td>
<td>3.31</td>
<td>9</td>
</tr>
<tr>
<td>Wordpress</td>
<td>Participant</td>
<td>P9</td>
<td>5.15</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Time to Reproduce (in min.)</td>
<td>P12</td>
<td>5.37</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td># of Steps to Reproduce</td>
<td>P16</td>
<td>5.40</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P11</td>
<td>5.37</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P5</td>
<td>5.08</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P14</td>
<td>5.18</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P4</td>
<td>5.40</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P15</td>
<td>5.18</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P5</td>
<td>5.08</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P14</td>
<td>5.18</td>
<td>4</td>
</tr>
</tbody>
</table>

Note: 3 min. for each participant.

Exc. = Excluded participant.
Table 9.7: Results of simulations with closed-source apps.

<table>
<thead>
<tr>
<th>App</th>
<th>Size</th>
<th>Usage</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CPU</td>
<td>Memory</td>
</tr>
<tr>
<td>News</td>
<td>25,951 kB</td>
<td>&lt;1%</td>
<td>0.002%</td>
</tr>
<tr>
<td>Weather</td>
<td>74,285 kB</td>
<td>&lt;1%</td>
<td>0.007%</td>
</tr>
<tr>
<td>Travel</td>
<td>69,865 kB</td>
<td>&lt;1%</td>
<td>0.097%</td>
</tr>
</tbody>
</table>

The group needed fewer steps to reproduce on average. The time difference is smaller as developers reading app reviews with context data reported to have already thought of comparable bugs from their experience.

The fourth bug, in the Wordpress app, was successfully reproduced by 4 of 4 members of the experiment group and by 0 of 4 members of the control group. The experiment group needed 5:08 minutes, with 4 interactions on average. We conclude that bugs depending on several configurations, e.g., the device model and system version, are too complex to be reproduced by developers in time without context data. Members of the control group often needed to switch between devices after figuring out that the chosen combination of device and system version was not affected. Switching devices led to the result that none of the members of the control group was able to reproduce the bug in time.

The results highlight that developers have a better understanding of the reported behavior when context data is given. As the textual bug descriptions, i.e., app reviews, often miss important information, developers ended up randomly browsing through the app to reproduce the reported behavior. With context data, developers were able to reproduce the bugs more precisely.

Simulations

We instrumented three closed-source apps to measure the CPU and memory usage overhead introduced by our approach to implicitly capture context information. Additionally, we determined the size of the collected context information. Table 9.7 summarizes the results of our simulation.

The CPU overhead is in all cases below 1%. Xcode Instruments uses a precision of 1% for the CPU usage. Thus, we were unable to observe any overhead in the 3 cases. The memory overhead is in the worst case 0.1%. The disk usage of the persisted context data is on average 250 kB per 1000 interactions, including app, device, sensor, and system context. The screenshot is in the worst case 391 kB, which highly depends on the screen resolution and content. The overall disk usage of our approach is minimal, with below 2%. Also, the context information is only stored temporarily on the device until it is transferred to the server.
9.2. Evaluation Results

Interviews

When asked for problems encountered with current bug reports, participant 13 (P13) states: “Being provided wrong steps to reproduce as well as app version or device model leads to incomprehensible bug reports and wastes valuable resources of developers and testers.”. P5 adds: “Having specific logs leading up to the event, or user specific information so I can fully reproduce their experience, is the most common missing information and some of the most helpful.”.

The statements additionally indicate that developers are supported by context information when reproducing non-crashing bugs. As a result, all participants reported they would integrate the approach to implicitly capture and isolate context information within their applications, also due to its easy integration, as no source code of the existing applications needs to be modified.

When asked for their concerns, participants brought up issues related to privacy, i.e., the transfer of personal user information. P1 recommends: “[…] provide an opt-out functionality for users to deactivate the capturing of context data.”, as also suggested by P3.

Further, the participants were also concerned about stability and performance, e.g., whether the approach leads to crashes or introduces additional runtime overhead.

Regarding the usability of our approach, P14 responds: “Maybe it makes sense to integrate a functionality to record audio comments in cases users are unable or do not want to type a text.”. P6 adds another idea: “A trust-index for reporters, which increments with every correct review [i.e., review validated by other users], might be helpful. The index could be used to filter false reviews.”.

Another important aspect is the integration of the collected information into developers’ and testers’ workflows. P2 and P9, both software testers, similarly state: “[…] a JIRA integration could save a lot of my daily work. Often, I forget to attach complete context data when creating bug reports. When developers ask me to provide the missing data, I have sometimes forgotten it and therefore need to reproduce the bug again.”.

P10 and P15 also highlight the importance of an integration for popular issue trackers. This integration should automatically create tickets on non-crashing bugs within issue trackers, including structured context information, such as steps to reproduce, and a screenshot. These tickets can directly be assigned to developers, e.g., those who contributed last to the affected features. Further improvements are an IDE integration showing similar bugs for developers to work on, as well as the mapping of user interactions to their executed source code.
9.3 Limitations and Threats to Validity

We ran an in-the-wild experiment since the real distribution of fake and regular reviews in app stores is unknown. We are the first to study fake reviews in app stores and consider our classifier as a first step towards automatically identifying fake reviews. The phenomenon of fake reviews has been studied in other domains already. For example, Streitfeld [267] reports that every fifth review submitted to Yelp is detected as dubious by internal filters. Based on these studies, we assume an amount of 15% fake reviews when comparing our classifiers. However, this amount could be significantly different in app stores and influence the selection of the classification algorithm.

We ran an experiment since we were unable to identify and reuse publicly available context data, including execution and user interaction context, which can be associated with app reviews. Steps to reproduce, e.g., captured with Google Analytics [106], are incomplete and miss the execution context. Also, interactions that lead to expected or unexpected app behavior can not be differentiated. Therefore, we reproduced bugs used in the experiment by students. The collected context information, which is presented to the developers, simulates the user behavior, and might differ from real user interactions. Also, we had to reproduce an app review describing a bug within the Wordpress app. We asked an iOS user who is unfamiliar with app development to write the review.

To help participants reproduce bugs without context data, we restricted the iOS simulators to three system versions in our setting, instead of using eight system versions. Using all simulators could increase the time developers require to reproduce bugs without context data.

Further, the measurements of the runtime overhead should be repeated with additional apps on different devices using alternative instrumentation tools.

To increase the generalizability of our results, we selected apps of different categories, complexity, and sizes. However, our selection was limited to open source apps that all participants did not use before, for all participants to have the same level of experience and to ensure comparable results. Choosing closed source apps developers are familiar with could influence the results, but these cannot be shared between participants that work for different companies due to contractual agreements.

In order to strengthen our results, we selected developers as well as testers with different levels of experience. However, the choice of participants could be biased to iOS developers whose experience is above average.

To increase the realism of the experiment, we selected real bugs in real apps. The experiment could be biased as a result of the app selection. One bug was implemented on our own as we were unable to identify a bug of this complexity within the selected apps, e.g., by browsing issue trackers or reading app reviews.
9.4 Summary

To evaluate the performance of machine learning algorithms when detecting fake reviews in a realistic setting, we conducted an in-the-wild experiment using datasets with different distributions of fake reviews. We found that the Random Forest classifier identifies fake reviews, given a proportional distribution of fake and regular reviews as reported in other domains, with a recall of 91% and AUC/ROC value of 98%.

To evaluate our approach that implicitly captures context information and isolates the occurrence of bugs, we conducted experiments with 16 experienced developers using four open-source apps. We found that isolated context information improves developers’ understanding and ability to reproduce bugs. Developers need between 30% to 71% less time (46% on average) to reproduce an issue while performing 67% to 89% fewer interactions (78% on average) with the application user interface.

A simulation showed that our approach to implicitly capture context information introduces no considerable runtime overhead. To the overall memory usage of the app it introduces, on average, below 0.1%, and to the disk usage below 2% overhead. Finally, we conducted interviews with 16 experienced iOS developers that underlined the benefits of captured and isolated context information and their willingness to integrate our approach into their apps and the applied software development workflow.
Chapter 10

Discussion

User feedback is of great potential value as it allows developers to develop apps according to the needs of users. In the previous chapters, we highlighted the problems developers face when working with user feedback. We presented solutions to identify fake user feedback, to augment user feedback with context information, and to isolate the occurrence of reported issues. With these solutions, we aim to increase the quality of user feedback to support software evolution. Further, we introduce release strategies and an approach for developers to quickly assess users’ satisfaction with app changes as expressed in their feedback, also for channels where no ratings are available. Our presented approaches focus on automation from the perspective of developers, to allow the integration and, thereby, the consideration of user feedback within DevOps processes. In this chapter, we discuss our findings.

The remainder of the chapter is organized as follows. We summarize the limitations of our findings in Section 10.1. Then, we discuss the use of blockchain technologies as one future direction to avoid fake user feedback in Section 10.2.

10.1 Limitations

There are three limitations of our work, which result from focusing on informal feedback channels, the perspective of developers, and popular apps.

Informal Feedback Channels. Our work focuses on informal feedback channels, such as app stores, social media, and user forums. We did not study formal channels, such as information provided within issue tracking systems. Moreover, we did not study any information provided via, e.g., source code control systems. Informal channels were not specifically designed for requirements-related feedback, such as bug reports. Formal channels support users in providing rele-
vant information, e.g., using structured templates that request specific context information as the app version, and might, therefore, include feedback of better quality. Also, we assume that these are tailored towards and used by more technical experienced persons. We base this assumption on our pre-study we conducted on different feedback channels (cf. Section 4.2). The study shows that the more effort users invest in providing their feedback, e.g., signing up for a specific feedback channel, the more requirements-related information is provided.

Even if developers offer, e.g., public issue tracking systems to their users already, we consider the user feedback provided via informal feedback channels as helpful for developers. Several studies (e.g., [200, 302]) already showed that by considering informal user feedback, additional bugs could be identified and bugs can be discovered earlier compared to issue tracking systems.

**Developer Perspective.** In this thesis, we focus on the perspective of developers. The problems we identify are mainly discussed from the development perspective, and our solutions presented aim to support developers during their work. We do not focus on app users.

App users, however, are affected by our solutions in several ways. Our approach to identify fake reviews is highly relevant to maintain users’ trust in feedback channels, such as app stores. Users often base their decision to download apps on the reviews. When being misled several times by fake reviews to download apps, their trust in these might decrease. In the worst case, app users could also refrain from providing reviews, and developers would no longer be able to use this valuable source of information.

Our chatbot approach decreases the effort for support teams by automatically requesting missing context information. At the same time, the approach increases the effort for reporting users. These are asked to provide additional information, in the worst case, several times until the required information is complete. The problem is therefore shifted from developers to users. To solve this issue, we also propose an approach to capture context information automatically. We assume shifting the problem towards users as a better alternative for reasons of scalability, i.e., there are many more users than developers. Similar applies to the context isolation approach.

**Popular Apps.** Our work focuses on popular apps. We mainly analyze the user feedback of apps that receive several hundred or thousand app reviews, tweets, and posts per day. We do not focus on smaller apps, e.g., published by individual developers.

The amount of feedback for less popular apps might be significantly lower. Moreover, these apps might not offer additional feedback channels, such as an
official support account on Twitter, besides the default feedback functionality offered by app stores. When identifying the characteristics of regular and fake reviews (cf. Section 5.2), we found that about 70% of all apps in the Apple App Store received less than ten app reviews. When receiving such low amounts of feedback, support teams do not exist, and our approach, e.g., to automatically request missing information from reporting users, is not needed. For such apps, automated approaches could even be counterproductive and developers should focus on communicating with each reviewer individually to extract additional information on users’ needs and ideas for the future development of their apps. Another limitation are specific use cases for that several alternative apps exist. For these, users might simply not willing to provide additional information when being asked but switch to an alternative app instead.

10.2 Untampered Feedback on the Blockchain

This section is based on the paper “ReviewChain: Untampered Product Reviews on the Blockchain” [187] by Martens and Maalej, published at the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain in 2018. My contribution to this publication is conducting all research, implementation, and analysis work, as well as leading the writing of the paper. My co-author supervised the work, inspired the research questions and goals, and made final edits to the paper.

When deciding between products, reviews have a considerable impact on consumers’ choice [47, 112]. Consumers that rely on reviews when deciding for a product have to trust at least two parties involved. These are the review authors and the operators of online portals. Untrustworthy reviews of single authors, e.g., an extremely positive review in between negative reviews, can possibly be recognized by consumers. However, this does not apply for larger amounts of reviews modified by online portal operators themselves. The operators act as central authorities throughout the complete review process. In the worst case, an operator can exclude consumers from submitting reviews, modify existing reviews, and introduce fake reviews to improve the ratings and rankings of products [90, 148, 204].

In this section, we discuss the use of blockchain technologies as one future direction to avoid fake user feedback. We present a decentralized reviewing approach. Our approach avoids central authorities by using blockchain technologies, decentralized apps and storage. It enables users to submit and retrieve untampered, i.e., authentic, user feedback. We highlight the implementation challenges encountered when realizing our approach on the public Ethereum blockchain. Then, we discuss possible design alternatives and their trade-offs regarding costs, security, and trustworthiness. Finally, we analyze which design
decision should be chosen to support specific trade-offs and present resulting combinations of decentralized blockchain technologies, also with conventional centralized technologies.

10.2.1 Ethereum Blockchain

Ethereum is a blockchain, developed in 2014 [79]. In comparison to the Bitcoin blockchain, which only handles accounts and transactions, Ethereum also stores programming logic. When paying for its execution, any Turing complete script can be run on Ethereum. Thereby, it enables decentralized apps without any possibility of downtime, censorship, or third-party interference.

Traditional transactions are digitally signed messages persisted on the blockchain. Each transaction is associated with an action, such as transferring cryptocurrency units. Transactions have a sender and receiver address that represent public keys belonging to specific users.

Ethereum also allows to store and execute programming logic, called smart contracts. These were first introduced in 1994 [273]. In Ethereum, these can be written by, e.g., using the Solidity programming language [306]. Contracts are executed on several nodes of the network within Ethereum virtual machines (EVM). After executing a contract, nodes must reach a consensus of the calculated result.

Transactions and scripts require a fee to be executed as a reward for the miner that executes the operation. This fee is paid using Ether (ETH). Ether is the cryptocurrency traded on the Ethereum blockchain. It is used to pay nodes of the network for executing requested operations, such as transactions.

10.2.2 Decentralized Reviewing Approach

Our decentralized reviewing approach focuses on creating and providing persistent access to untampered product ratings and reviews. Its fundamental goal is to avoid central authorities, being able to influence the review processes. Therefore, our approach utilizes technologies, such as the Ethereum blockchain as decentralized and immutable data storage, and decentralized apps to provide access to reviews. Consumers no longer need to rely on central authorities as blockchains operate decentrally across a network of several nodes, in which every user can participate [4]. Our approach enables consumers to submit and access untampered and trustworthy reviews.

A comparison of the conventional centralized review approach and our decentralized approach is shown in Figure 10.1. In the upper part of the figure, central authorities are able to exclude specific consumers from submitting reviews. Further, authorities store reviews in a centralized database. This enables them to modify existing reviews and introduce fake reviews by fictional con-
10.2. Untampered Feedback on the Blockchain

Figure 10.1: Centralized and decentralized reviewing approach.

sumers that did not buy the reviewed product. In the decentralized approach, shown in the bottom part of the figure, this is no longer possible. Once stored on the blockchain, reviews cannot be modified, neither can consumers be excluded from providing reviews. We refer to these reviews as *untampered reviews*.

Our decentralized reviewing approach must fulfill four requirements. These are extracted from existing review functionalities used for product reviews on Amazon or app reviews within the Apple App Store: 1) Authors must have purchased the reviewed product, 2) review authors must be distinguishable, 3) each author can only submit one review per product version, and 4) review submissions must not result in any costs for authors.

In the following, we apply our approach to the real-world scenario of mobile app reviews. Therefore, we enable authors to submit reviews within native apps and provide access to reviews via a web service. By doing so, we aim to highlight additional challenges which result from the usage of different technological environments. Moreover, we realize our approach on Ethereum to demonstrate the applicability and limitations of blockchain technology.

10.2.3 Implementation Challenges

We discuss major implementation challenges of our approach, as well as possible design alternatives and their trade-offs.

Submitting Transactions via Mobile Apps

After publishing a smart contract on the blockchain, allowing users to store reviews, it is executed by sending a transaction to the contract address. Each
Chapter 10. Discussion

Figure 10.2: Backend/app-side transaction creation.

transaction needs to be signed with the user’s private key. Figure 10.2 shows two alternatives to submit transactions via mobile apps. The first alternative implements the transaction handling within a backend. App and backend communicate using a REST API. The transactions, as well as the private keys, are generated by the backend. The keys are stored within a centralized database. After signing, the transactions are sent to an Ethereum node. Nodes can run on the backend itself. In the second alternative, the creation of transactions and private keys are contained within the app. Signed transactions are directly sent to a node.

In comparison, the first alternative offers several advantages. Computation-intensive operations, such as generating and de-/encrypting private keys, can be performed on the backend-side instead of using a resource-limited mobile device. Also, as fees have to be paid to execute transactions and our approach requires reviewers not to pay for these, the central database of the backend can be used to create a list of all participating users’ addresses. These can be funded beforehand with tokens to pay the transaction fees of review submissions. However, this alternative introduces a major drawback – the backend acts as a central authority. By not authenticating users, operators can exclude those from submitting reviews. Also, the stored private keys can be used to submit reviews without users’ behalf. As a result, a decentralized design, as presented in the second alternative, is preferable. In this case, the mobile app must self-contain the private key, as well as the transaction handling.

To realize the second alternative, an official implementation of the Ethereum protocol, called Go Ethereum (Geth) [100], can be cross-compiled to both Android and iOS. Using Java and Swift wrappers, it can be directly called from within the app’s native implementation. In addition, Geth generates native wrappers for smart contracts. Geth offers account management, remote node interfacing, and enables interactions with smart contracts. For example, private keys are generated and stored in an encrypted keystore. This keystore offers a standard and light security mode. However, due to the mobile device’s re-
source limitations, the author suggests using only the light security mode [100]. Although decentralized, this introduces a security risk. The keys are stored in the app’s document directory. On regular devices, both Android and iOS, this folder can only be accessed by the application itself. On jailbroken devices, apps can access all files on the mobile device. Thereby, the user’s private keys can be transmitted to an external server. A server’s high computational capacities can be used to brute-force the private key’s passphrase and perform transactions afterwards.

Restricting the Reviewing to App Users

After a user submits a review, the transaction, including the review, is created, signed, and sent to an Ethereum node. When the nodes of the network reached consensus, the transaction is immutably persisted on the blockchain. Since each person can participate in the network by downloading a copy of the blockchain and running a node, transactions, as well as smart contracts, are publicly visible. This enables participants of the network to copy, modify (e.g., by changing the review text), and resubmit transactions. The participants must not necessarily have downloaded the app the review targets. Thereby, fake reviews can be introduced to the blockchain.

Unfortunately, it is difficult to solve this issue without using a central authority. In the following, we describe three alternatives to exclude non-app users from submitting reviews to the blockchain. First, the smart contract can specify a white-list of addresses allowed to interact with it. Therefore, for each transaction targeting the contract, its sender address \( \text{msg.sender} \) is compared with the list. Since all addresses (i.e., app users) are unknown beforehand, the contract must implement a method to register additional addresses. As a drawback, this method can be used to register addresses that did not download the app. Second, an ERC-20 token [80] can be issued and distributed to app users. The smart contract methods must require the sender address to own a token before execution. Further, the contract must ensure that these tokens are untradeable between users. This alternative also requires tokens to be transferred to reviewers by a central authority. Third, a private key to sign transactions can be bundled with the app. By solely white-listening this key’s address in the smart contract, only app users are authorized to submit reviews. To use this key, its passphrase must be included in the app as well. This introduces a security risk since key and passphrase can be extracted so that fake reviews can be created from outside of the app. Since all users use the same key, we are unable to distinguish them. As a consequence, we cannot ensure that each user only submits one review per product version.
Reducing Costs of Data Storage

Another challenge is the storage of review data and its associated costs. The Ethereum yellow paper [306] states that a fee of 20,000 gas is required to store a 256-bit word on the blockchain, i.e., 625 gas per byte. Gas is a unit to measure the computational effort required to perform an action within the Ethereum network. The cost of a transaction, e.g., storing data on the blockchain, is calculated by multiplying the amount of gas with the gas price. The gas price is measured in Gwei, where 1 ETH equals 1 billion Gwei. The price is decided by the miners, who perform the transactions, based on the network conditions. If transactions specify a too low gas price, these will not be processed by miners. Vice versa, the higher the price, the faster transactions will be processed. In February 2018, the gas price to process a transaction below 5 minutes is 5 Gwei, while the median price of the last 1,500 blocks is 22 Gwei. The value of one ETH is $885.

A single 7-day Spotify release for iOS, for example, contains 3,025 reviews with an overall size of 270,110 bytes. To store this amount of data, an amount of 168,818,750 gas is required. Considering a gas price of 5 Gwei, the storage costs are 0.844 ETH or $747. This equals $0.247 per review. Using 22 Gwei, the costs are $3,287 or $1.09 per review. The costs can be reduced by defining a low gas price, which consequently requires more time for reviews to be processed.

In the following, we discuss three solutions to reduce storage costs. First, compression algorithms can be used. Alakujala et al. [5] introduce Brotli, a compression algorithm including a static dictionary and thereby appropriate for short texts. Second, instead of the complete review, only its hash can be persisted on the blockchain. The hash length is independent of the review. This reduces the costs of longer reviews. The actual review can be stored on a central server. When accessing reviews, their hashes are compared to those on the blockchain. However, this design introduces a single point of failure. Third, a better alternative is to use a a peer-to-peer distributed file system, such as Swarm [139] or InterPlanetary File System (IPFS) [140], see Figure 10.3. These seek to connect all computing devices with the same system of files. IPFS,
Table 10.1: Optimized trade-offs and their resulting configurations of design alternatives (A1 refers to alternative 1).

<table>
<thead>
<tr>
<th>Trade-Offs</th>
<th>Security</th>
<th>Trust</th>
<th>Costs</th>
<th>Design Alternatives</th>
<th>Optimized for</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Good</td>
<td>Medium</td>
<td>Medium</td>
<td>A2: Via App (Directly)</td>
<td>4.2: Authorization</td>
</tr>
<tr>
<td></td>
<td>Poor</td>
<td>Good</td>
<td>Medium</td>
<td>A2: ERC-20 Token</td>
<td>4.3: Data Storage</td>
</tr>
<tr>
<td></td>
<td>Good</td>
<td>Poor</td>
<td>Good</td>
<td>A2: Centralized</td>
<td>4.4: Transact. Fees</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A3: Smart Contract</td>
<td>4.5: Retrieval</td>
</tr>
</tbody>
</table>

Section 4.1: Submission | 4.2: Authorization | 4.3: Data Storage | 4.4: Transact. Fees | 4.5: Retrieval | Optimized for

for example, provides an immutable, content-addressed block storage model. Each item can be identified by a hash and accessed using an URL. By using a distributed file system, the single authority and point of failure is removed.

**Third-Party Transaction Fee Payments**

When performing a transaction, the sender (in our case the reviewer) must pay the fee. As our approach requires that submissions are free of costs for reviewers, we present and discuss three design alternatives. First, as we decided against using a centralized backend to store users’ private keys, we do not know their addresses. To collect these, a website can be deployed for users to (automatically) publish their addresses. Afterwards, the addresses can be funded with ETH. This website can be used without submitting a review. Second, free-of-charge transactions with a gas price set to zero can be used. These transactions are very unlikely to be picked by miners. Therefore, a central authority has to mine all transactions going to the contract’s address. As a drawback, the authority could decide against processing single transactions to, e.g., exclude users writing negative reviews. Also, the time to process a review highly depends on the mining capacity of the central authority. Third, to avoid a central authority, a smart contract can be used to reward miners. A miner that processed a transaction pointing to the address of the contract has to prove that the gas price was zero. The smart contract can then refund the miner for processing the transaction. The reward is paid from a pool address into which product vendors deposit tokens. It should equal the median transaction fees, normally paid by transaction senders so that these are as likely to be picked by miners as regular transactions.

**Retrieval of Untampered Reviews**

Our aim is to enable users to access reviews on the blockchain in a usable and untampered manner. Therefore, we deploy a decentralized app (dApp). This app must fulfill several requirements: It must be completely open-source, operate autonomously, and data must be decentrally stored to avoid central points of failure. Further, dApps must use tokens for access and to rewards miners, e.g., gas.

From an implementation perspective, the dApp consists of a frontend and backend. The frontend can be developed in any programming language, such as JavaScript. To fulfill the above mentioned requirements, it should be deployed on a decentralized storage. The backend is implemented as a smart contract and deployed on the Ethereum blockchain. The dApp connects to an Ethereum node using a remote procedure call (RPC) connection. As the reviews are timely distributed, the node has to have a full replica of the blockchain. In case
10.2. Untampered Feedback on the Blockchain

A remote node is used, the user has to trust that it returns unmodified results. To ensure that the dApp reads untampered data, users have to start a node by themselves. This downloads a large amount of data, which requires space and time and might be inappropriate for mobile devices.

10.2.4 Blockchain Design Trade-Offs

Table 10.1 lists three possible configurations of design alternatives (one per row). Each configuration combines different design alternatives with regard to their trade-offs. The first alternative is optimized for high security. Transactions are submitted directly from within the app, an ERC-20 token is used to authorize consumers to submit reviews, and data is stored decentrally (e.g., using IFPS or Swarm). Further, this option is based on smart contracts to reward miners processing the review transactions. For review retrieval, this alternative uses a dApp accessing a local Ethereum node. For the second alternative, trustworthiness is optimized. In comparison to the first, it bundles the app with a private pool key to sign transactions, and thereby completely avoids central authorities. As a drawback, this design reduces the implementation’s security as the private key and its passphrase can be extracted from the app.

As a drawback, this design reduces the implementation’s security as the private key and its passphrase can be extracted from the app. The last alternative focuses on optimizing costs. It is the same as the first alternative, except for using a centralized database to store the complete reviews (not hashes). Thereby it reduces storage costs, with the drawback of reducing the trustworthiness of reviews, as the database contents can be modified by its operator.

Depending on the trade-offs to be achieved, such as optimizing for security, trustworthiness, or costs, different configurations of design alternatives need to be chosen. This can either lead to a completely decentralized approach, based on blockchain technologies, or a combination of decentralized and centralized design alternatives, such as a blockchain to store review hashes and a conventional database to store its contents.
Chapter 10. Discussion
Chapter 11

Conclusion

Software systems must continually evolve to remain satisfactory to their users [170]. Users describe their needs in feedback provided via app stores, social media, and user forums. Research has shown that user feedback includes requirements-related information, such as feature requests or bug reports [224].

Unfortunately, a large number of users provide feedback of insufficient quality as our results highlight. Users are incentivized to provide unauthentic feedback, with the aim of manipulating apps’ ratings and rankings in app stores. Similar to regular feedback, fake user feedback includes bug reports and feature requests, which might mislead developers when trying to understand real users’ needs. Moreover, users miss to provide relevant context information, such as the affected app version, resulting in issues that are non-actionable, i.e., not understandable and reproducible, to developers.

We propose automated approaches to increase the quality of users feedback. In our evaluation we show that user feedback of sufficient quality, i.e., being real and containing relevant context information, is of great potential to developers.

The quality-improved user feedback should be integrated within continuous software evolution practices, and complement the collection of operational metrics. Continuously involving users into the evolution of software systems through the collection of explicit and implicit user feedback, improves developers’ understanding about the occurrence and impact of reported issues. Further, in oppose to operational metrics that refer to technical aspects of the software system, such as the performance, monitoring user feedback allows to understand users’ satisfaction with evolutionary software changes. This helps developers to develop their apps according to the needs of users.

The remainder of the chapter is organized as follows. We summarize our findings and contributions in Section 11.1. Afterwards, we outline future work in Section 11.2.
11.1 Summary of Contributions

This thesis makes four contributions to improve the quality and integration of user feedback for software evolution. Figure 11.1 integrates the thesis contributions into continuous software evolution (DevOps) phases.

The first two contributions improve the quality, i.e., authenticity and actionability, of user feedback. We increase the authenticity of provided user feedback to help developers trying to understand real users’ needs (contribution A in Figure). Further, we increase the actionability of user feedback to help developers understand and reproduce reported bugs (contribution B).

The remaining two contributions integrate user feedback into the continuous software evolution processes. We isolate the occurrence of reported issues using crowdsourcing (contribution C). Moreover, we provide an integrated overview and enable developers to monitor users’ satisfaction with evolutionary software changes as expressed in their feedback provided via different channels (contribution D).

In the following, we describe and separate each contribution by their sub-contributions.

11.1.1 Authenticity of User Feedback

We support developers in understanding and detecting fake reviews in app stores to increase the authenticity of user feedback. We found that part of the fake reviews contain requirements-related information, such as feature requests and bug reports. When not removing fake reviews before further analysis, developers might be misled by, e.g., fake feature requests, when trying to understand real users’ needs. To our best knowledge, we are the first to analyze fake reviews in app stores. Our contributions can be summarized as follows:

1. We analyze the fake review market, including fake review providers, their offers to manipulate ratings and rankings in app stores, and the policies submitted fake reviews must comply with in Chapter 3. We focus on the following aspects:

   a) We study by whom fake reviews are offered and what strategies fake review providers follow. We show that developers buy reviews from paid review providers or deal with reviews in exchange portals.

   b) We identify the exact services fake review providers offer and their conditions. Providers offer fake ratings, reviews, and installs. Fake reviews are offered for relatively expensive prices of a few dollars.

   c) We analyze the providers’ policies fake reviews must comply with and extract initial indicators to detect fake reviews. We collect the
11.1. Summary of Contributions

User Classify
user feedback
Remove fake feedback
Augmentation
of context
information
Isolation
of context
information
Explicit Feedback
Implicit Feedback

Figure 11.1: Integration of thesis contributions into continuous software evolution (DevOps) phases.
policies from providers’ websites and through a disguised question-
naire. Our results show that providers invest huge effort to disguise
fake reviews and that fake reviews do not mean short reviews. Fake
reviews are written to sound authentic, i.e., including custom key-
words or pre-defined texts that discuss actual app features, and are
submitted by humans.

2. We present a classifier to identify fake reviews submitted to app stores in
Chapter 5. We focus on the following aspects:

a) We extract and compare about 62 million regular app reviews with
about 60,000 fake reviews to reveal empirical differences between the
reviews and their corresponding apps and reviewers. We found that
the properties of the corresponding app and reviewer differ most be-
tween regular and fake reviews.

b) We create a gold-standard truthset for fake reviews and describe the
development of a supervised machine learning classifier to detect fake
reviews.

c) We identify the features that can be used to detect fake reviews,
compare the results of different machine learning algorithms, and
optimize the classification results.

d) We perform an in-the-wild experiment to understand how the classi-
fiers perform on datasets with different distributions of regular and
fake app reviews since the proportional distribution of fake reviews in
app stores is unknown. We found that the Random Forest classifier
identifies fake reviews, given a proportional distribution of fake and
regular reviews as reported in other domains, with a recall of 91%
and AUC/ROC value of 98%.

3. We discuss the use of blockchain technology as one future direction to avoid
fake user feedback in Chapter 10. We focus on the following aspects:

a) We present a decentralized reviewing approach that can be realized
using blockchain technologies, e.g., smart contracts. The approach
avoids central authorities, such as app store operators, storing re-
views within a centralized database, and being able to, e.g., exclude
specific users from submitting reviews.

b) We discuss the major implementation challenges encountered when
realizing our approach on the Ethereum blockchain, including the
submission of transactions via mobile apps, restricting the reviewing
to actual users of the app, and reducing costs for data storage. Fur-
ther, we discuss how costs that are associated with executing smart
11.1. Summary of Contributions

contracts can be avoided for reviewers. Last, we describe the usage of decentralized apps to retrieve submitted reviews via an appropriate user interface without the possibility of the reviews being manipulated.

c) We discuss the trade-offs developers have to consider when realizing our decentralized reviewing approach using blockchain technologies. Developers can optimize our approach for security, trust, or costs by combining different presented design alternatives.

11.1.2 Actionability of User Feedback

We increase the actionability of the issues reported in user feedback by collecting and isolating context information. Thereby, we support developers in understanding and reproducing reported issues, especially non-crashing bugs. Our contributions can be summarized as follows:

1. We analyze the relevance of context information for developers to understand and reproduce reported issues in Chapter 4. We focus on the following aspects:

   a) We present the development of an unsupervised approach that uses pre-defined keyword lists, word vector representations, and text patterns to extract basic context items, including the platform, device model, app version, and system version, from informal user feedback.

   b) We evaluated our approach against a manually labelled truthset of about 3,000 tweets, the approach achieved precisions from 81% to 99% and recalls from 86% to 98% for the different context item types.

   c) We apply our context extraction approach to about 3 million tweets from support accounts of the three popular apps Netflix, Snapchat, and Spotify to highlight the effort support team invest in clarifying missing context information. We found that users and support teams exchange context information related to iOS or Android in about every tenth conversation. More than half of all extracted context items are provided only after the engagement of support teams. Support teams participate in about 40% of the conversations including context items, until relevant items are present.

2. We present two complementary approaches to augment newly created and existing user feedback with relevant development information in Chapter 6. We use the collected information to auto-populate issue trackers with structured bug reports that are actionable to developers. We focus on the following aspects:
Chapter 11. Conclusion

a) We present an in-situ context capturing approach that augments newly created user feedback with context information implicitly captured during app executions. This information is automatically attached to their explicitly provided user feedback. We highlight the implementation challenges developers face when realizing our approach on the iOS platform.

b) We present a combination of a context extraction approach and a chatbot approach that identify and explicitly request missing context information for existing user feedback, e.g., submitted to social media, from reporting users. The approach is complementary and can be applied to feedback reported without the use of the implicit context augmentation approach. We describe its implementation as a web-based application.

11.1.3 Isolation of Reported Issues

We integrate user feedback into continuous software evolution practices. Thereby, we aim to benefit from the feedback of multiple software users. Our contributions can be summarized as follows:

1. We present an approach to isolate the provided context information using crowdsourcing in Chapter 7. The approach highlights relevant context items to developers and removes irrelevant information. We focus on the following aspects:

   a) We use crowdsourcing to validate the occurrence of reported issues under different configurations, e.g., system versions. Therefore, when opening the app view possibly affected by the reported bug, the user is presented a validation form, which depicts the bug report's textual description and optionally an annotated screenshot. A server collects the response and context information of all users validating the issue. The validation also allows detecting temporary issues.

   b) We present an approach that isolates both the execution context and the interaction context by comparing the information of the reporting users and the validators. The approach identifies recurrent patterns, e.g., which system versions are affected or which user interactions facilitate the defective behavior. This information is used to highlight relevant context items to developers and remove irrelevant information.

2. We present an evaluation to determine if context augmented and isolated user feedback supports developers in understanding and reproducing non-crashing bugs in Chapter 9. We focus on the following aspects:
11.1. Summary of Contributions

a) In experiments, 16 professional iOS developers reproduced four real bugs from open source apps that were described within app reviews. The developers were either given or not the corresponding context information. In the experiments, we found that context information improves developers’ understanding of reported issues. Developers need 30% to 70% less time to understand and reproduce reported issues. The improved understanding is also reflected by the lower number of interactions, 78% on average, developers perform during bug reproduction.

b) In simulations with three closed-source apps, we show that our approach to implicitly capture context information and isolate the occurrence of reported issues does not produce observable runtime overhead and memory usage.

c) In interviews, we asked the developers for problems encountered with current bug reports. We collect and discuss their concerns regarding our approach to extract possible future improvements. Developers highlighted the importance of integrating user feedback into continuous software evolution practices.

11.1.4 Monitoring Users’ Satisfaction with App Evolution

We further integrate user feedback into continuous software evolution practices to complement the collection of operational metrics that only refer to technical aspects of the software system, such as the performance, and do not mirror users’ satisfaction with evolutionary software changes. Our contributions can be summarized as follows:

1. We support developers in monitoring users’ satisfaction with evolutionary app changes and suggest strategies to release changes in Chapter 8. We focus on the following aspects:

   (a) We introduce and discuss the use of sentiment analysis tools to extract users’ emotions from written feedback and allow developers to quickly assess the impact of implemented changes for channels where no ratings are available, such as Twitter.

   (b) We empirically analyze about 7 million app reviews of popular apps to determine if users’ emotions correlate with the rating, price, or content. We found a moderate correlation between the star rating and the sentiment that can be used to quickly assess user satisfaction in channels where no ratings are available. The results highlight that sentiment analysis tools need to be adjusted to the domain


Chapter 11. Conclusion

of software engineering, e.g., for words such as 'bug' that refers to negative experiences with an app.

(c) The emotionality of app reviews can vary over time. For example, new app features might increase the average sentiment. We analyze the development of the sentiment of all apps within our dataset that received more than 1,000 app reviews to identify emotional patterns. We identified four emotional patterns. These are consistent emotion, inconsistent emotion, steady decrease/increase, and emotion drop/jump.

(d) We compare the emotional patterns to introduced app changes to derive release strategies for software practitioners. We present five release lessons that guide software practitioners in deciding how to release software changes without decreasing users’ experience and, therefore, satisfaction with the app.

11.2 Future Work

This section separates and describes future work by aspects related to research and the implementation.

11.2.1 Research

We particularly focused on the Apple App Store and Twitter as our feedback channels. Our collected results need to be replicated on further feedback channels, such as Google Play or smaller app stores, such as f-droid [82].

The classification results, e.g., for detecting fake reviews, need to be replicated with larger and more diverse datasets. Especially for fake reviews, creating a gold-standard truthset only including actual fake reviews was challenging.

Results related to users’ emotions in app reviews need to be replicated with sentiment analysis tools that are adjusted to the domain of software engineering. We deliberately decided to use SentiStrength [280] as it is commonly used as baseline for emotion classification. SentiStrength is designed for short informal texts and does not consider words, such as 'bug', that have a different meaning within the domain of software engineering in their current implementation and configuration files. Novielli et al. [216] conduct a benchmark study to assess the performance of three sentiment analysis tools fine-tuned for the software engineering domain. Future studies should consider the use of fine-tuned tools and perform a custom retraining of the used classifiers as recommended by the authors.
11.2 Implementation

Future work should consider the integration of augmented and isolated user feedback within current development environments. For example, a prototypic implementation of a JIRA plugin that displays relevant steps to reproduce could be evaluated in experiments and interviews with developers.

Also, our approach to capture context information should be realized on different platforms, such as Android or web. For Android context information, such as user interactions, can be collected on system level in oppose to the iOS platform that only allows capturing context information on app level [102]. This allows capturing context information globally without the need of integrating a context capturing approach into every app.

Future implementations should also consider the grouping of reported issues. Especially when a large portion of users is affected by a bug, this is reported through different channels by different users. The issues should be grouped to compare the provided information, e.g., different system versions affected. For bugs frequently reported an additional context augmentation and isolation phase as described in Chapter 6 and Chapter 7 might thereby no longer be required. Moreover, the reported issues can be compared to already existing reports within issues trackers. Specific bugs that have been reported by users might have already been resolved but not released yet. Here, reporting users can be automatically informed about the current progress.

The privacy of users should be considered more carefully within future implementations. For example, the user interactions might reveal details about the app usage, e,g., viewing specific political accounts on Twitter that the app user does not want to reveal. Here, options must be introduced for reporters and validators to anonymize the collected information. Also, the use of anonymization approaches should be considered to replay interactions that require a login and are dependent on the user’s information [50].

Moreover, future work should study and integrated features that motivate reporters and validators to provide the information requested by developers.
Appendices
Appendix A

A Study of Fake User Feedback

A.1 Disguised Questionnaire

Dear Sir or Madam,

I am currently looking for a service that offers reviews. My company offers both, an iOS and Android app in a highly competitive field. We have several competitors which gain more and more market share. For this reason we are looking for both positive and negative reviews, positive for our apps and negative for our competitors’ apps. Are you able to provide both types of reviews?

Additionally I would like to clarify if it is possible to provide keywords or even completely written reviews which you just publish. Do you ask real users to provide the reviews or do you just use fake account? Is there a guarantee that these reviews won’t be deleted from the app stores? Is it also possible to distribute the reviews over different countries? We offer our apps worldwide and also need reviews in foreign languages, if yes which languages do you offer?

If possible, can you provide us some example reviews already published, to verify the reviews’ quality?

I am looking forward to hear from you.

Best regards
Appendix A. A Study of Fake User Feedback
Appendix B

Evaluation

B.1 Interview

The semi-structured interview with professional iOS developers is shown in Figure B.1, Figure B.2, and Figure B.3.
Interview: Augmenting App Reviews with Context Data

1. General Questions

1. How many years of development experience (general) do you have?

________________________________________________________________________

2. How many years of development experience (iOS) do you have?

________________________________________________________________________

3. What is your focus domain?

________________________________________________________________________

4. What kind of work do you do?
   (e.g. development, project management, quality assurance)

________________________________________________________________________

2. Questions regarding App Reviews

1. Do you use app reviews as a source of information during your work?
   ☐ Yes ☐ No

   If true: Do you use app reviews to gather requirements related information?
   ☐ Yes ☐ No

   If true: Which requirements related information do you extract from app reviews?
   ☐ Bug reports
   ☐ Improvement request
   ☐ Feature requests
   ☐ Other: ________________________________________________________________

2. Did you identify problems when working with app reviews?
   ☐ Yes ☐ No

   If true: What problems did you identify?
   ______________________________________________________________________
   ______________________________________________________________________

Figure B.1: Questions of the interview with professional iOS developers (1/3).
3. **Experiment: Reproduce Bugs**

1. **Given:** App review
   
   Time req. Bug\textsubscript{1} \_____________ : _____________________________________________

2. **Given:** App review + Context data
   
   Time req. Bug\textsubscript{2} \_____________ : _____________________________________________

4. **Questions regarding Approach**

4. **Did the provided context data help you while understanding and reproducing the bugs?**
   
   ☐ Strongly disagree ☐ Disagree ☐ Neutral ☐ Agree ☐ Strongly agree

4. **Is the library easy to integrate into applications?**
   
   ☐ Strongly disagree ☐ Disagree ☐ Neutral ☐ Agree ☐ Strongly agree

4. **Would you use the library for your applications?**
   
   ☐ Strongly disagree ☐ Disagree ☐ Neutral ☐ Agree ☐ Strongly agree

4. **Do you already capture context data during application executions using other tools? (e.g. crash reporter)**
   
   ☐ Yes ☐ No

   *If true: What tools do you use?*

   _____________________________________________

4. **What context data are the most helpful?**

   _____________________________________________

---

Figure B.2: Questions of the interview with professional iOS developers (2/3).
Appendix B. Evaluation

6. What are your concerns when using the library inside your applications?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

7. Do you have improvement requests regarding our approach/library?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

8. Do you have feature requests regarding our approach/library?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

9. Additional Notes

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

Figure B.3: Questions of the interview with professional iOS developers (3/3).
List of Figures

1.1 Integration of thesis contributions into continuous software evolution (DevOps) phases. ........................................ 6

2.1 Staged model of the software life cycle [234]. ......................... 13
2.2 Different issue types reported in issue tracking systems. ......... 14
2.3 Example of a structured bug report regarding the Firefox app for iOS within the issue tracking system of Mozilla [37]. ....... 16
2.4 Typical issue states and workflow within an issue tracking system. 17
2.5 Context information divided by execution and interaction context. 18
2.6 Example user interaction trace including three app views (v_i) and three events (e_i) captured from the Spotify iOS app. ........... 20
2.7 Continuous delivery through development and operations loop, i.e., DevOps (source: [221]). ......................................... 24
2.8 User involvement in software development separated by its stages and forms (according to [62, 219]). ............................. 26
2.9 Analysis model of user feedback, separated by explicitly and implicitly provided feedback. ................................. 28
2.10 Example of explicit user feedback regarding the WhatsApp app provided via the Apple App Store and Google Play. .......... 29
2.11 Example of explicit user feedback provided via Spotify’s official Twitter support account. ..................................... 30
2.12 Example of explicit user feedback provided via Spotify’s official user forum. ...................................................... 31
2.13 Screenshot of a crash reports as displayed within the Apple Crash Reporter (adapted from: [13]). ................................. 33
2.14 Non-crashing bug reported within InstaBug (source: [57]). ....... 34

3.1 Research method including structured Google search, extraction of fake review providers’ offers and pretended fake review characteristics. ........................................ 45
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>Screenshot of a fake review provider, offering a) app developers to buy reviews and b) non-/developers to sign-up and write rewarded reviews. Section c) shows a list of apps requesting fake reviews against monetary rewards (obfuscated by the author).</td>
<td>46</td>
</tr>
<tr>
<td>3.3</td>
<td>A fake review request with predefined rating and review on a review exchange portal.</td>
<td>47</td>
</tr>
<tr>
<td>3.4</td>
<td>Reviewing strategies of fake review providers.</td>
<td>47</td>
</tr>
<tr>
<td>4.1</td>
<td>Overview of the research method including the data collection, data preparation, truthset creation, and data analysis phase.</td>
<td>62</td>
</tr>
<tr>
<td>4.2</td>
<td>Example of a conversation between user and support team on Twitter to obtain missing basic context items (i.e., the device and platform).</td>
<td>63</td>
</tr>
<tr>
<td>4.3</td>
<td>Docanno tool used to manually annotate basic context information within tweets of the truthset.</td>
<td>66</td>
</tr>
<tr>
<td>4.4</td>
<td>Distribution of requirements-related issues within different feedback channels.</td>
<td>69</td>
</tr>
<tr>
<td>4.5</td>
<td>User and developer exchanging context items on Twitter to reproduce a non-crashing bug, separated into context items initially provided by the user, requested by the developer and provided by the user, and missing although requested by the developer.</td>
<td>70</td>
</tr>
<tr>
<td>4.6</td>
<td>Overview of context items within issues reported on Twitter.</td>
<td>71</td>
</tr>
<tr>
<td>4.7</td>
<td>Creation of the filtered Android device list using pre-defined keyword lists and word vector representations.</td>
<td>72</td>
</tr>
<tr>
<td>4.8</td>
<td>Ten most similar words to 'iPhone 8 Plus' (i.e., misspellings or abbreviations), identified using the trained fastText model.</td>
<td>74</td>
</tr>
<tr>
<td>4.9</td>
<td>Intersections between versions included in the pre-defined lists (non-comparable versions are shaded).</td>
<td>76</td>
</tr>
<tr>
<td>4.10</td>
<td>Version matcher processing the input ‘version 8.0.1.785’.</td>
<td>77</td>
</tr>
<tr>
<td>4.11</td>
<td>Percentage of conversations per number of tweets included.</td>
<td>82</td>
</tr>
<tr>
<td>4.12</td>
<td>Distribution of context items in conversations (user tweets only).</td>
<td>84</td>
</tr>
<tr>
<td>5.1</td>
<td>Research method including the data collection, preparation, and analysis phase.</td>
<td>95</td>
</tr>
<tr>
<td>5.2</td>
<td>Screenshots of fake reviews before submission to the app store used as proof for fake review providers, depicting a) fake review included in our study, b) cut-off fake review excluded from the study, and c) non-English fake review also excluded from the study.</td>
<td>97</td>
</tr>
<tr>
<td>5.3</td>
<td>Distribution of fake and official reviews over ranges per app.</td>
<td>102</td>
</tr>
<tr>
<td>5.4</td>
<td>Ratio of fake reviews for non-/deleted apps.</td>
<td>102</td>
</tr>
<tr>
<td>5.5</td>
<td>Number of reviews provided per fake or official reviewer.</td>
<td>103</td>
</tr>
<tr>
<td>5.6</td>
<td>Distribution of star ratings between official and fake reviews.</td>
<td>105</td>
</tr>
</tbody>
</table>
5.7 Distribution of review length (in characters) between regular and fake reviews. .................................................. 105
5.8 Percentage of reviews received per day after app release (day 0 is release of app update). .................................. 106
5.9 Delta between occurrences of most-common words in official/fake reviews, negative rankings indicate that the specific word is more common for fake reviews while positive rankings denote that the word is less common. .................................................. 107
5.10 Overview of fake review classification. .......................... 108
5.11 Precision of random forest classifier by number of features selected. 112
5.12 Relative importance of extracted features to detect fake reviews. 113
5.13 Reviews within truthset classified as bug report and/or feature request. .................................................. 115

6.1 Implicit capturing approach to create structured bug reports including context information. .................. 123
6.2 Screenshot of the report creation form allowing the user to provide explicit feedback in the form of a textual description and annotated screenshot (e.g., to hide private data). .................. 124
6.3 Combination of extension and method swizzling used to implement the app-independent tracking of context information. ........ 126
6.4 Using the UIView hierarchy to generate unique view identifiers. .................. 127
6.5 Explicit capturing approach setting to auto-populate issue trackers with structured bug reports including context items mined from user tweets. .................................................. 129
6.6 Example response of tweets included in Twitter timeline. .................. 131
6.7 Information included in a tweet. ................................. 132
6.8 Template code of the Twitter package for Node.js. .................. 133
6.9 Chatbot functionality to respond to a tweet in order to clarify missing context information. .................. 133
6.10 Issue reported on GitHub including basic context information. .................. 134
6.11 Creating issues using octonode and the GitHub API. .................. 135

7.1 Main phases of the context isolation approach. .................. 142
7.2 Screenshot of the validation request asking additional users if they are being affected by a reported bug. .................. 144
7.3 (a) Unfiltered user interaction trace (on app view level), including all visited app views, by confirming and non-confirming users, and their possible transitions; (b) Filtered user interaction trace including all visited app views by confirming users only. .................. 147
List of Figures

7.4 Filtered user interaction trace on event level. Trace \( v_1 \) — \( v_2 \) — \( v_6 \) — \( v_8 \) includes the mandatory event \( e_2 \) on app view \( v_6 \) (highlighted in red). .................................................. 147

7.5 Overview of the framework architecture (adapted from [182]). . . . 148

7.6 Visualization of isolated non-crashing bug report. ............... 151

8.1 Research method separated into the data collection, data preparation, and data analysis phase. ................................. 157

8.2 Sentiment per star rating. .................................................. 162

8.3 Sentiment per content category. ......................................... 163

8.4 Four recurring emotional patterns in user feedback. ............... 164

9.1 Overview of the evaluation method to evaluate the impact of our approaches on the quality, i.e., authenticity and actionability, of user feedback. .................................................. 177

9.2 Steps of the experiment to reproduce bugs that are timely dependent or independent from the participants. .................. 181

9.3 Classification scores of machine learning algorithms on imbalanced datasets, including 90\% to 0.1\% fake reviews, plotted on a logarithmic scale. .................................................. 183

9.4 Classification scores of appropriate machine learning algorithms for datasets with class imbalance, i.e., including 90\% to 0.1\% fake reviews, plotted on a logarithmic scale. .................. 185

10.1 Centralized and decentralized reviewing approach. ............... 199

10.2 Backend/app-side transaction creation. .............................. 200

10.3 Overview of data storage. ................................................. 202

10.4 Overview of decentralized app (dApp). .............................. 205

11.1 Integration of thesis contributions into continuous software evolution (DevOps) phases. ......................................... 209

B.1 Questions of the interview with professional iOS developers (1/3). 222

B.2 Questions of the interview with professional iOS developers (2/3). 223

B.3 Questions of the interview with professional iOS developers (3/3). 224
## List of Tables

2.1 Example of execution context captured from the Spotify iOS app. 19  
2.2 Overview of captured events, as part of the interaction context. 20  
2.3 Example of automation tools used in DevOps. 25  
2.4 Selected research on classifying requirements-related information in explicitly provided user feedback. 35  
2.5 Overview of sentiment analysis tools. 38  
2.6 Selected research applying sentiment analysis tools to explicit user feedback. 39  
2.7 Selected research on release engineering based on information extracted from explicit user feedback. 40  
3.1 Offers and their prices (in US$) of paid review providers. 49  
3.2 Summary of disguised questionnaire showing offers of paid review providers. 51  
3.3 Review characteristics extracted from review exchange portal policies. 52  
4.1 Key figures of the crawled and cleaned datasets. 64  
4.2 Key figures of the truthset. 67  
4.3 Performance of the approach compared to the truthset (T/FP = True/False Positive, F/TN = False/True Negative). 79  
4.4 Overview of context items extracted from the cleaned dataset. 83  
4.5 Effort invested by support teams until context items in conversations are present. 86  
5.1 Overview of collected fake reviews and apps requesting fake reviews. 96  
5.2 Overview of the official and the fake reviews datasets. 98  
5.3 Category ranking, number of apps, and percentage of reviews per category within the fake reviews and official reviews dataset. 100  
5.4 Features selected for the classification of fake reviews. 109  
5.5 Classifiers’ scores to detect fake reviews on balanced dataset. 111
## List of Tables

6.1  Instrumented UIKit classes and methods to capture context data. 125

7.1  Example of isolated execution context reported by three confirming and two non-confirming app users. . . . . . . . . . 145

8.1  User reviews by categories (N = 7,396,551). . . . . . . . . . 159
8.2  Dispersion of sentiments across content categories. . . . . . . 163

9.1  Participants of the bug reproduction experiment and interview. . 178
9.2  iOS apps used in the bug reproduction experiment. . . . . . . 179
9.3  App reviews shown to developers in the experiment. . . . . . 180
9.4  Confusion matrix of a two-class problem . . . . . . . . . . . . 183
9.5  Classification scores on imbalanced datasets with skews of 90% to 0.1% fake reviews (DT: DecisionTreeClassifier, MLP: MLPClassifier, RF: RandomForestClassifier). . . . . . . . 186
9.6  Quantitative experiment results of bug reproduction sessions using both app review and context data or using the app review only as bug report. . . . . . . . . . . . . . . . . . . . . . 189
9.7  Results of simulations with closed-source apps. . . . . . . . . 190

10.1 Optimized trade-offs and their resulting configurations of design alternatives (A1 refers to alternative 1). . . . . . . . . . 203
List of Publications

In the following my peer-reviewed publications, partly verbatim, included in this thesis:


Additional, my co-authored and peer-reviewed publications not included in this thesis:

List of Tables
Bibliography


Bibliography


234


243
Bibliography


Bibliography


Bibliography


Bibliography


