2018-01-15

Analytical Release Management for Mobile Apps

Nayebi, Maleknaz

UNIVERSITY OF CALGARY

Analytical Release Management for Mobile Apps

by

Maleknaz Nayebi

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN ELECTRICAL AND COMPUTER ENGINEERING

CALGARY, ALBERTA

JANUARY, 2018

Abstract

Developing software products and maintaining software versions for adding or modifying functionality and quality to software is affected by several factors that have been traditionally analyzed under the terms “when to release” and “what to release”.

Along the emergence of mobile apps, the release practices are changing. App stores are markets for many small sized software products which provide an open platform for users to express their opinions about using apps. The rise in popularity of mobile devices has led to a parallel growth in the size of the app store market, intriguing several research studies and commercial platforms on mining app stores. Large numbers of similar software products in a market make app design highly competitive.

On the other side, app store reviews are currently the primary tool being used to analyze different aspects of app development and evolution. However, app users’ feedback does not only occur on the app store. In fact, despite the mass quantity of posts that are made daily on social media, the importance and value that these discussions provide remain largely unused.

While release management of software products has been long the subject of studies, mobile apps bring new opportunities and threats to the release practices. This thesis is a series of eight papers contributing to:

1. Understanding the opportunities and threats for release management of mobile apps.

2. Analyzing evolution of mobile apps over different releases.

3. Providing new formulation to integrate the opportunities of app stores into planning models.

4. Providing decision support for release management of mobile applications.

Acknowledgements

I never crossed days of my calendar or counting hours to finish my PhD and this is all because of the excellent mentors, supporters, and friends I had. My family and my supervisory committee are my twin pillars that without them I could not stand. My supervisors inspired me and shaped my life impermeably and forever.

My ultimate inspiration comes from my best friend, my supervisor Guenther Ruhe. Guenther never gave me any idea that I couldn't do whatever I wanted to do or be whomever I wanted to be. He filled my PhD journey with joy, encouragement, collaborations, and books unflagging in his efforts to give me role models from Jane Austen to Marry Shaw to Angela Merkel. As he guided me through these incredible four years I don't know if he ever realized that the person I most admired was himself. I am proud to be his student. Thank you Guenther, for everything.

Thomas Zimmermann was a kind, decent and unfailingly brilliant mentor for me and the excitement of Tim Menzies was always encouraging. Thanks to both of you for being fantastic role models and well-hearted supporters. I would like to thank my PhD examiners Dr. Gail Murphy, Dr. Rei Safavi Naini, and Dr. Laleh Behjat. Thanks for helping me on making this thesis better.

I would like to thank Dr. Andreas Zeller for the internship opportunity and all the guidance. I also like to thank Dr. Frank Maurer, Dr. Dietmar Pfahl, Dr. Bram Adams, Dr. Walid Maalej, Dr. Christof Ebert, Dr. Krzysztof Wnuk, and Dr. Daniel Mendez Fernandez for their advice and strong support on our joint publications. I am grateful to Dr. Mark Harman, Dr. Ye Yang, Dr. Federicca Sarro, Dr. Abram Hindle, Dr. Mahmood Mousavi, Dr. Allessandra Gorla, and Dr. Emad Shihab for their advice and support along the road.

I would also like to thank my students Navid Pourmomtaz, Steven Zhang, Samarth Sinha, Kurtis Jantzen, Ada Lee, Rachel Quapp, Shane Sims, Homayoon Farrahi, Ron Ittyipe, Liam Dicke, Paul Chen, and Henry Cho. Thanks for your encouragement, great work, and all the memories. I am thankful to all the SEDS lab members as well as Timo Johan, Mahshid Marbouti, and Konstantin Kuznetsov. Also, I owe a lot to all the anonymous reviewers of my papers, all the people who listened to my ideas and talks and motivated me or criticized my work. Both are equally valuable to me.

Especial thanks to the industry partners, their openness in sharing experience and opinions was truly valuable to me. I would like to thank ATB financial, BrightSquid, and CodeExcellence and their affiliated employees and in particular Dr. Ian Hargreaves and Mr. Chris Carlson.

Table of Contents

List of Figures and Illustrations

List of Tables

Chapter 1

Introduction

Adaptive software development is designed for facilitating the application of changes and requires a high degree of user involvement [206]. As part of any incremental and iterative development, release management (considering both release planning and engineering) is important. Release planning is the process of assigning features to upcoming releases (or iterations) such that the overall product evolution is optimized [173]. Release engineering is a group of activities that are managing the pipeline of transforming source code into an integrated, packaged, and tested products that are ready for release [3]. Adaptive software development as an iterative approach benefits from release management for accommodating changes within development iterations. To have more influential adaptation to changes, software owners try to minimize the distance between customer feedback and development processes. The platform-mediated networks that software products are released on (such as app stores) enables fast feedback from customers [63]. The conjunction of open access to customers' feedback as well as the similar software products that are potentially competing, opens up the opportunity to look beyond the fence of organization boundaries for software development.

The increasing desire for real-time decision making and continues delivery create new challenges for the release methodologies. The border between decision making and applying decisions on the software product is getting narrower for dynamic and user intensive environments. Several approaches have been proposed to apply user feedback in product development by gathering information from customers in different phases of the software development process [173]. Designing

feedback loops through the analysis of software usage, observing user attitudes and directly asking users about their experience have been discussed by many researchers [151]. In particular, for mobile apps, several studies focus on the usefulness and impact of customers' feedback [155].

Traditionally, release planning has been discussed as an iterative and evolutionary approach within which a set of features is proposed to minimize the degree of conflicts between stakeholders' priorities [79]. While optimization of release plans proved to be useful [79, 229, 259], these approaches are not flexible enough to accommodate the degree of changes happening in user and competitor intensive environments such as app stores and for real-time decision making [152].

The emergence of the app stores and pervasive use of software products in a daily life of people made general public the primary user of several software applications which in turn makes software engineers responsible for developing products that are desirable and helpful for them. In this designation, I particularly address problems related to release management of mobile apps. The initial evaluation of mobile app release practices and development problems [109, 174] revealed differences between release practices for mobile apps and other software products. In particular app developers stated their uncertainties about the impact of their release decisions, on the customers and market competition. By understanding the users, developers, the platform of distribution, and release process of mobile apps, I focus on providing decision support methods for release management of mobile applications.

1.1 Thesis objectives

In this thesis, which is a collection of papers (manuscript-based thesis), I targeted three research objectives in the context of mobile applications:

Objective 1: Mining information for release management.

Objective 2: Optimization to support release decisions.

Objective 3: Decision support for release functionality and marketability.

Objective 2 and Objective 3 are both aimed to facilitate decision making for app developers. This implies some overlap between the two objectives. To achieve the above objectives, I applied a series of various forms of analytics ranging from descriptive to predictive [27] as demonstrated in Figure 1.1:

- *Descriptive analytics* emphasizes on answering the questions regarding, "What has happened?". Descriptive analytics can be classified as the simplest analytical techniques since it uses the data to simply summarize what happened.

- *Diagnostic analytics* goes beyond descriptive analytics by also explaining why things happened. In this sense, this technique provides insights about the past and answers, "Why did it happen?".

- *Predictive analytics* analyzes current or historical data to provide predictions, such as a probable future outcome or the likelihood of a situation occurring. This type of analysis provides insight towards, "What will happen?".

- *Prescriptive analytics* is also looking into the future, but the emphasis is more on providing insight in the form of concrete actions by recommending one or more possible courses of

Table 1.1: List of related papers to this thesis.

ID	Ref.	Objective	Title	Status
P1	[174]	Motivation	Nayebi, M., Adams, B., Ruhe, G., Release Practices for Mobile Apps–What do Users and Developers Think?. SANER 2016 (Vol. 1, pp. 552-562). IEEE.	Published
P2	[176]	Objective 1	Nayebi, M., Cho, H., Ruhe, G. App Store Mining Is Not Enough for App Improvement, Empirical Software Engineering Journal (EMSE).	Under revision
P3	[189]	Objective 1	Nayebi M., Ruhe, G., Zimmermann, T., Mining Treatment-Outcome Constructs from Sequential Software Data, submitted to TSE, 2017	Submitted
P4	[188]	Objective 2	Nayebi, M., Ruhe, G., Optimized Design of Supper Apps, International Conference on requirements Engineering (RE), 2017	Published
P5	[187]	Objective 2	Nayebi, M., Ruhe, G. Asymmetric Release Planning, Compromising Satisfaction against Dissatisfaction, Transaction of Software Engineering (TSE)	Submitted
P6	[179]	Objective 3	Nayebi, M., Farrahi, H., Ruhe, G., Which Version Should be Released to App Store?, Empirical Software Engineering and Measurement (ESEM) 2017	Published
P7	[181]	Objective 3	Nayebi, M., Quapp, R., Ruhe, G., Marbouti, M., Maurer, F., Crowdsourced exploration of mobile app features, ICSE 2017, IEEE Press.	Published
P8	[227]	Future work	Ruhe, G., Nayebi, M., Ebert, C., The Vision: Requirements Engineering in Society, International Conference on requirements Engineering (RE), 2017	Published

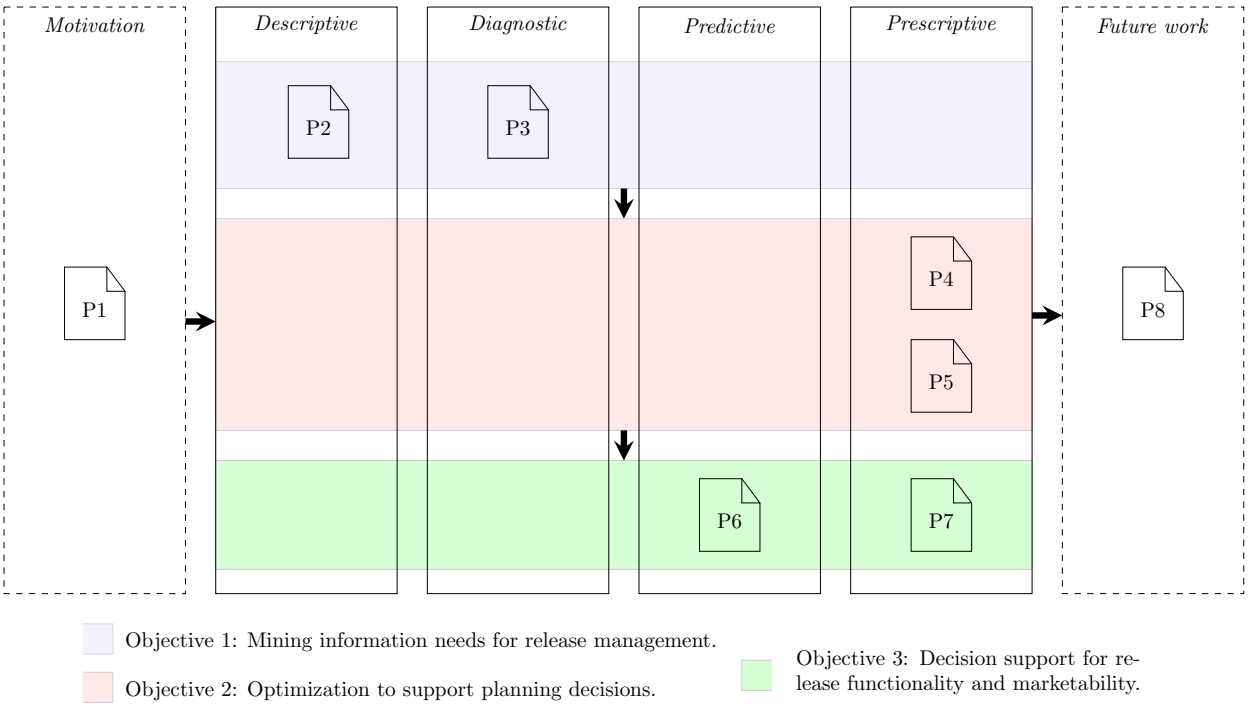


Figure 1.1: Relation between publications, type of analytics and thesis objectives.

Table 1.2: List of developed tools.

ID	Description	Papers
T1	**Gitlyzer:** Structuring and outputting log of Github projects structured based on releases. The data of commits, file authors, sequence of authors, extend and nature of changes, files evolution, issues, cycles, pull requests all structured based on releases.	P7, P8, P9, P10, P11
T2	**GWM Tool:** This tool supports the three steps of GWM (encoding, abstraction, and synthesis) and provides thechniques to compare the results with other methods.	P9, P10
T3	**MAPFEAT Tool:** This tool supports the process of mapping app features into tweets. Tool get the input of a set of tweets about an event and it mines, crowd evaluates, and priorities features for a mobile app that can supports management of that event.	P14
T4	**Asymmetric planner (SDO):** The tool get the stakeholders, their weight, and the features as input. Stakeholders interactively answer Kano questionnaire. The tool calculates satisfaction and dissatisfaction per feature using Kano model and prioritize features by solving this as integer linear programming (ILP) problem.	P13

action, thus answering, “What should be done?”.

My thesis is based on a subset of papers I published over the period of my PhD as detailed in Table 1.1. I demonstrated the relation between eight papers and their relation to the three research objectives in Figure 1.1. There are also seventeen publications that I did not include in my thesis. They are listed in Table 1.3. I developed a set of tools for supporting my studies as detailed in

Table 1.3: List of PhD publications not included in this thesis.

ID	Ref.	Title	Status
E1	[190]	M. Nayebi, K. Kuznetsov, A. Zeller, G. Ruhe, Why, When, and How Developers Delete Functionality, ICSE 2018	Submitted
E2	[180]	Nayebi, M., Farrahi, H., Ruhe, G., Cho, H., App store mining is not enough. ICSE 2017 (pp. 152-154). IEEE Press.	Published
E3	[190]	M. Nayebi, S.Kabeer, G. Ruhe, C. Carlson, F. Chew, Hybrid labels are the new measure, IEEE software, 2017	Accepted
E4	[178]	Nayebi, M., Farrahi, H., Ruhe, G., Analysis of marketed versus not-marketed mobile app releases. RELENG 2016 (pp. 1-4). ACM.	Published
E5	[177]	Nayebi, M., Farrahi, H., Lee, A., Cho, H., Ruhe, G., More insight from being more focused: analysis of clustered market apps. WAMA 2016 (pp. 30-36). ACM.	Published
E6	[226]	Ruhe, G., Nayebi, M., What Counts is Decisions, Not Numbers – Towards an Analytics Design Sheet in Perspectives on Data Science for Software Engineering	Published
E7	[173]	Nayebi, M., Ruhe, G. (2015). Analytical product release planning. The Art and Science of Analyzing Software Data, 550-580.	Published
E8	[175]	Nayebi, M., Adams, B., Ruhe, G., From Perception to Evidence: Release Patterns in Mobile App stores, EMSE	In preparation
E9	[173]	Nayebi, M., Ruhe, G., Mota, R., Moufti, M., Analytics for Software Project Management–Where are We and Where do We Go?, ASEW, 2015	Published
E10	[191]	M. Nayebi, K. Wnuk, Summary of the 1st international workshop on Open Innovation in Software Engineering (OISE 2015), ICSSP 2015	Published
E11	[184]	M. Nayebi, G. Ruhe, Analytical Open Innovation for Value-Optimized Service Portfolio Planning, ICSOB 2013	Published
E12	[185]	M. Nayebi, G. Ruhe, An open innovation approach in support of product release decisions, CHASE 2013	Published
E13	[142]	Maalej, W., Nayebi, M., Johann, T., Ruhe, G. (2016). Toward data-driven requirements engineering. IEEE Software, 33(1), 48-54.	Published
E14	[7]	D.AlAlam, M. Nayebi, D. Pfahl, G. Ruhe, A Two-staged Survey on Release Readiness, EASE'17, 2017	Published
E15	[91]	A. Hemmati, C. Saunders, C. Carlson, G. Ruhe, M. Nayebi, Analysis of Software Service Usage in Health care Communication Services, QRS 2017	Published
E16	[66]	D. Méndez Fernández, S. Wagner, et al., Naming the pain in requirements engineering: contemporary problems, causes, and effects in practice, EMSE journal	Published
E17	[183]	M. Nayebi, G. Ruhe, An open innovation approach towards feature elicitation for product release planning, CSER 2013	Published

Table 1.2.

It is worth to note that the main content of my PhD thesis included 17 papers. However, including all these would exceed the page limit of a thesis. Consequently, I relied on eight main papers.

In the following sections, I provide an overview on the purpose and main content of the papers selected under each of the three objectives.

Objective 1: Mining information for release management

Release planning is a well-established problem in software engineering [228]. Release planning is a decision-centric problem that needs comprehensive information and knowledge. A release is a new or upgraded version of an evolving product that is characterized by a collection of new or modified features. Typical decisions about release planning are concerned with the functionality (what to release?), time (when to release?), and quality (how good should the release be?) of an upcoming release. Continuous delivery of releases [152] is an attempt to simplify release planning by releasing more frequently and reducing the scope of releases. While substantial research has been done on release planning, on rapid releases and their impact on web and desktop applications [113, 247], no such study has ever considered release planning for mobile apps. While it is known that big organizations such as Mozilla or Facebook have dedicated release strategies for their mobile apps, the difference of releases management for mobile apps versus other software products is still at an early stage.

CHALLENGE: *The release management and decision making process for mobile apps is not much known and has not been studied in research community.*

In this direction, I performed a survey study with app developers and users to understand the release strategies, release decision-making criteria, and the impact of release management on the mobile app user. By looking into the similarities and differences between state of the art release planning of mobile apps and general software products, I investigated on mining the information that is needed for planning mobile app releases.

The Stakeholders' satisfaction, values, interests, preferences, and evaluation have been mentioned in six out of the 28 release planning approaches studied by Svahnberg et al. [251]. Other than different satisfaction and value measures, the variety of stakeholder types is of importance. Svahnberg et al. [251] found that too often critical stakeholders are overlooked, and priorities are given in an ad-hoc fashion. More recently, discussion forums and user groups have been used to get access to stakeholder opinions [45]. Many researchers mined mobile app reviews to infer user

values and plan for app evaluation.

Martin et al. [155] provided an overview on the state of the art studies in the app store. Within this study they analyzed 45 studies on app store reviews performed between 2012 and 2015. Several studies focused on the techniques for extracting app features from reviews [84, 105], while other studies focused on developing models and tools for automated classification of reviews into predefined categories [42, 77, 80, 141]. Furthermore, mining app reviews have been used as a basis for decision support. Palomba et al. [197] analyzed 100 Android apps and showed that addressing user reviews increases the chance of apps' success. Villarroel et al. [263] introduced CLAP for categorizing reviews into bug reports, clustering related reviews, and prioritizing reviews. More recently, Ciurumelea et al. [44] defined a two-level taxonomy based on 1,556 reviews with the aim to assist mobile app developers with planning maintenance and evolution activities. Di Sorbo et al. [54] defined a taxonomy of reviews and provided a tool for categorizing reviews.

However, the completeness of the information for making a decision about app evolution is not clear. Release planning models by now, have prioritized the features being available in a feature pool and used a selected set of known stakeholders [228]. App stores are one source of information where we can mine the information that is needed for release management of mobile apps.

CHALLENGE: *Release planning approaches consider requirements as a known set of features and release planning processes are using rather closed and inflexible feature prioritization and planning processes. With the paradigm shift of the people involved in the decision-making, planning processes can no more rely on the assumption that the user priorities have been given. Unbiased and comprehensive understanding of users' needs, and perceived value of software is a pre-requisite for project planning.*

By relying on the state of the art studies on the impact and value of app reviews, and motivated by what we foresaw for the involvement of crowd in future of software engineering [142] ([E13] in Table 1.1), I looked beyond the fence of app stores for gathering users data. I investigated the data of social media and compared them with users reviews in mobile app stores ([P2] in Table 1.1). In these studies, I showed that while app store is a rich source of user information but the app

store mining is not enough. Complementary information, however mobile app users can be found on Twitter.

On the other side, mining the best practices and diagnosing their impact of them on the products is not trivial. In particular, in the domain of mobile apps, the impact of developers' decisions are unexplored. Developing software products and maintaining software versions is affected by several factors that have been traditionally analyzed under the terms "when to release" and "what to release". The initial evaluations showed that release cycles have rarely been analyzed as a factor that may affect the software performance and customer values. Analyzing the time for releasing a software version has been studied as the "when to release" problem traditionally. The "when to release" problem has been studied in literature and addressed by release readiness more recently [243]. The fixed release cycles for adaptive software development [206] or traditional software development processes are introduced and investigated. In the context of mobile app release planning, the time line of a project was analyzed; although the best practices for release management is unknown. Mcilroy et al. [158] found that releases to the end user do not necessarily perceive positively for mobile app. We observed that the sequence of events happening over a period of time is a game changer itself ([P3] in Table 1.1).

CHALLENGE: *The sequence and variation of release cycle times have never been considered as part of any of the existing release planning approaches. Practitioners [3] mentioned the importance of time frame analysis. Also, the effect of time-based release management in open source projects was discussed via interviews. It was concluded that lack of time frame consideration might affect development processes in open source software systems.*

Papers and detailed research questions related to this objective are presented in Part I (Chapter 3 and Chapter 4).

Objective 2: Optimization to support release decisions

Release planning approaches for answering "what" to release and "when" to release problems are modeled by defining planning objectives and constraint, and in consideration of feature values, feature dependencies as well as stakeholder opinions and priorities stated by known stakeholders. It has been assumed that a decision on priority of requirements have been made upfront and assignment of requirements into different releases is conducted while planning for releases. During the feature prioritization process, stakeholder priorities are gathered, and the optimization process is applied. The existence of multiple mobile applications in the app store along with openly accessible features, functionalities, and users' feedback provides a unique opportunity to take the best of the competitors' practices. Proposing learning models in consideration of competitors status on the market is desirable by developers [174].

CHALLENGE: *Prioritizing user requests for mobile apps have been studied [43, 158, 263]. However, none of these studies provide a release planning method and formal formulation for mobile apps considering the established state of the art release planning or prioritization methods [2, 228, 251].*

I investigated the release practices for mobile apps considering users and developers perspectives ([P1] 1.1). Knowing that users tend to write a review for an app either they are very satisfied or dissatisfied with [88, 114], I focused on satisfaction and dissatisfaction of users [P5]. It has been proved that satisfaction and dissatisfaction are two sides of the same coin and their co-existence is undeniable. Having a huge set of users involved in the decision process, it is not enough to only focus on increasing satisfaction and positive sentiments about the product, one should also pay attention to the dissatisfaction and negative sentiments. Kano model [110] showed that the relation between satisfaction and dissatisfaction is assymetric. Knowing the importance of users' voice for mobile app success [114, 197], I investigated a novel method of release planning for fine-grained consideration of customer value using the Kano model [110, 140]. The Asymmetric release planning method searches for release plans that balance satisfaction with dissatisfaction ([P5] Table

1.1).

In addition, the existence of similar apps that are potentially competing is a threat that can be turned into an opportunity. Considering the functionality of similar applications and users feedback on these functionality, I provided a formulation for optimized super app functionality ([P4] in Table 1.1). By looking into similar apps and considering app store as a pool of features, I proposed a bi-criterion optimization approach that balances app feature value with the cohesiveness of features ([P4] in Table 1.1). Information on these attributes comes from reasoning on feature composition of the (existing) similar apps. I proposed a new model using bi-criterion integer programming. I made suggestions for optimized super app functionality that are based on two key aspects: First, the estimated value of features, and second, the cohesiveness between newly added features and cohesiveness between existing and the features to be added.

Papers and detailed research questions related to this objective are presented in Part II (Chapter 5 and Chapter 6).

Objective 3: Decision support for release functionality and marketability

Market and user characteristics of mobile apps make their release management different from proprietary software products and web services. Developers release a version of the app on a platform which was also known as the app store or marketplace. Many open source mobile app releases have never been released on the app store ([P7] and [E4]). These releases are significantly different from the marketed releases. Users can download the app from the platform instead of getting it directly from the software owner as it is done traditionally. Figure 1.2 illustrates the fundamental difference in offering software app products. While an increasing number of software products are designed and developed for platform-mediated environments, I only studied releases of (open source) mobile apps.

A *marketed release* is a release that was introduced in a Git repository as well as in the app store. A *not marketed release* is a release that was only introduced in the Git repository of the app [178]. "Marketability" refers to the decision of shipping a new release into the market or not.

CHALLENGE: *The difference between release process and marketability decision has not been discussed. Our survey of release managers [179] showed that marketability decisions are of significant importance for app developers. The adopted planning process for mobile apps needs additional decision support.*

I investigated the release process of mobile apps and the difference with desktop and web applications and found that marketability is a considerable problem that can be facilitated using decision support systems. By characterizing marketed and not marketed releases, we provide recommendations to transition a non-marketable release into a successfully marketable one. We performed analogical reasoning using the experience of the same app and looking across apps ([P6] in Table 1.1).

Along with degree and nature of changes and the buginess of the release ([E4]), sequence and duration of release cycle times have significant impact on the apps' success ([P3], [E8]). Considering these factors, I predict marketable and non-marketable releases of mobile apps using analogy based reasoning ([P6] in Table 1.1).

Deciding on the functionality of the software product is not trivial and it has been known as a human intensive problem [142]. In [P2] I showed that app store mining is not enough for deciding

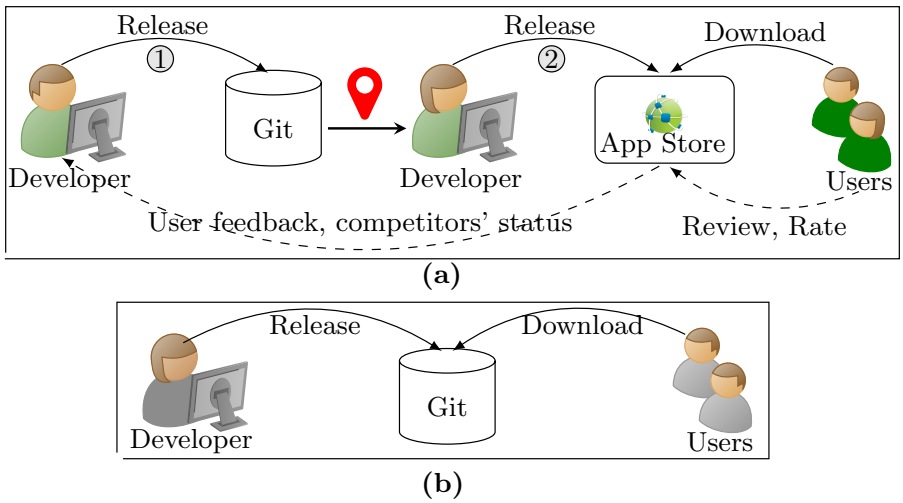


Figure 1.2: Distribution of a product release to the end user for (a) app versus (b) non-app open source software products [179].

on the app improvement. Motivated by these results, I looked into communication of general public about an event in social networks and investigated if and how the users' needs could be mapped into software technologies in the format of a mobile application.

CHALLENGE: *Analysis of users' need as stated in app store reviews have been the only source for requirements elicitation and decide functionality of mobile apps [155]. The pervasiveness of mobile applications can come to help people in daily life in case app functionalities match their expectations.*

I proposed MAPFEAT ([P7] in Table 1.1) for mining and prioritizing mobile app features that help people with a special event (such as a wildfire emergency). MAPFEAT is an automated method designed for the purpose of mapping general public needs that were communicated via Twitter into mobile app features. By applying MAPFEAT into the tweets regarding the Fort McMurray wildfire of 2016, we found 139 new features with 85.7% of them considered useful by the general public. For specific case of wildfire mobile apps, I showed that MAPFEAT is able to find a significant number of additional features which were not in the currently existing wildfire apps.

I presented related papers and detailed research questions related to Objective 3 details are followed in Part IV (Chapter 7 and Chapter 8).

1.2 Research methodology

To achieve the stated research objectives, I used both empirical and non-empirical methods as well as quantitative and qualitative methods. In what follows, I provide an overview of the research methodology across different papers under this thesis.

Empirical methods

Qualitative survey: A survey is a comprehensive system for collecting and describing behavior [201]. Surveys can be supervised (with one to one connection between researcher and respondents) or unsupervised. The selection of the survey type and the design of instrument depends on the

study's objective [201]. Qualitative research is concerned with collecting information about the opinions of the subjects. Surveys can reach a huge population of respondents, and they provide easy to analyze data as the questions are mainly designed based on defined hypotheses. One should be careful about generalizing the results of the survey [271]. I summarize the qualitative surveys I used within this thesis:

1. To understand the release strategies of mobile apps and evaluate the impact of release strategies on users ([P1] in Table 1.1). This research included 36 app developers and 654 app users.

2. To evaluate the importance of marketability problem for mobile apps I surveyed 22 release engineers. I also evaluated the results of marketability prediction with seven app developers in [P6] of Table 1.1.

3. To evaluate the value and desirability of asymmetric release plans I designed a survey for human experts to prioritize release plans using Analytical Hierarchy Process (AHP) [234] and evaluated the usefulness of proposed release planning model in [P5]. I surveyed six app developers and 14 students to show the pros and cons of asymmetric plans ([P5] in Table 1.1)

4. I performed crowdsourced surveys with two purposes. First, to evaluate and calibrate the results of machine learning in [P2] and [P7] of Table 1.1. Second, to evaluate the value and importance of suggested features or releases in [P4] and [P7] in Table 1.1.

In the above surveys, I performed convenient sampling [117].

Case study: A case study investigates a single entity in its real-life context. Case studies are used to research projects and activities. Through the process, the data is gathered or retrieved for a well-defined purpose of the study. A case study is aimed to track one or a set of attributes [271]. Case studies do not require a strict boundary between the object of the study and its environment. Different types of a case study are conducted in software engineering. Exploratory (understanding

the phenomenon and generating hypothesis for research), descriptive (describing the status of a phenomenon), explanatory (explaining a phenomenon in the form of causal relation), and improving (improving a certain aspect of a phenomenon) [232]. I performed the following case studies:

1. To understand the completeness of information from the app store, I compared the app reviews with the tweets about apps. Starting with 30,793 apps for an exploratory case study, I switched into the in-depth analysis of 70 apps for a descriptive case study in [P2].

2. The exploratory case study in [E4] with open source mobile apps hosted on F-Droid showed the difference between marketed and not marketed releases. Following on this, in [P6] I performed analogical reasoning to support the decision for releasing a version to the app store. I evaluated the results over 58 release of 39 apps.

3. In [P3] I ran three different case studies to mine code ownership patterns, file authorship patterns, and release cycle time patterns. With this I demonstrated applicability of proposed Gandhi-Washington Method (GWM) in different software engineering problems.

4. In [P5] we ran an improving case study [232] with a real world mobile application to demonstrate how the integration of asymmetric release planning model with a bi-criterion optimization formulation increase the satisfaction and decrease the dissatisfaction of users.

5. In [P7] I ran an explanatory and improving case study about the Fort McMurray wildfire of 2016. Analyzing 69,680 tweets about Fort McMurray wildfire, I used MAPFEAT to elicit and prioritize mobile app features based on tweets. The results showed that we could improve the design of apps functionality with the proposed method.

Non-empirical methods

[P3], [P4], [P5], and [P7] have significant methodological novelties and contributions. I categorize the non-empirical strategies following the classification provided by Glass et al. [76] for research methods.

Formulative process, method, algorithm: I developed methods and applied algorithmic and mathematical techniques as part of my thesis:

1. Many investigations in empirical software engineering look at sequences of data resulting from development or management processes. I proposed an analytical approach called the Gandhi-Washington Method (GWM) to investigate the impact of recurring events in software projects ([P3] in Table 1.1).

2. App stores are markets for many small sized software products which provide an open platform for users to give feedback on using apps. Moreover, the functionality and status of similar software products can be retrieved. While this is a competitive risk, it is at the same time an opportunity. In [P4] I propose a new model using bi-criterion integer programming. We make suggestions for optimized super app functionality that are based on two key aspects: (i) the estimated value of features, and (ii) the cohesiveness between newly added features and cohesiveness between existing and the features to be added.

3. Maximizing satisfaction from offering features as a part of the upcoming release(s) is different from minimizing dissatisfaction gained from not offering features. This asymmetric behavior of the value of features has been approached in the past by the Kano analysis method [110]. However, it has never been utilized for release planning. I accommodate this asymmetry and study *Asymmetric Release Planning* (ARP). I have formulated and solved ARP as a bi-criteria optimization problem. In its essence, it is the search for optimized trade-offs between maximum stakeholder satisfaction and minimum dissatisfaction. While this is a general release planning method, I dedicatedly applied and evaluated it for a mobile application.

4. Using the synergy between information in social media and mobile app stores [180], I investigated a method to suggest features that are useful for apps. My proposed method called MAPFEAT, combines various machine learning techniques to analyze tweets in conjunction with crowdsourcing and guides an extensive search in app stores to find currently missing

features in apps. MAPFEAT is evaluated by a real-world case study of the Fort McMurray wildfire emergency of 2016.

Data analysis: Different methods exist for analyzing qualitative and quantitative data, and I used both since the data of this thesis is both qualitative and quantitative [271].

Statistical analysis: I used descriptive statistics and inferential statistics in this thesis. Descriptive statistics such as box-plots, standard deviation, variance, and mean values are used to help in interpreting the data. Descriptive statistics give the feeling about data and its distribution. This results in forming a hypothesis and opens up the way for inferential statistics [121]. With inferential statistics, one drives conclusions from data (mainly) by testing a hypothesis [121], which includes both parametric and non-parametric tests. All of the papers included in this thesis other than [P4] and [P8] include statistical analysis.

Machine learning: I used machine learning based classification, clustering, and topic modeling. A considerable amount of information about mobile apps is quantitative and in form of text whether it is app description, user reviews, or user feedback and experience report in the social media. Building on the existing literature [83, 141], I used Support Vector Machine (SVM) and Naive Bayes to classify reviews and tweets in [P2] and [P7]. I used Random Forrest, SVM, and Decision Trees to classify marketable and non-marketable releases in [P6].

Analogical reasoning has been successfully used in different fields of software engineering [244] and beyond [219]. Reuse of software knowledge is known to be inherently difficult because, in a strict sense, each software project is unique. However, reuse of experience is essential, and it is intuitive to learn from experience. To assist decision making about release marketability, in [P11] I used analogical reasoning for transferring a not marketable release into a successfully release using analogical reasoning [173].

To compare the content of app reviews with the content of app related tweets I used topic modeling. Topic modeling is the process of finding the topics of a large and possibly unorganized collection of documents. I used Latent Dirichlet Allocation (LDA) [30] for this purpose. LDA [30] is an established method used to find topics in a set of natural language text documents. Topics

are created when words that tend to appear together frequently are found in the documents of the corpus. LDA assumes that there is a fixed number of topics. It assigns each document a probability distribution over topics. By looking into the words with heaviest weights, we can assign descriptive names to the probabilities assigned to words. I used topic modeling in [P2] and [P14].

Crowdsourced data analysis: Combining machine learning and human intelligence is an emerging trend [164]. I used crowdsourcing to evaluate and adjust the machine learning results to verify the results of [P2] and [P7]. While I compared and setup the machine learning techniques carefully for the best possible performance, I went one step further and evaluate and adjust the results by using crowd intelligence.

1.3 Data collection methods

Self-administrated questionnaire

The self-administrated questionnaire contains questions that should directly be answered by the subjects of the study [120]. On the positive side, in this type of questionnaires, more questions could be asked. However, lack of clear question or description endangers the correctness of the results. In these surveys a comprehensive instruction should be provided along with the questions. Questionnaires may include closed questions (subjects should select answer from a list) and open questions (subjects can openly frame their response)

I used self administrated questionnaire in [P1], [P6], and [P5] of Table 1.1.

Mining archival data

Archival data refers to documents that were previously gathered and measured in an organization [231]. Archival data is third-degree data that were not developed for the intention of a case study research. Researchers access this data retrospectively and use it to conduct case studies. For this type of study, a tool is a major source as it may support the collection of data for multiple studies.

I developed tools to assist data gathering in case a data source was used multiple times following the suggestion of Menzies and Zimmermann [160] (see Figure 1.3).

Data from GitHub: I gathered data from `GitHub` repositories for the purpose of multiple studies. I designed and developed the tool `Gitlyzer` as described in Table 1.2 to mine and structure `GitHub` logs based on software releases. I used `GitHub` data to conduct case studies in [P2], [P3], and [P6] of Table 1.1.

`Gitlyzer` uses an interface based on commands (like a miniature shell) just for analyzing `Git` repositories. `Gitlyzer` gets a repository name as an input and collects its Git log. By mining the log, `Gitlyzer` provides information on release name, release dates, file authors considering the life cycle of the project, file authors per release, commits messages for each release, number of commits, code churn in a release, file churn in a release, files changed per release, branches per release, issues opened per release, closed issues per release, issues open at the time of release, developers opening the issues per release, and the developers closing the issues per release.

App store data: I mainly focused on analyzing Android mobile apps as a repository for the open source Android applications. I set up an `Apache Nutch` crawler with an `Apache Solr` indexer for fast search and performed a bi-weekly crawl of 100,000 apps over eight months. Our existing process of crawling the website with `Apache Nutch` integrated with `Apache Solr` and parsing the resulting data with a custom `Java` parser, provided a way to collect data for a large number of apps every week. This data included up to 40 user reviews for each app.

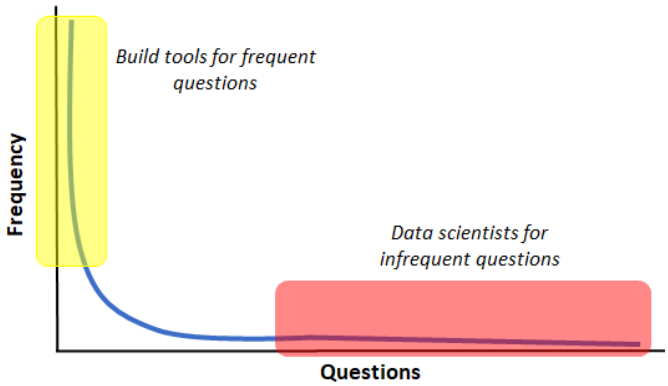


Figure 1.3: Relation between frequency of questions and tool development [160].

When a user visits the Google Play Store they can view in sliding panes that are navigated through by buttons leading to the next and previous panes (a `JavaScript` behavior). The page loads with the first 40 reviews included, but when a user clicks through to the end of the reviews, an `AJAX` (`Asynchronous JavaScript and XML`) request to the server loads more reviews so the user may continue. Therefore, in order to pull more user reviews I used `Selenium play` to trigger the `AJAX` request. The `Selenium Play` is implemented with a `Java` program.

I used the data from Google Play in [P2], [P3], [P5], [P6], and [P7] in Table 1.1. For [P14] I used the iTunes API as it was free of cost for unlimited search queries into the store. Note that, at the time of starting my research, repositories such as Androzoo [8] did not exist. Hence, I developed tools for gathering data for my research.

Data from Social Networks: I used the open source modules for collecting social media data for studies in [P2] and [P7] as of in Table 1.1. Tweets about the apps for [P2] were collected using the Twitter API using `CliPS Pattern` module [248]. Using the `CliPS` open source module, I connected to the python implementation of the Twitter API, which allowed for the mining of content, date-stamp and unique ID of tweets. I implemented various strategies to ensure that the tweets collected were complete and accurate. Using the `Pattern` module to access Twitter, I ensured that all of the tweets collected were in English.

The process is set up to gather set of reviews based on keywords over time. Also, I implemented methods for cleaning up the tweets from duplicate content, spams, and advertisement.

1.4 Vision

While there are many possible perspectives for the future of software engineering I published few possible directions in [P8] as of Table 1.1. The pervasiveness of software in the daily life of human beings is undeniable. The variety of online mobile devices increased the demand of society for the better support of software applications. Having a device that collects our personal and situational data and using social networks to report our status to the friends increases the chance for decision

Table 1.4: List of contributions and the related chapters.

ID	Contribution	Objective	Chapter
P1	Extracting six release strategies of mobile apps and their impact on product and users and comparing them with traditional release planning strategies	Motivation	Chapter 2
P2	Demonstrating app store mining is not enough for making improvement decisions and complementary information about mobile app change requests exists in social media	Objective 1	Chapter 3
P3	Introducing Gandhi-Washington Method (GWM). GWM is a general approach to analyze the relationship between sequential events in software processes or products and an outcome. GWM can be applied to a large class of software engineering decision problems related to sequences of events.	Objective 1	Chapter 4
P4	- Propose a new model using bi-criterion integer programming based on (i) the estimated value of features, and (ii) the cohesiveness between newly added features and cohesiveness between existing and the features to be added. - Demonstrating the process with an illustrative example.	Objective 2	Chapter 5
P5	- Formulation and modeling of the Asymmetric Release Planning (ARP) problem to balance customer satisfaction against dissatisfaction. - Design of a bi-criteria solution approach called *Satisfaction-Dissatisfaction Optimizer (SDO)* for ARP. SDO generates optimized trade-off plans. - Case study evaluation of SDO for a problem with 36 features on a mobile app. We performed (i) a comparative analysis of SDO with random search and eight planning heuristics and (ii) an evaluation of the results that included 20 stakeholders.	Objective 2	Chapter 6
P6	Introducing and confirming the importance of marketability decision for mobile apps which have not been investigated so far. - Compared three machine learning methods to predict release marketability. - Performing analogical reasoning using the experience of the same app and looking across apps, to transition a not marketable release into a successfully marketable one. - Evaluating results of analogical reasoning with actual changes that transitioned under question releases (internal validation over time). As well as, performing a survey of app developers for external validation.	Objective 3	Chapter 7
P7	- Introducing an automatic method (MAPFEAT) to map tweets into mobile app feature with the purpose of designing applications that meet users' need. - Case study evaluation on the Fort MacMurray wildfire of 2016.	Objective 3	Chapter 8
P8	Presenting the vision and study possible options to perform requirements engineering in socio-technical systems, but get closer to make the results happen and evaluate their actual impact.	Vision	Chapter 9

makers to provide better software. In [P8] as it is in Table 1.1, I discuss the treatments and research advancements that changed the path of requirement engineering.

1.5 Main contributions and organization of this thesis

While this thesis is based on the manuscripts, each chapter is directed toward one of the research objectives and includes several papers. Each paper has a contribution toward he body of knowledge and the objectives of this thesis. The list of contributions presented in each chapter is in Table 1.4.

Part I describes the motivation of the thesis and comprises of one paper ([P1] in Table 1.1). Part II is aligned with Objective 1 and is a combination of papers [P2] and [P3]. Part III covers Objective 2 and is a collection of two papers ([P4] and [P5]). In Part IV, I presented two papers that are directed toward Objective 3 ([P6] and [P7]). Finally, Part V presents two papers with a vision to future of software engineering ([P8]).

Part I

Motivation

Chapter 2

Release Practices for Mobile Apps - What do Users and Developers Think?

Authors: Maleknaz Nayebi; Bram Adams; and Guenther Ruhe SANER 2016

Abstract - Large software organizations such as Facebook or Netflix, who otherwise make daily or even hourly releases of their web applications using continuous delivery, have had to invest heavily into a customized release strategy for their mobile apps, because the vetting process of app stores introduces lag and uncertainty into the release process. Amidst these large, resourceful organizations, it is unknown how the average mobile app developer organizes her app's releases, even though an incorrect strategy might bring a premature app update to the market that drives away customers towards the heavy market competition. To understand the common release strategies used for mobile apps, the rationale behind them and their perceived impact on users, we performed two surveys with users and developers. We found that half of the developers have a clear strategy for their mobile app releases, since especially the more experienced developers believe that it affects user feedback. We also found that users are aware of new app updates, yet only half of the surveyed users enables automatic updating of apps. While the release date and frequency is not a decisive factor to install an app, users prefer to install apps that were updated more recently and less frequently. Our study suggests that an app's release strategy is a factor that affects the ongoing success of mobile apps.

This paper has been published in the proceeding of Software Analysis, Evolution and Re-engineering Conference (SANER 2017) by ***Maleknaz Nayebi, Bram Adams, and Guenther Ruhe*** [1].

[1]First and third authors are with SEDS lab at University of Calgary
The second author is with MCIS lab at Polytechniques of Montreal

2.1 Introduction

Release planning of software products is a decision-centric problem that requires comprehensive information and knowledge. A release is a new or upgraded version of an evolving product that is characterized by a collection of new or modified features. Typical decisions made by release planners concern questions about the functionality (What to release?), time (When to release?) and quality (How good should the release be?) of an upcoming release. Depending on the market and risk involved, answering these questions requires thorough data collection and analysis about development and bug fixing progress, user escalations and competitors' progress. The recent practice of continuous delivery [152] is a parallel attempt to simplify release planning by releasing more frequently and reducing the scope of releases.

While substantial research has been done on release planning, on rapid releases and their impact on web and desktop applications [113, 247] (there is even a dedicated workshop on the topic of release engineering [230]), no such study has ever considered release planning for mobile apps. This is surprising, because Chuck Rossi (release engineering manager at Facebook) claims that "*mobile deployments are more challenging than Web deployments because we do not own the ecosystem, so we cannot do all the things that we would normally do*" [3]. Indeed, instead of releasing their app twice a day (like their Web site), Facebook and other major companies like Netflix release their mobile apps once every two weeks.

Since there are more than one million mobile apps across the major app stores, with thousands of developers specializing in mobile apps, competition is fierce [47]. Hence, release decisions matter even more for small, mobile app companies. In contrast to organizations like Facebook or Google, who have large budgets and substantial resources to develop release plans and tools, a typical mobile app company [49] consists of just a handful of developers, with 40% of all app developers having a separate main job and 21% working only part-time on apps. These small app vendors cannot afford to make mistakes and need to react in an agile way to market opportunities. In other words, making the right release decisions is key for them, without the luxury of having

all required information or knowledge available or having ample time to process such information. Because release practices as known in traditional software development might not apply in this new context, we are motivated to investigate the release practices that app developers follow. More specifically, we want to understand the extent to which release management of apps is based on developers' intuition compared to a formal rationale. In this paper, we refer to the thought processes and decisions going into release planning as release strategy. These also include the timing aspects of a release, such as decisions on the release date, the duration of a release cycle, andor frequency of releases.

In particular, we performed one survey with mobile app developers and one survey with app users to get a better understanding of the perceived value and impact of the different release practices in use. We address the following research questions from a developer (RQ 1-3) and a user (RQ 4-6) point of view:

RQ1: What strategies do developers use to release apps?

About half of the app developers has an explicit release strategy. Developers with a strategy tend to choose the strategy in the initial releases, then stick with it throughout the app's life.

RQ2: What is the perceived impact of time-based release strategies on app development?

The majority of developers are willing to bend their time-based strategies to accommodate users' feedback. Developers believe that apps with frequent updates likely deliver less

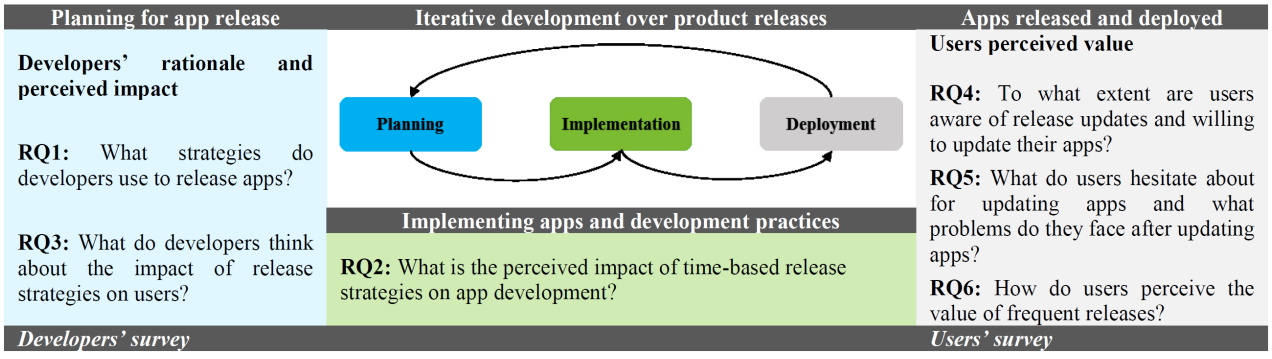


Figure 2.1: Summary of findings that led to challenges.

changes of functionality and quality in each

RQ3: What do developers think about the impact of release strategies on users?

Developers believe that the rationale for release decisions affects users' feedback. While the majority of developers believe that the time-based release strategies in general affect the feedback, the impact of release frequency is unclear.

RQ4: To what extent are users aware of release updates and willing to update their apps?

Almost all users are aware of mobile app updates, with only half of them always automatically updating their apps. For the majority of users, release date is not a deal-breaker, although they do prefer to install recently updated apps. Also, majority of users observed regularity in time-based strategies in the form of monthly releases.

RQ5: What do users hesitate about for updating apps and what problems do they face after updating apps?

Most participants have hesitated at one point to update apps and have experienced problems after updating their apps, especially due to lack of memory space, as well as phone and app crashes, security and privacy issues, and functionality loss. When it comes to the actual problems caused by updates, device or app crashes are the major problem, followed by low speed of the device, feature and functionality loss and bugs.

RQ6: How do users perceive the value of frequent releases?

Users have mixed feelings toward frequent app releases. Although they like apps with frequent updates and do not perceive it to include low quality apps, at the same time, frequent updates may be discouraging and could negatively affect users' decision for selecting, downloading or even uninstalling apps.

In Section 2, we briefly highlight the current state of research on mobile apps in software engineering and we discuss the few empirical studies that exist in this domain. In Section 3, we

describe our survey methodology and the conceptual model behind the study design. In Section 4, we characterize survey participants and introduce our classification of users and developers. We describe the results of the survey for different groups of users and developers in Section 5, then discuss our findings in Section 6. The limitations of our results are presented in Section 7, followed by the conclusions of the paper.

2.2 Related Works

Mobile apps are focused software products designed for mobile devices that are distributed through centralized mobile app stores [88]. The dynamism of these markets in combination with the multiple variables affecting market feedback (i.e., app rating, number of downloads and reviews) triggered substantial research in recent years. Considering app rating as an indicator of success for mobile apps, Linares-Vásquez et al. [131] analyzed the effect of API changes and fault tolerance of mobile apps on app success. Addressing the question of how effective apps have been created, reuse of mobile apps has been studied by Mojica et al. [167] and Linares-Vásquez et al. [132]. As another important quality or success characteristic, the security of apps has been analyzed [15, 78]. Only a small fraction of developers releases a large number of apps [200]. Those developers mainly belong to larger companies, while the majority of the developers work on their own. The mobile app domain also has seen a number of survey studies to analyze app users' behavior [55, 97, 98]. Related to the software engineering aspects of mobile apps, the survey that is closest to our study is the one by Joorabchi et al. [109], who studied developers' challenges for cross-platform app development by performing a survey with 188 developers.

Although app stores have initiated a variety of research directions, the release engineering aspects of mobile apps have not been tackled so far, in contrast to web and software apps, where release management is well-established [228]. Several studies have been done to analyze the effect of rapid releases on software [103, 148, 152]. Khomh et al. [115] empirically analyzed the impact of the migration of the Mozilla Firefox release process towards shorter cycle time, on the software

quality. They found that although bugs are fixed faster with rapid releases, proportionally fewer bugs are being fixed.

Recently, McIlroy et al. [158] empirically analyzed the update frequency of the top 10,713 mobile apps across 30 mobile app categories. Their results showed that 14% of the apps are updated frequently. 45% of the frequently-updated apps do not provide the users with any information about the rationale for the new updates. They also found that users highly rank frequently-updated apps and frequently-updated apps remain popular based on app stores' ranking. In contrast, we are interested in understanding why developers opt for a specific release strategy and whether there is in fact a plan, hence our study is not only limited to the frequency of releases. Furthermore, instead of deriving the impact of this strategy in terms of ratings and app store ranking, which might be affected by other factors, we directly include this as a question to the users.

In summary, while app stores are dynamic and many releases are observed in the app store [14], the how and why of practices and strategies for release management in this domain have not yet been studied.

2.3 Methodology

We used surveys to collect information from mobile app users and developers to describe and explain their behavior, attitude, and knowledge of using and releasing mobile apps. We performed two different survey studies, i.e., one with mobile app users and one with mobile app developers.

Scope. To study release strategies and practices, we targeted the planning, deployment and implementation phases of the app development lifecycle. The research questions related to each phase and the intended survey participant are shown in Figure 2.1. For a sequence of releases, we investigated the visibility of app updates for users, and investigated if users and developers think the frequency of releases affects the users' satisfaction and willingness to update apps. Also, we were interested to see what hesitations and actual problems are related to updating to a new mobile app version.

Instrument. We created a five-minute survey for mobile app users (including 16 close- and two open- ended questions) and a separate 10-minute survey for app developers (including 18 close- and six open- ended questions), both with descriptive design in order to explore and capture the description of release practices in mobile apps [202]. The surveys were approved by the University of Calgary Conjoint Faculties Research Ethics Board. To manage the survey distribution, we used the Qualtrics platform . Both the Ethics board and seven grad and undergrad software engineering students evaluated the surveys in terms of understandability, reliability and validity of the questionnaire. Three of these participants are doing active research in release planning and engineering, and hence were able to perform content validity of the questionnaire, while the remaining four students performed face validity [117]. Their feedback was collected and the questionnaires were adjusted. The questionnaires can be found online.

Participants. For both surveys, we used a non-probabilistic convenience sampling method [26], which is a proper method to use when subjects are easily accessible. The prevalence of smart devices and mobile apps, especially for social networks, inspired us to advertise the survey on Twitter and Facebook.

For the *user survey*, our advertisements asked people to participate in a short academic survey on mobile apps and enter a raffle of five 15 dollars amazon gift cards. We targeted the ad specifically at users of ages 15-85 in the US, Canada, Germany, and the UK who spoke English. We only published this advertisement on mobile news feeds. The ad reached 6,486 people and received 2,197 clicks. In addition, we created a task in Amazon Mechanical Turk accessible only to people with experience with mobile apps. The advertisement and task were live for one week during August.

For the *developer survey*, we advertised the survey for one month on both Facebook and Twitter. We targeted the ad specifically at users in the US, Canada, Germany and UK of ages 15-60 who spoke English. We published this ad both in mobile and desktop news feeds. In total, we received 46 clicks on the ads. The advertisements were live for six weeks in August and September.

Data Preparation and Analysis. First, we excluded incomplete survey entries. We used de-

(a) Users

	Low usage	***High usage***
Low involvement	*Category 'LILU'* 40.3% of participants	*Category 'LIHU'* 30.5% of participants
High involvement	*Category 'HILU'* 9.3% of participants	*Category 'HIHU'* 19.9% of participants

(b) Developers

	Low success	***High success***
Low experience	*Category 'LELS'* 5.6% of participants	*Category 'LEHS'* 36.1% of participants
High experience	*Category 'HELS'* 5.6% of participants	*Category 'HEHS'* 52.7% of participants

(c) Usage

value	#downloaded apps (AQ1)	#purchased apps (AQ2)	Classification
1	≤ 5	= 0	S = value(AQ1) + value(AQ2) If S < 5 Then *Usage is Low, Otherwise High*
2	5 < # ≤ 10	0 < # ≤ 5	
3	10 < # ≤ 15	5 < # ≤ 15	
4	> 15	>15	

(d) Success

value	#developed apps (AQ1)	#developers (AQ2)	Classification
1	1	≤ 1	S = value(AQ1)+ value(AQ2) If S < 5 Then *Success is Low,* Otherwise *High*
2	1 < # ≤ 5	2 ≤ # ≤ 3	
3	6 < # ≤ 10	4 ≤ # ≤ 5	
4	>10	> 5	

(e) Involvement

value	#rated apps (AQ1)	#reviews (AQ2)	Classification
1	≤ 5	= 0	S = value(AQ1) + value(AQ2) If S < 5 Then *Involvement is Low,* Otherwise High
2	5 < # ≤ 10	0 < # ≤ 5	
3	10 < # ≤ 15	5 < # ≤ 15	
4	> 15	>15	

(f) Experience

value	highest #downloads (AQ1)	best rating (AQ2)	Classification
1	<500	≤ 1	S = value(AQ1)+ value(AQ2) If S < 5 Then *Experience is Low,* Otherwise *High*
2	500 < # ≤ 50,000	2 ≤ # ≤ 3	
3	50,000 < # ≤ 500,000	4 ≤ # ≤ 5	
4	>500,000	> 5	

Figure 2.2: Classification of users (a) and developers (b), applying classification algorithms for (c) usage, (d) success, (e) involvement and (f) experience.

scriptive statistics to characterize survey participants and to analyze survey results. In addition, we classified participants in each survey into different groups based on demographics (see Section 4). We then analyzed survey results for the different groups of participants. We used the Kruskal-Wallis test (a non-parametric test for analyzing significant differences between two or more groups) along with the Tamhane post-hoc test (conservative pairwise comparison) to analyze the significance of results between user groups and between developer groups.

For open questions, two software engineering students (including one of the authors) performed an open card sorting [100] technique using the Uxsort tool [100] to summarize the results. The Cohen's Kappa [29] agreement between two authors were between 87% to 95%. Card sorting is a set of activities for grouping and naming concepts or objects and to derive taxonomies from qualitative data [100]. In open card sorting, users define categories, while in closed card sorting the categories are pre-defined.

2.4 Demographic Analysis of Survey Participants

Since we used a convenience sampling method [118] for both of the surveys, it is important to first get a better insight into the characteristics of the survey participants. For this, we classified both the surveyed users and developers based on demographic questions that we asked.

Users

Through advertisements, we received 2,197 clicks and 674 people eventually participated in the survey. Among the answers, we excluded the users that left the survey incomplete and the ones who provided meaningless answers (like repetition of letters) to at least one open question. By excluding 20 entries, we analyzed the responses of 654 participants in the user survey. The following information was gathered via the survey and is used to characterize the developers:

- The app stores the participant has ever download apps from.
- Number of downloaded apps: The number of times they personally installed apps on their devices.
- Number of purchased apps: The number of times they paid for an app.
- Number of rated apps: The number of times they rated apps.
- Number of times wrote a review: The number of times they have reviewed apps.

While the selection of the app stores was not exclusive (participants could select all the app stores they had experience working with), most of the participants (67.6%) used Google Play. After that, iTunes (with 26% of participants) and Microsoft Store (with 2.8% of participants) were used by our participants. Most of the users downloaded more than 15 apps. In terms of ratings, most of our participants rated apps five to ten times. Finally, most of the users wrote reviews less than 10 times and purchased an app less than five times. Using the demographic information of participants, we classified the participants into four different usage groups based on their degree

of usage and degree of involvement (in terms of providing feedback). We used the number of downloaded apps and the number of purchased apps as the *degree of usage* from app stores. Along with that, we used the number of rated apps and number of times they wrote a review as the *degree of involvement* for a participant. The groups were named low involvement and low usage (LILU, 40.3% of participants), *low involvement and high usage* (LIHU, 30.5% of participants), *high involvement and low usage* (HILU, 9.3% of participants) and *high involvement and high usage* (HILU, 19.9% of participants). Detailed descriptions of these categories and how we obtained them using a heuristic method are presented in Table 2.2-(a). The thresholds used are based on analysis of the survey results. We discuss them in more detail in Section 7. As expected, people who rarely use app stores are rarely involved in feedback loops, hence the high involvement and low usage class, has the lowest number of participants. In reporting the survey results, we will compare the answers between these four user classes to analyze whether or not their perceived value of mobile app releases differs from one another

Developers

Developers were persuaded to participate in the survey via advertisements, which received 46 clicks and 36 participants. All these 36 answers were complete and acceptable for our analysis. To analyze the characteristics of the participants, we asked questions about:

- Number of developed apps: The number of apps they developed personally or in a team.
- Years of experience in app development: The period of time during which they were involved in app development.
- Highest number of downloads among apps: the highest number of downloads among all the apps that they developed.
- Rating of the most successful app: Rating of their most downloaded app.

- Years of experience in app development: The period of time during which they were involved in app development.

We considered the number of developed apps, duration of app development and number of co-developers as the developers' degree of expertise. We used the highest number of downloads among apps and rating of the most successful app as the degree of success for the app developer. Based on this data, we classified the developers into four groups, i.e., *low expertise and low success* (LELS, 5.6% of participants), *low expertise and high success* (LEHS, 36.1% of participants), *high expertise and low success* (HELS, 5.6% of participants), and *high expertise and high success* (HEHS, 52.7% of participants). The details of these classifications are presented in Figure 2.2-(b). Based on common success criteria of apps, most participants had successful apps (88.8% of them) and they mainly have high experience in development (52.7% of them). This implies that our observations regarding release strategies correspond to strategies that worked in practice. Further studies are necessary to analyzed failing strategies. We discuss this in more detail in Section 7.

2.5 Findings

In this section, we summarize the findings from the developer and user surveys and discuss the differences and commonalities between the perception of developers and users regarding app releases. We do this by analyzing the differences between the four groups of developers and users identified in Figure 2.2. After running Kruskal-Wallis tests, if we did not find *any* significant *difference between any two groups* we use the notation ☒. In contrast, if the p-value of Kruskal-Wallis was <0.05, we test for significant differences between each two groups by running Tamhane's test and we tagged this as ☑ Group1 vs. Group 2.

Release Strategy and its Impact – Developers' Perception

Since mobile app developers have a wide variety of expertise and knowledge on software production in app stores [49], it is unclear to what extent they follow actual release management practices.

In this section, we analyze these practices and to what extent their release strategies are based on rationale (explicit strategy) or intuition (no clear strategy).

App Release Strategies (RQ1)

Release management practices were studied in medium to large software organizations and open source teams and the applicability of its principles into mobile app development is questionable. Mobile app teams tend to be smaller, with sometimes only one developer who needs to manage all of the development and release process. Moreover, app developers might be inexperienced in developing apps or developing software products in general [49] and not being experienced in the release management of software products.

Half of the app developers (52.78%) follows a rationale-based release strategy. To understand the state-of-the-practice of mobile app release management, we initially asked developers if they follow an established rationale for releasing their apps (release strategy) or whether they merely follow their intuition for defining what to release in the app version and when to release it.

☑HEHS vs. LELS and LEHS (p-value=0.007 and 0.005 respectively): Our findings show that developers with high experience (and high success) tend to follow significantly more a rationale for releasing apps comparing to the ones with low experience in app development. The latter instead follow their intuition.

The survey results show that release strategies are determined early in the app's lifecycle and most of the time developers do not change their strategy after releasing their apps. Half of the participating developers with a certain release strategy stated that they followed the same strategy from the beginning of their apps' lifecycle. Also, the majority (66.67%) of participants that followed release strategies stated that they do not change it at all:

Follow a release strategy	No 47.22%		Yes 52.78%
Have changed release strategy	No 66.67%		Yes 33.33%
How long been following a strategy	Same from start 50%	For half of the releases 31.25%	Only recently 18.75%

Table 2.1: Release strategies as stated by developers.

Category	**Description**	**Frequency of categories**	**Total**
Time-based	Weekly or bi-weekly	45%	80.00%
	Yearly	15%	
	Quantitative descriptions (agile releases, releasing often,)	10%	
	Time-based strategy for pre- or post-release versions	10%	
Marketing considerations	App localization and country targeting	25%	40.00%
	Blog spill-outs	10%	
	Promotions	5%	
Quality (test) driven	Test coverage and internal quality assurance	10%	25.00%
	Over the air installation and testing	10%	
	Test with a portion of customers	5%	
Feature-based	Implementing planned features	10%	20.00%
	Release each implemented feature	10%	
Size-based	Size of the update	10%	10%
Occasional	Release app during the launch of a new device of a well-known brand	5%	5%

These findings did not show statistically significant differences between the four groups of developers.

We identified six different categories of release strategies, as summarized in Table 2.1. Among all the strategies, the time-based (84.2%) and marketing-based strategies such as country targeting and promotional versions (42.1%) are ranked first and second, respectively. Some app developers follow more than one release strategy. For example, the following developer considers both the quality-driven and the *time-based release* strategies:

"Merging contributions, releasing alphas after each merge, freeze for beta with parallel alphas still going on, stable release when beta is good enough within one to six weeks."

In this way, stabilizing a release within a reasonable time forms the rationale for the above developer. The following developer bases his strategy on two pillars, i.e., offering localized (translated) versions of his apps to reach more markets and advertisement versions to entice users via popular forums or blogs. In this case, the release cycle time takes months instead of weeks.

"Let's say we have two types of strategy distribution and publishing. For the first we localize the app distribution for the second we send a simplified but polished and attractive version to few bloggers and 1-2 forums to advertise and do more test in this time. The whole process should take

Table 2.2: Reasons of users to hesitate to update apps.

Category	Type of concern	Frequency of statement	Total
Quality of the app	Device and app crash	9.00%	37.40%
	Security and privacy	8.40%	
	Bugs and compatibility concern	13.80%	
	Speed	4.70%	
	Data loss	1.50%	
Release and version specification	Download and update size	5.60%	16.40%
	Lack of update description	2.00%	
	Time to stabilize before nload	1.20%	
	Improper timing for update	5.00%	
	Frequent updates	2.60%	
Device-related issues	Space	13.30%	15.60%
	Battery problems	2.30%	
Functionality of the app	Uninteresting update	5.20%	12.70%
Negative arguments	Feature and functionality loss	7.50%	10.10%
	Negative experience from past updates of the app	1.40%	
	Reviews	5.20%	
	Anecdotal reports	3.50%	
App-related issues	Unused app	4.20%	5.30%
	Advertisement and paywall	1.10%	
Gut feeling	–	–	2.50%

4-5 months and we cannot do it so often."

Finding 1. *The majority of developers make rationale-based decisions for mobile app releases, which is observed significantly more often for developers with high experience.*

Finding 2. *Time-based and marketing-based considerations are ranked as the first and second most popular rationale among developers, respectively.*

Perceived Impact of Time-based Release Strategy on Development (RQ2)

RQ1, we found that only one third of the developers with a release strategy have changed their release strategy. We also asked all developers with a release strategy (including those that never changed it) whether they could imagine changing their time-based strategy in order to achieve more success as a measure to understand their flexibility on time restriction:

	Disagree	Neutral	Agree
Change time-based strategy to achieve higher success	22.23%	13.77%	*64%*

In general, the main drivers for the developers to change time-based release strategy are user feedback and personal experience with earlier releases. In particular, the participants who agreed to change their time-based strategy stated the following reasons for doing so:

Accommodating user feedback	36.1%
Gained experience in similar cases	16.6%
Do not feel obligated to follow self-defined schedule	13.9%
Occurrence of specific occasions or events	8.3%
Status of competitor apps and their releases	5. 6%
Gaining higher market visibility by analyzing country targeting stats	8.1%
Following the main schedule in addition to hot fixes	6.3%
Time needed to stabilize app due to the results of over-the-air testing platforms such as Testflight or Hockeyapp	5.1%

On the other side, the disagreeing participants stated the following reasons for not changing their release strategy:

Users are waiting for the release	35.7%
Special occasion may pass	27.1%
Competitor apps would release the product sooner	24.6%
Limited resources and capacity	12.6%

☒ These findings were not statistically significant for different groups of developers.

The majority (61.1%) of developers believe that apps with frequent updates (more than once per 3 weeks) deliver less functionality and quality changes in each version. On the other hand, developers were unsure whether apps with infrequent updates had an overall higher quality:

Apps with frequent updates deliver less changes in each version	Disagree 27.8%	Neutral 11.1%	Agree 61.1%
Less releases means higher quality	Disagree 38.9%	Neutral 22.2%	Agree 38.9%

☑ HEHS vs. LEHS (p-value= 0.003): Developers with high experience (and high success) believe significantly more that apps with frequent updates have lower quality than developers with low experience.

Furthermore, **a minority of the participants stated that the apps' time-based release strategy may affect the development practices, team size, and effort needed to develop them**. Whereas 44.5% of the participants did not agree with this statement, 36.12% state that their se-

Table 2.3: Actual problems that users faced after updating mobile apps.

Category	**Type of concern**	**Frequency of statement**	**Total**
Quality of the app	Phone and app crash Speed Bugs Data loss Compatibility concern Security and Privacy	51.20% 13.20% 9.10% 3.10% 2.80% 2.20%	81.60%
Functionality of the app	Feature and functionality loss Undesirable UI	10.30% 1.30%	11.60%
Device-related issues	Space Battery draining	2.50% 2.00%	4.50%
App-related issues	Advertisement and Paywall	2.30%	2.30

lected time-based release strategy changed the way they developed the app in terms of number of developers involved, test coverage, code quality, code review and so on. In particular, they stated the following changes:

- More development effort. *"... more developers and planning and team meetings needed".*

  "... somehow, more work!"

- Less provided functionality than targeted. *"... Spend time on testing, (so) we stop (including) new feature developments"*

- Change in packaging and testing the apps. *"... I pack all changes each 2 weeks and give around 4 weeks to stabilize ..."*

  "If you're looking for a 'Hollywood-blockbuster' style (where v1.0 has a big download spike, then quickly trails off - typical for many apps), you'll want a long beta test with a healthy amount of testing and development before v1.0, If you're looking for a 'slow growth' style (particularly common f the app is for an already-well-known company [...]) you should still beta test, but it's OK to launch with more of a 'minimum viable product' and grow from there."

- Change in design and implementation of the app.

"... We need more design for features in each release so we need to pay more for designers and hire one part time." "I need to test and develop and design overall a lot of cathartic work so I do a bit of each instead of finishing one and go to the other."

Despite the majority of developers claiming no effects on development practices, team size and development, 44.5% of participants agree that frequent releases need more developers:

Time-based release strategy affects development practices, team size and development	Disagree *44.5%*	Neutral 19.5%	Agree 36.1%
More frequent updates need bigger teams and more effort	Disagree 36.1%	Neutral 19.5%	Agree *44.5%*

☒ These findings were not statistically significant for different groups of developers.

The contradictions between some of these results suggest the need for further research on this topic.

Finding 3. *Developers are flexible to deviate from their apps' time-based release strategies, mainly to gain better user feedback.*

Finding 4. *Developers believe that apps with frequent updates deliver less changes and do not need additional development effort.*

Perceived Impact of Release Strategy on Users (RQ3)

Prior research has shown that the feedback provided by user reviews, emergence of successful apps and the general ecosystem dynamics affect new versions of apps [141, 197, 236]. In our survey as well, **44.5% of developers agree that the release strategy affects user feedback** in the form of ratings, reviews, and number of downloads:

Release strategies affect user feedback	Disagree 36.1%	Neutral 19.4%	Agree *44.5%*

☑ HEHS vs. LEHS (P-value: 0.029): Developers with high experience significantly stronger agreed compared to the developers with low experience (and high success). In particular, the majority of participants (61.1%) agreed that time-based strategies affect user feedback (rating, number of downloads, and reviews), yet developers are unsure about the exact effect of update

frequency on app users. While 38.9% of participants agreed that frequent app releases have a negative impact on user feedback, a similar number disagreed.

Frequent updates causes negative user feedback	Disagree 38. 9%	Neutral 22.2%	Agree 38.9%
Time-based release strategy affects users' feedback	Disagree 27.8%	Neutral 11.1%	Agree *61.1%*

☒ These findings were not statistically significant for the different groups of developers.

Finding 5. *The release strategy affects user feedback, something that developers with high experience are most convinced about.*

Finding 6. *Developers believe that, in general, the release strategy affects user feedback.*

Release Strategies and its Impact – Users' Perception

Having learned the developers' perspective on release strategies, we now investigate the users' perception.

Awareness of app Updates (RQ4)

Many mobile apps provide an automatic updating facility that automatically updates an app to its latest release, without user intervention. Apart from making life easier for users, such a functionality helps a company focus on maintaining just one app version in the field as opposed to having to maintain a heterogeneous mixture of old releases. Whereas for desktop systems like browsers large companies have a clear idea about which versions are still being used, for mobile app companies, especially the smaller ones, this is not necessarily the case.

To investigate the app users' stance towards automatic updates, we asked users explicitly about their awareness of updates to their installed mobile apps. The results showed that **almost all users (96.1%) notice the app updates and are aware of it**. Also, users do not blindly enable automatic updates for all mobile apps, as **41% of the users only partially and 14.1% never enable automatic updates**.

Of the users that do not always allow automatic updates, 18.5% rarely update their mobile

apps and 45.6% update less than once a month. In particular, power users seem pickier regarding automatic updates.

Noticed app updates	Yes 96.1%				No 3.9%	
Allow automatic updating of apps	Always *44.9%*	Partially 41%			Never 14.1%	
		Rarely 18.5%	Monthly *45.6%*	Weekly 25.5%	More often 10.4%	

☑ LUHI vs. LULI (P-value=0.002): Users with low usage and high involvement allow automatic updates significantly less often than users with low usage and low involvement.

☑ HUHI vs. LULI (P-value = 0.007): Also, users with high usage and high involvement allow automatic updates significantly less often than users with low usage and low involvement.

Finally, the majority of users (59.6%) state that the recency of the latest version of an app has never prompted them to buy/install an app. However, 61.9% state that, given the choice between two equivalent apps, they would choose the one having more recent release date. So, while the last release date could potentially break the tie between two similar apps, this information in app stores is not decisive for choosing an app for 40.38% of the users. Power users again exploit this kind of information more than others do:

Prefer to install app with recent release date in case of equal functionality and quality	No 38.1%	Yes *61.9%*
Ever decided to install an app because of last release date	No *59.6%*	Yes 40.4%

☑ HUHI vs. LULI (p-value: 0.025): We found that users with high usage and involvement consider release date to be significantly more important compared to the users with low usage and low involvement.

Given that the "last release date" is available along with other app information in the app stores, 79.2% of participants reported that they observed regularity in release cycles (35.4% did this rarely, while 43.8% did this frequently). Among the participants who observed regularities, 28.7% reported having observed weekly updates and 47.4% reported having seen monthly updates. 2.4% of users reported other types of regularities such as *bi-weekly updates, updates by closing the*

app, restarting the device or logging-out of the application:

	Never	Rarely		Occasionally		
Observed regularity in app updates	22.1%	34.1%		*43.8%*		
		Occasional updates 16.5%	Yearly updates 5.1%	Monthly updates *47.4%*	Weekly updates 28.6%	Other 2.4%

☒ We did not observe any statistical significance between user groups for the above statements.

Further, we did not find a correlation between the manual update habits and the observed regularity in app updates. This means that for example, the users who usually update their apps once a month manually, did not necessarily report monthly regularity in app updates.

Finding 7. *Users are aware of mobile app updates. Less than half of them always turns on automatic updates, especially users with low usage and low involvement.*

Finding 8. *While the release date is not a decisive factor to install an app, users prefer to have apps that were updated more recently.*

Finding 9. *Users observed regularity in the release cycle of mobile apps independent from their preference for manual or automatic updating of apps.*

Users' Hesitations with App Updates (RQ5)

60.3% of participants have some hesitations to update mobile apps, and almost half of the participants (47.7%) have encountered a problem after updating apps.

We categorized the participants' hesitations into seven categories in Table 2.2, while the actual problems experienced after app updates are categorized in four groups in Table 2.3.

The top reason why users hesitate to update mobile apps (stated by 13.3%) is the lack of memory space on their device. Phone and app crashes (9% of participants), security and privacy (8.4% of participants), and feature and functionality loss (7.5% of participants) are the next most common reasons:

"... (App updates) sometimes takes too long and jacks up my phone." [Speed]

"... There were reviews how buggy it was." [Reviews - Bugs]

"Didn't want to lose date/game scores." [Data loss]

"Requesting permissions for things the app doesn't need like contacts info and pictures." [Security and Privacy]

"Occasionally an update will cause configuration and compatibility problems." [Compatibility concern]

When it comes to actual problems experienced, 51.2% of the users reported device or app crashes as the major problem they faced after updating their apps. After that, low speed of the app (13.2% of participants), feature and functionality loss (10.3% of participants) and bugs (9.1% of participants) were reported as the largest problems users faced:

"(App) starts sending me messages without asking me and I think it might have started accessing my data without my permission." [Security and privacy]

"Incompatible features, higher battery usage, memory usage, ads." [Compatibility concern, Battery draining, space, advertisement]

"Usually the app would crash or other problems so I try to update as little as possible. Unless there are new features." [Device and app crashes]

☑ HUHI, HULI vs LUHI, LULI (P-value: 0.009): The users with low usage (both in LUHI and LULI) encountered significantly more problems after updating their apps, compared to those who make more use of the apps (both in HULI and HUHI categories).

Finding 10. *The majority of users have concerns regarding updating apps and have faced update-related problems before. Users with low usage encountered significantly more problems comparing to users with high usage. Device and app crashes were the main reported problems.*

Awareness about Release Frequency (RQ6)

Overall, users have mixed feelings toward the frequency of releases, so while they like it at times they may also find it bothersome. 70.8% of users agreed that they are happy with their installed apps that have frequent updates. At the same time, 35.3% users claimed that they did uninstall apps because of too frequent releases:

Have hesitation for updating apps	No 39.7%	Yes *60.3%*
Encountered problems by updating apps	No *52.3%*	Yes 47.7%

Also, among apps with the same functionality and quality, 54.6% of participants prefer to install apps with less need for frequent updates. On the other side, frequent updates do not imply lower quality to the users. 66.2% of participants disagreed that frequently updated apps have lower quality.

Prefer to Install apps with less frequent updates in case of equal functionality and quality	No 45.4%	Yes *54.6%*	
Apps with frequent update have lower quality	Disagree *66.2%*	Neutral 11.5%	Agree 22.3%

☑ HULI vs. LUHI and HUHI (P-value=0.000 and 0.0002 respectively): The users with high usage and low involvement uninstall apps significantly more than the users with high involvement (both LUHI and HUHI).

☑ HULI vs HUHI (P-value=0.005): The users with high usage and low involvement prefer to download apps with frequent updates significantly less than the users with high usage and high involvement.

Finding 11. *Users have mixed feelings toward frequent app releases. They like apps with frequent updates but at the same time, frequent updates may be discouraging and could negatively affect users' decision.*

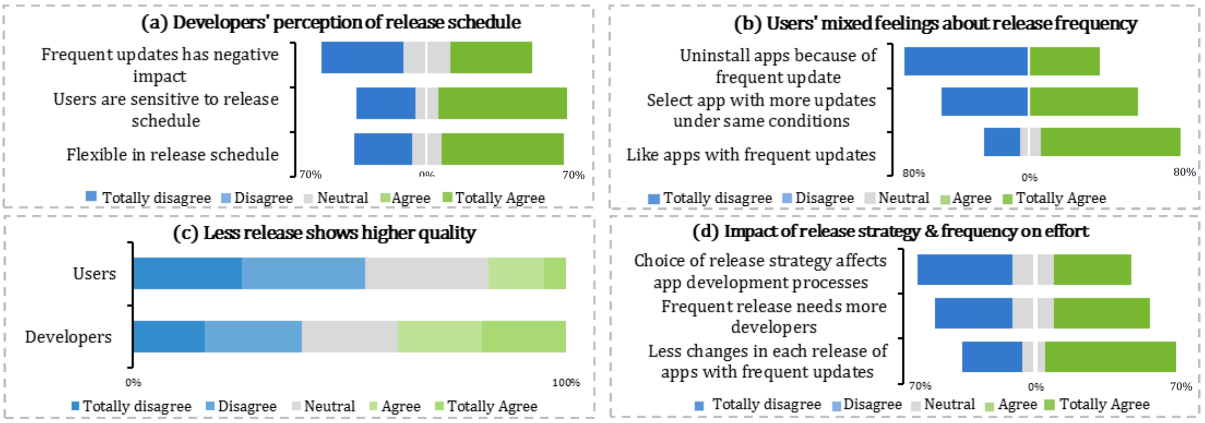


Figure 2.3: Summary of findings that led to challenges.

2.6 Discussion

During our analysis of the update attitude of mobile app users, we found that manual updating of mobile apps is still quite common (RQ4), despite the automatic update facilities of mobile app platforms. This might be linked to the sometimes negative experience of users regarding the frequency of releases (RQ6). Moreover, this might result in a huge diversity of running versions of an app among different users and cause maintenance issues. From this point of view, we asked both users and developers about their perception of frequent app updates.

Developers believe that the time-based release strategy in general affect user feedback, but the effect of release frequency in particular is unclear (Finding 6). On the other side, users have mixed feelings about release frequency: while they like apps with frequent updates, they might find it discouraging or might even uninstall the app for this reason (Finding 8). The perception of developers and users on the effect of release frequency is demonstrated in Figure 2.3 – (a) and Figure 2.3 – (b).

Challenge 1. *The impact of release frequency on mobile app users is unclear.*

The results of our analysis of the relation between release frequency and app quality was ambiguous. Half of the developers believed that less releases meant higher quality, while on the users' side the majority believed that less releases does not necessarily imply better quality of the app. The feelings of developers and users about the relation between release frequency and app quality is demonstrated in Figure 2.3 – (c) and Figure 2.3 – (d).

Challenge 2. *The relation between release frequency and app quality is unclear.*

There is an ongoing discussion in the software engineering community on the commonalities and differences of mobile apps and traditional software products. The results of this survey show that the release management of mobile apps is different in a variety of aspects from traditional software release management. In particular, the problem of when to release and release readiness [228] are affected by other variables. For example, the frequency of releases, reluctance of

updating apps on one side in conjunction with the development effort needed affect the decision of when to release an app.

There is an ongoing discussion in the software engineering community on the commonalities and differences of mobile apps and traditional software products. The results of this survey show that the release management of mobile apps is different in a variety of aspects from traditional software release management. In particular, the problem of when to release and release readiness [228] are affected by other variables. For example, the frequency of releases, reluctance of updating apps on one side in conjunction with the development effort needed affect the decision of when to release an app.

In particular, developers who participated in our survey mostly believed that the choice of a frequent release strategy does not affect development process. However, they did believe that a larger team might be necessary to accomplish this. Given that apps traditionally have been developed by smaller organizations [47], the decision to support frequent releases might have a large impact on the average mobile app company compared to release practice for traditional software development. Empirical investigation of this phenomenon is needed.

Challenge 3. *While the developers believe that release strategy and frequency do not affect development practices and total effort needed, empirical analysis using app store mining will be necessary to holistically evaluate and compare to traditional software release practices.*

The qualitative results of the developer survey provide insight into how common release strategies are (only half of the developers had an explicit strategy) and the specific release strategies that they follow such as size-based strategies. While these strategies are not known for releasing in traditional software development and as the size of updates is a concern for users (see Table 2.2), the popularity and success of such strategies needs further research.

Challenge 4. *The popularity and efficiency of release strategies and comparison with release strategies in traditional software products is needed to define the state of practice in mobile apps' release management.*

A recent survey on challenges in product management revealed that half of the product man-

agers consider "the process" the biggest challenge. The survey included almost 900 respondents from around the globe and from a wide variety of industries and company sizes. While this is true for "traditional" software-based products, in this paper we have investigated some specific questions related to processes in releasing mobile apps. Further research is necessary to answer questions such as *Which release strategies could provide higher popularity and visibility for an app? What decision variables should be considered by developers for releasing their app?*

Inspired by app stores and their emerging impact on software engineering, it was predicted that requirement engineering will hugely be affected by users and their feedback [142]. This study could be useful for developers and app owners to understand and address users' problems and hesitations. For example, better app descriptions could help users better understand changes, explicitly defining consumption of device resources could address battery draining concerns [109] and quality assurance and usability testing could address the huge concerns regarding quality issues of new app versions [19, 109].

2.7 Threats to validity

Several limitations should be considered while interpreting and reusing the reported results.

Construct validity. Although we experimented with automated clustering of the users and developers, which led to seemingly random groups without clear rationale, we used manual clustering based on clear criteria. We believe that users with different degree of usage of mobile apps and involvement in providing feedback experience different feelings about the release practices, while for developers the role of experience in managing and valuing the releases is undeniable. However, one should consider the following as the limitations of these classification:

- The classification resulted in unbalanced grouping, especially in terms of developer success. This is partly by design, since the thresholds that we used to cluster apps are based on common success criteria for apps (divide it into upper and lower half), and we want to learn the release strategies used by successful app developers. Furthermore, despite our survey setup, convenience

sampling has the inherent risk of attracting only particular sub populations.

- The terms 'low' and 'high' for the degree of usage and involvement of users, and for the degree of experience and success of developers are relative. So we categorized a user downloaded more than 15 apps and never purchased an app as a user with low usage relative to the other users in survey.

In addition, the sample size of users is much bigger than developers (674 in comparison to 36). This unbalanced sample might endanger the validity of the study. However, number of app users is much more than the number of mobile app users as well. Also, in the surveys we only targeted the planning, implementation and deployment phases of the software development lifecycle as they likely have the most effect on release management. We designed questions based on our own release planning and engineering knowledge and from the available literature about software projects. Many other release management practices and topics could be addressed, but were not in order to keep the survey focused (to avoid scaring participants). We have mitigated this risk by designing open-ended questions.

Internal validity. The survey results reflect the subjective opinion of the participants, which may be different from reality. For example, when comparing quality and functionality offered in each release (RQ2) or talking about battery drain (RQ5), the meaning of "high" and "low" are subjective.

External validity. App stores attract many developers and users from all around the world. Compared to the number of involved users (645) and developers (36), our sample in this study includes a limited number of users and developers. Yet, this is still a larger sample than previous studies on user behavior [55, 97, 98], and, given the large effort going into the planning and analysis of a survey, in general the study of developers in this context is rare [55]. Also, we used convenience sampling [117] to obtain responses for both surveys. This increases the risk of bias of targeted population. For users survey, we mitigated this risk by targeting the advertisement only to Facebook and Twitter app users to make sure that they are familiar with the usage of mobile apps. Finally, the open questions and the qualitative findings, even if the sample would be biased,

provide a wealth of information for further research.

2.8 Conclusion

We performed two surveys (i) with 36 app developers to extract release strategies for mobile apps in general and the time-based release strategies in particular, to find the developers' rationale for release decisions, and the impact of their choice of strategy on development process and users; and (ii) with 654 app users to explore problems they have with updating apps and their feeling toward strategies with frequent updates. Our findings show a partial difference between release strategies of mobile apps and traditional software. In particular, we found relatively unknown marketing-based and size-based strategies in this context. Our study showed that half of the developers follow rationale-based release strategies and that time-based strategies are used by the majority of these developers. The mixed feelings of users about the frequency of releases and the uncertainty of developers about the impact of updates should be further supported by complementary app store analysis. Moreover, our study shows that release decisions should be re-evaluated and aligned toward user's needs and convenience.

Part II

Objective 1: Mining Information for Release Management

Chapter 3

App Store Mining is Not Enough for App Improvement

Authors: Maleknaz Nayebi; Henry Cho; Guenther Ruhe EMSE 2017

Abstract - The rise in popularity of mobile devices has led to a parallel growth in the size of the app store market, intriguing several research studies and commercial platforms on mining app stores. App store reviews are used to analyze different aspects of app development and evolution. However, app users' feedback does not only occur on the app store. In fact, despite the mass quantity of posts that are made daily on social media, the importance and value that these discussions provide remain mostly unused. In this paper, we study how Twitter can provide complementary information to support mobile app development. By analyzing a total of 30,793 apps over a period of six weeks, we found strong correlations between the number of reviews and tweets for most apps. In comparison to reviews, through the application of automatic classification, topic modeling and subsequent crowdsourcing we successfully mined 22.4% additional feature requests and 12.89% additional bug reports from Twitter. We also found that 52.1% of all the feature requests and bug reports were in common between reviews and tweets. Further, from performing sentiment and content analysis for 70 randomly selected apps, we found that tweets provided more critical and objective views on apps than reviews from the app store. These results show that app store review mining is indeed not enough; other information sources ultimately provide added value and information for app developers.

This paper has been submitted to Empirical software Engineering Journal (EMSE). This is a joint paper of ***Maleknaz Nayebi****,* ***Henry Cho****, and* ***Guenther Ruhe*** [1].

[1]Authors are affiliated with SEDS lab at University of Calgary

3.1 Introduction

The high demand for mobile devices, combined with the relative ease of app development has caused an exponential growth of the app market in the past half-decade. The resulting quantity of available apps caused competition between app developers to earn a spot on mobile users' devices. Thus, now more than ever, app developers need a source where they can collect user demands, feature requests, and general opinions to cater to the needs of their consumers effectively. The paramount importance of this developer-user feedback loop is confirmed by numerous studies in the area [155].

Where do app developers currently search to find all this information? Looking into third party data analytics platforms[2,3,4] and the existing software engineering body of knowledge [155] [174] [197], it can be seen that the most popular source utilized by app developers and researchers for mining user feedback is the app store. The app store is used to collect reviews and user meta-data to analyze how to enhance the apps [155]. However, the conversations about apps do not only happen in the app store. App users often employ social media platforms to discuss their experience with an app, whether it would be praise [141], feature requests or users sharing the app with their social media connections [197]. As a follow up of our former work [180], the *main objective* of this paper is to show that the evaluation and analysis of these app discussions on social media serve as valuable resources for app developers. We focused on Twitter to evaluate the added value and insight that developers can attain from tweets when compared to reviews.

Over a period of six weeks (June 24^{th} to Aug 7^{th}, 2016), we investigated the relationship between the number of reviews and number of tweets for a sample set of 30,793 Google Play apps. This sample included 7,933 apps from Google Play top charts (all the top chart apps across different categories) as well as random selections of apps with diverse rating and number of reviews [155]. For the purpose of an in-depth content analysis, we randomly selected 70 apps and compared the

[2] `https://www.appannie.com`
[3] `http://www.appbrain.com`
[4] `http://www.searchman.com`

content of app store reviews with the content of app-related tweets. These comparisons provided insight into Twitter's potential to give developers extra information that cannot be obtained with app store reviews. To achieve that, we applied automated classification and subsequent topic modeling, following the process employed in previous studies done in this context [42] [83] [141] [155]. We also went one step further and evaluated the quality of the extracted topics with automatic classification through crowdsourcing. Specifically, we investigated the following research questions:

RQ1: When considered over a period of time, is there a correlation between the number of tweets and the number of reviews about mobile apps? If so, which apps exhibit the strongest correlation?

Why and How: Prior to an in-depth analysis of content, we focused on the quantitative analysis of review and tweet trends. This means that we wanted to understand if an increase or decrease in user attention on an app brings about similar responses from both the app store and Twitter. We were mostly interested in exploring the types of apps that exhibit stronger/weaker correlations (between number of reviews and tweets) and consequently the types of apps that are potentially suitable for in-depth content analysis. To do so, we investigated the correlation between the number of tweets and the number of reviews over time and compared the distribution of the number of reviews and tweets for each app.

RQ2: What type of information can developers gain from tweet analysis that cannot be attained from the app store reviews?

Why and How: We were interested to see if it is worthwhile to look at tweets in addition to studying the more technically-oriented reviews. To ensure that we compare feature requests and bug reports within tweets with feature requests and bug reports within reviews (respectively), we first classified tweets and reviews into feature requests, bug reports, and others (including user experience, rating, and advertisement). For the two categories of bug reports and feature requests, we applied topic modeling and compared the topics of tweets with the topics of reviews to see if tweets include any extra information for app development in comparison to app reviews.

RQ3: How do app store reviews and app related tweets compare with regards to sentiments, degree of specification, and understandability?

Why and How: Sentiment analysis was used to understand users and prioritize their needs for app evolution and improvement [155] [84] [80]. Based on this, we were further interested in the degree of sentiment alignment between tweets and reviews of mobile apps. We also were interested in understanding how the reviews and tweets are characterized and related to each other with regards to technical specification and understandability. To investigate these questions, we compared the sentiment values of reviews and tweets. The sentiment analysis was done based on polarity (positivity and negativity) and subjectivity (factual or opinionated) using machine learning. The degree of specification and understandability was crowd evaluated for *all* tweets and reviews.

The results of the paper show the importance of analyzing alternative information sources other than just app store reviews. Specifically, *app store mining is not enough* in the sense that additional and complementary information can be extracted from other media to support app development, where Twitter is just one prominent example. With our results, we go beyond the existing work of just analyzing one feedback mechanism at a time. While supporting app development currently is limited to app store mining [155] [197] [263], our work shows the importance of looking beyond the fence [142] of the app stores to support app development.

We published initial results related to RQ2 in [180]. In comparison to that, in this paper we:

- Added two research questions (RQ1 and RQ3).
- Provided a motivating example.
- Extended the related work section.
- Analyzed a bigger sample of apps for unbiased app selection and comprehensively discussed data gathering and processing.

- Provided insight to the quantitative relation between the number of reviews and tweets as part of RQ1.
- Provided further discussion and examples for RQ2.
- Compared reviews and tweets with regards to sentiment, the degree of specification and understandability.
- Discussed the applicability and threats to validity.

In the next section, we provide a motivating example for our study. We give an overview of the related work in Section 3.3. In Section 3.4, we describe the data gathering and pre-processing steps. Then in Section 3.5, we discuss the variety of statistical, natural language processing, and machine learning techniques that we used. In Section 3.6, we present our results of comparing tweets with app reviews. We sum up the paper by investigating threats to validity in Sections 3.7. A discussion of results is provided in Section 8. The paper is wrapped up by conclusions and outline of future work in Section 3.9.

3.2 Motivating Example

In this section, we briefly illustrate the main idea of the paper through a recent example. Pokemon Go, the free-to-play and location-based game with augmented reality features was released on July 5^{th}, 2016. It got a lot of attention from users, turning it into a global phenomenon. Pokemon Go users wrote 254 reviews on the first day after the app release. For the same day, we mined 2,103 tweets related to its Android mobile app. Figure 8.1 compares the number of reviews with the number of tweets for Pokemon Go over six weeks period of time. We observe a similarity in trend, showing that the app drew increased attention in both media. During the first six days of the app release, we can see that users tweeted about the app more than reviewing it in the app store. Beginning of August, there is a jump in both numbers of tweets and number of reviews, but the

jump in reviews comes a few days later. Twitter users did not only react faster, but also reported servers and feature crashes to the app owners.

For example, on July 28^{th}, Twitter users warn each other about the new update of Google Maps which adversely affected the accuracy of the Pokemon Go map. The official Pokemon Go News account used this form of Twitter crowd intelligence and officially warned the users about this update and provided a workaround. In comparison, the reviews mined from the app store show very few mentions about Google Map changes, and the reviews are written three days later (on Aug 3^{rd}). Having a comprehensive understanding of users' needs and desires in a timely manner and being aware of the need to change existing features is a key advantage that could be attained by analyzing tweets, not just app store reviews.

Similarly, when a pay wall and yearly subscription fees for accessing several features was introduced for the Evernote app in a release (on Jun 13^{th}), users reacted both in app store reviews and tweets. Looking into the tweets, we found that Twitter users repeatedly submitted the request of "accessing passcode pin for free"; app owners made this feature available in the most recent release of the app.

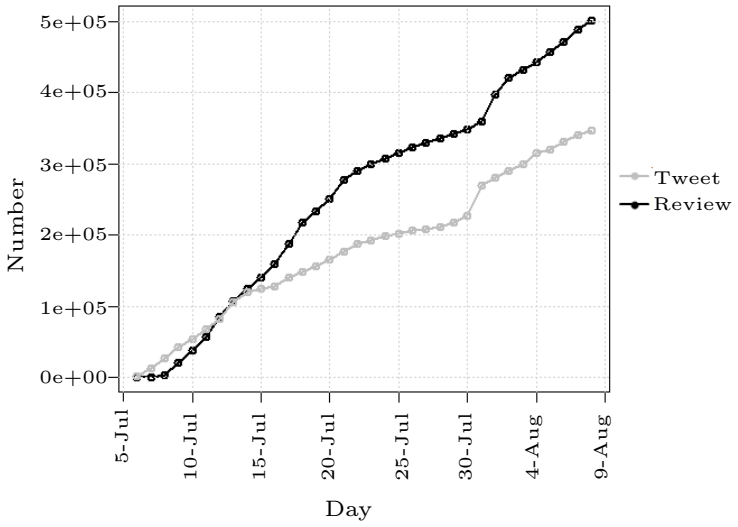


Figure 3.1: Cumulative number of app store reviews and tweets about Pokemon Go app for a period of six weeks from June 24^{th} to Aug 7^{th}, 2016

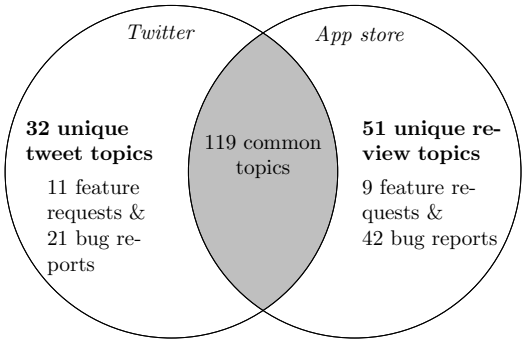


Figure 3.2: Comparison of the number of feature request and bug reports between app reviews and tweets for the Pokemon Go app.

Figure 3.2 shows an example result of this paper for Pokemon Go. We mined both app store reviews and app related tweets and found the commonalities and differences between tweets and reviews that are informative for developers.

3.3 Related Work

App store review analysis and Twitter analysis are established fields of research. In this section, we discuss the most related studies of these two fields.

App Review Analysis

Reviews have been analyzed with different objectives in previous studies. An overview was provided by Martin et al. [155] where they analyzed 45 studies on app store reviews performed between 2012 and 2015. Several studies focused on different techniques for extracting app features from reviews [105] [84], while other studies focused on developing models and tools for automated classification of reviews into predefined categories [42] [77] [80] [141].

Maalej and Nabil [141] discussed machine learning methods for classifying app reviews. They classified app reviews as bug reports, feature requests, user experiences, and ratings. They described the pre-processing and stemming steps and its threats for app reviews. They also compared

the performance of *Naive Bayes*, *Decision Tree* and *Max Ent* classifiers on mobile app reviews and found that *Naive Bayes* achieved high accuracy with a smaller training set and much less time in comparison to other classifiers. Based on these findings, we used *Naive Bayes* as one of the learning methods in our methodology.

For the content analysis of app reviews, Chen et al. [42] suggested the use of topic modeling in AR-Miner. AR-Miner [42] first classifies reviews into informative and non-informative and applies topic modeling within each class. We combined this method with the classification suggested by Maalej and Nabil [141] to analyze the content in depth. Furthermore, mining app reviews have been used as a basis for decision support. Palomba et al. [197] analyzed 100 Android apps and showed that addressing user reviews increases the chance of apps' success. Villarroel et al. [263] introduced CLAP (Crowd Listener for releAse Planning) for categorizing reviews into bug reports, clustering related reviews, and prioritizing reviews. More recently, Ciurumelea et al. [44] defined a two-level taxonomy based on 1,556 reviews with the aim to assist mobile app developers with planning maintenance and evolution activities. Di Sorbo et al. [54] defined a taxonomy of reviews and provided a tool for categorizing reviews.

Sentiment analysis of mobile app reviews for analyzing positivity or negativity of users' feedback have been discussed by different authors [155]. Chen et al. [42] used sentiment of app reviews to identify informative reviews. Guzman et al. analyzed review sentiments in support of evolution decisions by prioritizing changes considering users' sentiment. In a nutshell, sentiment analysis was used to identify informative reviews, user complains, and priorities tasks for app evolution.

In short, app reviews formed the basis for many studies and decisions ranging from feature extraction to release planning of mobile apps. The underlying assumption for these studies is that developers would benefit from looking into reviews for prioritizing bugs [42] [77] and planning for releases [106]. While this is true, our study shows that the app store review data is not enough and media such as Twitter can provide complementary information for app development. In fact, many app development companies already have an active and official channel on Twitter to communicate with their users [83].

Tweet Analysis

Analyzing Twitter to gain information about peoples' communication is the subject of study in different fields including but not limited to journalism, emergency management, bio-medicine, stock market analysis, and most importantly software engineering. On the other hand, several research studies on natural language mining, social media analysis, communications, and sentiment and opinion analysis were introduced as enabling technologies for using Twitter information. While these studies are outside software engineering, we relied on their promising results we introduce and use them for this study.

Tweet analysis in software engineering

Twitter was analyzed from different software engineering perspectives. Bougie et al. [33] studied the potential of Twitter and micro-blogging for software development while Tian and Lo [255] analyzed the software developer microbloggers and their communities on Twitter. Wang et al. [**?**] studied tweets for Drupal software and reported that Twitter is used to solicit contributions. In a very recent study, Nayebi et al. [182] introduced MAPFEAT to map tweets (as a source of user requirements) into the mobile app features.

In a very recent study, Guzman et al. [83] studied the contents of tweets and investigated their potential for software requirement engineering. They analyzed the content of 6,437,282 tweets across 22 software applications and defined their relevance for different stakeholder groups. Furthermore, they manually categorized 1,000 tweets to define the tweet sentiment (Likert scale 1 to 5) and the content categories. They ended up with 26 content categories that were extracted manually. They also discussed the automation potential for classifying the content of tweets. They compared Decision Tree and Support Vector Machine (SVM) classifiers. Their results showed that SVM has better performance when compared to Decision Tree analysis [83]. Based on their results, we selected SVM as another classifier for our study. Prasetyo et al. also suggested SVM. [210] for automatic classification of tweets into software engineering related and unrelated classes.

The value added by Twitter analysis was determined by studying the differences between contents of tweets and app store reviews. We employed both Naive Bayes as suggested by Maalej and Nabil [141] and SVM as suggested by Guzman et al. [83] to classify the content and hence contrast tweets and reviews. We also applied topic modeling, which was previously used by Chen et al. [42] for app review analysis and for Twitter analysis [94]. Using topic modeling, we compared feature requests and bug reports in Google Play and Twitter.

Sentiment and content analysis of tweets

Sentiment analysis of tweets is broadly investigated by researchers to understand feelings of the general public toward specific events. Agarwal et al. [4] investigated on tweet features for training better learners. Their study showed that use of Twitter-specific features such as hashtags does not tremendously change the accuracy of the classifiers. They conclude that sentiment analysis of tweets does not extensively differ from sentiment analysis in other genres. Thelwall et al. [253] analyzed tweet sentiments of specific events. They found that peaks of interest have stronger positive sentiment during the event in comparison to the time before that event.

Kouloumpis et al. [125] and Smedt et al. [248] studied lexical features of tweets for sentiment analysis. Mohammad et al. [166], Rosenthal et al [225] proposed different methods with high accuracy for sentiment analysis of tweets. These works had triggered many other studies for designing high-performance systems for sentiment and content analysis of tweets.

Content analysis of tweets was done manually in some studies (for example [172]) while many studies use topic modeling for this purpose. Ramage et al. [213] used LDA for topic modeling in consideration of emotions, social signals, and hashtags on Twitter. However, O'Connor et al. [195] used syntactic filtering, language modeling for extracting topics from tweets. Hong and Davison [94] compared multiple topic modeling techniques on the tweets. Their results showed that topic models learned from tweets of each user have superior performance in classification problems. These topic modeling techniques were widely adopted to different contexts and were used for content analysis of microblogs.

3.4 Data Gathering and Preparation

We gathered app related data from both Google Play and Twitter. The apps we selected for analysis are documented in Subsection 3.4. The data collection processes for reviews and tweets are detailed in Subsections 3.4 and 3.4, respectively.

Selection of Apps

Following the arguments and techniques proposed by Martin et al. [153], we selected a diverse set of apps for the analysis. We selected top-chart apps as well as non-top chart apps having different prices (free vs. priced), different number of reviewers and different ratings. For our investigation, 30,793 apps were analyzed, with each of them belonging to one of the following categories:

i. 7,993 trending apps from Google Play top charts (all the top chart apps).

ii. 7,600 apps: Rating $\geq$ 4.0 AND # of reviewers $\geq$ 1,000.

iii. 7,600 apps: Rating $\geq$ 4.0 AND # of reviewers $\leq$ 100.

iv. 3,800 apps: Rating $<$ 4.0 AND # of reviewers $\geq$ 1,000.

v. 3,800 apps: Rating $<$ 4.0 AND # of reviewers $\leq$ 100.

The 7,993 trending apps were collected from all six Google Play defined top charts - *Top Free, Top Paid, Top Grossing, Top Free Games, Top Paid Games and Top Grossing Games*. Top charts are published by the Google Play team based on popularity and market trends. We crawled the top charts *daily* to track changes in trending apps as well as rise and fall in apps' popularity. The remainder of the apps were collected randomly from a sample of the Google Play app store, according to the above classification. For this, we crawled 100,000 apps with a random crawler[5].

Looking into the rating distribution of the sample apps, we found that about 33% of them had a rating of 4.0 or lower. The rest (67% of apps in our random sample) had ratings above 4.0. We

[5] `http://nutch.apache.org/`

defined categories of apps having below 4.0 and equal or above 4.0 rating. Looking into the 10% percentile of our set of 100,000 random apps, a number of 100 or fewer reviewers is classified as a *low* number of reviewers. Similarly, a number of 1,000 or more reviewers representing about the top 80% of the reviewers, is called *high*. Having these categories, we randomly picked apps among each category in a way to make sure they did not belong to top charts. As the result of this selection, we compared top chart apps with (i) high rating AND high # of reviewers, (ii) high rating AND low # of reviewers, (iii) low rating AND high # of reviewers, and (iv) low rating AND low # of reviewers. For each app, the respective package names were collected. This enabled the investigation of the apps' market-side data, as to be detailed in the next section.

In **RQ1**, we quantitatively analyzed the relationship between reviews and tweets for all the above apps. In **RQ2** and **RQ3**, we focused on 70 apps for a more in-depth sentiment and content analysis involving 1,267,895 reviews and 358,860 tweets. We selected these apps from all Google Play *top charts* only, as we did not get a sufficient number of reviews and tweets for the ones outside top charts for the selected six week period as we will discuss in **RQ1**. For app selection, we considered the ratio between # of tweets and # of reviews. The *# of tweets* to *# of review* ratio was in the range $[0.083, 251]$ for the 7,993 apps considering all the data within the six weeks period.

We selected apps with the five lowest and five highest ratios between # of tweets and # of reviews from each top chart. We also randomly selected 5-10 other apps from each category (based on the number of apps in each top chart). Following this method, we picked 20 apps from each of the *top free* and *top free games* categories. We also randomly selected 15 apps from each of the *top paid* and *top paid games* categories. As we will report later, non-top chart apps did not receive many reviews and tweets in the six weeks period, hence were not selected for our study.

Data from Google Play

For each of the above 30,793 apps, the market-side data was extracted from the Google Play platform. Attaching the individual app's package name at the end of the following URL `"https:`

`//play.google.com/store/apps/details?id="` allows access to the app's unique web page. Once this page was accessed, the HTML data was crawled and parsed to collect the following information about each app - *app name*, *rank in top chart* (if the app is on the top chart), *ratings*, *app description*, *# of downloads*, *# of reviews*, *last updated date*, and *user reviews*. The average length of reviews in our dataset is 97 words.

Data from Twitter

Tweets about the apps were collected using the Twitter API, as provided through the `CliPS Pattern` module [248]. Using the `CliPS` open source module, we connected to the Python implementation of the Twitter API, which allowed for the mining of content, date-stamp and unique ID of tweets. We implemented various strategies to ensure that the tweets collected were complete and accurate.

First, the search was performed with the word *"app"* and *"mobile"* attached to the end of the app name (ex. *"Facebook"* - *"Facebook mobile app"*). This may have filtered out some relevant tweets but ensured that all tweets collected were relevant to the mobile application, rather than its web or desktop counter-parts. We also excluded tweets having any of the words *"apple"*, *"ios"*, *"iphone"*, *"blackberry"*, *"desktop"*, *"Mac"*, *"window"* and *"win"* from our search query, to ensure that all tweets were relevant to the Android application.

Using the `CLiPS Pattern` module to access Twitter, we ensured that all tweets collected were in English. Second, in order to make the search more comprehensive, we integrated common acronyms and short-hand names of an app (e.g. *"Facebook"* and *"FB"*) into the search algorithms. The acronyms were manually extracted by searching the Internet (e.g. we sent the query *"Facebook acronyms"* to the Google search engine and got *"FB"* as a common short-hand). Finally, we cross-referenced tweet ID between searches to ensure that no duplicate tweets existed in the data collected. We performed the search using this search algorithm every two days to ensure the stable number of *likes* (❤) and *re-tweets* (🔁) starting *Jun* 24^{th}, 2016 for all the 30,793 apps. The resulting data contained 8,349,308 tweets. These tweets were filtered to ensure the accuracy and quality of

the data. The average length of tweets in our dataset is 28 words.

Eliminating duplicate tweets

Duplicate tweets often emerged from the data collected. These duplicate tweets were either (i) one user repeatedly posting the identical tweet, (ii) a user re-tweeting another user's tweet, or (iii) identical tweets posted by two different users. All three types of duplicate tweets were removed from the dataset due to two main reasons.

First, the analysis of the tweets in later sections included the sentiment analysis. Large numbers of duplicate tweets would have skewed the results of this analysis. For example, if one user repeatedly posted the identical negative feedback of an app, and all of the tweets were considered, the app would be deemed to have an overly negative review, despite the fact that only one user had a negative experience and re-tweeted it several times. Second, applying machine learning techniques for tweet classification and later topic modeling is computationally intensive. Duplicate tweets would have hindered the ability to apply these techniques in a timely fashion. Eliminating duplicate content increased the performance of automated techniques without jeopardizing the content analysis, as only one instance of each tweet would remain in our set.

Eliminating irrelevant and spam tweets

As discussed earlier, we sent queries in the form of *"app name" AND "mobile app"* for retrieving tweets about each of the 30,793 apps. However, irrelevant tweets and spam tweets still inevitably emerged in the tweets collected. Irrelevant tweets were tweets that do not pertain to the app itself. These tweets are prevalent in apps with a common name - for example, apps with the name *"Photo Editor"* have a lot of irrelevant tweets as opposed to an app with a unique name such as *"Clash of Clans"*. We filtered out 41 apps with generic names because of the high likelihood of large amounts of irrelevant tweets such as apps named *"Music Player"* or *"One"* from our sample set to mitigate the risk of irrelevancy. For example, the tweet *"Is anyone aware of a music player app for Android?"* may not be talking about the "Music player" app with the package name

`media.music.musicplayer`. To detect apps which are related to a significant amount of irrelevant data (more than 10% of all the tweets), two of the authors manually inspected all the tweets per app and selected the apps to be filtered out from our set.

Furthermore, spam tweets are irrelevant. In defining spam, we used Twitter's official definition of spam, "Spam can be generally described as unsolicited, repeated actions that negatively impact other users[6]". Spam tweets were detected using three methods. First, following observations noted in Guzman et al. [83], we removed tweets where the posting user's name contains the word *"bot"*, as they have a high likelihood of being spam. Second, the tweets were manually observed to pinpoint users that share spam tweets. These users were added to our spam list as spammers [22]. All tweets posted by spam users were deemed to be spam and were removed. Three software engineers (including two of the authors) went through the tweets sorted per user. At least two of the three software engineers checked each user and detected if the user is a spammer or not. Third, it was observed that spam tweets were often detected and deleted after a few days period by Twitter. Following this, we searched for each tweet using its tweet ID after *three* days. If the search request for the tweet failed, it indicated that the tweet had been deleted. Hence, these tweets were excluded from our sample set.

Among the 8,349,308 tweets which we gathered over a six weeks time period, we filtered out 3,481,438 tweets due to irrelevancy, duplication or spam content. Of the 3,481,438 tweets that were removed out of the 8,349,308 collected, 2,868,704 were deleted for being a duplicate, 490,187 were deleted for having 'bot' in the username, and 122,547 were deleted due to being posted by a manually identified spam user. Thus, we continued our analysis with the 4,867,870 remaining tweets.

3.5 Methodology

To approach the stated research questions, we applied several statistical tests and analytical methods to analyze (i) the correlation between the number of reviews and number of tweets, (ii) the

[6] `https://support.Twitter.com/articles/64986`

comparison between the content of app reviews and app related tweets in terms of the insight they provide for app developers, and (iii) the alignment of sentiments between tweets and app store reviews. Statistical methods were done for exploratory purposes. To investigate the availability of extra information obtained from mining tweets, we performed classification and topic modeling of reviews and tweets independently and subsequently analyzed the similarities and differences between the extracted topics. Once we observed that extra development information is available in the tweets, we compared the sentiment, degree of specification and understandability of the tweets and reviews. Overview of the techniques we used to answer each research question is provided in Figure 3.3.

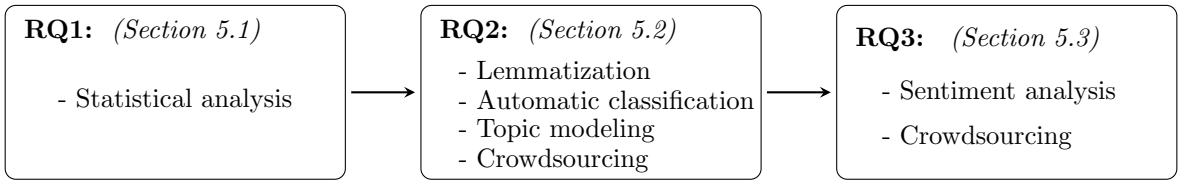


Figure 3.3: Overview of the techniques used to answer each RQ.

RQ1: Statistical Analysis

The investigation involved extensive comparisons between data sets to study correlations and distribution of data. Various statistical methods were used to achieve this. These methods were used to answer **RQ1**. First, the Pearson correlation test [74] was utilized to study the numerical alignment between the time series data of tweets and reviews. We tested the relationship between the number of reviews and the number of tweets using this test.

Second, the Mann-Whitney U-test [74] was used to study the commonalities and differences between two independent sets of non-parametric data. We used the test to compare the distribution of the number of tweets and the number of reviews in **RQ1**.

RQ2: Extracting and Comparing App Review and Tweet Topics

To compare the content of app reviews with the content of app related tweets, we first classified tweets and reviews into different categories, following Chen et al. [42]. We used two different and independent classifiers, *Naive Bayes* and *SVM*. In past works, Naive Bayes performed best to classify app reviews [141] and SVM performed best for software engineering tweet classification [83]. We filtered out reviews and tweets that were classified inconsistently by the SVM and Naive Bayes classifiers. We used the taxonomy suggested by Maalej and Nabil [141] as:

- We were interested in the improvement requests and the evolution of the app (and not the praise, user experience, etc.).
- There is no taxonomy based on tweets and reviews of mobile apps and a more fine-grained

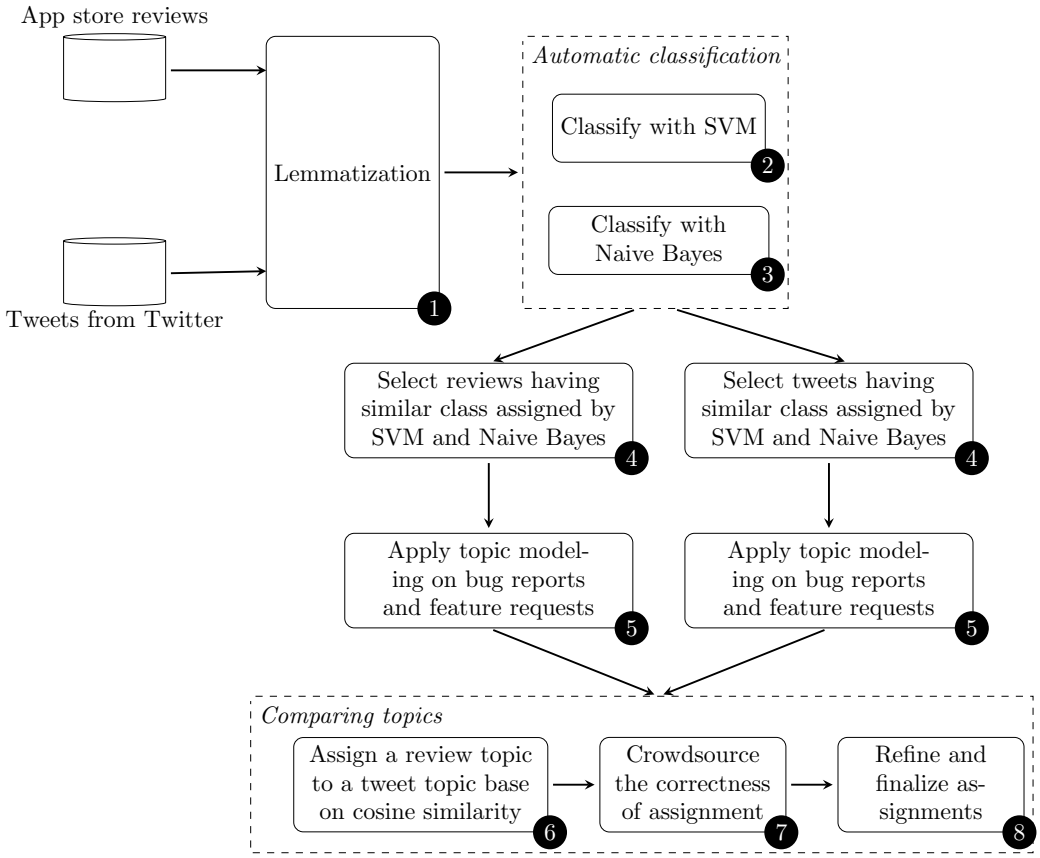


Figure 3.4: Process per app by comparing textual content of reviews and tweets.

taxonomy is biased either toward tweets [83] or toward reviews [54] [53].

Subsequently, we applied topic modeling for tweets and reviews in each category (being a feature request or bug report) [42]. Finally, we calculated the similarity between the topics extracted from tweets and the ones extracted from app store reviews. The process is demonstrated in Figure 3.4. This analysis was applied to 70 randomly selected apps - entailing 358,860 tweets and 1,267,895 app store reviews. The results were used to answer **RQ2**.

Going from analyzing numbers to content, we applied multiple natural language processing and machine learning techniques to analyze app reviews and tweets, as formulated in **RQ2** and **RQ3**. The process overview is shown in Figure 3.4.

We used an open source module, named `Pattern` [248], which is built on top of the `NLTK` comprehensive Python package [139]. Python's `NLTK` has been used in a variety of recent app store mining studies in software engineering [88]. `Pattern` is an open source module that we calibrated, with additional Twitter ids and specified list of top words, and used for the purpose of this study. The program is used for tweet collection via the Twitter API, as well as for sentiment analysis on the tweets collected. `Pattern` is trusted by many academic scholars, as demonstrated by its wide usage [248].

Lemmatization

Having app reviews and app related tweets, we first removed stop words such as "with", "my", "them" from textual data (Step ❶ in Figure 3.4). To do so, we inspected and customized the default set of stop words to keep words such as "should" or "must" as suggested by Maalej et al. [141]. We applied lemmatization to map English words into their dictionary form (also named Lemma). For instance, through lemmatization, we mapped *"helped"*, *"helps"*, and *"helping"* to *"help"*. We chose lemmatization over Porter's stemming [208] as the contexts of the review and tweet are highly important. Lemmatization maps *"worse"* to *"bad"* while the Porter's stemming misses this link [106] [141]

Automatic classification of tweets and reviews

Building on top of past research, we automatically classified reviews and tweets into fine-grained categories based on their content as shown in Step ❷ and Step ❸ of Figure 3.4. We then applied topic modeling on each class separately [42]. Maalej et al. [141] defined four categories for app reviews: bug reports, feature requests, user experience, and rating. In another work, Guzman et al. [83] defined 21 categories for software engineering related tweets after manually analyzing 450 tweets. However, several of these tweet categories are not applicable to app reviews. With the aim of comparing app reviews with app related tweets, we classified reviews and tweets. In a way that each tweet or review belongs to *at most two* of the following categories:

Feature requests, are users' suggestion on functionality or content to enhance the app in future releases.

Bug reports, describe problems with an app such as crashes or performance issues.

Others, this category covers individual and social interactions among users who are looking to share their experience on using the app. This category also encompasses ratings (simple text reflecting numeric rating such as *"great app"*) and advertisements made by users or companies.

Maalej and Nabil [141] compared different classifiers to categorize *app reviews*. They reported that *Naive Bayes* achieves high accuracy with a small training set and needs much less time in comparison to other studied methods. On the other hand and with the aim to analyze *tweets* for software development, Guzman et al. [83] suggested the use of SVM method instead of Decision Trees (C4.5 algorithm) as it has higher precision and ability to filter out irrelevant tweets. In Step ❹ of Figure 3.4 we applied SVM and Naive Bayes separately for reviews and tweets of each app.

Naive Bayes is relying on the strong assumption that existence of a word in the review or tweet is independent of the existence of another word. For example, this would mean that if the tweet contains the word *"tracker"*, this gives us no further information about *"calorie"*. This algorithm is simple and proven to perform well for small training sets on mobile app reviews [141].

Support Vector Machine (SVM) is a universal classifier which finds the decision boundary between each two classes. SVM learns independently from dimensions of the feature space (words in the context of reviews or tweets) [240].

We evaluate SVM and Naive Bayes classifiers in terms of precision and recall for category i (i being *feature request* or *bug report*) as below:

$$Precision(i) = \frac{TP(i)}{TP(i) + FP(i)} \tag{3.1}$$

$$Recall(i) = \frac{TP(i)}{TP(i) + FN(i)} \tag{3.2}$$

$$F1(i) = \frac{2 \times Precision(i) \times Recall(i)}{Precision(i) + Recall(i)} \tag{3.3}$$

TP(i): # of reviews or tweets truly classified as category i.

FP(i): # of reviews or tweets falsely classified as category i.

FN(i): # of reviews or tweets falsely classified as not being category i.

For training and evaluating classifiers, we randomly selected 4,500 tweets and 8,300 reviews across different apps. Three of the authors classified these reviews and tweets manually (with the Cohen's Kappa [29] agreement's degree between 82% and 90%). We used this sample to evaluate our automated classification models applying 10-fold cross validation. The accuracy of

Table 3.1: Precision, recall, and F-score (F1) of Naive Bayes and SVM classifiers separately for tweets and for reviews (average results of 10 time 10-fold cross validation).

Reviews

Classifier	**Feature request**			**Bug report**		
	Precision	Recall	F1	Precision	Recall	F1
Naive Bayes	0.89	0.81	0.84	0.91	0.82	0.81
SVM	0.84	0.79	0.81	0.78	0.73	0.75

Tweets

Classifier	**Feature request**			**Bug report**		
	Precision	Recall	F1	Precision	Recall	F1
Naive Bayes	0.76	0.61	0.67	0.77	0.70	0.73
SVM	0.56	0.50	0.52	0.61	0.54	0.57

Naive Bayes and SVM for tweets and reviews is shown in Table 8.1. In our subsequent content analysis, for the purpose of the high reliability of results, we only considered reviews and tweets that exhibited the same classification. That way, 72.3% of reviews and 51.8% of all tweets were finally considered.

To understand the content of reviews and tweets, we applied topic modeling only to the ones classified as *bug report* or *feature request* as demonstrated in Step ❺ of Figure 3.4. We considered the other classes as non-informative [42] as they likely would not contribute to software development.

Topic modeling and crowdsourcing

We used Latent Dirichlet Allocation (LDA) to extract topics from app reviews and app related tweets (Step ❺ of Figure 3.4). LDA [30] is an established method used to find topics in a set of natural language text documents. Topics are created when words that tend to appear together frequently are found in the documents of the corpus. LDA assumes that there is a fixed number of K topics. It assigns each document a probability distribution over topics. By looking into the words with heaviest weights, we can assign descriptive names to the probabilities assigned to words.

Because the number of topics existing for tweets and reviews is unknown and the content might be unrelated, we extracted topics from reviews and tweets separately. To decide which number of topics K makes the most sense, we varied K and decided based on (i) the perplexity measure, where low perplexity indicates better generalizability of a topic [30], and (ii) intuitive meaningfulness of the results [42]. In this way, we extracted topics from both tweets and from reviews for each app. Across all 70 apps, the number of topics ranged from [0, 85] for reviews and [0, 170] for tweets.

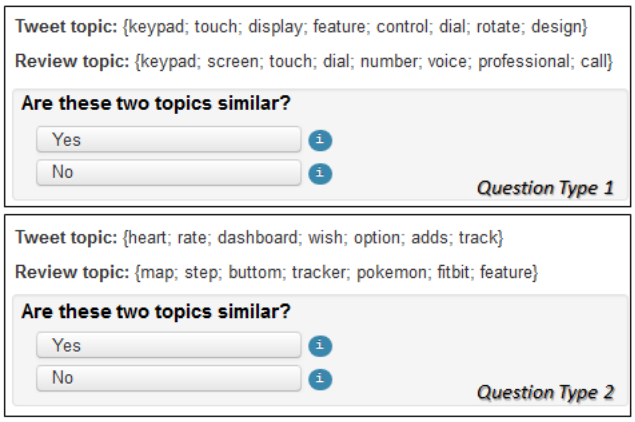
To determine the consistency and inconsistency between any topics discussed in Twitter and Google Play, we measured cosine similarity which was applied successfully in this context before [213]. This is shown as Step ❻ of Figure 3.4. Based on human judgment we determined a threshold value of 0.6. For any pair of tweet and review topic with cosine similarity above the threshold, we considered that topic as a *common topic between tweets and reviews*.

As the final step (Step ❼ of Figure 3.4), to confirm the results of automated topic modelling and its subsequent similarity assignments, we performed an additional two-step evaluation process of human judgment [41]. First, we asked human subjects if they see a similarity between any pair of topics or not. Second, we presented a tweet topic with no assigned review topic and asked if a human subject can find similarity with any other topics (we asked for one-by-one topic comparison for each tweet with no similar review topic). This evaluation was done by crowdsourcing. We asked the following two types of questions via Amazon Mechanical Turk [198]:

Question Type 1: Is a tweet topic similar to its assigned review topic?

Question Type 2: Is a tweet topic, with no similar review topic, related to any other unassigned review topics?

The idea of relying on crowd-based judgment for validating the results of LDA was introduced by Chang et al. [41]. First, we randomly submitted 25% of data for calibrating similarity threshold. We calibrated the threshold of our cosine similarity measure to 0.6 as we found that any lower threshold, resulted in a substantially more inaccurate assignment of topics. Then, we randomly submitted 75% of all tweet and review topics across all apps to 658 crowd workers in a way that at least three workers judge the similarity and dissimilarity of "a tweet topic and a review". We



Did our similar topic assignemnt make sense to humans?

Did we miss to assign a tweet topic to a review topic?

Figure 3.5: Sample questions submitted to crowd workers for evaluation of topic modeling and similarity analysis. Each question was answered by at least three workers.

considered the majority of votes among three as the final decision (democratic voting principle). A sample of questions asked in Amazon Mechanical Turk is presented in Figure 3.5.

We found that our topic modeling has an accuracy of 57.1% *precision* and 51.5% *recall*. Crowd workers also determined similar reviews for unassigned topics for 31.3% of the cases. The precision and recall of our approach are slightly better in comparison to past studies on Twitter [94]. We finally in Step ❽ of Figure 3.4, adjusted our topic assignment using crowd categorization to increase the accuracy of the result for **RQ3**.

RQ3: Comparing Sentiments, Degree of Specification and Understandability of Tweets and Reviews

Having lemmatized texts of the tweets and reviews, we further compare their content with regards to the sentiment, degree of specification and degree of understandability of language and intention.

Sentiment analysis

Sentiment analysis has been studied to characterize the attitude of people writing reviews and tweets [83] [141]. App reviews and tweets carry people's opinion, and opinions reflect people's sentiment. A user's opinion and sentiment could be positive or negative, defined as *polarity*. Tweets and reviews were studied for sentiments from this polarity aspect [141] [155]. For instance, words like *"perfect"* and *"good"* are very positive while the word *"recommended"* is slightly positive. In addition, the sentiment might be factual or opinionated, outlined by subjectivity [133]. For instance, the word *"hope"* is subjective while the word *"scientific"* is objective. Sentiment analysis of tweets from aspects of both polarity and subjectivity is well established [193]. In this paper, we compared both polarity and subjectivity of app reviews and app related tweets. We performed this sentiment analysis to answer **RQ3**.

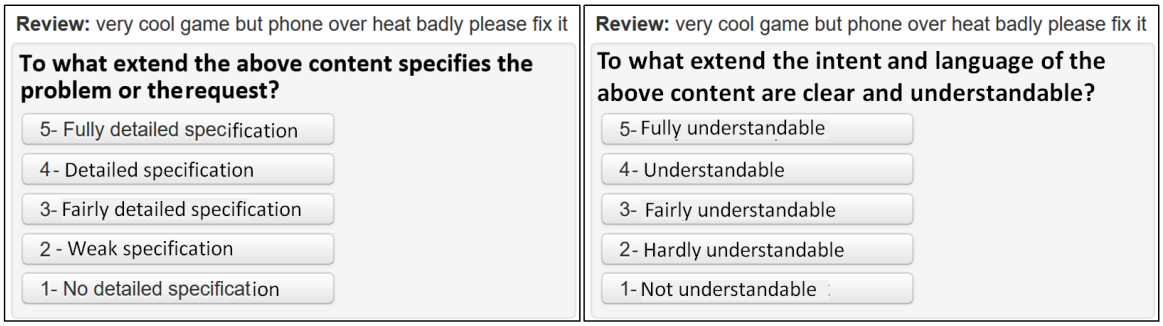


Figure 3.6: Sample questions submitted to crowd workers for evaluating the *degree of specification* and *understandability* of tweets and reviews. Each question was answered by at least three workers.

Crowdsourcing to evaluate the degree of specification and understandability

We used crowdsourcing to evaluate and adjust the results of automatic topic modeling. Further, in

RQ3 We compared the tweets and reviews based on:

- *Degree of specification*: to what extent the content of a tweet or review specifies a problem or request?
- *Understandability*: to what extend the content of the tweet or review is clear and understandable considering the language and intent?

To compare, we performed another round of crowdsourcing. We submitted all of the informative tweets and reviews to the crowd in Amazon Mechanical Turk and asked them to evaluate each tweet and each review separately on a five-point Likert scale. In the design of this evaluation, we provided descriptions and examples to familiarize the crowd with the intention. Each tweet and each review were evaluated at least by three workers based on these two criteria. Figure 3.6 shows sample questions submitted to the crowd.

Each review and each tweet were judged initially by three crowd workers. In case the majority agreed with a particular Likert category, the degree of specification or degree of understandability were decided accordingly. Otherwise, the task was resubmitted up to the time that at least two of the workers vote for the same category. 8.1% of the tweets and 11.3% of reviews were evaluated by more than three crowd workers.

3.6 Results

Results are organized according to the three stated research questions and are presented in subsections 3.6, 3.6 and 3.6. Answering **RQ1** is based on the analysis of reviews and tweets of 30,793 apps. For comparing their content (**RQ2**) as well as for comparing sentiments and other characteristics of app reviews and tweets (**RQ3**), we randomly selected a sample of 70 apps. This sample size is comparable to similar investigations done in this context [155].

RQ1: When Considered Over a Period of Time, do the Number of Tweets and the Number of Reviews Has Correlation and for Which Apps the Correlation Is Stronger?

For a total of 30,793 apps, we calculated the correlation between the number of reviews and number of tweets when looking at them over a period of six weeks. This initial investigation was aimed to explore if the app related tweets are related in terms of amount and trend with reviews of the same app. We want to know for which apps the relation between a number of reviews and number of tweets is stronger. We performed this comparison for different top charts apps and for apps having combinations of high or low ratings with the high or low number of reviews [153]. We tested the following null hypothesis using Pearson's correlation:

H_0: *Considered over time and per app, there is no correlation between number of user reviews and number of user tweets.*

The null hypothesis was rejected for 98.6% of the apps. The relationship between the number of tweets and number of reviews for all top chart apps (separated by categories) is visualized in Figure 3.7. The average Pearson r effect size (with 0.1 being small, 0.3 being medium, and 0.5 considered of being large) of comparing tweets and reviews for *top free* apps was 0.46, for *top paid* was 0.67, for top free games was 0.54, for top paid games was 0.81, for top grossing was 0.18, and for top grossing games was 0.23. We can see that apps in the top free category have a

higher correlation in comparison to other top chart apps. *Top free games* have the second highest correlation compared to other top chart categories.

The results showed a very high correlation between apps outside top charts ($median = 1$ for all of non top chart apps). For most of the apps outside top charts, the high correlation appears as the app did not receive any review or any tweets. In other words, the daily variance of tweets and reviews with six weeks of our study is very low. We compared the distribution of the number of tweets and number of reviews for apps outside top charts. The results of Mann-Whitney test showed that the distribution of tweets and reviews for apps outside top chart are not significantly different ($p - value = 0.748$). As a result, we selected a random sample of apps for the next two questions considering the top chart mobile apps.

RQ2: What Type of Information Can Developers Gain from Tweet Analysis?

In Figure 3.8 the distribution of the number of feature requests, the number of bug reports and number of other types of tweets and reviews are compared between tweets and reviews. Looking into the category of informative tweets and reviews for the 70 apps, we found that 45.6% of the

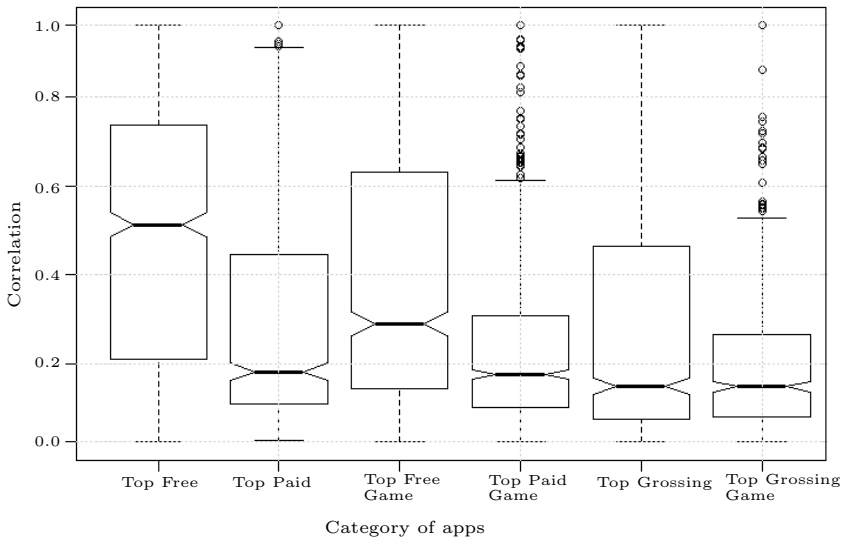


Figure 3.7: Correlation between number of reviews and number of tweets for different top chart categories.

tweets are not reporting any bugs or requesting features in comparison to the 39.6% of all the informative reviews not reporting a bug or requesting a feature. However, the reviews are mainly bug reports (41.1%) while 36.9% of tweets are dedicated to this category. We observed that 17.5% of informative tweets and 19.3% of informative reviews are *feature requests*.

We found that in 97.1% of the cases (68 out of 70 apps), mining tweets would provide complementary information to developers about users' feature requests and bug reports. In the rest of cases, we found more than 65% commonality between the feature requests and bug reports in Twitter and app store. Figure 3.9 compares the number of tweets and reviews from Twitter and app store for all the apps. Among all the 70 apps we analyzed, we found a total of 198 feature requests and 246 bug reports in the app related tweets that we did not find in the app store reviews. We observed that the feature requests in users' tweets have a wider scope than the ones articulated in app store reviews. Bug reports appear to be more detailed in reviews, often specifying the device and Android version that they catch a crash on it. Figure 3.9 shows the final results of our study on the content of the 70 apps. We found that on average across all the apps 40.78% of feature requests and bug reports are common between app related tweets and app store reviews.

Again, we provide more detailed information for one of the apps introduced in the motivation of the paper. For the Evernote app, we extracted 45 bug reports and 39 feature request from reviews as well as 35 bug reports and 47 feature request tweets. 14 bug reports and 11 features were in common between reviews and tweets. In other words, within six weeks of our analysis, we found

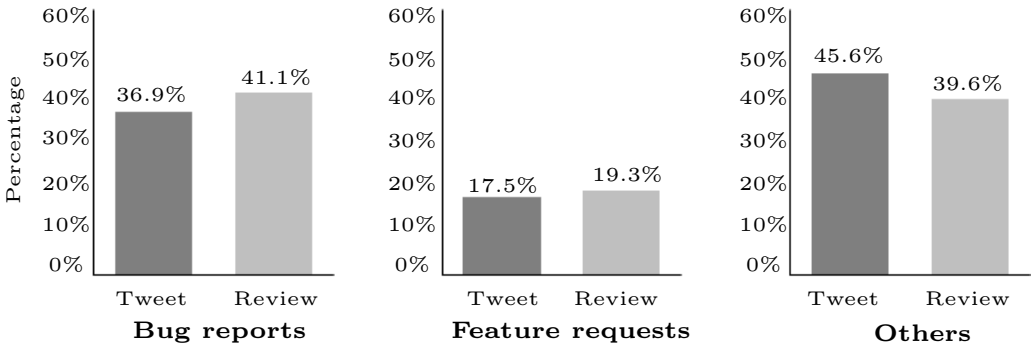


Figure 3.8: Comparison of informative reviews and tweets for the 70 apps. The comparison shows the percentage of tweets and reviews in each category.

28 feature requests and 31 bug reports in Evernote tweets that were not stated in the app store reviews. Below we provide *a sample* of bug reports and feature requests unique in reviews and tweets and in common between both reviews and tweets for Evernote app:

App store reviews:

- *Need support for mathematical equations.*
- *Ability to hand-write on an inserted photo.*
- *In Jellybean, camera option is not working.*
- *It won't always allow me to check items off in a checklist.*

Tweets:

- *I need PIN Lock Feature on mobile.*

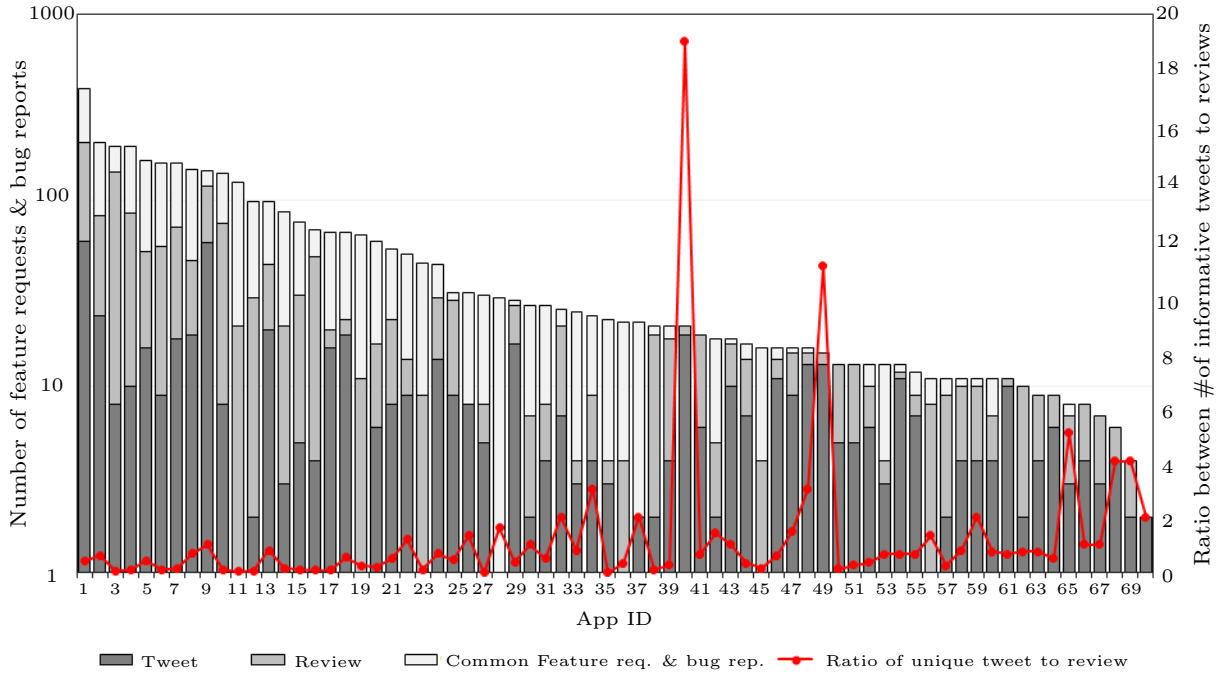


Figure 3.9: Number of topics found after classification, topic modeling and crowdsourcing evaluation. Topics are grouped into (i) Twitter unique, (ii) Google Play unique, and (iii) common topics. The line chart on the second y-axis represents the percentage of tweet unique topics detected.

- *Rhyming feature designed for Songwriters.*

- *My smart watch app doesn't work when this is running*

- *It's slow as death. I type a letter a minute later it appears*

▶ & 🐦 **Mutual between tweets and reviews:**

- *Integration with SwiftKey keyboard.*

- *Add reminders to the shortcut section.*

- *Can't the camera widget zoom.*

- *Can't access the PDF normally when using the app.*

Considering all the tweets and reviews we analyzed, we mined 1,064 feature requests and 2,277 bug reports in total. Among them 22.4% of feature requests came from Twitter *only* and 34.09% from app store reviews *only*. Also, 12.8% of bug reports were mined from Twitter *only* while 30.59% *only* came from the app store. 43.51% of feature requests and 56.61% of bug reports were in common between Twitter and Android app store. While in two cases (Apps #27 and #35 in Figure 3.9) we could not gain extra information from mining tweets, in four apps with no informative reviews (Apps #26, #28, #37, and #70 in Figure 3.9) we mined several feature requests and bug reports from mining tweets (see Figure 3.9). We selected the diversified apps across top charts, and the results showed that we find valuable development information for the majority (97.1%) of these apps (including game applications) by mining Twitter.

We did not find any significant difference between paid and free apps in terms of ratio between #of informative tweets to reviews using Mann-Whitney test. We also did not find any significant difference between game and not game apps in terms of ratio between #of informative tweets to reviews.

RQ3: How do app store reviews and app related tweets compare with regards to sentiments, degree of specification and understandability?

We focus on 70 randomly selected apps for this analysis. Sentiment analysis was performed from both polarity and subjectivity perspectives. The distributions of review and tweet sentiments for the two sample apps (Pokemon Go and Evernote) mentioned at the beginning are given in Figure 3.10. While we have fewer tweets than reviews for both apps, the scatter plots of review and tweet sentiment are largely the same. Both reviews and tweets are rather subjective (≥ 0.5) and have slightly more positive (≥ 0) sentiment. However, if we look closer into the middle charts in Figure 3.10, Evernote app has more objective and positive tweets in comparison to its reviews.

To test the difference between review and tweet sentiment of all apps, we compared polarity and subjectivity of reviews and tweets by using the Mann-Whitney test. We tested the following null hypotheses:

$H_{0\_Polarity}$: *There is no difference in the polarity distribution (over time) of tweets and reviews.*

$H_{0\_Subjectivity}$: *There is no difference in the subjectivity distribution (over time) of tweets and reviews.*

Running the Mann-Whitney test, we rejected both null hypotheses $H_{0\_Polarity}$ ($p\text{-}value = 0.021$) and $H_{0\_Subjectivity}$ ($p\text{-}value = 0.004$). The degree of polarity and subjectivity for all apps is shown in the left column of Figure 3.10. Looking into these charts for a particular app, tweets are usually less positive and less subjective in comparison to app store reviews. The most right charts of Figure 3.11 show the difference between overall polarity and subjectivity of tweets and reviews.

We also compared the degree of understandability and degree of specification of all the informative reviews, and that of all informative tweets. The results of this comparison are presented in Figure 3.11. The bigger portion of reviews was evaluated as a fully detailed specification in comparison to the tweets (32.3% versus 25.8%). However, comparing the least specific category, the bigger portion of reviews were judged as they have "no detailed specification" (24.1% vs. 19.6%). Looking into the reviews and tweets classified between three (detailed specification) to five (fully

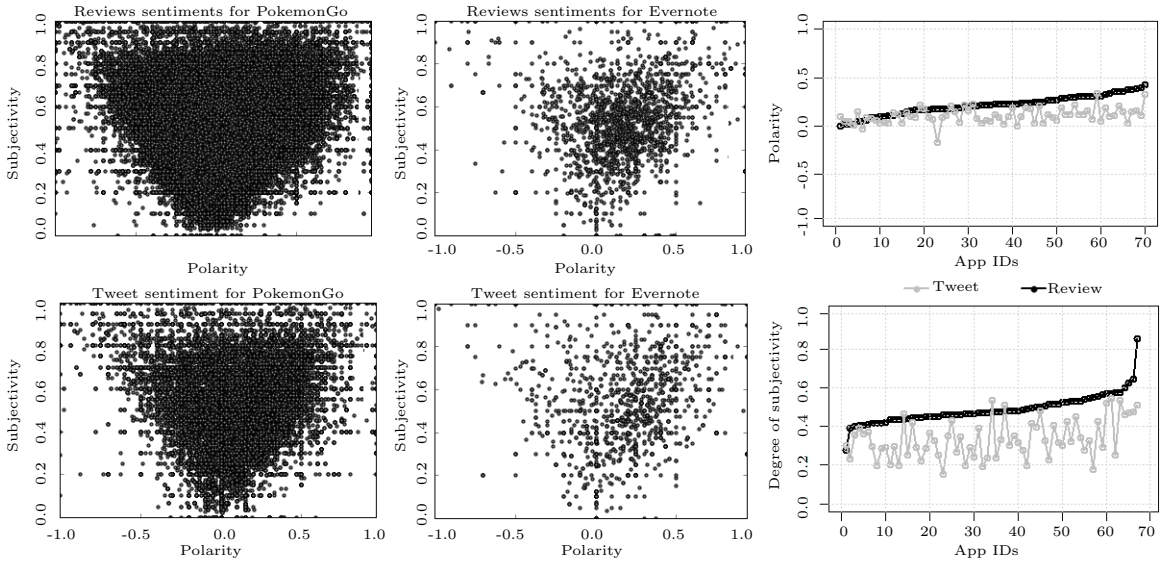


Figure 3.10: First two columns are the comparison of polarity and subjectivity between tweets and reviews for Pokemon Go and Evernote reviews and tweets. The last column demonstrates the comparison of sentiments between tweets and reviews of an app - The most right side charts compare the polarity (top) and subjectivity (bottom) of tweets and reviews for 70 apps.

detailed specification), reviews are *slightly* more specific in defining the problem or the request as 66.3% of reviews was judged as having detailed specification or better while this includes 63.1% of the tweets. Also, our analysis did not demonstrate any significant difference between paid and free apps nor between game and non-game apps in terms of polarity and subjectivity of tweets in comparison to reviews.

Looking into the degree of understandability and clarity of the intent and language, reviews and

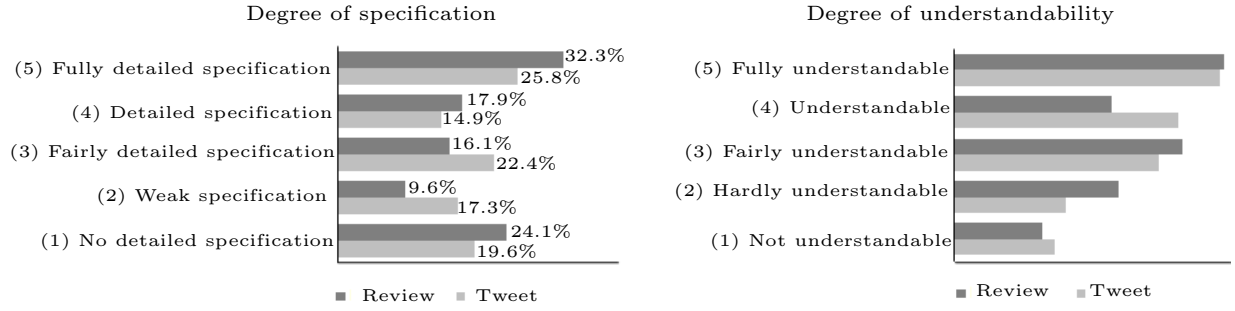


Figure 3.11: Comparison between tweets and reviews based on the “degree of specification” and the “degree of understandability”.

tweets are almost judged equally clear (29.8% vs 29.3%). On the other hand, more tweets considered not understandable at all (11.1%) in comparison to the reviews (9.7%). 72.2% of reviews and 76.6% of tweets were considered understandable, fairly understandable or fully understandable. We speculate that the length limitation of the tweets made the users communicate their intention right on to the point. Our analysis did not demonstrate any significant difference between paid and free apps nor between game and non-game apps in terms of degree of understandability and degree of specification for tweets in comparison to reviews.

3.7 Threats to Validity

As with any study, there exist threats to the validity of our results. We discuss these threats following the categorization which is given by Wohlin et al. [271]:

Conclusion validity- Are we drawing a right conclusion about treatment and outcome relation?: The conclusions made were solely based on apps and related reviews and tweets taken from the Google Play (Android) app platform within six weeks period. App store reviews were collected from the Google Play app store only. Furthermore, tweets from Twitter were searched to specifically target tweets about the Google Play version of the apps. We excluded tweets that pertain to alternate stores through taking out tweets with words such as iOS, iPhone, etc. However, a tweet from a non-Android user might have occurred in our sample if the user did not use any of the mentioned keywords. We also analyzed apps within a limited time frame and compared tweets and reviews only for this period. However, a feature may have been requested or a bug reported in app store prior to this time frame and before the tweet occurred. We argue that 26.2% unique feature requests and bug reports on average over six weeks period is not trivial and not by accident. Also, 42.8% of the apps in our study have been released for the first time or at least had one release within the six weeks period. For more comprehensive analysis the evaluation of data further than six weeks period could be considered.

Taking intersection between two classifiers may potentially take out informative reviews and

tweets and affect the conclusion. However, F1 for reviews is high by both classifiers (See Table 8.1). In addition, we performed a manual inspection on this subject for the 70 apps but did not find any such case which affects the results of the study.

Another threat refers to the trustworthiness of crowd evaluation. We used our experience in designing multiple crowdsourced surveys [182] [186] and adopted the guidelines of Kittur et al. [123] for designing crowd surveys. To test the accuracy of the crowdsourcing in **RQ2**, we randomly selected 500 tweet and 500 review topics. Among them, 250 topics were marked as similar and 250 topics were marked as different by the crowd. We asked three app developers (with whom we have formerly worked with) to manually label these 500 topics.. They answered the same questions as the crowd did (see Fig 5). We compared the results and found that developers classified 99.4% of the topics in the same way as the crowd did In **RQ3**, we used crowdsourcing for defining the degree of specification and understandability. In addition, we also ran a small experiment to understand the degree of difference between surveying an unknown set of developers (crowd) and a set of known developers. We randomly selected 250 tweets and 250 reviews and asked three developers to identify the degrees of specification and understandability. Comparing the results of crowd classification with the classification made by the developers showed a maximum difference of 3.4% for the degree of specification and a maximum difference of 3.6% for the degree of understandability.

These threats are considered non-critical for the type of investigation made in this paper. The investigations were targeted at a proof of the existence of additional information and insight. Restricting samples is not contradictory to that. It only means that we might have even missed information available.

Construct validity - Are we measuring the right things? Data for this study was collected over a six week time period. Thus, for apps that were released to the market in the middle of our data-collection cycle or occurred in a top chart some time in between, the collected data does not span the full six-weeks. However, we selected apps for content analysis considering different amounts of the ratio between # of tweets and # of reviews. In this way, if an app (such as Pokemon Go)

occurred at the end of our third week we still included it in our study due to huge number of reviews and tweets.

We introduced two additional measures to increase construct validity. First, from performing two independent classifications of both reviews and tweets, we have a higher reliability that the content of the messaging is indeed related to the category it is assigned to (this is highly important for tweets). Second, from topic modeling and comparison of all topics extracted, we have increased the validity of comparing the actual content. When the majority of our crowd workers detect a mismatch or found a newly matched topic, we tuned the results of machine learning. However, we expect that by altering the topic modeling technique or defining a more fine-grained classification, the accuracy of the machine learning model would become higher.

The use of sentiment analysis tool might risk the results [108]. This potentially may impact the results of **RQ3** in our study. We carefully analyzed this impact by comparing `Patter` with `Stanford NLP` [145] and `Vader` [104]. Testing the $H_{0\_Polarity}$, using each of these libraries showed significant difference between reviews and tweets.

Internal Validity - Can we be sure that the treatment indeed caused the outcome? As we ensured that the reviews and tweets were explicitly related to the apps selected, the risk of violating internal validity appears to be low. Of course, number and sentiment of messages can be influenced by other factors. We would argue that the number of apps and messages studied exclude the random influence of this phenomenon. Despite the rigorous analysis there is still the risk that some of the existence of spam and irrelevant tweets remained in the sample. To scope the extent of remaining spams in our data we randomly sampled 250 spam tweets and 250 of non-spam tweets, and we asked three developers to read the tweets and label them as spam/not spam. Then, we compared the results between developers' tagged spams and filters we described in Section 3.4. The results showed that developers detected 257 non-spam tweets and we filtered out tweets that were not spam, however; the difference is about 2.8%. We believe that few spam/not spam tweets on one app will not alter the accuracy of our results.

External Validity - Can the results be generalized beyond the scope of this study? All the inves-

tigated apps and related reviews and tweets were randomly taken from Google Play (Android). A generalization of results (getting additional insight) to other social media is outside consideration (even though considered to be likely). A generalization to Twitter seems possible as we showed already for the given sample set of apps that substantial additional insights can be gained. We also observed that the number of tweets and reviews for the majority of apps outside the top chart is low over the considered period of six weeks. Apps must possess a certain degree of visibility and popularity for users to review and tweet about the app. Thus, developers of apps with little or even no client-base will likely find it difficult to utilize tweets to support their development process.

3.8 Discussion

We selected the taxonomy suggested by Maalej and Nabil [141] for the purpose of this study. Maalej and Nabil categories each review into *feature request*, *bug report*, *user experience*, or *other*. We used this high-level taxonomy as a more fine-grained taxonomy is either based toward tweets [83] or reviews [44, 54]. In this section we discuss why the use of the existing finer grained taxonomies does not suit the purpose of this study.

We compared the review taxonomies suggested by Di Sorbo et al. [53] [54], and Ciurumelea et al. [44] and the tweet taxonomy suggested by Guzman et al. [83]. We took two arguments (called Argument 1 resp. 2):

Argument 1: We compared the tweet taxonomy [83] of software products with app reviews [44].

Argument 2: We analyzed $7,233$ Tweets using SURF as suggested by the reviewer. This sample covers all the tweets of Evernote, Dropbox, and FaceTune.

Argument 1- Two of the authors separately compared the taxonomy of tweets provided by Guzman et al. in [83] with the taxonomy of reviews reported by Di Sorbo et al. [54] and by Ciurumelea et al. [44]:

- **Comparing tweet taxonomy [83] with review taxonomy by Di Sorbo et al [54]:** We could find only one commonality between these two taxonomies which is "pricing". While tweet taxonomy has separate "Feature shortcoming", "Feature strength", "Feature requests", review taxonomy by Di Sorbo et al. has "feature and functionality". Also, review taxonomy by Di Sorbo et al. suggest "improvement" which could be either "bug report" or "feature request" considering the tweet taxonomy. The rest; 21 categories of tweet taxonomy (80.7%) and nine categories (75%) of Di Sorbo et al.'s review taxonomy; are not explicitly associated.

- **Comparing tweet taxonomy [83] with review taxonomy by Ciurumelea et al. [44]:** We could map 2 of the categories "hardware" and "pricing". We did not find any explicit association between the rest of the taxonomies (92.3% of tweet taxonomy and 83.3% of Ciurumelea et al. review taxonomy).

- **Comparing review taxonomy by Di Sorbo et al. [54] with review taxonomy by Ciurumelea et al. [44]:** We found that there exists a wide array of differences in the taxonomies. Comparing review taxonomies suggested by Di Sorbo et al. and Ciurumelea et al. we only found three categories "app", "GUI" and "Pricing" common between two taxonomies (25% compatibility).

As the result of this inconsistency we used a higher level taxonomy to observe if extra information exists in app relate tweets or not. Future studies should further focus on the taxonomy that can fit both reviews and tweets of mobile applications.

Argument 2- We used SURF tool which was designed based on Di Sorbo et al. [54] taxonomy to categorize in total $7,233$ Tweets. We analyzed the accuracy of the results manually. The tweets were compiled into a compatible XML file, and the SURF script was run to produce the output XML file. Afterward, we assigned a True/False classification for the assignment of the category done by SURF and made a holistic judgment regarding the validity of its use. We found that (i) SURF categorized 6,187 of the tweets we considered informative as irrelevant and ignored them, and (ii) For the rest of the 1,046 tweets, SURF misclassified 62.3% of tweets as it was tailored and

tuned for reviews (and not tweets). Table 3.2 shows the status of what we found through utilizing SURF with tweets from the Evernote, Dropbox, and Facetune apps.

We often found that the tweets do not belong to any of these categories. A prominent example of this is the pay-wall that Evernote implemented that disables users from having more than two accounts without a paid account. SURF categorized most of the complaints regarding the pay-wall as a bug while it is originally a request to get back a feature. The results showed that development information in tweets is different from the one obtained from reviews. Potentially, an app developer can benefit from looking into sources of users feedback outside app stores. While at this stage we could not generalize the difference between the information inside and outside the app stores, it appears that there exist useful and complementary user reviews outside the app stores that helps in the evolution process of an app.

Also, analyzing users who prefer tweeting to writing app reviews and vice versa could help in understanding users' need better. We analyzed the usernames related to tweets and reviews of the 70 apps by adopting the heuristics suggested by Wiese et al. [270] and found 181 similar usernames (screen names) between Twitter and Google Play. We then manually looked into the Twitter profile and Google Play of users in case they were publically available. We also checked their personal website or LinkedIn profiles manually in case available. Among these 181 users:

- We could not verify the similarity for 103 users: In this case, either two of the fields "name", "last name", "gender", "age", "occupation", or "location of the person" was totally different between profiles. So we can not say the person on Twitter was the same one on Google Play.

- We could not access the profiles of 74 users: In this case, the user has either deleted or protected her account in Google Play or Twitter. So, we could not check the identity of the user. This is mainly happening as we do this analysis retrospectively and after some delay from the initial study.

- We could verify that four users have been commented on both Google Play and Twitter: among these, one user has commented on the same app and three users have been commented

on different apps in Google Play and Twitter.

3.9 Conclusions and Future Work

Understanding and fulfilling users' need is the ultimate goal of app software development which is mainly achieved by providing new features, revising features, fixing bugs, and improving the quality of users' experience with the software. In support of this development and for prioritizing change requests, an increasing number of research studies and businesses have been formed around mining app reviews. The key idea of this paper is to improve this support by looking "beyond the fence". We analyzed the possibility to combine information sources for supporting development decisions. The analysis was done in a three-staged process: (i) studying the alignment between number of tweets and reviews over time, (ii) comparing the content of tweets and reviews with regards to development information using a variety of machine learning techniques and crowd-sourcing, and (iii) similarity and differences between user sentiments on app store reviews and tweets as well as understandability and degree of problem or request specification.

The results of this study imply that the research community should look into the additional sources of information while performing empirical studies on users feedback for app evolution. Our results also showed that App developers could find more information about users' requests by mining social media. Comparing developers' perception along with the actual evidence, we found as well that evaluating the impact of reviews on release decisions for mobile app evolution is a possible future direction of research. The same is true for comparing the posts of users within Twitter and within app store to understand the characteristics of media and content of requested changes.

Table 3.2: Results of classifying tweets with a tool and taxonomy provided by Di Sorbo et al. [54] for reviews

. Evernote app

SURF Category	Total # of Tweets	# of Correctly Classified Tweets	% of misclassified Tweets
App	121	55	54.5%
Company	4	0	100.0%
Contents	41	22	46.3%
Download	13	2	84.6%
Feature/Functionality	140	61	56.4%
GUI	29	14	51.7%
Improvement	15	9	33.3%
Model	54	28	48.1%
Pricing	26	15	42.3%
Resources	1	1	0.0%
Security	14	2	85.7%
UpdateVersion	19	11	42.1%

DropBox app

SURF Category	Total # of Tweets	# of Correctly Classified Tweets	% of misclassified Tweets
App	121	44	63.6%
Company	17	4	76.5%
Contents	26	5	80.8%
Download	10	0	100.0%
Feature/Functionality	157	50	68.2%
GUI	40	15	62.5%
Improvement	13	1	92.3%
Model	52	22	57.7%
Pricing	12	4	66.7%
Resources	8	0	100.0%
Security	13	2	84.6%
UpdateVersion	15	9	40.0%

FaceTune app

SURF Category	Total # of Tweets	# of Correctly Classified Tweets	% of misclassified Tweets
App	20	4	80.0%
Company	1	0	100.0%
Contents	5	1	80.0%
Download	4	1	75.0%
Feature/Functionality	19	5	73.7%
GUI	16	4	75.0%
Improvements	1	0	100.0%
Model	7	1	85.7%
Pricing	11	2	81.8%
Resources	0	0	0
Security	1	0	100.0%
UpdateVersion	0	0	0

Chapter 4

Mining Treatment-Outcome Constructs from Sequential Software Engineering Data

Authors: Maleknaz Nayebi; Guenther Ruhe; and Thomas Zimmermann TSE 2017

Abstract - Many investigations in empirical software engineering look at sequences of data resulting from development or management processes. In this paper, we propose an analytical approach called the Gandhi-Washington Method (GWM) to investigate the impact of recurring events in software projects. GWM takes an encoding of events and activities provided by a software analyst as input. It uses regular expressions to automatically condense and summarize information and infer treatments. Relating the treatments to the outcome through statistical tests, treatment-outcome constructs are automatically mined from the data. The output of this process is a set of treatment-outcome constructs. Each treatment in the set of mined constructs is significantly different from the other treatments considering the impact on the outcome and/or is structurally different from other treatments considering the sequence of events. We describe GWM and classes of problems to which GWM can be applied. We demonstrate the applicability of this method for empirical studies on sequences of file editing, code ownership, and release cycle time.

This paper has been submitted to Transaction of Software Engineering journal (TSE). This is a joint paper of ***Maleknaz Nayebi***, ***Guenther Ruhe***, *and* ***Thomas Zimmermann*** [1].

[1]*First and second authors are affiliated with SEDS lab at University of Calgary, & the third author is with Microsoft Research

Software development deals with a large amount of data created from sequential activities, events, and decisions. Many investigations in empirical software engineering look sequences of events as the independent variables and analyze the related outcomes. With the increasing amount of results gained from empirical investigations in software engineering, there is a substantial need to structure and synthesize this knowledge. We propose a method for empirical studies where the ”cause constructs” [271] are ”sequences of activities, events or decisions”. In this context, we extract treatments that are significantly related to an outcome by mining a dataset.

Developers change source code, perform code reviews, commit code, run builds and test cases, and release software iteratively. Customers use the software, navigate through features, and submit bug reports or feature requests. Many of the questions that software engineers have are related to such sequence of events, activities, and decisions. Often the impact of the activities, events and decisions are studied empirically. As a result, questions like “Should developers commit code before reviewing the changes” or “should they review changes before committing the code?” are asked and investigated empirically [221, 222]. Therein, commit-then-review and review-then-commit represent sequences of review and commit activities.

The experiment principle [271] is shown in Figure 4.1. For performing an empirical study in software engineering, we first form a hypothesis around cause and effect. Then, we test that hypothesis against our observations by conducting an experiment (see Figure 4.1). In this paper,

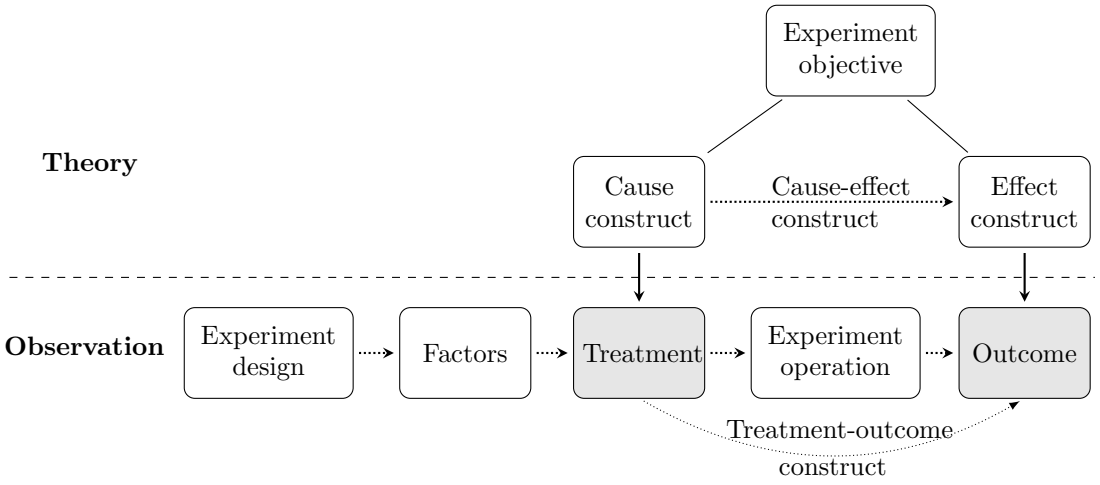


Figure 4.1: The principle and process of experimentation adopted from Wohlin et al. [271].

we propose a method exclusively for testing hypothesises that considers sequence of activities as the treatments. Treatment is often referred to an intervention which is a method or an independent variable that cause some measurable factor to change [122]. The results of our method are treatments that had a significant impact on a specified dependent variable being know as the outcome [271]. Having a unified *methodology* for mining and analyzing the impact of event sequences is the objective of this research.

We introduce an analytical approach called the Gandhi-Washington Method[2] or shortly GWM. The basic steps of GWM can be summarized into three phases:

1. **Encoding.** A software analyst (or engineer) encodes a sequence of events (we later refer to them as items) into a sequence of characters taken from an alphabet of her choice.

2. **Abstraction.** An automated step that summarizes the encoded sequences by using regular expressions.

3. **Synthesis.** An automated step that applies statistical tests on the regular expressions. This step merges the categories that have insignificant relationship with the specified outcome.

t

The output of GWM is a set of treatment-outcome constructs [271] or in short *TrOC*. TrOCs are regular expressions that show the condensed sequence of events (treatments). These regular expressions are correlated with a specified dependent variable (outcome). GWM is a semi-automated approach. Once an analyst encodes the problem and selects the desirable performance measures, a series of statistical tests automatically retrieve treatment-outcome constructs (TrOCs).

For example, an analyst hypothesized that "the sequence of commit and review activities is significantly correlates with the number of bugs in the code" (the cause-effect construct). She designed an experiment and observed the relation between two treatments "commit-then-review"

[2]The name is inspired by the homological diversity and discipline that Gandhi and Washington both stood for. This is a metaphor for the alternative structuring and occurrence of events for developing a software product.

and "review-then-commit" and defined outcome as the "number of bugs". She made the treatment-outcome construct that "review-then-commit is correlated to the significantly less number of bugs comparing to the commit-then-review sequence". However, by having GWM, the analyst only needs to encode the activities commit and review and provides a dataset of the experiment. GWM then automatically mines treatments and their relation with the outcome.

The contribution of this paper is the Gandhi-Washington Method. GWM is a general approach to analyze the relationship between sequential events in software processes or products and an outcome. GWM can be applied to a large class of software engineering decision problems related to sequences of events. In empirical software engineering, researchers often evaluate alternative sequences and measure the related outcomes. GWM facilitates this process by (i) providing a unified method to mine treatments from data (ii) synthesizing treatments based on the impact on a specific outcome (iii) make the process easily replicable. In Figure 4.1, the role of GWM in the process of running empirical studies is illustrated. To apply GWM for a new decision problem, a developer or analyst only has to model the problem by adjusting the encoding and the rest has the potential of automation and could be supported by the tool that is openly accessible.

In this paper we first describe GWM. Then, to illustrate its wide applications, we provide three case study examples from different contexts: file editing patterns, file ownership patterns and release cycle time patterns. Within each example, we show the new results that were achieved by using GWM.

In the next section we provide two example scenarios to motivate the discussion. In Section 4.2, we describe the details of GWM, followed by Section 4.3 in which we discuss three applications of GWM. Prototype tool support is the content of Section 5. Implementation aspects such as multiple encoding and computational complexity are elaborated in Section 4.5. Some limitations of the applicability of GWM are discussed in Section 7. In Section 4.7, we present related work and compare existing sequential data analysis methods with GWM. Finally, we conclude the paper in Section 4.8.

4.1 Motivation

We present two motivating scenarios to explain the problem under consideration. The first scenario relates to the relationship between code healthiness and code review practices. The second scenario looks into the relation between the number of conflicts and sequence of code verification periods.

Scenario #1: Project manager Alice is looking into ways to enhance the healthiness of the code developed by her team. The team doesn't have a unified way for committing, testing and reviewing the code. Some of the sub-project teams follow commit-test-review process, while the others apply review-test-commit, and the rest do not exclusively follow either of these strategies to review their code. Alice wonders if the selection of one of these strategies affects the number of code bugs. To answer this question, she selects GWM and encodes each method for testing and reviewing a file by the sequence of testing (T), committing (C), and reviewing (R) activities on it. She also selects "number of bugs" as the outcome.

Cause-effect construct: sequence of activities for testing, committing, and reviewing the code has impact on the number of code bugs.

Outcome: number of bugs.

Treatments: TCR, TRC, CRT, CTR, RTC, RCT.

Applying GWM, she observes that pieces of code which are alternating CRT sequence for commit, test and review have significantly fewer number of bugs compared to the other approaches.

Scenario #2: Product manager Bob wants to define a cycle for periodic verification of the code's trunk branch for their repository [170]. In this strategy, a number of commits are integrated to the trunk branch and after a period of time the product is built and later verified. The time between two verifications is defined by the teams. The product manager, being afraid of large merge conflicts, defines a fixed verification period of two days. Currently, he wishes to delay the

Table 4.1: Applying Gandhi-Washington Method on Scenario #2.

Step 1: Encoding (Analyst)	
What to encode?	The length of verification periods.
How to encode?	Discretize the length into short (S), medium (M) and long (L) periods.
What to select as performance measure?	Number of conflicts.
Sample encoded periods (and their number of conflicts)	*SSSSSS* (23), *LLLLLLLLL* (47), *MMSMMS* (73), *SSSSLLLLLL* (52), *MSLMSS* (66).
Step 2: Abstraction (Automated step in GWM)	
What to abstract?	Abstract the encoded verification periods.
How to abstract?	Use regular expressions to classify encoded strings.
Sample abstracted strings	S^*, L^*, $(M^*S)^*$, S^*L^*, $(MS^*L^*)^*$.
Step 3: Synthesis (Automated step in GWM)	
What to synthesize?	Classes of regular expressions.
How to synthesize?	For each regular expressions compare mean of conflicts by applying Mann–Whitney test.
Sample output of GWM	Set of TrOCs (such as L^*, which represents consecutive long verification periods) indicating unique sequences of events with differing mean number of conflicts.

verification by three more days due difficulties in running the example project. Bob is concerned with the impact of this unscheduled variation and thinks that the conflicts would be increased significantly. In other words, he assumed that a fixed and very short verification period decreased the number of conflicts in the code. Bob collects the data from all the projects in his company and perform experiment.

Scenario #2 - traditional design: Bob assumes that sequence of fixed and very short verification periods significantly decrease the number of conflicts (cause-effect construct). He defined two treatments. First, sequence of fixed and very short verification periods and second, all the other variation of the verification periods. He divide the data into two subsets; first group the projects that had fixed short verification periods and second group the data of all the other projects. He then compares these two groups by running the Mann–Whitney test on the number of conflicts in each dataset.

Scenario #2 - using GWM: Bob discretizes the verification periods into short (S), medium (M), and long (L). Bob applies GWM as shown in Table 4.1 to check the correctness of the cause-effect construct. GWM shows that L^* (consecutive long verification periods), $(SL)^*$ (consecutive pairs of short and long release cycles), and all the rest of processes $(S^*M^*L^*)^*$ has significantly different impact on the number of conflicts.

Cause-effect construct: length and variation of verification periods has impact on the number of conflicts.

Outcome: number of conflicts.

Treatments: all the variations and combinations of S, L, M.

Having GWM at hand, Bob has more flexibility in two ways. First, he can encode verification periods more diversely. So he can introduce a medium verification period as well. Second, he does not need to pre-assume the existence of two treatments. By using GWM Bob can explore a range of treatments.

In both scenarios, the analysis is concerned with the sequence of events that are happening in the project and how these sequences relate to the specified dependent variable (outcome).

4.2 Gandhi-Washington Method

In this section we introduce the Gandhi-Washington Method (GWM). We define *items* as the basic information for our mining process. *Items* are software related activities and events that are stored as transactions in the software repositories. *Itemsets* are groups of items that occur together in sequence and are grouped by a time-stamp [143]. Within this paper, as long as we are discussing itemsets, we are referring to *ordered* itemsets. In the context of GWM, treatment-outcome constructs (TrOC) are recurring sequences of events that are statistically related to software product measures (dependent variables).

GWM uses the notion of regular expressions and their grammatical relation within a formal language to summarize the structure of itemsets. Next, observations are synthesized by using statistical tests to extract structure of sequences that affect a specific dependent variable in the software or process (outcome). The selected dependent variable is called the *outcome*. The rest of this section discusses the three phases of Gandhi-Washington Method in detail.

Encoding

A formal language L over an alphabet Σ is a set of all strings permitted by the rules of formation and is a subset of Σ^*, where Σ^* is the set of all possible combinations (considering sequence) of the letters over the alphabet. In this context, the cardinality of Σ is the number of items encoded in an itemset. Ordered itemsets can also be presented as a string of items. A regular expression provides a structure to express a class of strings. The length of a string is the number of characters within that string (for example, the length of $'AABA'$ is 4).

Encoding assigns one character to all the items (events) of similar type. For example, in the Scenario #2 (verification patterns of repository trunk branches), $'S'$ is used to encode all the verification periods which had a length between 1 to 4 days. Software development is a series of events (items). Each event or event type can be named by a letter. While a big portion of sequential data in software engineering is on nominal or ordinal scales, but for synthesizing results, attributes on other scales can be mapped into ordinal scale. An analyst should identify the types of events in the processes and projects (problem space) and assign a symbol from Σ to each event. Encoding is powerful and flexible and requires a deep understanding of the problem space. In other words, encoding is the art of preparing a model for applying abstraction and synthesis. In GWM, an encoded itemset is formed by assigning a symbol to each item.

Categorical items: Categorical (nominal) data include fixed number of possible values. For example, if the people involved in the commit-review process is of interest (like Alice, Bob and Carol), we assign one letter to each, i.e $'A'$ for Alice, $'B'$ for Bob and $'C'$ for Carol. Then the Alice, Carol, and Bob process is encoded in the form of $'ACB'$. If their organizational position in

the process is of interest and having Alice as developer and Bob and Carol as code reviewers, we assign $'D'$ to Alice and $'R'$ to Bob and Carol so the Alice, Carol, and Bob process is encoded as $'DRR'$.

Non-categorical data: While continuous data can range between negative and positive infinity, only some important distinctions might be of interest [159]. Discretization creates different groups (bins) of data. For encoding ordinal data, different discretization or clustering algorithms can be used [159]. Also, experts' opinions and their perception of groups in the data could be used for discretization. In the Alice, Carol, and Bob example, we are interested in the number of bugs each of them reported as their performance measure (outcome). We categorize the number of reported issues between 0 and 4 as low (L), 5 to 10 as medium (M), and more than 10 as high (H). If Alice, Bob and Carol reported 2, 11, and 14 bugs respectively (itemset looks like 2, 11, 14), then the encoded itemset is $'LHH'$.

Abstraction

The abstraction receives encoded itemsets as the input and groups these itemsets into different classes with regards to the sequential commonality between them. Each of these classes is represented by a regular expression. In this step, we move from encoded strings to the set of strings represented by a regular expression. So instead of focusing on the concrete strings such as ABB or

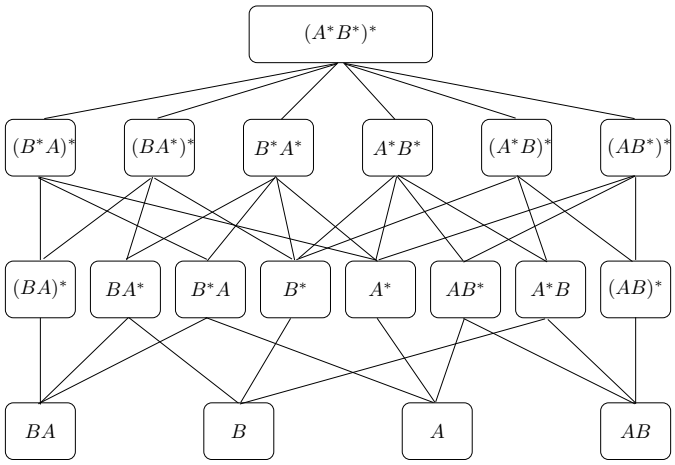


Figure 4.2: A hierarchy of regular expressions over $\Sigma = \{A, B\}$.

ABBBBBB we focus on the sequence of items' occurrence in the form of AB...B.

Regular expressions provide a compact view of an itemset and bring focus to the occurrence of sequences. In a regular expression, the use of the Kleene star (*) shows zero or more occurrences of an item type. Following this, both encoded strings *ABBBBBB* and *ABB* are categorized by the regular expression AB^*. To abstract each itemset in the form of a regular expression, an enumerated set of regular expressions is formed over a formal language Σ (see Figure 4.2). Then, a bottom up search categorizes each encoded itemset into one of the enumerated regular expressions that matches the itemset's sequence.

Regular expressions have structural differences (for example, $AB^* \neq B^*A$). Some regular expressions are positioned as the parent of the others. A regular expression is the parent of another one if it can produce the strings of the child regular expression. The parent-child relation is is expressed by an edge in the hierarchy. Figure 4.2 shows the hierarchy of regular expressions for the alphabet $\Sigma = \{A, B\}$. When comparing two regular expressions within a hierarchy, the greater the distance of a node from the root node, the more specific the regular expression. Consequently, each node is more specific than its parent, and thus can produce less variety of sequences.

The hierarchy of regular expressions could be extended by using production rules. Production rules replace A^* with $A...AA^*$ and B^* with $B...BB^*$ so an infinite number of strings and children can be produced for each regular expression. In this way, a more detailed hierarchy could be defined over $\Sigma = \{A, B\}$. Substitution of symbols can be recursively performed to generate diverse regular expressions. The more details required for the analysis, the more such production rules could be applied to expand the hierarchies over each alphabet.

Having a hierarchy of regular expressions over different sets of alphabet, each encoded itemset (which is a string of letters from an alphabet) is abstracted into the most specific regular expression. To categorize the encoded itemset in a hierarchy of regular expressions, a search is started from the leaf node with highest distance from the root. Each encoded itemset is compared with the hierarchy nodes until a match is found. One encoded itemset might match with more than one regular expression when two regular expressions have parent-child relation. GWM always classifies each

encoded itemset in the most specific regular expression possible. In this way, itemset $\{A,A,A,A,B\}$ is categorized in A^*B and not in A^*B^* even though both regular expressions match the itemset.

As the result of this step, each itemset is categorized in one regular expression and we moved from the actual encoded itemsets into the regular expressions that abstracted sequence of items. Looking into the sample regular expression hierarchy in Figure 4.2, some of the regular expressions might remain empty after abstraction. That means no such sequence occurred in the existing data. Non-empty regular expressions are the treatments for our experiment.

Synthesis

The synthesis receives items that are categorized in the hierarchy of regular expressions. Synthesis is the application of a series of statistical tests on a selected dependent variable (outcome). Synthesis step analyzes the commonality of recurring sequences and merges hierarchy nodes based on the results. Merging two nodes in this context means to systematically transfer strings that are classified with a specific regular expression into a more general regular expression in the hierarchy. Merging is applied based on the impact of recurring sequences on a specified outcome.

Outcome is the dependent variable which we analyze the effect of recurring sequences(treatments) on it. Outcome can range from code related issues (such as the number of bugs, the number of commits, etc.) to market related concerns (such as market share of a product, number of downloads, etc.). In this step, GWM identifies the construct between treatment and the outcome.

Synthesis – Algorithm: The complete synthesis steps is demonstrated in Algorithm 1.

In this process, a node may have more than two parents and may have many siblings. The Mann-–Whitney test is performed for all the nodes. Among the nodes with the greatest distance from the root, the node with highest priority is selected. In the rest of this section, we refer to lines of Algorithm 1 and describe it in depth.

Order in the synthesis: *(Algorithm 1, Line 9-16)* The synthesis process is applied on all the hierarchy nodes until all of them have been statistically compared with their parents and siblings. To synthesize structures and extract treatments with a significant impact on the outcome, the node

with the greatest distance from the root that satisfies the following conditions, is selected:

1. The node has not been decided in the synthesis. This means that no decision was made for merging or not merging the node with its parent.

2. The node that does not have any child or any undecided child.

3. Between nodes with highest distance, the priority is given to the node that does not have siblings and then to the node that has only one parent.

Algorithm 1: Synthesis

```
1  Hierarchy with nodes being regular expressions as the output of abstraction step
2  Array MWS[][]; //Mann–Whitney results for nodes & siblings
3  Array MWP[][]; //Mann–Whitney results for nodes & parents
4  Array DISTANCE[]; //Greatest distance of a node from root
5  Var i = Maximum distance in a tree;
6  Func Synthesize (Regular expression hierarchy)
7   for (all the nodes with DISTANCE[i] from root) do
8       while (all the nodes are not decided or merged) do
9           if (a node does not have sibling) then
10              select it as NODE
11          end
12          if (a node has one parent only) then
13              Select it as NODE
14          else
15              Select the node with most number of not rejected H0
16          end
17          Apply Mann-Whitney test between NODE and its' parent(s);
18          Apply Mann-Whitney test between NODE and all siblings;
19          if (NODE rejects all H0) then
20              mark NODE as decided;
21          else
22              Merge NODE with parent returning the highest p-value;
23              if (NODE has decided children) then
24                  Transmit the children into NODE parent;
25                  Update MWS and MWP of parent and its children;
26              end
27          end
28          i = i − 1;
29      end
30 end
```

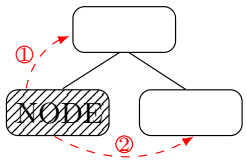
Synthesis – Base case: *(Algorithm 1, Lines 17-19, and 22)* The synthesis merges nodes with one parent only in the case where no significant relationship between the regular expression groups and the selected outcome is detected. Assuming that outcome is ordinal, the non-parametric analysis of variance, Mann–Whitney, is used in synthesis. Mann–Whitney test provides a comparison between all the itemsets that GWM categorized in two regular expression groups. The output of the synthesis is a set of regular expressions that has significant relation with outcome, called TrOC (treatment-outcome construct). The simplest synthesis on a node which has one parent and one sibling only is explained below.

Synthesis: Base case

Apply Mann-Whitney test to parent and sibling.

Continuous and ordinal data ⇒ **If** H_0 isn't rejected in at least one of the tests (① or ②), merge the NODE with its parent.

⇒ **Else**, keep the NODE unmerged as a pattern. It is now referred to as a significant node.



Synthesis - A node with siblings: *(Algorithm 1, Line 18)* When GWM considers synthesis of a node that has siblings, the base case of synthesis is extended a step further. In this case, a node is merged with a parent if it shows less significance (higher p-value) in terms of outcome compared to its siblings. Although, before making the decision to merge a node, GWM will Fight to the Last Breath (FLB). Fighting to the last breath is performing an exhaustive search to find the node which has the least significance in two ways:

1. It will select a node to synthesize, among all siblings, which has the greatest number of not rejected null hypotheses from the Mann–Whitney tests.

2. If a node has more than one parent, the node would be merged with the parent that showed the higher p-value in the Mann–Whitney test.

Synthesis – A node with multiple parents: *(Algorithm 1, Lines 21 and 22)* Nodes with more

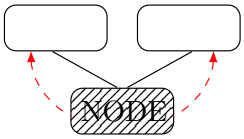
than one parent in the regular expression hierarchy need an extension of the synthesis base case. The extended synthesis uses the results of the statistical test to decide on merging or not merging the node. The process for a node with two parents is explained below.

Synthesis of a node with two parents

Apply Mann–Whitney test to both of the parents.

⇒ **If** both tests reject H_0, the node is not merged.

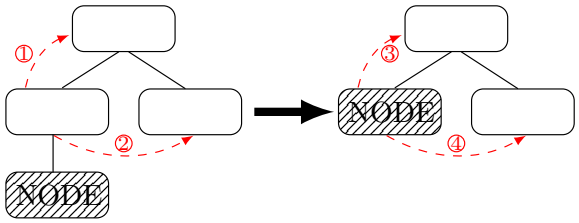
⇒ **If** only one test rejects H_0, the link between NODE and the parent which doesn't reject H_0 is deleted.



⇒ **If** both tests don't reject H_0, NODE is merged with the parent returning the higher test's p-value.

Synthesis – Child transmission in merging: *(Algorithm 1, Lines 23-25)* In the synthesis, if we merge a node which itself has a significant child node, the significant child node would be transmitted to the new combined parent and its significance would be reevaluated considering its new position in the tree as described below.

Transmit Significant Nodes



Apply Mann–Whitney test to both parents.

1- Merge NODE parent with its parent.

2- Transfer NODE to the new parent.

3- Apply Mann–Whitney test to new parent.

4- **If** H_0 isn't rejected, apply Mann–Whitney to the sibling.

⇒ **If** the H_0 isn't rejected merge the NODE with parent.

Nodes with no siblings and only one parent have higher priority than others. When all such nodes are analyzed, the nodes with the greatest number of not rejected null hypotheses are selected. When we test relationship of a node and its siblings based on outcome and decide not to merge

that node, we call this node a significant node. If this significant node is not merged with any other nodes during the recursive synthesis, the final regular expression related to the node shows the pattern in the sequential data that affects outcome.

Output: GWM determines a set of TrOC, T_1 to T_L, which meet the following conditions:

1. Each itemset is categorized in exactly one TrOC.
2. $T_i \neq T_j \ \ \forall i, j$.
3. Mann–Whitney (T_i, T_j) rejects H_0 for all pairs TrOC T_i, T_j if T_i and T_j are siblings in the hierarchy.
4. Mann–Whitney (T_i, T_j) rejects H_0 for all pairs of T_i, T_j where T_i is the immediate child of T_j.

In this step we mined the constructs between treatments (with sequential nature) and a selected outcome. The output is a subset of treatments that are:

- Structurally different from each other considering the sequence of items, and/or
- Has significant impact on the selected outcome.

4.3 Applications

Several applications of GWM are briefly introduced in this paper. We discuss three of these applications in depth.

First, file editing patterns- Subsection 4.3:

What? For each file we consider two status either being under edit or not. We consider the *edit time* as the time that a file is being edited by a developer and the *idle time* as the period that a file is not changed. Considering idle and edit time, we applied GWM to mine if the sequence of the edit and idle intervals has a significant impact on the bugginess of files.

How? We first replicated a study [279] on a sample of 22 open source software products with in

total 176,487 files. Then we applied GWM for mining patterns of file editing and compared the results to show the added value of GWM.

Cause-effect construct: sequence of the edit and idle intervals has a significant impact on the number of bugs in a file.

Outcome: number of bugs in a file.

Encoding: C (extended file idle intervals), and D (shorter idle intervals).

Treatments: C^*, D^*, CD^*, D^*C, $(CD)^*$, C^*D, C^*D^*, $(C^*D)^*$, $(DC^*)^*$, $(C^*D^*)^*$.

TrOC: C^* (consecutive long idle intervals) has significantly higher number of bugs in a file compared to the $(C^*D^*)^*$.

Second, file ownership patterns - Subsection 17:

What? Code ownership has been studied from different perspectives [28, 157, 279] to find the best way to decide who is most competent to receive a change request. Many models have been made to help this, although the effect of sequence of contributors in changing a file was not considered. We applied GWM to mine if the sequence of touching a file by major and minor contributors has significant impact on file bugginess.

How? We defined the major and minor contributors of 176,487 files over 22 open source projects and applied GWM by considering *number of post release bugs* as the performance measure.

Cause-effect construct: sequence of major and minor contributors touching a file has impact on the number of bugs in a file.

Outcome: number of bugs in a file.

Encoding: A (file owner), B (other developers committing to a file).

Treatments: 17 treatments (all the nodes of the hierarchy in Figure 4.2 other than $(BA)^*$ and BA^*.

TrOC: $(A^*B^*)^*$ has significantly more number of bugs in a file compared to A^*B^*.

Third, release cycle time patterns - Subsection 17

What? Looking into a software product, iterative release decisions, such as duration of release cycle (short cycle or long cycles) is of importance to plan for a software release. More recently, several questions have been raised in this context, questioning the trade-off between release duration, effort needed, and type of changes in releases [3, 20]. Some empirical studies have been designed to observe and report the impact of release duration on specific products [115, 162].

How? For 6,003 apps from Google Play store, we mined the duration between duration between release cycle time. Then, we encode the duration between two consecutive releases into short, medium or long release cycle and applied GWM. We will discuss correct and incorrect inferences from GWM results along with the extracted patterns.

Cause-effect construct: sequence of release cycle time has impact on the apps' rating.

Outcome: Apps' rating.

Encoding: S (short release cycle), M (medium release cycle), and L (long release cycle).

Treatments: 73 treatments over $\Sigma = \{S, M, L\}$.

TrOC: 7 treatment-effect constructs between L^*M^*, L^*, L^*S^*, M^*, $(M^*S^*)^*$, S^*, and $(S^*M^*L^*)^*$.

While we discuss the applicability of GWM in three case studies as example but GWM is not limited to these cases. GWM can be applied on several software engineering experiments that the cause construct is about a sequence of items. The Gandhi-Washington Method extracts the

recurring sequences of events with regards to their effect on a context specific factor. In a nutshell, GWM is applicable to:

1. Data which has sequence by nature. For example, time sequences, process sequences, or development sequences.

2. Data presented in a system alongside numerical factors which provide a measure to the system performance (outcome). For example, code healthiness or estimated effort of a task.

3. Cases where multiple instances of data are available. Mining TrOCs is meaningful when we compare multiple itemsets.

Example: File Ownership Patterns

We first replicate a study on file editing patterns performed by Zhang et al. [279] and subsequently apply GWM on the same dataset and compare the results. Furthermore, we demonstrated how we can get more information and automatically extract all the treatment-outcome constructs using GWM. Following the approach and keywords reported by Ray et al. [216], we retrieved the number of commits related to bug fixes in each file. The extraction of bug related issues was done manually by two software engineers and the conflicts were resolved by one of the authors.

Case study data

We used the data of 22 trending projects from GitHub in the category of programming languages[3]. These projects had 176,487 files, 1,773 releases, and 4,501 developers in total. For each file we mined the number of post-release bugs per file from git logs by considering the release and commit dates.

We also estimated the edit and idle time for each file in these projects. Zhang et al. [279] considered the time span for editing a file as *edit time* and the time interval where no one was editing the file as the *idle time*. Zhang et al [279] studied Mylyn project which gave the actual

[3] `https://github.com/showcases/programming-languages`

value for edit and idle intervals. However, not having access to that data we used open source projects and estimated these time intervals.

Algorithm 2: Edit and idle time calculation

```
1  //datetime is the date and time of a commit.
2  for (each file) do
3      Ignore the commit that creates the file;
4      for (each following commit) do
5          Set commit's datetime as (A);
6          Set the datetime of the most recent previous commit, anywhere in the project, of
             this developer as (B);
7          edit time = (A) - (B);
8          (P) is the next developer committed to the file;
9          Set the datetime of the most recent previous commit, anywhere in the project, of
             developer (P) as (Y);
10         if (Y)<(A) then
11             conflict time = (A) - (Y);
12             idle time = 0;
13         else
14             idle time = (Y) - (A);
15         end
16     end
17 end
```

We used Algorithm 2 to estimate the edit and idle time intervals for the files of our GitHub projects. We made the assumption that if a developer commits changes only to File I and File II at Datetime 1 and made changes to File III at Datetime 2, then edit time of File III is defined as $Datetimet\ 2 - Datetime\ 1$.

Note that we use this case study to demonstrate the added value of GWM and the contributions of this paper is not the actual and precise results of this case study and the estimated values are calculated to this end. In what follows, we go through different GWM usage scenarios using this dataset.

Replication of the former study

Previous research by Zhang et al. [279] investigated file editing patterns. They hypothesized the existence of four file editing patterns namely concurrent editing, parallel editing, extended editing,

and interrupted editing patterns. Patterns are the sequence of changing a file. These patterns were then analyzed in relation with the number of developers, number of co-editing files, and the edit and idle time intervals. To demonstrate the benefit of GWM we only discuss interrupted editing patterns likewise one can apply it on extended patterns.

To replicate their study, we calculated the maximum idle intervals of each file (IdleTime). Following Zhang et al. [279] a file follows the interrupted editing pattern "if and only if its IdleTime is greater than the third quartile of all IdleTime values". We calculated IdleTime of files in our case study and mined TrOCs.

The case study by Zhang et al. [279] reported that files following interrupted editing pattern are 2.0 times more likely to have future bugs. They used Fisher's exact test in conjunction with Odds Ratio for this reasoning. In the same way, we found that files following the interrupted editing pattern are 1.86 times more likely to experience future bugs. With Fisher's exact test (p-value$<$0.001) and $OR = 1.86$Our findings are aligned with the results reported in [279].

The study by Zhang et al. [279] examined predefined patterns and tested their likelihood of having future bugs. In other words, they assumed that specific treatments (sequence of idle or edit times) exist. Replicating their method but using GWM, we did not assume the existence of any treatment (idle or edit patterns) and we just encode the data as they proposed and GWM could mine the treatment-outcome construct. Our findings are aligned with their reported results.

Now we go one step further. In the past study among all the idle and edit time intervals of a file, the decision was made based on the longest edit and idle interval for each file (i.e., the EditTime and IdleTime). Applying GWM we discuss three different scenarios in the next subsections:

- we encode all the idle times for each file (Subsection 17),
- we change the encoding for fine grained analysis of idle times (Subsection 17), and
- extract patterns (treatments) from the idle and edit time instead of assuming the existence of two patterns (i.e., extended editing and interrupted editing patterns) and extracting the TrOCs for testing the assumption (Subsection 17).

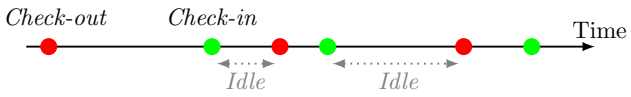


Figure 4.3: Abstract model of idle time.

Scenario 1: Synthesizing patterns

From applying GWM, we are interested in answering the following research question:

RQ1: Do any idle time sequence exist related to the bugginess of a file?

Two sequences with significant impact on number of post release bugs exist and files having consecutive long (extended) idle interval have significantly more number of post-release bugs as compared to the rest of the files.

Encoding: We encoded the time between a check-in and check-out of a file as idle time as shown in Figure 4.3. Having the idle time of all the files, an alphabet consists of two letter were used for encoding.

C: Encodes the idle intervals greater than the third quartile value of all idle interval values ($\geq$ 133.49 *hours*). This is called *extended idle time*.

D: Encode all the other idle intervals (less than third quartile value).

In this model, each itemset represents a sequence of idle time intervals being longer than or equal to the third quartile values (encoded by C) or not (D). Comparing this with the Zhang et al. model [279], we did not assume that there is a extended idle time pattern but we are looking to extract sequences of idle times with different lengths with significant impact on the outcome (number of bugs).

Categorizing: We used the regular expression hierarchy over $\Sigma = \{C, D\}$. Using regular expressions, for example $\{C, C, C, D, D\}$ is categorized as C^*D^* and $\{C, C, C, C\}$ categorized as C^*.

Synthesizing: We applied Algorithm 1 on the categorized itemsets and used the average number of post-release issues per file as outcome. Using GWM, we mined two treatments having

significant impact on the number of post release bugs:

C^***:** Consecutive long (extended) idle interval. 1.2% of files followed this sequence.

$(C^*D^*)^*$**:** Combination of extended and not extended idle intervals. 98.8% of files followed this sequence.

The boxplot distribution of the number of bugs (outcome) for each of the above patterns is shown in Figure 4.4−(a). C^* had significantly more bugs.

Discussion. We considered all idle intervals for each file and applied GWM. The results showed that the consecutive extended idle intervals affect the average number of bugs significantly differently than other patterns. Also using OR, we found that files that are edited with consecutive extended idle intervals are 2.3 times ($OR = 2.3$) more likely to experience future bugs in comparison to other files.

In addition to the results achieved by Zhang et al. [279], we found that the occurrence of extended idle intervals in each file affect the likelihood of a file's bugginess. *More specifically*, files having consecutive extended idle intervals are more defect prone (C^*). This might be because these parts of code are not maintained for a certain period of time. Also, considering the turn over of team members this might occurred because of lack of knowledge to the specific files. While GWM does not indicate causality, the analysis of the reasons could be the subject of other research.

Scenario 2: Change model granularity

To demonstrate how different encodings can solve different problems using GWM, we answer RQ2 using the same dataset:

RQ2: How results of more fine-grained encoding of idle time into four categories compare to the results of Scenario 1 with two encoded idle time intervals?

GWM extracted five patterns. All these five sequences that have a significantly different effect on

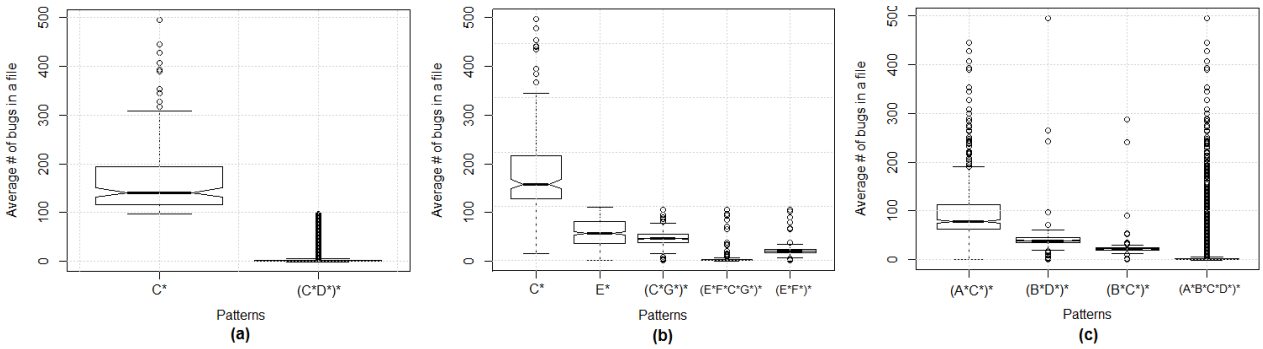


Figure 4.4: Boxplot of average # of bugs in a file for patterns of (a) idle intervals (b) fine grained idle intervals (c) idle and edit intervals in conjunction.

the number of post release bugs.

Encoding: We want to extract more fine grained idle time sequences as compared to the previous scenario. To acquire such sequences, we change the encoding of GWM and use an alphabet with four letters to encode idle time:

C: Encodes the idle time greater than or equal to the third quartile ($\geq$ 133.49 hours, called *extended*) of all idle time values (like in Scenario 1).

E: Encodes the idle time less than or equal the first quartile ($\leq$ 13.24 hours, called *very short*) of all idle time values.

F: Encodes the idle time greater than the first quartile and less than or equal the second quartile of all idle intervals (between 13.24 and 43.81 hours, called *short*).

G: Encodes the idle time greater than the second quartile and less than or equal the third quartile (between 43.81 and 133.49 hours, called *long*).

In this model, an itemset represents the sequence of encoded idle intervals for a file.

Categorizing: We use the regular expressions over $\Sigma = \{C, E, F, G\}$ to categorize itemsets. The items extracted from our dataset were categorized in 47 regular expressions.

Synthesizing: We applied Algorithm 1 on categorized itemsets, using the average number of post release bugs as outcome. The results show five patterns of idle time intervals with significant impact on number of post release bugs:

C^***:** Consecutive extended idle interval (same as Scenario 1-RQ2). 1.2% of all files followed this sequence.

E^***:** Consecutive very short idle interval. 1.7% of all files followed this sequence.

$(C^*G^*)^*$**:** Combination of long extended idle intervals. 0.6 % of files in our case study followed this sequence.

$(E^*F^*)^*$**:** Combination of short and very short idle intervals. 3.9% of all files followed this sequence.

$(C^*E^*F^*G^*)^*$**:** Combination of idle intervals with different length. 92.6% of all files followed this sequence.

The boxplot distribution of number of post release bugs (outcome) is shown in Figure 4.4−(b).

Discussion: Encoding enables us for flexible modeling of the problem. While in this scenario we followed the same model as Scenario 1, we retrieved more fine grained sequences. We calculated the Odds Ratio(OR) and Fisher's exact test values (As suggested by Zhang et al. [279]) and found that files that have consecutive long idle intervals are more than 4.201 likely to experience future bug comparing to the files having consecutive short idle times. While comparing them with the mixture of all types of intervals i.e. $(C^*E^*F^*G^*)^*$ they are 8.73 time more likely to have future bugs. On the other side, in comparison to the files with a combination of long and extended idle times, they are less likely to experience future bugs ($OR < 1$). Additionally, files with a combination of long and extended idle times are 9.1 times more likely to experience future bugs in comparison to files with a combination of idle times with differing lengths.

In comparison to Scenario 1, we observe that we have more detailed and specified information about the treatments we extracted previously (C^*). In addition, we can now capture and analyze all the idle time intervals this leaded us into capturing treatments such as $(C^*G^*)^*$ being 9.1 times more likely to experience future bugs.

It worth to note that while Mann–Whitney test result for E^* and $(C^*G^*)^*$ were not significant

($= 0.068$), GWM keeps both as TrOCs because they relate to different branches of a regular expression hierarchy. In other words, the treatments were structurally different as these two treatments (sequences) are not siblings nor having parent-child relation with each other.

Scenario 3: Solve more complex problems

So far, we studied just the sequences of idle time for files. We are also interested to observe the sequences of edit and idle times of a file in conjunction and analyze the impact on files' bugginess. We capture both edit time and idle time of a file. In this way, an itemset represents the sequence of both edit and idle intervals for a file. This model is shown in Figure 4.5.

RQ3: Do any idle and edit time exists that has impact on the bugginess of a file?

GWM extracted four sequences that are related to the average number of bugs significantly differently from each other.

Encoding: We encode the items using an alphabet of four letter and follow the below encoding:

A: Encodes the edit time greater than or equal the third quartile (≥ 42.39) of all edit time values.

B: Encodes all the other edit intervals (less than the third quartile of edit time).

C: Encodes the idle time greater than or equal the third quartile (≥ 133.49) of all idle time values.

D: Encodes all the other idle intervals (less than third quartile).

Each itemset represents a sequence of edit and idle intervals for a file.

Categorizing: We used the regular expressions over $\Sigma = \{A, B, C, D\}$ to categorize itemsets. Extracted itemsest were categorized in 59 regular expressions.

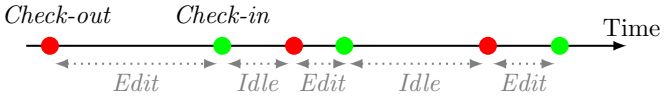


Figure 4.5: GWM can encode idle and edit times together.

Synthesizing: By applying Algorithm 1 into the categorized items. We found four sequence that impact the number of bugs significantly different from each other:

$(A^*C^*)^*$**:** 0.4% of files follow extended edit intervals and extended idle intervals.

$(B^*C^*)^*$**:** 0.8% of files had not-extended edit interval and extended idle time.

$(B^*C^*D^*)^*$**:** 1.3% of files have the combination of idle interval types with not-extended edit time (all types of encoding other than A).

$(A^*B^*C^*D^*)^*$**:** 97.5% of files have a combination of different types of edit and idle intervals.

The boxplot distribution of number of post release bugs (outcome) is shown in Figure 4.4−(c).

Discussion: Considering the idle and edit intervals together and measuring the likelihood of future bug occurrence using Fisher's test and Odds Ratio (As suggested by Zhang et al. [279]) we found that files that were not edited for a long interval but were idle for a long period (i.e. $(B^*C^*)^*$) are 8.6 times more likely to experience future bugs in comparison to files that have a combination of all types of edit and idle time intervals (i.e. $(A^*B^*C^*D^*)^*$). Additionally, files that have not been edited in extended periods of time (i.e. $(B^*C^*D^*)^*$) are 1.97 time more likely to experience future bug in comparison to the files following extended edit and idle interval sequences (i.e. $(A^*C^*)^*$). Files that follow $(B^*C^*D^*)^*$ are 7.1 more likely to experience future bugs in comparison to the files following $(A^*B^*C^*D^*)^*$.

Example: Code Ownership sequences

The writer (developer) of a line of code has the most knowledge about that part of the code. The owner of a piece of code is determined as the developer who wrote the most lines in that piece of code.

File ownership is defined as the percentage of lines of code that a developer owns in a file [75]. Code ownership was studied in many different ways in order to find the most competent and knowledgeable person to change the code [28,75,157]. We analyzed the relationship between code

ownership sequences and the number of bugs related to that file. We studied the following research question:

RQ4: Do any code ownership sequences exist related to the healthiness of a software product? *Applying the Gandhi-Washington Method on a large scale dataset from GitHub, we found two sequences of file ownership that have a statistically significant relationship with the healthiness of a program file.*

Encoding: Analyzing code ownership for each file, we considered two types of contributors; the contributor who owns the file (she has the most code churns in that file) and the rest of the developers who committed changes to the file. Using $\Sigma = \{A, B\}$:

A: For each file, $'A'$ represents the *file owner*. We calculated the code churn (number of changed lines of code) made by each developer in the whole project and across all releases per file. We considered the person with the highest number of churns in each file as the file owner.

B: For each file, $'B'$ represents the *rest of the contributors to a given file*. These developers contributed changes to a file but they do not own the most lines of code in the file.

A release file owner could be $'A'$ or $'B'$. After we defined file owners, we assigned $'A'$ or $'B'$ to each release of a file. This way we encoded each file with its release owners. For every file in a project, we created an itemset within which the items show if the owner of the file overall is also the owner in each of the releases.

For example, an itemset such as $\{A, A, B, B, B\}$ shows that $'A'$ created the file and initially committed changes while the most recent changes were by other developer(s). The file owner of this file ($'A'$) cumulatively made most changes to this file (note that $'B'$ in not one specific developer but the other developers not being the file owner). $\{A, A, A, B, B, B\}$ shows that the person with highest contribution to the file and the owner of the file in early releases, has not contributed as extensively as others to this file in the last three releases. In this case study we ended up with 175,995 itemsets as some of the files were never changed in their projects' lifecycles. We selected the average number of post-release bugs for a file as outcome to analyze the relationship between

the ownership sequence and the bugginess of the files.

Abstraction: We used the regular expression hierarchy over $\Sigma = \{A, B\}$ (like the one in Figure 4.2) to categorize the itemsets. This enumerated hierarchy has 19 nodes, of which 17 had encoded itemsets categorized into them. Using regular expressions $\{A, A, A, B, B, B\}$ is categorized as A^*B^* while $\{B, B, B, B, B\}$ is categorized as B^*.

Synthesis: We applied Algorithm 1 on the categorized itemsets and used the average number of post-release bugs per file as the outcome. As the results of this process, we found two sequences (treatments) as A^*B^* and $(A^*B^*)^*$. Files with the ownership sequence A^*B^* have significantly more post-release bugs as compared to the rest of the files. Figure 4.6 shows the distribution of the number of post-release bugs in the files with A^*B^* and $(A^*B^*)^*$ ownership sequence. A^*B^* represents files for which the developer who mainly owns the file in the project has the ownership of the file in early releases, but later the release file ownership is transferred to another developer.

The result of this analysis shows that files which have ownership sequences of A^*B^* have significantly more post-release bugs compared to other files. Significantly more bugs in A^*B^* might be because of unfamiliarity of later stage file owners ($'B^*$) with the code that were maintained largely by the main owner for a while (A^*). Further investigations are needed to find the actual cause for this problem.

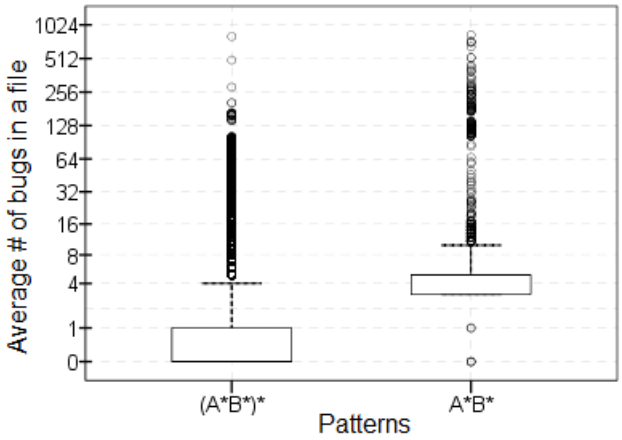


Figure 4.6: Boxplot of apps rating for release cycle time sequences among 6,003 apps in Android market.

Discussion: Code ownership TrOCs are not targeting predictions, but rather determining if there is a statistically significant relationship between sequence of owning a file and the number of post-release bugs. Bird et al. [28] reported that the number of low expertise owners has relationship with both post-release and pre-release failures. They showed that higher level of ownership for the top contributor results in fewer failure. Their study considered the proportion and level of ownership, but the impact of sequences of ownership (as studied here) was not discussed. The results of our study showed that files following A^*B^* sequence (consecutive edits by the file owner and later consecutive edits by other contributors) are 1.74 times more likely to experience future bug (p-value of Fisher's test = < 0.001 and $OR = 1.74$).

Example: Release Cycle sequences

A product manager is concerned about the date for next release of a software product. She is following a fixed cycle schedule to release a version of her code every two weeks. Now she is facing a major bug in the code. Fixing the bug would delay the version release. She decides to release the product as she believes that users are expecting to receive the new version based on the schedule. In other words, she assumes that following a fixed release cycle increases customer satisfaction. GWM can find the relation between fixed, short release cycles and customer satisfaction.

Defining release cycle strategies for software product are a challenge for many practitioners and researchers [3, 115]. Traditional software development processes were changed with the introduction of iterative processes. Later, agile practices enhanced the development process a step further by using short release cycles. The effect of short and long release cycles on different aspects of software products like teams' productivity, requirements engineering and specification were discussed.

We have always discussed the fixed short release cycles (agile development with short scrums) versus the long release cycles (traditional iterative processes). Mapping this terminology into GWM, we represent short release cycles by S and long release cycles by L, then analyzing the consequent short release cycles is presented as S^*, and the analysis of consequent long release

cycles is presented as L^*. Having this in mind, the GWM contributes in two ways:

1. It discovers more in-depth release strategies among different software. In this way, we can extend the discussion of traditional versus agile iterations to include more diverse release strategies such as long release cycles followed by a consequent short cycles (for example for stabilizing a major release).

2. It statistically analyzes the effect of these sequences (treatments) on a context specific performance measure (outcome). For example, the effect of consequent short cycles, S^*, on the number of bugs.

RQ5: Do any release cycle time sequence exist with significant impact on the mobile apps' rating? *Applying GWM we mined seven sequence of release cycle time that has significant impact on the app rating.*

To answer this question, we analyzed 6,003 apps from GooglePlay and we selected the app rating as outcome. The apps in this sample have different numbers of releases, ranging from 3 to 186. These 6,003 apps had 60,588 releases in total. Each app has a rating between zero and five which is the average rating granted by the app's users.

Release cycle is the time between two consequent releases. In this sample, the release cycle durations follow a power-law like distribution. In this data, one quarter of the release cycles are less than five days, while one quarter take more than a month. Applying GWM to a set of 6,003 apps, we found seven particular release cycle time TrOCs.

Encoding: Having release cycles of each app as items in the itemset, we applied frequency

Table 4.2: Release cycle TrOCs - Star indicates the insignificance of the p-value.

ID	Treatment	Description	% of occurrence
P1	L^*M^*	Combination of L and then M cycle types.	14.1%
P2	L^*	Subsequent release cycles of [23, 1365) days.	23.7%
P3	L^*S^*	Combination of L and then S cycle types.	9.5%
P4	$(S^*M^*L^*)^*$	Combination of all cycle types.	35.1%
P5	M^*	Subsequent release cycles of [7,23) days.	3.5%
P6	$(M^*S^*)^*$	Combination of M and S cycle type.	7%
P7	S^*	Subsequent release cycles of [0, 6) days.	7.1%

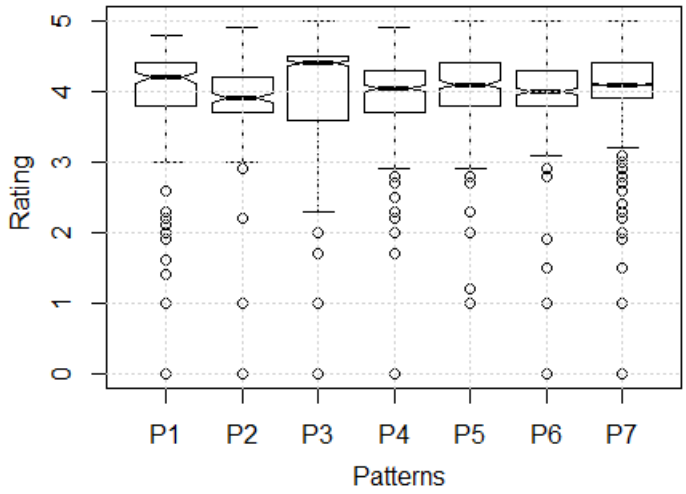


Figure 4.7: Boxplot of apps rating for release cycle time TrOCs among 6,003 Android apps.

based distribution on the release cycles. This resulted in three clusters called *S* (short), *M* (medium), and *L* (long).

***S*:** represents cycles between 0 to 7 days of duration (less than a week),

***M*:** represents cycles between 8 to 21 days (up to three weeks) and,

***L*:** represents cycles 22 days or more (i.e., more than three weeks).

For example, {20, 16, 19, 15, 13, 21, 20} and {21, 122, 81, 61} represents release cycle times for two different apps where the first one is encoded to *MMMMMMM* and the second itemset is encoded as *MLLL*. Encoding in GWM is flexible and can be customized in any context. Alternatively, in this case, encoding could be different using experts' defined ranges or clusters defined by using the k-means algorithm [159].

Abstraction: A regular expression hierarchy over *S*, *M*, *L* with 130 nodes was used. Categorizing the 6,003 release cycle sequences, 73 nodes of this hierarchy were filled with at least one itemset. Using the regular expressions, *MMMMMMM* is categorized in M^* and *MLLL* categorized in ML^*.

Synthesis: We applied Algorithm 1 on the 73 nodes of the regular expression hierarchy. Using app rating as outcome, we obtained seven unique release sequences that significantly affect the

Table 4.3: Sample of correct and wrong inference from GWM results

×	*One should release her/his app within consecutive short cycles (s^*) to receive better rating.*
Why?	GWM found that S^* affects app rating significantly and higher than other release sequences, but this does not mean causality. GWM work in the observation space rather than theory. We can't state that these apps are getting higher ratings because they follow consecutive short cycles. Short release cycles might be one of the reasons and should be considered in the models we are making to predict apps' success.
Application	If we are making models for performing research on the success factors in the app store market, release cycle time sequences should be part of it, as we see they have an effect on app success, although other factors may affect app success in parallel.
✓	*A long release cycle followed by a short cycle $(LS)^*$ does not affect app rating significantly differently comparing to a strategy such as $(S^*M^*)^*$ or $(LM)^*$.*
Why?	$(LS)^*$ doesn't appear as a separate TrOC in Table 4.2.
Application	The app developer is following $(LS)^*$ to release her alpha and beta versions of her app. She released alpha version and she planned to release the beta within the next six days (short cycle). After two days she receives several negative comments on app defects. She can't fix all the bugs in remaining days and is thinking about changing schedule and release the app later. Using the results of GWM, she does not need to be concerned about the negative affect of deadline extension on customer satisfaction.

rating of the apps within our sample (seven TrOCs). Table 4.2 and Figure 4.7 show the final treatments extracted by GWM and their outcome distribution. Apps that have consecutive long release cycles (more than 3 weeks) followed by consecutive short cycles (less than a week) have the highest median of app rating (L^*S^*), followed by apps that has consecutive long and short release cycles (L^*M^*). The lowest median app ratings correspond to apps with long release cycles exclusively (L^*).

A detailed sample of synthesis steps is described in Appendix A. To elaborate more on the application of release cycle TrOCs and express GWM findings more clearly, we summarize a sample of a correct (✓) and a wrong (×) inference in Table 4.3.

4.4 GWM Tool Support

We provided a tool support for the proposed method. The GWM support tool is a `Windows` based program which has been tested on `Python 3.5.2`, though it may work on earlier versions. The tool has a graphical user interface (GUI) supported by `PyQt` as well as a shell access to support users of all levels.

The tool has a separate interface for each of the three phases of GWM as well as a dedicated interface for analyzing the fitness of statistical tests to support in-depth analysis and inferences by running multiple statistical tests:

Encoding: The encoding step translates lines of data from a `csv` file and outputs a string of characters representing an itemset. If the input data is non-categorical, the tool provides support for expert-based discretization and frequency-based discretization. The output of this phase is a `csv` file that automatically transforms as the input for abstraction.

Abstraction: The abstraction step matches encoded strings into enumerated regular expressions. The results of the process are given in the Categorized Strings window and shows that which itemset were categorized in which regular expression. The output is a `csv` file is used as the input for synthesis phase.

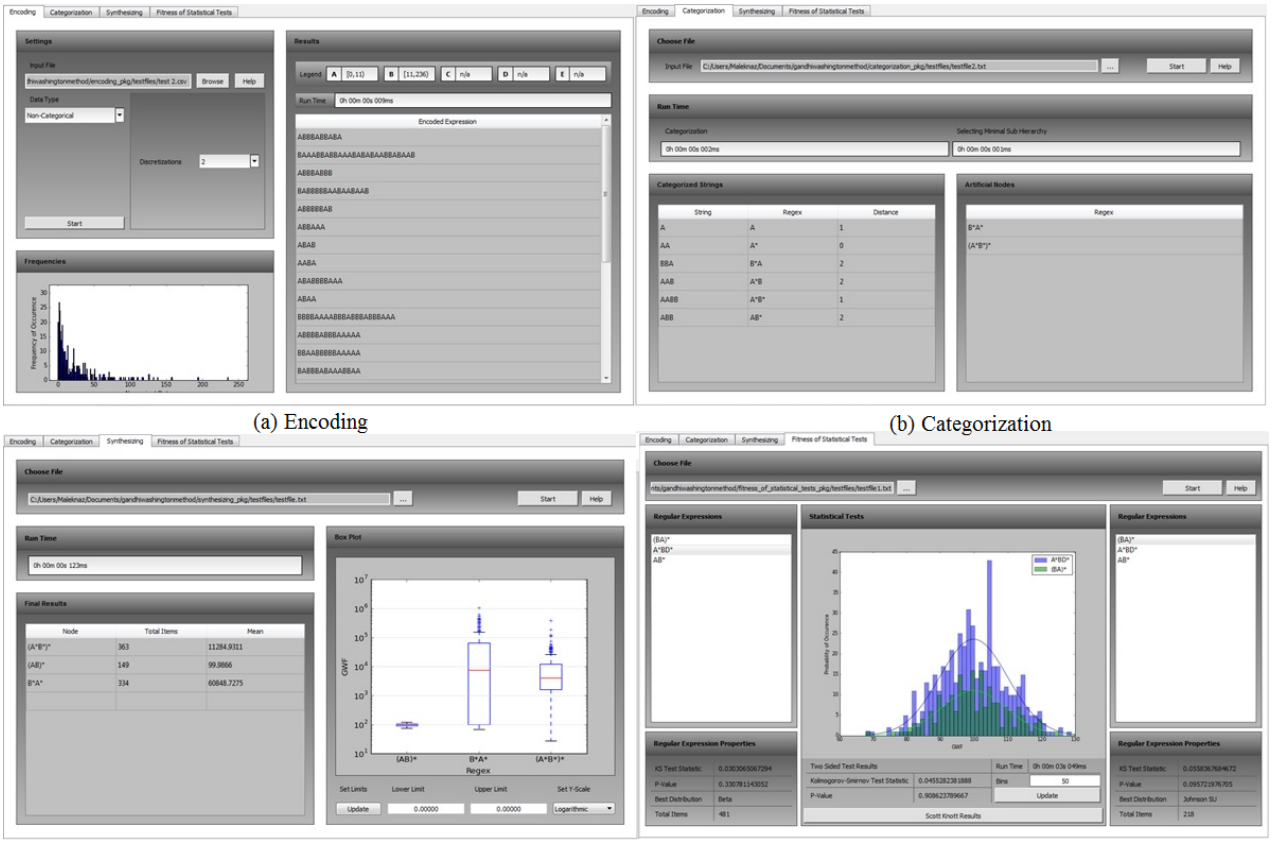
Synthesis: Synthesis phase merge separate regular expressions where the Gandhi Washington Factors categorized into them are statistically different using Mann-Whitney U-test as a default (that could be alternated). The output is provided in the format of a table showing the treatments, mean of the group and the sample size (number of itemsets) for each treatment. Also the distribution of the outcome are visualized trough boxplots.

Fitness of statistical tests: Each two treatments can be selected and their constructs can be compared trough a series of statistical tests and visualizations. The format of the input file is the same as in the synthesizing and so is useful to verify certain results.

We provided screen shots of the tool in Figure 4.4. A test generator tool is also available along with the Gandhi-Washington tool. The test generator will help to run controlled experiments with the tool as well as logging the time and determinism (as we used it in Section 4.5). The tool support up to five encoded items. However, the design is extensible.

GWM implementation, test generator and described test cases in Section 4.5 are all accessible via our website[4].

[4] `http://ucalgary.ca/mnayebi/tools-and-data-sets`



(a) Encoding

(b) Categorization

(c) Synthesis

(d) Fitness of Statistical tests

Figure 4.8: The GUI screen-shot of the Gandhi-Washington tool support. The tool supports all the three phases of the GWM process and additional support for verification of the results.

4.5 Discussion

In this section we discuss some further aspects of the implementation of the method. We look at possible solutions for more complex encoding for mining treatment-outcome and we will show that GWM is deterministic. In addition, we discuss the scalability and complexity of the method.

Multiple encoding

Encoding is flexible and in the case where multiple and related variables exists, multiple encoding is possible. In the *Alice, Carol, and Bob* example (Section 4.2), the *lines of code (LOC)* in conjunction with the number of bugs each of them reported is of interest. In this case, multiple encoding

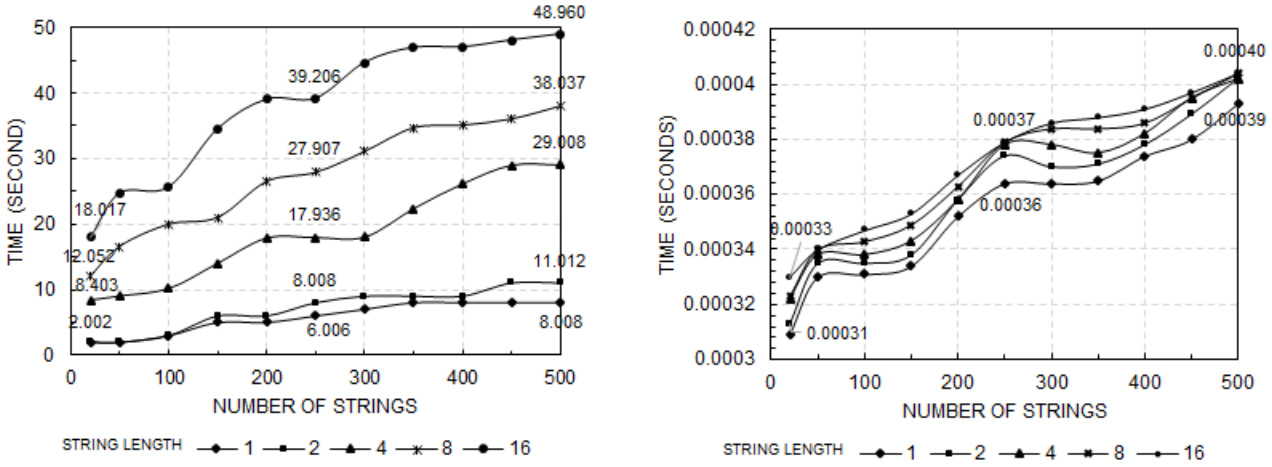


Figure 4.9: Time performance of the abstraction (on left) and synthesis (on right) for 2500 cases. Each line shows the performance for 500 strings of the same length.

is applied. We discretize the LOC into two groups naming 'A' for LOC below 5,000 and 'B' for LOC equal to or above 5,000. Each of Alice, Bob and Carol might be in one of the groups below.

Bug group	LOC group	Encode Item
L	A	U
M	A	V
H	A	W
L	B	X
M	B	Y
H	B	Z

In this way, the encoded itemset for Alice, Carol, and Bob sequence might be such as 'UWX'. Once an analyst models the problem and encodes the events, GWM abstraction and synthesis steps automatically retrieve TrOCs. The abstraction and synthesis steps do not need the analyst's participation. It is therefore easy for an analyst to try different encodings and find the proper match.

Fitness of statistical tests in synthesis

The Mann–Whitney test is selected as the main statistical method for the synthesis with the assumption that outcome is ordinal. The Mann–Whitney test is non-parametric and is not concerned with normal distribution of the variable and sample size [246]. In case the distribution of outcome between the two groups are identical, the Mann–Whitney test compares the medians. If the distribution of outcome between two groups are not identical, this test compares the mean rank between two groups. The degree of similarity between the mean ranks resulting from the Mann–Whitney test is the basis for deciding to merge two regular expression classes (or not).

P-values of Mann–Whitney tests are used to make the decision when multiple options are possible during the synthesis. The higher p-value shows the less significant difference between groups. Consequently, if a node has more than one parent and the synthesis has decided to merge it, the node will be merged with the parent that returns the higher p-value in a pairwise Mann–Whitney test. In cases of ambiguity in selecting one node among all its siblings, the node that rejects the null hypothesis least frequently will be selected. If there is a draw between the siblings for the fewest rejected null hypotheses (as they reject the test same number of times), then the one with the highest p-value will be merged first.

The Mann–Whitney test in synthesis is applicable for ordinal outcomes. The test could be replaced by other statistical inferences if a non-ordinal outcome is used. If we apply Mann–Whitney test on two groups of itemsets with non-identical distribution of outcome, the test compares the mean rank between the groups. Replacing the Mann–Whitney test with tests such as the t-test or even group tests such as ANOVA or Kruskall-Wallis is possible, although one needs to adjust the interpretation of TrOCs in each case. Furthermore, adjusting the p-values by using error correction techniques (such as Bonferroni) for the tests with multiple comparisons (such as ANOVA) would be needed.

Determinism and Complexity

The main computational complexity of GWM comes from the abstraction and synthesis steps as the encoding is done by the analyst. In order to exhibit the performance of GWM in practice and later to support the applications, we implemented GWM using Python. In this section, we report the computational complexity and actual run time of an implementation of abstraction and synthesis. The run time is measured using an `Intel CORE i7-260M` CPU. The results show the scalability of this method.

We also examined determinism of the abstraction and synthesis. We used syntax-based coverage criteria [11] and enumerated regular expressions to generate test cases. We developed a test generator tool for automated testing of GWM. For testing the abstraction, the testing tool checked the strings to see if they were categorized back into the regular expression that generated them. To test the synthesis, we assigned outcomes to the strings in a way to retrieve specific regular expressions as a TrOC. We then compared the results of test cases with our expected outcomes.

Abstraction: Abstraction consists of two sub processes, (i) reaching nodes in the regular expression hierarchy and (ii) sequence matching (analyze if the string is producible by the regular expression). Scanning all of the regular expressions to find a suitable node has complexity of $O(n)$, with n being the number of nodes in a regular expression hierarchy. While visiting each node in this hierarchy, the string is analyzed against the regular expression stored in the node. To do so, a finite state automaton based search is used with computational complexity of $O(|\Sigma|m)$, m being the number of strings processed by the abstraction process and $|\Sigma|$ being the size of the input alphabet Σ. Hence, the complexity of this process is $O(|\Sigma|nm)$.

To validate abstraction, a series of strings was produced by a generator tool. The generator tool takes each enumerated regular expression and produces strings using rules of formation. These strings were used as the input for our implemented abstraction. We tested how many strings were classified in nodes equal to their generator regular expressions. 10,000 sample itemsets were generated from 500 regular expressions by using the test generator. Output from the abstraction

mapped one-to-one into the generated test cases and was 100% accurate and deterministic. All of the generated strings were categorized in the regular expressions that they were produced from (accuracy), which was the most specific regular expression they could match to. We ran each test case multiple times and the results were consistent in all the runs.

To demonstrate the scalability and performance of the abstraction, we created five different sets of 500 strings. Each bin of 500 strings had the same length from 1 to 16 letters. The time performance of the synthesis over this dataset is illustrated in Figure 4.9.

Synthesis: Following Algorithm 1, a graph traversal approach is employed to apply a series of Mann–Whitney tests between nodes in the hierarchy. Using a Depth First Search (DFS) [194] approach and including pairwise comparisons, the algorithm has the time complexity of $O(n(m+n))$, with m being the number of edges and n the number of nodes in the hierarchy graph. Examining nodes for merging needs another graph traversal and updates to some of the test results in case a node is merged. This has the time complexity of $O(n(m+n))$. To test the correctness of the synthesis we tested two conditions,

(i) If the synthesis merges nodes with similar outcome distributions, and

(ii) If the synthesis returns nodes that are significantly different in terms of outcome.

We generated 100 strings per regular expression over four different alphabets $\{A,B\}$, $\{A,B,C\}$, $\{A,B,C,D\}$, and $\{A,B,C,D,E\}$. In order to check (i) the following partitioning strategies were used:

- We assigned a random outcome value x to all the strings belonged to half of the regular expressions and the random outcome y to the rest, where $x \neq y$. We expected to have two final treatments.

- We assigned a random outcome value x to the strings belonged to one third of regular expressions, one third a random outcome y, and one third a random value z, where $x \neq y \neq z$. We expected to have three final treatments.

- We assigned a random outcome x to all the strings belonged to one fourth of the regular expressions, one fourth a random outcome y, one fourth a random value z, and one forth a random value w where $x \neq y \neq z \neq w$. We expected to have four final treatments.

We ran each partitioning strategy on four different sets of alphabet. We observed over 12 runs of synthesis that all the strings were merged as expected (100% accuracy).

To test condition (ii), in each test we targeted specific regular expression to be retrieved by GWM as the TrOC. For the enumerated regular expressions over the four different sets of alphabet, a random node was selected. Random outcomes were assigned to the strings generated by the selected regular expression. We assigned outcome with random value x to all the other strings belonging to the rest of the regular expressions (making them merge together). We ran 20 tests (five tests for each alphabet set). For all the tests, GWM returned the targeted regular expression as TRoC and matched with our expected outcome (100% accurate). We ran each test case multiple times and the results were consistent in all the runs.

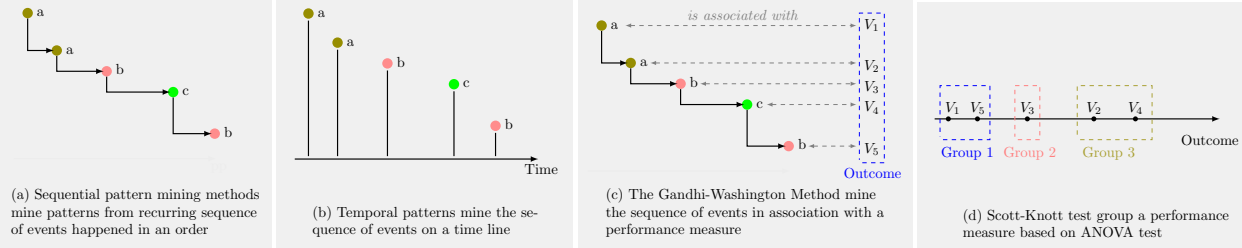
To demonstrate the scalability and performance of the synthesis, we created five different sets of 500 strings. Each bin of 500 strings had the same length from one to 16 letters and had random outcomes. The time performance of the synthesis over this dataset is illustrated in Figure 4.9.

Table 4.4: Comparison of temporal patterns and Gandhi-Washington Method.

Approach	Solution	Confinement	Instances
Apriori-based	Construct all the possible sequences by generating the candidate sequences and build patterns one item at a time iteratively and traverse the search space.	Any sub pattern of a frequent pattern must be frequent as the measure for pattern interestingness.	Apriori [5] SPADE [278] SPAM [16] GSP [250]
Pattern-growth-based	Search of patterns in a specific part of a given database by making the suffix and prefix trees of data. These algorithms do sequence pruning in order to prune candidate sequences early in the process.	Based on sampling and compression. Looking into frequency and periodicity as the measure for pattern interestingness.	FreeSpan [85] PrefixSpan [86] SPIRIT [71]
Temporal patterns	Search for the patterns of interactions in a time ordered input sequence.	Looking into frequency, length, and periodicity as the measure for pattern interestingness	Time-based methods [90] [272]

Table 4.5: Comparison of temporal patterns and Gandhi-Washington Method.

Mining temporal sequences	Gandhi-Washington Method
Consists of events with no natural points indicating the start or stop of an event.	Start and stop of events are defined be consecutive version releases i.e., the release cycles.
Use event-folding technique (sliding time window) to partition sequence of event.	Natural sequence of events are de-fined by releasing a new product into the market.
Create additional candidate episodes from subset of maximal episodes (top-down approach; from genera to more specific).	Create artificial nodes (bottom-up from the most specific to the more general regular expressions).
Use frequency, preciosity and length of patterns for interestingness.	Use of statistical inferences on a context specific factor (outcome).
Looking for patterns that follow interval sequences as a pair of (timestamp, event).	Extracting patterns of timestamps and characterize them by events.



(a) Sequential pattern mining methods mine patterns from recurring sequence of events happened in an order

(b) Temporal patterns mine the sequence of events on a time line

(c) The Gandhi-Washington Method mine the sequence of events in association with a performance measure

(d) Scott-Knott test group a performance measure based on ANOVA test

Figure 4.10: Comparison of the input data for sequential pattern mining, temporal pattern mining, Gandhi-Washington Method and Scott-Knott approach. In the figure a, b, c are the events and V_1 to V_5 are the values of the impact factor.

4.6 Limitations

GWM facilitates the structuring and packaging of knowledge gained from empirical investigations. However, there are various limiting factors for its applicability:

- **Scope:** GWM is applicable for empirical studies that are concerned with understanding the impact of structure and impact of sequences of activities and events on a performance measure. If a study does not target mining and understanding sequential data, then GWM is not applicable.

- **Granularity:** The idea of GWM is finding commonality in sequences and their outcome. The definition of structural commonality as done in abstraction imposes abstraction from details in the sense that sequences $\{A,A,A,A,A,A,B\}$ and $\{A,A,B\}$ are considered the same

(expressed as A^*B).

- **Cognitive limitation of modeling:** The encoding phase of GWM needs human experts to define items in a way to model the problem. Different modelings of the items would solve different problems. Correct modeling for solving the right problem is essential in GWM. Encoding is analysts' responsibility and human error is unavoidable which may threat the validity of the results.

- **Run time and algorithm complexity**: So far we discussed the run time and complexity of the method using up to five categories for encoding. The relation between regular expressions make it hard to define the enumerated hierarchy of items. It is expected that more computational resources are needed for higher number of categories.

4.7 Related work

In *software engineering*, patterns are used to "encapsulate knowledge for constructing successful solutions to recurring problems" [218]. Patterns have been used on:

- Code-level constructs: such as program generation, re-usability, and code defects,

- Design level constructs: such as design skills, and architecture design, or

- Knowledge and communication: such as work experience, pattern organization, and documentation [218].

Pattern selection, application, and modularity of patterns in software engineering were evaluated in most cases by domain experts, researchers, or participants in an empirical study [218].

Xie et al. [274] and Hassan and Xie [89] defined three broad categories for software engineering data, naming *sequences*, *graphs*, and *text*. We focused on sequential data in this paper. Xie et al. introduced "execution traces collected at run time, static traces extracted from source code, and co-changed code locations" as the most prominent examples of sequential software engineering

data which are mainly related to programming, bug detection, maintenance, and debugging tasks. Software development deals with a large amount of data created from sequential activities, events, and decisions. One of the foreseen challenges for mining sequential data were complexity of data and mined patterns [274].

In the field of data mining, sequential pattern mining discovers frequent patterns in a database. Sequential pattern mining approaches targets databases with sequences of ordered events with or without the concrete notion of time [143] such as sequence of customer's transaction in an online store. A wide range of applications for these approaches has been discovered from web-access patterns to the analysis of DNA. Several solutions were proposed to efficiently mine sequential patterns and several comprehensive and comparative analysis of these approaches exist [143, 169, 199]. In Table 4.4 we summarized the three main methodologies and the instance of approaches that follows each method.

We compare and differentiate Gandhi-Washington Method with existing sequential pattern mining methods and clustering methods based on group means. Different algorithms for mining sequential data reflect different levels of information, and the choice of algorithm depends on the task's mining requirements [274].

Comparison of GWM with sequential pattern mining methods

Sequential pattern mining approaches are mainly categorized as Apriori-based [5] and pattern-growth methods [85]. These methods use frequency (in terms of user specified minimum support thresholds) and length as the measure for pattern interestingness. In one instance of sequential pattern mining approaches regular expressions are used [71] which enable users to express the specific category of sequential patterns that are of interest to them. The solution and constraints for all these approaches is compared in Table 4.4.

Song et al. [249] used Apriori-based algorithms to mine methods to predict defect associations and defect correction effort. Michail also used Apriori-based algorithms to mine patterns in code library usage [161]. Pattern growth-based algorithms were used by Lo et al. [137] to mine software

behavioral specification.

Gandhi-Washington Method analyzes the hierarchical relation between events by using regular expressions and merges the sequences when it cannot find statistical differences between them, while previous approaches used frequency, length, and periodicity as the factor to select patterns. With this aim, these approaches are fundamentally different while both target sequential and ordered itemsets. In Figure 4.10 we demonstrate the difference between sequential pattern mining methods and GWM.

The most well-known sequence is defined in the order of *time*. The Mining temporal sequences for discovering patterns [90] is searching for the patterns of interactions in a time ordered input sequence. This approach is looking into frequency, length and periodicity of patterns as the measure for interestingness. First, it partitions the event sequence into the maximal episodes and creates an initial set of candidates from these episodes. Second, the approach is generating additional candidate episodes as a subset of maximal episodes and evaluates the interesting episodes by computing the compression ratio to select the interesting episodes as the final patterns. The comparison between temporal pattern mining and the Gandhi-Washington Method is presented in Table 4.5.

Wasylkowski and Zeller [268] used temporal patterns to mine violations of operational preconditions in code. Also, Herzig and Zeller [92] mined temporal process patterns that encode key features of the software process and validate them automatically. Uddin et al. [258] proposed a method for mining temporal API patterns to detect API usage patterns in terms of their time of introduction into client programs.

Comparison of GWM with clustering methods

Using the analysis of variance to split treatments into homogeneous sets [239] formed a class of clustering methods. This specific class of clustering algorithms introduced and known as Scott-Knott test [239] which use group means to partition the data.

Scott-Knott is a hierarchical clustering algorithm that creates non-overlapping groups of treatments using the ANOVA test. The method of Scott-Knott uses a hierarchical and divisive clustering

method imposing a hierarchical groups of means. In each step, the best clustering is selected by the sum of squares within groups. Scott-Knott's termination criterion is based on results of ANOVA test at each stage of the procedure which were corrected for Type I error [37]. Tian et al. [256] used Scott-Knott test to rank feature importance in characterizing high rated mobile apps.

In comparison to Scott-Knott, the Gandhi-Washington Method considers the sequence of event occurrences along with the distribution of treatments and hence different. This makes GWM suitable for analyzing impact of processes and decision sequences on the software metrics. This is visualized in Figure 4.10.

We compared the input data of different methods mentioned above in Figure 4.10 and in Table 4.4.

4.8 Summary

The Gandhi-Washington Method is an approach to analyze the sequence of recurring events, and items in relation with a context specific performance indicator. GWM represents a structural and unified method to determine the effect of different software engineering decisions and event sequences on projects, processes and products performance. GWM combines the use of regular expressions with the application of statistical tests. The encoding phase provides a flexible means for analysts to model the problem using a set of alphabets. Abstraction and synthesis are automated steps in GWM which condense the data and later aggregate sequences of items objectively and based on their common effect on a software performance measure.

GWM shows which recurring sequences in software processes do significantly affect performance measures and retrieve them as TrOCs. From the potentially broad range of applications, we demonstrated usefulness of GWM on code editing, code ownership and release cycle time analysis. Like most statistical methods, GWM is not intended to claim causality, as confounding factors cannot easily be excluded and the patterns retrieved by GWM are tied and limited to the context specific performance measure and changing this context specific factor in the process of GWM

may resulted in different patterns. Also, further analysis is needed to measure the sensitivity and robustness of the results gained in dependence of the underlying datasets.

Acknowledgments

We wish to thank Enoch Tsang, Kurtis Jantzen, Samarth Sinha, and Shane Sims for their help on the tool implementation of this research. We are grateful to Bram Adams for helpful discussions on a former version of the paper. This research was partially supported by the Natural Sciences and Engineering Research Council of Canada, NSERC Discovery Grant 250343-12 and Alberta Innovates Technology Futures.

Part III

Objective 2: Release Planning Models and Formulation for Mobile Apps in Presence of Multiple Similar Applications

Chapter 5

Optimized Functionality for Super Mobile Apps

Authors: Maleknaz Nayebi; and Guenther Ruhe RE 2017

Abstract - Functionality of software products often does not match user needs and expectations. The closed set-up of systems and information is replaced by wide access to data of users and competitor products. This shift offers completely new opportunities to approach requirements elicitation and subsequent planning of software functionality. This is, in particular true for app store markets. App stores are markets for many small sized software products which provide an open platform for users to provide feedback on using apps. Moreover, the functionality and status of similar software products can be retrieved. While this is a competitive risk, it is at the same time an opportunity.
In this paper, we envision a new release planning approach that leverages the new opportunities for decision making. We propose a new model using bi-criterion integer programming. We make suggestions for optimized super app functionality that are based on two key aspects: (i) the estimated value of features, and (ii) the cohesiveness between newly added features and cohesiveness between existing and the features to be added. The information on these attributes comes from reasoning on feature composition of existing similar apps. The approach is applicable to the development of new product releases as well as to the creation of completely new apps. We illustrate the applicability of our model by a small example and outline directions for future research.

This paper has been published in the proceeding of International conference on Requirements Engineering (RE 2017) by ***Maleknaz Nayebi****, and* ***Guenther Ruhe*** [1].

[1]Authors are with SEDS lab at University of Calgary

5.1 Introduction

A paradigm shift in requirements engineering and software evolution towards data-driven processes in open environments was projected by Maalej et al. [142]. Ease of access to users feedback and usage data, the pro-active participation of users through social media, and the maturity of analytics tools and algorithms opened new opportunities for requirements engineering arise. This is particularly true for software app markets. Big data on similar products, together with access to customers' feedback, provide a unique way to develop new software products based on the success and failure of similar products. Our previous study with developers and users [174] showed that app developers look into their competitors to decide on their software releases and updates. All the available data openly accessible in the app stores can be used for planning apps' evolution.

Release management is a decision-centric process and part of incremental and iterative software evolution. In particular, *release planning* is the process of assigning features to upcoming releases (or iterations) such that the overall product evolution is optimized. Release planning approaches for answering *what* and *when* to release problems have been modeled and solved by different researchers and with different formulations, methods, and solutions. Aspects of release planning for mobile apps were studied in [158, 174, 178, 263]. However, none of these studies provided an optimized solution approach with reference to data analytics for reuse of features and their cohesiveness.

In this paper, we outline the direction to compose "super" mobile apps by combining functionality available across existing products with similar purposes. Deciding on the functionality of a new product release is of fundamental importance for product success. The key dilemma is that often users do not know what they want or they changed their mind quite often and quite rapidly. In the case of widely accessible apps and their existing functionality, one way to approach this problem is to extract and reuse features that have proven successful and offer them in conjunction with the current features. We provide a new formulation for proposing optimized super app functionality. The formulation applies in the same way to the brown field and green field product

planning of mobile apps:

Brown field design: Design of a new release for an existing app, and

Green field design: Design of a completely new product from scratch.

In both cases, the idea is to reuse existing functionality and to combine it in an innovative way. In particular, we answer the below research question:

Research question: *How to determine value-cohesiveness-optimized app functionality by reuse of features from similar apps retrieved in the app store?*

The main contribution of the paper is a new method to find the best set of features for a new release of a mobile app using optimization techniques. We formulate this search for optimized apps as a bi-criterion optimization problem. Using quadratic integer programming [46], we search for the trade-off between the total value of a release and the total cohesiveness of the features selected for the new version of an app. We demonstrate these concepts and the related approach by an illustrative example.

5.2 Information Needs

Search for super app functionality is based on various sources of information. We describe them in the sequel.

Similar apps

App markets provide access to a pool of software products, some of them are designed for a similar purpose. We call them *similar apps*. This gives users the opportunity to use a similar product if they are dissatisfied with a particular app. The openness of these platforms on providing meta-data such as app descriptions, reviews, and app ratings makes it easier to perform a holistic search for finding similar apps.

When publishing an app in the market, the owners classify their apps in a specific category. Looking into these categories is one way to find similar apps. For example, the category "travel and local" includes a variety of apps for booking hotels, finding flights, restaurants, etc. The other way to identify similar apps is to search app stores with specific keywords pertaining to the desired purpose such as booking a hotel. To this end, one might search "booking AND hotel" in the app store.

Feature Extraction

Feature extraction of mobile apps was approached either by mining app descriptions or app reviews [155]. Analyzing app reviews to extract feature requests and bug reports were comprehensively approached by Maalej and Nabil [141].

Feature extraction from app descriptions has been studied in recent years [155], and a method was proposed by Harman et al. [88] to extract *featurelets* from app descriptions. The authors defined featurelets as "a set of commonly occurring co-located words using bi-grams and then aggregate similar featurelets into features using clustering [88]. For our proof-of-concept implementation, we used this method. However, alternative methods can be used [155]. Mining app descriptions and app store reviews do not provide a complete set of app features and looking into alternative information sources has been recommended [180].

Feature value

The question of what constitutes the value of a feature is difficult to answer. Value definition is context specific and user specific. A comprehensive software value-map is suggested by Khurum et al. [116]. Recent studies on mobile apps, used rating, the number of downloads, the content of reviews, and sentiment of reviews as the indicator of users' perceived value. In the illustrative example provided later in this paper, the value of app features is determined from crowdsourcing [142].

In the proof-of-concept example, by presenting the app features to Amazon Mechanical Turk

workers (who are considered knowledgeable in using apps), we ask them to assign a value between 0 to 100 to each feature. The final feature value is the average of the values defined by all users. The value indicates how desirable the feature for a (potential) user is expected to be.

Cohesiveness between pairs of features

There are multiple technical and non-technical benefits from offering features in conjunction. While cohesiveness is difficult to measure, we propose a proxy measure combining feature co-occurrences and apps' rating. Two features $f(n)$, $f(m)$ are considered to be (functionally) connected if they often co-occurred within the set of existing apps, and even more so if this co-occurrence is happening for apps with high ratings.

In addition to the number of co-occurrences, we also consider the rating of apps. The app rating is provided by users in the app market. Rating is expressed on a five-point-scale ranging from 1 (lowest) to 5 (highest). The rating of an $app(k)$ is called $rating(k)$ which is the average of all customer ratings within the apps' life cycle. The product of average rating and number of co-occurrence is called *feature cohesiveness*. The cohesiveness between two features is the higher, the more often they co-occur among all the existing apps and the higher the average rating of these apps considered for co-occurrence. Formally, for any pair of features $f(n)$ and $f(m)$

- $\alpha(n,m)$ represents the number of co-occurrences of these features among all apps currently available in that category.
- $\beta(n,m)$ represents the average rating across all the apps offering $f(n)$ and $f(m)$ in conjunction.

For example, considering similar apps in the category of *travel and local*, the feature *search a location* and the feature *sort results base on distance from a location* often co-exist in these apps. However, the features *counting calories* and *searching a location* co-occurred only twice across all apps from the app store. We observed that two features *counting calories* and *searching a location*

in the app category of *travel and local* co-occurred in two existing App 1 and App 2. App 1 with $rating(1) = 3.0$ and App 2 with $rating(2) = 3.5$. This means that $\alpha(n,m) = 2$, and $\beta(n,m) = 3.25$.

Effort estimation

Implementation of features consumes effort. We make the simplifying assumption of just looking at the total amount of (estimated) effort needed per feature. Different methods are applicable for providing estimates [245]. Often, effort estimation methods are hybrid in the sense that they combine formal techniques with the judgment and expertise of human domain experts. For our proof-of-concept example, we used three context expert app developers and each of them estimated the effort for implementing each feature in person hours.

5.3 Problem formulation

Inspired by state of the art release planning methods [173], we provide a novel formulation for proposing optimized super app functionality. The key idea is to look at two objectives and formulating the search for reuse-based app functionality as an optimization problem:

Objective 1: The total value of an app is defined as the sum of the feature values elicited from existing similar apps. This objective emphasizes the individual attractiveness of features for users which is the traditional linear value function used in release planning [228].

Objective 2: The degree of cohesiveness of features measured by their co-occurrence in existing apps. This objective is independent of the first one and looks at how often (quantity), and on which occasions pairs of features co-occurred in apps within similar context (quality).

The first optimization objective is purely value related. The actual value of new features is difficult to predict. In the case of brownfield development, one way to measure feature value is to look at its usage. Other alternatives were elaborated by Khurum et al. [116].

The second search objective emphasizes the cohesiveness of the features as observed from their former occurrence. Both objectives are independent and important. The projected value of features is what makes the new release valuable. The cohesiveness of features ensures that semantic connections between pairs of "successful" combinations of features should be maintained. This second objective is important as we are targeting semantically cohesive apps, not just a collection of individually high valued but unrelated features as studied for theme-based release planning [111].

Let $F1 = \{f(1)\ldots f(N1)\}$ be a set of features implemented in the current product version called *version y.z*, and $F2 = \{f(N1+1)\ldots f(N)\}$ the features extracted across a set of similar apps for being candidates for designing super app functionality of *Version* $y.z+1$. Each new app release *version* $y.z+1$ is characterized by an N-dimensional Boolean decision vector x with components $x(n)$ defined as:

$$x(n) = \begin{cases} x(n) = 1 \text{ if feature f(n) is offered, and} \\ x(n) = 0 \quad \text{otherwise} \end{cases} \tag{5.1}$$

By definition,

- each feature $f(n)$ from $F2$ has occurred in at least one of the existing apps, and
- $x(n) = 1$ for all $n = 1...N1$.

Each feature $f(n)$ has a perceived individual value called $value(n)$. Upcoming feature value is inherently difficult to predict. We consider crowdsourcing, user forums or stakeholder evaluation as possible options to estimate value. Our first objective is to maximize the total value of a release. It is expressed by value function $C1(x)$ and defined as:

$$C1(x) = \sum_{n=N1+1\ldots N} value(n) \times x(n) \tag{5.2}$$

Each feature $f(n)$ has an estimated effort called $effort(n)$ needed for its implementation. With an assumed total effort capacity Cap available for implementation of version $y.z+1$, all new app

releases encoded in vector x need to fulfill the effort constraint expressed as Equation (3):

$$\sum_{n=N1+1...N} effort(n) \times x(n) \leq Cap \tag{5.3}$$

This formulation so far looks like a traditional release planning problem [79] with just one release. Exclusively applying Equation (2) as an objective for optimization would result in a composition of features maximizing the total release value. What we potentially get is a collection of attractive features, but not necessarily a set of features that have been proven to be cohesive with regards to the experience.

To address this additional concern, we formulate our second objective $C2(x)$ with the target to maximize total cohesiveness of the features selected for the super app functionality. As outlined above, for all pairs $f(n)$, $f(m)$ of features we use the product of the number $\beta(n,m)$ of co-occurrences with its average rating $\alpha(n,m)$ as a proxy for their cohesiveness. In other words, the more often two features co-occurred, the higher their cohesiveness. Also, co-occurrence of two features in the apps with higher rating increases cohesiveness compared to apps with lower rating.

The objective then is defined as Equation (4) with the summation taken over all pairs of features $f(n)$, $f(m)$:

$$C2(x) = \sum_{f(n),f(m)} \alpha(n,m) \times \beta(n,m) \times x(n) \times x(m) \tag{5.4}$$

Consideration of semantic cohesiveness between features was studied for the purposed theme-based release planning [111]). The novel part in our formulation is that for the selection of additional features, the cohesiveness between them, as well as the connectivity to existing features is taken into account. This is something that is ignored in all existing release planning approaches but should be considered important to ensure features' cohesiveness.

Both objectives $C1(x)$ and $C2(x)$ are independent and are competing. While we want to maximize the value, we also want to maximize the cohesiveness of the features. This is because best-valued apps are not automatically best in terms of cohesiveness and vice versa. In other

words, we are looking for Pareto solutions for the bi-objective optimization problem to maximize $(C1(x), C2(x))$.

We investigate the question: *"For a given app release, which set of features should be added to increase the total value of the app and to maximize the cohesiveness between all pairs of features of the new release."*

All app releases fulfilling the effort constraint of Equation (3) are called *feasible app releases*. An app release x^* is a *Pareto solution* if no other feasible app release x' exists that is better in one criterion (value or cohesiveness) and at the same time not worse in the other. This is expressed in Equation (5):

$$(i) \quad C1(x') \geq C1(x^*) \quad \textit{and}$$

$$\text{(ii) C2(x')} \geq C2(x^*) \quad \textit{and}$$

$$\text{(iii)} \quad \text{(C1(x'),C2(x')} \neq (C1(x^*), C2(x^*)) \qquad (5.5)$$

For simplicity, we do not look at additional (detailed effort or technology related) constraints. Note that this technically would not change the proposed solution approach as we keep value and cohesiveness as the optimization objectives.

5.4 Solution Approach

So far, we discussed possible methods for similarity analysis to find similar mobile apps as well as methods for extracting mobile app features, effort and value estimation in Section 5.2 and Section 5.4. Main steps of the solution approach are outlined in Figure 5.1.

For this paper, the emphasis is on the search process for finding an optimized set of feature for extending app functionality over releases. This problem formulation (Equations (1) to (5)) by nature is an Integer Quadratic Programming problem [81]. Each particular vector x with integer variables as defined in Equation (1) describes a specific composition of a new app release. The problem has a quadratic component as it contains combinations of two features as expressed in the

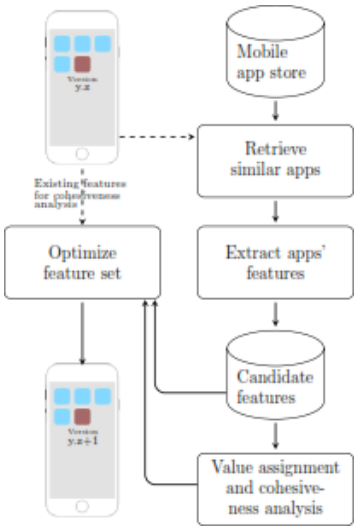


Figure 5.1: Key steps of the process of finding optimized super app functionality. The two dashed arrows are only applicable for brown field development.

second objective $C2(x)$.

To solve the problem, one can transforms the original bi-objective optimization problem into a sequence of single objective problems [46]. Following that, we have transformed our problem into a sequence of problems $G(\gamma)$:

$$G(\gamma,x) = Max\{\gamma \times C1(x) + (1-\gamma) \times C2(x) : \text{x fulfills (3)}\} \quad \text{for all } \gamma \text{ from (0,1)} \tag{5.6}$$

The result of Equation (6) is a set of features for to be implemented in a new app release. From multi-criteria integer programming [46] it is known that the feature sets we receive from solving a series of the parametric single-criterion problems are Pareto optimal. For computing optimal (super) app releases for our proof-of-concept evaluation, we used an academic license of the proprietary Gurobi optimizer [82]. While the solution approach does not depend on this tool, Gurobi has proven to be both computationally efficient and scalable [82]. In a comparative analysis with top commercial solvers `CPLEX` (from IBM), `XPRESS` (from Fico) and also comparing to open source solvers such as `CBC`, `Gurobi` performed best concerning run time and the percentage of

solved instances from a pool of benchmark problems[2].

5.5 Proof-of concept evaluation

We demonstrate the key idea of the approach by a small illustrative example which was taken from a more comprehensive case study formerly conducted [173]. The application context is *Over the Top TV (OTT)* apps (such as Netflix or Hulu) taken from the Android app store (Google Play). In this category, we gathered a set of 84 similar apps. Using the method proposed by Harman et al. [88], we extracted 10 features of OTT apps (which is a random subset of the total set of 34 features originally extracted). For all these features, the observed implementation effort and the feature value (based on crowdsourcing) are shown in Table 5.1. We applied methods for crowd-sourced feature evaluation and effort estimation as they were described in Section 5.2. We investigate the question:

For a given OTT app and for given capacity to develop the next release, which features should be added to increase the total value of the app and to maximize the total feature cohesiveness?"

To illustrate our main idea, we assume that the first four features listed in Table 5.1 constitute the current version *Version 3.7* of the app under consideration, denoted by App^*. That means, App^* currently has features $f(1)$, $f(2)$, $f(3)$, $f(4)$. Different combinations of features $f(5)$, $f(6)$, ...,$f(10)$ are candidates for creating the next release *Version 3.8*.

Following the problem formulation in Section 5.3 to answer the above question we consider:

(i) The perceived individual value of each candidate feature $f(5)$, $f(6)$,...,$f(10)$, and

(ii) The degree of cohesiveness between all the pairs features. This includes the cohesiveness between existing features $f(1)$, $f(2)$, $f(3)$, $f(4)$ and newly added features as well as the cohesiveness of the newly added features with each other.

[2] `http://plato.asu.edu/ftp/milpc_tables/1thread.res`

Table 5.1: Feature set of the illustrative example for optimized super app design and its evaluation. Features f(1), f(2), f(3) and f(4) have been already implemented.

ID	Features	Value	Effort (in person hours)
f(1)	Live channel coverage	58.2	10.4
f(2)	Support of multi-screen	77.4	27.7
f(3)	Switching between horizontal and vertical views	49.1	10.3
f(4)	EPG	28.6	3.8
f(5)	Aspect ratio change	63.5	37.9
f(6)	Remote control	37.9	26.3
f(7)	Support for devices without a touch screen	43.8	20.4
f(8)	Video on demand	91.3	53.8
f(9)	YouTube integration	26.3	13
f(10)	Capability to select source signal	57.3	25.8

For (ii), the assumption is that features that have been occurred together in the past in (successful) competitor apps are good candidates to offer them in conjunction again. One reason might be that the two features depend on each other. Another one is that their co-occurrence creates value synergy. We provided the matrix of feature co-occurrence and average ratings as Table 5.2. Therein, the values above the diagonal of the main matrix represent the average rating values ($\beta(n,m)$), the ones below the diagonal represent the number of co-occurrences ($\alpha(n,m)$). We observe that feature pairs $f(3)$, $f(8)$ and $f(5)$, $f(8)$ occur most frequently. At the same time, the pair $f(7)$, $f(8)$ co-occurs in one app with Rating 5.

Having feature value, cohesiveness and estimated implementation effort for features as discussed in Section 5.2, we assumed $Cap = 58.3$ as the implementation effort budget for Version 3.8. Then, we applied formulation (6) and obtained four optimized alternatives for extending the current release Version 3.7 by adding features to create the new release Version 3.8:

Optimized Alternative O1: Adding f(9) and f(10) with (C1(x), C2(x)) = (296.9, 96.6)

Optimized Alternative O2: Adding f(7) and f(10) with (C1(x), C2(x)) = (314.4, 71.3)

Optimized Alternative O3: Adding f(4) and f(9) with (C1(x), C2(x)) = (303.1, 87.9)

Optimized Alternative O4: Adding f(4) and f(7) with (C1(x), C2(x)) = (320.6, 61.5)

Table 5.2: Cohesiveness $\beta(n,m)$ (below the diagonal) and average rating $\alpha(n,m)$ (above the diagonal) for all pairs of the 10 features.

Features	f(1)	f(2)	f(3)	f(4)	f(5)	f(6)	f(7)	f(8)	f(9)	f(10)
f(1)		0	4.9	3	0	0	0	3.4	0	0
f(2)	0		3.2	0	4.6	3.8	0	4.9	0	0
f(3)	2	4		3.4	4.1	0	4.7	3.1	5	4.9
f(4)	2	0	8		4.5	4.6	0	4.5	3.3	0
f(5)	0	4	5	3		0	4.9	3.4	4.1	0
f(6)	0	1	0	2	0		0	0	0	0
f(7)	0	0	1	0	2	0		5	4.6	4.9
f(8)	1	7	14	2	15	0	1		4	4
f(9)	0	0	2	8	3	0	2	2		3.9
f(10)	0	0	1	0	0	0	2	1	9	

All the four optimized feature sets are best in combining both value and cohesiveness for a new app release. That means, there is no alternative being better in one aspect and not worse in the other.

To visualize this, we formed all the possible feature combinations of $f(5)$, $f(6)$, $f(7)$, $f(8)$, $f(9)$, and $f(10)$ which needs implementation effort less than the assumed capacity of 58.3. We plotted the value and cohesiveness of these combinations along with the four optimized alternative solutions in Figure 5.2[3]. In Figure 5.2, Optimized Alternative 1 to 4 are denoted by $O1$ to $O4$. $A1$, $A2$, $A3$ and $A4$ are non-optimal alternatives of feature combinations with required effort less than the available capacity ($= 58.3$). $A1$ is composed of $\{f(7), f(9)\}$, $A2$ is $\{f(6), f(9)\}$, $A3$ composed of $\{f(6), f(7)\}$, and $A4$ is composed of $\{f(6), f(10)\}$.

5.6 Related Work

Sophisticated analysis on the mobile apps, including reviews, ratings, the number of downloads, and dynamic changes in the competitors' apps provides opportunities for proactive analysis and decision making within this context [155, 197]. The results of different studies showed that the user's feedback in the form of ratings and reviews have an impact on the app development decisions [197] and tools were investigated to assist app developers in reacting to their customers more

[3]While the problem is discrete we visualized the Pareto front in a continues format for simplicity

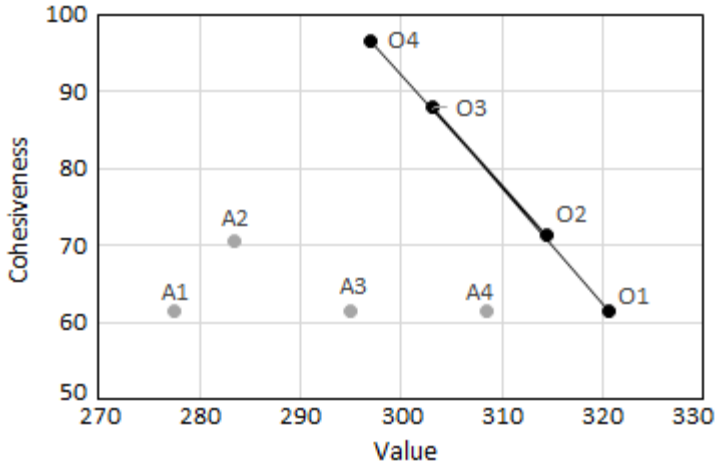


Figure 5.2: Existing apps and optimized super app.

efficiently [53]. App store mining and analysis has been widely studied in the context of software engineering from different aspects and reuse of code was discussed in number of studies [155]. However, requirement engineering and design of apps by reusing features from the other apps are rather untouched.

Seyff et al. [242] studied the usage of Facebook to overcome current limitations of requirements' engineering (RE) tools in terms of the end user acceptance and involvement. They found that existing features of Facebook, e.g., the possibility to build dedicated groups, comment on posts, or like posts, support RE activities such as requirements elicitation, prioritization, and negotiation. We believe our proposed method, is the first step toward systematically integrating the voice-of-the customer into optimized mobile app design.

In a recent systematic literature survey on release planning models, Ameller et al. [9] confirmed the increasing trends to look at multiple objectives. In line with that, in the design of super app functionality, we looked into feature value and cohesiveness between features as the objectives. Release planning of mobile apps were studied [158, 174, 178, 263]. However, none of these studies provide a formal formulation considering the established state of the art release planning or prioritization methods [228, 251] and often refer to prioritization of bug fixes and features as release planning of mobile apps.

5.7 Limitations

When looking at cohesiveness of feature combinations, we are currently limited to pairs of two features. However, additional synergies might occur for larger combinations (e.g., three or four features co-occurred often together). Also, detecting similar apps and their features is not an easy task and need further elaboration. Looking into the app store categories does not guarantee the similarity as the functionality range of apps in the same category is rather broad. However, searching for specific keywords, using domain experts, and snowballing the app store recommendations can be used in addition.

The scalability of the method needs to be proven. This is not only a computational but also an information retrieval and validity problem. Finding similar features and determining their value and cohesiveness is computationally intensive and is best working on the apps with adequate number of similar apps. We foresee the need for more comprehensive evaluation of the applicability of the proposed solution idea and have started applying the process to real-world app design [182].

5.8 Next RE Research

Offering the most attractive functionality to users is the ultimate goal of app release planning. We proposed a model to frame the functional design of apps as a systematic and optimization based decision-making process. So far, the limited proof-of-concept evaluation shows the applicability of our model to both new app development and extending existing apps with new functionality. The proposed model and its algorithmic implementation are flexible to accommodate more complex synergies and types of feature dependencies. With a powerful solver like Gurobi, solving quadratic integer programming is scalable and is expected to solve problems with several hundred of features.

Our initial idea was presented for brownfield app development considering an existing product in the market. However, the same idea can also be applied to enhance the functionality of a new app (Greenfield app design). The only difference is that the current set of existing features in the

app is empty. We evaluate the above ideas trough fully-fledged real-word case studies as the next steps of this research.

Finally, the current model looks exclusively at features that have been occurred somewhere before. However, there might be new features coming from other information sources. As outlined and evaluated by Nayebi et al. [182], features can be extracted also from the mining of social media platforms such as Twitter. by adjusting the cohesiveness function $C2(x)$ to reflect "requirements co-occurrence" instead of features' co-occurrence, the formulation and solution approach can be applied in the same way for this more comprehensive application scenario.

Acknowledgments

This research was partially supported by Alberta Innovates Technology Futures AITF (first author) and by the Natural Sciences and Engineering Research Council of Canada, NSERC Discovery Grant 250343-12 (second author).

Chapter 6

Asymmetric Release Planning

Authors: Maleknaz Nayebi; and Guenther Ruhe TSE 2017

Abstract - Maximizing satisfaction from offering features as a part of upcoming release(s) is different from minimizing dissatisfaction gained from not offering features. This asymmetric behavior of the value of features has been approached in the past by the Kano analysis method. However, it has never been utilized for release planning. We accommodate this asymmetry and study *Asymmetric Release Planning* (ARP). We have formulated and solved ARP as a bi-criteria optimization problem. In its essence, it is the search for optimized trade-offs between maximum stakeholder satisfaction and minimum dissatisfaction. As a prerequisite, and to increase the validity of the feature evaluation, we define a continuous Kano analysis that allows for modeling and analysis of the degree of membership to feature categories being *attractive*, *one-dimensional*, *must-be* and *indifferent*. As a proof of concept, the proposed solution approach called *Satisfaction-Dissatisfaction Optimizer* (SDO) is validated via a real-world case study project. From running three replications with varying effort capacities, we demonstrate that SDO generates optimized trade-off solutions being (i) of a different value profile and different structure, (ii) superior to the application of random search and heuristics in terms of quality and completeness, and (iii) superior to the usage of the fuzzy Kano analysis as an underlying feature classification model. An additional survey with 20 stakeholders confirmed the usefulness of the generated results.

This paper has been submitted to Transaction of Software Engineering (TSE) by ***Maleknaz Nayebi****, and* ***Guenther Ruhe*** [1].

[1]Authors are with SEDS lab at University of Calgary.

6.1 Introduction

Making proper decisions about the functionality of evolving software product releases are critical for the success or failure of a product. The process of understanding stakeholder needs and prioritizing them (see [2, 25, 220]) is a prerequisite for making good release decisions. Without a proper understanding of features and their value, product development becomes risky. Some features generate satisfaction, but not automatically create dissatisfaction if they are not offered. Vice versa, some features might create high dissatisfaction if not offered but not necessarily create satisfaction if offered. We call this relationship between customer satisfaction and dissatisfaction an *asymmetric value impact* of a feature. The release planning process in consideration of asymmetric value impact of features is called *Asymmetric Release Planning* (ARP).

Customer [2] value or customer satisfaction were considered by several release planning approaches [251], but none of them handled the conjoint effect of satisfaction and dissatisfaction on planning for future releases. This says, by considering satisfaction or dissatisfaction as one planning objective, then the other objective should be considered as well. The asymmetric behavior of feature value requires a different modeling and solution approach when compared to the former (symmetric) planning methods because the results are expected to be different.

The main research questions addressed by this paper are:

RQ1: (Modeling) Which models exist for measuring satisfaction and dissatisfaction in the release management process?

RQ2: (Method) What release planning method creates trade-off solutions by accommodating the asymmetry between satisfaction and dissatisfaction?

RQ3: (Evaluation) For a real word case study, and for the method found in RQ2, how do the optimized plans (i) compare to plans generated randomly or by greedy search, (ii) are ranked

[2] While formally a *customer* is a specific form of a *stakeholder*, both terms are used interchangeably in this paper to accommodate the terminology used in different contexts such as Kano (customer) analysis and stakeholder driven release planning.

by stakeholders, and (iii) compare to manual plans created by managers in terms of structure and quality?

The paper is subdivided into eight sections. To better illustrate the key idea of the paper, a motivating example is introduced in Section 2. In Section 3, modeling and problem statement of ARP are presented. This is followed in Section 4 by the description of our proposed solution approach SDO. In Section 5, we present a case study. It serves as validation of the problem formulating ARP and as evaluation of SDO. Applicability, limitations, and usefulness of SDO are further discussed in Section 6. We reported related work in Section 7. Finally, in Section 8, we provide conclusions and an outlook to future research.

6.2 Motivating Example

In this section, we motivate the use of ARP by a small sample project. For simplicity, we assume that we just have nine features F1 ... F9 and just one customer. All features have been evaluated by the customer in terms of satisfaction and dissatisfaction. Feature priorities are defined on a nine-point scale ranging from 1 (very low) to 9 (very high). For example, a satisfaction score of 9 for a feature F1 means that the customer is very satisfied if this feature is offered. However, this does not tell anything about the dissatisfaction of the customer in case the features are not delivered. Similarly, a dissatisfaction score of 9 for feature F9 means that the customer is very dissatisfied if this feature is not offered. Feature F7 has the same degree of dissatisfaction but differs in the degree of satisfaction once it would be offered. The resulted feature priorities are shown in Table 6.1.

All features F1 to F9 behave asymmetrically as they impact customer satisfaction and dissatisfaction unequally. They consume an (simplified) effort of a one-person day to be implemented. We further assume that the project has a capacity of three person days to implement features. Also, we only plan for the features of one (the next) release.

In total, there are $(9 \times 8 \times 7)/6 = 84$ possibilities to pick three features for the next release.

Table 6.1: Satisfaction and dissatisfaction score of features.

ID	Feature	Satisfaction score	Dissatisfaction score
F1	Instant streaming	9	1
F2	Multi-casting	9	2
F3	Replay	9	3
F4	Video on demand	8	4
F5	Playlist	7	7
F6	Video recommendation	4	8
F7	Video history	3	9
F8	Parental control	2	9
F9	Share video	1	9

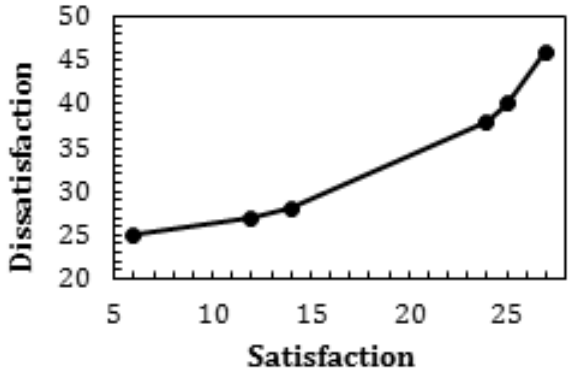


Figure 6.1: Visualization of solutions listed in Table 3.

Our proposed solution will determine six plans P1 ... P6. The plans are described in Table 6.2. P1 provides the least satisfaction (= 6) but is best in the sense that it causes the least dissatisfaction (= 25). On the other end of the spectrum, P6 provides the highest satisfaction (= 27) but also causes the highest dissatisfaction (= 46). The values of the six plans are plotted in Figure 6.1[3].

In Table 6.2 the satisfaction of a plan is the sum of all individual feature satisfactions of that plan. Plan $P1 = \{F7, F8, F9\}$ creates a satisfaction of $3+2+1 = 6$. The dissatisfaction of $P1$ is the total dissatisfaction of all features not being offered, i.e., $1+2+3+4+7+8 = 25$.

We discuss three planning scenarios:

Scenario 1: Planning is directed towards maximizing satisfaction. This scenario would result in the unique solution P6 of implementing feature set $\{F1, F2, F3\}$.

[3]While the problem is discrete we visualized the Pareto front in a continues format for simplicity

Table 6.2: Asymmetric release plans (to be) found by SDO.

Plan ID	Feature sets	Satisfaction	Dissatisfaction
P1	F7,F8,F9	6	25
P2	F5,F7,F8	12	27
P3	F5,F6,F7	14	28
P4	F3,F4,F5	24	38
P5	F2,F3,F5	25	40
P6	F1,F2,F3	27	46

Scenario 2: Planning is directed towards minimizing dissatisfaction. This scenario would result in the unique solution P1 implementing feature set $\{F7, F8, F9\}$.

Scenario 3: Planning is based on both maximizing satisfaction and minimizing dissatisfaction (with equal weights). This scenario would result in two solutions P3 and P4 with feature sets $\{F5, F6, F7\}$ and $\{F3, F4, F5\}$, respectively.

We noticed that all solutions corresponding to the three scenarios are not only different in value, but also largely different in terms of the features offered. In summary, our proposed asymmetric release planning approach SDO will overcome former symmetric planning limitations in two main aspects:

(i) SDO considers the impact of both satisfaction and dissatisfaction that is caused by providing (resp. not providing) features.

(ii) SDO generates not just one solution, but a portfolio of trade-off solutions, compromising between satisfaction and dissatisfaction.

6.3 RQ1: Modeling Feature Satisfaction and Dissatisfaction

The results of various studies showed that the relationship between satisfaction and dissatisfaction is asymmetric (see for example [223], [205]). So far, in software engineering, the conjoint relation between satisfaction and dissatisfaction has been ignored. Khurum et al. [116] provided a comprehensive value map for software products and reported that acquiring value for comprehensive

feature prioritization needs careful analysis. Thus, software feature prioritization methods and, even further, release planning approaches have assumed that priorities are coming from stakeholders.

So far, conjoint selection of satisfaction and dissatisfaction for prioritization or planning is optional. For example, in EVOLVE II, planning criteria could be *satisfaction* and *time to market*. However, a deeper analysis of users' satisfaction and dissatisfaction behavior showed that these two factors are two sides of the same coin and creation of satisfaction undeniably create dissatisfaction [70].

Selecting a model for measuring satisfaction and dissatisfaction is context specific. Hence, in what follows, we describe three models to predict the impact of offering or missing features in upcoming releases. The questions to be answered per feature and by all stakeholders are:

One-point estimates: The degree of satisfaction and dissatisfaction per feature and per stakeholder is the result of a one-point estimate.

Pairwise comparison of features: The degree of satisfaction and dissatisfaction per feature and per stakeholder is the result of aggregating pair-wise comparisons between features.

Kano analysis: For each feature a customer should answer functional and dysfunctional question. Satisfaction and dissatisfaction are calculated as the result of answering these two questions.

One-point estimates

Nine point evaluation of features have been used in software release planning and feature prioritization literature [228]. Applying it to ARP means that each feature is evaluated by all stakeholders twice to express her satisfaction of receiving a feature and her dissatisfaction of not receiving the same feature.

Satisfaction: To what extend you would satisfied by receiving this feature?

Dissatisfaction: To what extend you feel dissatisfied from not receiving this feature?

Because of the asymmetry between the two criteria, both questions have to be answered all the time. To express the satisfaction and dissatisfaction, each feature is evaluated by each stakeholder based on a nine-point scale defined below, where all even scores are described as the values between the two neighboured odd values.

1 - Very low 3 - Low 5 - Medium 7 - High 9 - Very high

For feature $F(n)$ and stakeholder s, the answers result in scores *sat(n,s)* and *dissat(n,s)*. To differentiate between stakeholders importance, the weight of the stakeholder $wstake(s)$ is applied. It ranges from very low (= 1) to very high (= 9). A weight $wstake(s) = 0$ indicates that the stakeholder is not considered at all.

We compute weighted averages $S(n)$ and $DS(n)$ expressing the estimated total impact on satisfaction respectively dissatisfaction when delivering (not delivering) feature $F(n)$. The weighted averages are defined in Equations (1) and (2):

$$S(n) = \sum_{s} wstake(s) \times sat(n,s) / \sum_{s} wstake(s) \tag{6.1}$$

$$DS(n) = \sum_{s} wstake(s) \times dissat(n,s) / \sum_{s} wstake(s) \tag{6.2}$$

Impact prediction from pair-wise comparisons

As an extension of the one-point estimates, applying the Analytical Hierarchy Process (AHP) [234] in this context means to perform pair-wise comparisons between all pairs of features from the perspective of both satisfaction and dissatisfaction. AHP allows decision-makers to assign ratio scale priorities to alternative features applying a sequence of pair-wise comparisons.

The process is applied independently for satisfaction and dissatisfaction perspective. Assuming two features called $F(n)$ and $F(m)$, a nine-point scale ranging from equally preferred (value = 1) over strongly preferred (= 5) to extremely preferred (= 9). A pair-wise comparison matrix M is used to describe all these scores. For example, if feature $F(n)$ is strongly preferred to feature $F(m)$

Table 6.3: Kano evaluation table [110].

Customer requirements		**Dysfunctional questions (D)**				
		(i) Like	(ii) Must-be	(iii) Neutral	(iv) Live with	(v) Dislike
Functional questions (U)	(i) Like	Q	A	A	A	O
	(ii) Must-be	R	I	I	I	M
	(iii) Neutral	R	I	I	I	M
	(iv) Live with	R	I	I	I	M
	(v) Dislike	R	R	R	R	Q

in terms of satisfaction (for a given stakeholder), then $M(n,m) = 5$ and $M(m,n) = 1/5$. Applied for both objectives, AHP converts these evaluations to numerical values which are used as scores $S(n)$ resp. $DS(n)$.

Once these values are determined, weighted averages taking into account stakeholder weights $wstake(s)$ are determined similar to Equations (1) and (2). It is known that the whole AHP process provides more reliable scores. At the same time, it needs additional effort in comparison to one-point estimates.

Kano model

The traditional Kano questionnaire [110] overcomes point-wise evaluation by including two questions related to each feature. The first question (called *functional*) evaluates customers' reaction related to the presence of a specific feature. The second question (called *dysfunctional*) evaluates customers' reaction in the absence of the same feature:

Functional question: If feature F(n) is offered in the product, how do you feel?

Dysfunctional question: If feature F(n) is not offered in the product, how do you feel?

The answer to both questions has five choices as outlined below. In the traditional model, each customer must select exactly one single answer to each question:

(i) I like it that way

(ii) It must be that way

(iii) I am neutral

(iv) I can live with it that way

(v) I dislike it that way

The Kano model categorizes features by combining the answers of the functional and dysfunctional questions [110]. As described in Table 6.3, each feature is classified into one of the following categories:

Must be (*M*): The prerequisite features that the customer assumed as granted. Delivering these features do not affect the satisfaction level but prevent the occurrence of dissatisfaction among stakeholders.

One-dimensional (*O*): The fulfillment of these features will linearly increment the degree of customer satisfaction.

Attractive (*A*): These features are not explicitly requested. The absence of these features in the offered product will not cause dissatisfaction although the presence of them is expected to lead to greater customer satisfaction.

Indifferent (*I*): The customer feels indifferent towards the availability or unavailability of these features. The feature might be considered as needed inside an organization, for example as an enabling technology.

Reverse (*R*): Stated that the customer not only does not want this feature but even expected the feature not to be in the product. These features are avoided as they just waste resources without any desirable impact.

Questionable (*Q*): This category is the indicator of a problem in phrasing the question or understanding the question. Therefore, these answers are not included in the analysis.

In what follows, we call these categories *Kano attributes*. In the traditional Kano model, each feature is assigned to exactly one of the above attributes. Based on this classification, Berger et

al. [26] defined product satisfaction and dissatisfaction coefficients. They proposed to count the number of Attractive (#A) and the number of One-dimensional (#O) features and relate this number to the sum of #A, #O, #M, and #I. This also means that R and Q attributed features do not count in this evaluation. Stakeholders' dissatisfaction with a product is defined analogously, with the focus on the number of Must-be features (#M) and a number of One-dimensional features (#O). The two formulas for estimating stakeholders' satisfaction and dissatisfaction of receiving a product are given as Equations (3) and (4) below:

$$Sat(P) = \frac{\#A + \#O}{\#A + \#O + \#M + \#I} \tag{6.3}$$

$$Dissat(P) = \frac{\#M + \#O}{\#A + \#O + \#M + \#I} \tag{6.4}$$

$Sat(P)$ is a coefficient for the stakeholder satisfaction expected from receiving product P (compare [237]). Similarly, $Dissat(P)$ is a coefficient telling how much providing product P avoids stakeholder dissatisfaction. From these equations:

- Increasing (#A) and (#O) will increase satisfaction. This aligns with reality, as the percentage of features contributing to satisfaction is increasing.
- Increasing (#M) will increase the amount of dissatisfaction. This aligns with reality, as the percentage of features contributing to dissatisfaction is decreasing.
- Increasing (#I) will both reduce satisfaction and dissatisfaction. This aligns with reality, as resources are wasted for features not making direct contributions.

These coefficients were extracted experimentally. Later, other researchers analyzed these and provided more sophisticated formulations [163]. We relied on Berger's coefficient for simplicity as the baseline [26].

The *continuous Kano model* overcomes the limitations of the traditional model as it allows stakeholders to express their sentiments by selecting multiple responses for each question. These

responses are aggregated with continuity in the feature scores (between categories) and include the degrees of importance per stakeholder. Extending the former fuzzy approach described by Lee et al. [128], continuous Kano analysis has four main steps:

Step 1 –Normalizing stakeholders' responses: For a given feature and a specified stakeholder, U_i, U_{ii}, U_{iii}, U_{iv}, and U_v [4] represent the normalized degrees of responses to the functional question. The different indexes refer to the five choices for each question. Normalization means that the sum of all five responses to a question is equal to 1. Similarly, D_i, D_{ii}, D_{iii}, D_{iv}, and D_v represent stakeholders' response to the dysfunctional question.

Step 2 –Calculating Kano attribute scores for each feature: We denote the result of the evaluation of the attractiveness of feature $F(n)$ in terms of Kano attribute A by stakeholder s as $score_A(n,s)$. This score is determined by the summation over all the values that correspond to A cells in Table 6.3.

$$score_A(n,s) = (U_i \times D_{ii}) + (U_i \times D_{iii}) + (U_i \times D_{iv}) \tag{6.5}$$

Equations (6) to (9) are defined following Lee et al. [128]. The equations are based on the classification of features given in Table 3. For example, $score_A(n,s)$ of Equation (7) is determined by summing up all parts of stakeholder's answers resulting in category A (Attractive).

The values $score_O(n,s)$, $score_M(n,s)$, and $score_I(n,s)$ are defined similarly. For One-dimensional, Must-be, and Indifferent Kano feature attributes, formulas are given as Equations (8), (9), and (10), respectively.

$$score_O(n,s) = (U_i \times D_v) \tag{6.6}$$

$$score_M(n,s) = (U_{ii} \times D_v) + (U_{iii} \times D_v) + (U_{iv} \times D_v) \tag{6.7}$$

[4] To keep notation simple, we excluded any reference to both the feature and stakeholder in this context

$$score_I(n,s) = (U_{ii} \times D_{ii}) + (U_{ii} \times D_{iii}) + (U_{ii} \times D_{iv}) + (U_{iii} \times D_{ii}) + (U_{iii} \times D_{iii}) + (U_{iii} \times D_{iv}) + (U_{iv} \times D_{ii}) + (U_{iv} \times D_{iii}) + (U_{iv} \times D_{iv}) \tag{6.8}$$

Step 3 –Calculating weighted averages for aggregating stakeholders' score: To differentiate between stakeholders' importance, we use the weighted average for aggregating stakeholder scores as described in Equation (9).

To compute the overall attractiveness $F_A(n)$ of a feature $F(n)$, the individual scores $score_A(n,p)$ from all stakeholders are added to form the weighted average of Equation (7). The scores related to the other continuous Kano attributes are defined correspondingly.

$$F_A(n) = \frac{\sum_s wstake(s) \times score_A(n,s)}{\sum_s wstake(s)} \tag{6.9}$$

Step 4 –Calculating feature satisfaction and dissatisfaction values: Modeling feature satisfaction and dissatisfaction follows the same idea as expressed for product evaluation by Berger et al. [26]. For this purpose, we extend the product related Equations (3) and (4) based on the number of (Boolean) occurrences to feature related (continuous) degrees of occurrences. This results in Equations (10) and (11). To calculate stakeholder satisfaction $S(n)$ of feature F(n), all stakeholder responses related to satisfaction elements (Attractive and One-dimensional) are divided by the sum of the Attractive, One-dimensional, Must-be and Indifferent portions of that feature:

$$S(n) = \frac{F_A(n) + F_O(n)}{F_A(n) + F_O(n) + F_I(n) + F_M(n)} \tag{6.10}$$

Similarly, $DS(n)$ is calculated by adding all responses with dissatisfaction elements (One-dimensional and Must-be) and dividing it by the total amount of relevant responses:

$$DS(n) = \frac{F_M(n) + F_O(n)}{F_A(n) + F_O(n) + F_I(n) + F_M(n)} \tag{6.11}$$

The detailed calculation of $S(n)$ and $DS(n)$ is illustrated in Appendix I.

6.4 RQ2: Asymmetric release planning

In this section we describe the objectives and constraints of asymmetric planning and introduce the solution approach SDO for providing trade off solutions to balance satisfaction and dissatisfaction.

Features

Decisions in product release planning are related to features and their assignment to releases. In the definition of features, we follow Wiegers and Beatty [269] who defined a product feature as *a set of logically related requirements that provide a capability to the user and satisfies the business objectives*. A more comprehensive list of release planning information needs is provided by Nayebi and Ruhe [173].

Let $F = \{F(1) \ldots F(N)\}$ be a set of N candidate features for development during the upcoming K product releases. A feature is called *postponed* if it is not offered in one of the next K releases. Each release plan is characterized by a vector x with N components $x(n)(n = 1 \ldots N)$ defined as:

$$x(n) = k \quad \text{if feature } F(n) \text{ is offered at release } k \tag{6.12}$$

$$x(n) = K + 1 \quad \text{if feature } F(n) \text{ is postponed} \tag{6.13}$$

Objectives: Satisfaction versus dissatisfaction

The objective of our planning approach is to maximize stakeholder satisfaction and simultaneously minimize stakeholder dissatisfaction. These two objectives are independent and competing with each other. Pursuing each objective in isolation will create different release planning strategies.

For modeling of the satisfaction objective, we follow the proven concepts of the EVOLVE based algorithms, in particular, the most recent EVOLVE II [228]. For a given time horizon of K releases, there is a discount factor making the delivery of a feature less satisfactory when it is offered later. While using a weighting (discounting) factor $w(k)$ for all releases $k = 1...K$, we assume that $w(K+1) = 0$, $w(1) = 1$ and

$$w(k) > w(k+1) \qquad (k = 1 \ldots K-1) \tag{6.14}$$

This assumption implies that the value of delivering a feature will be the higher the earlier it is delivered. For a plan x assigning features to releases, *Total Satisfaction* $TS(x)$ is defined in (15). It is based on the summation of the discounted feature values $S(n)$ taken over all assigned features and all releases.

$$TS(x) = \sum_{k=1...K} \sum_{n:x(n)=k} w(k) \times S(n) \rightarrow Max! \tag{6.15}$$

Total Dissatisfaction $TDS(x)$ of a plan x follows the same idea as just introduced for satisfaction. The longer a feature is not offered, the higher the dissatisfaction. Similar to satisfaction, we introduce factors describing the relative degree of dissatisfaction between releases. $z(k)$ is the dissatisfaction discount factor related to release k. As dissatisfaction of non-delivery increases over releases, we assume $z(1) = 0$, $z(K+1) = 1$ and

$$z(k) < z(k+1) \quad (k = 1 \ldots K) \tag{6.16}$$

If plan x would not offer any features at all, total dissatisfaction $TDS(x)$ would be the summation of all feature dissatisfaction values. More general, if a feature is offered in release k, then this creates a dissatisfaction of $z(k) \times DS(n)$. If it is offered in the next release, no dissatisfaction is create at all. Total dissatisfaction TDS(x) created by a plan x is modeled as the summation of all adjusted feature values $DS(n)$, and this function needs to be minimized:

$$TDS(x) = \sum_{k=1\ldots K+1} \sum_{n:x(n)=k} z(k) \times DS(n) \rightarrow Min! \tag{6.17}$$

Resources

Implementation of features consumes effort. We make the simplifying assumption of just looking at the total amount of (estimated) effort needed per feature. The estimated effort for implementation of feature $F(n)\,(n = 1\ldots N)$ is denoted by *effort(n)*. When planning K subsequent releases, the consumed effort per release is not allowed to exceed a given release capacity. For all releases k $(k = 1\ldots K)$, this capacity is denoted by $Cap(k)$. More formally, a *feasible release plan x* needs to satisfy all constraints of the form:

$$\sum_{n:x(n)=k} effort(n) \leq Cap(k) \quad \text{for k = 1}\ldots\text{K} \tag{6.18}$$

We can also provide a more fine-grained model that subdivides the effort per features into more specific types of effort related to analysis, coding, and testing (as an example). The additional constraints resulting from that would not create principal new difficulties for the proposed solution approach but are ignored here to keep the model more simple.

ARP problem formulation

ARP extends the well-established symmetric release planning by considering both satisfaction and dissatisfaction and by treating them as independent criteria. This will be shown to have an impact on the number and structure of plans generated. Having $TS(x)$ and $TDS(x)$ as planning objectives, we are looking for trade-off solutions that maximizes satisfaction and minimized dissatisfaction.

Among all the plans fulfilling resource constraints (known as *feasible plans*), a plan x^* is called a *trade-off solution* for ARP if no other plan exists that is better on one criterion and at the same time not worse in the other. This means that we are looking for feasible plans x^* with the property that there is no other feasible plan x' (also called a *dominating plan*) such that:

$$(i) \quad TS(x') \geq TS(x^*)$$

$$\text{(ii) TDS(x')} \leq TDS(x^*), and$$

$$\text{(iii) (TS(x'),TDS(x')} \neq (TS(x^*), TDS(x^*)) \qquad (6.19)$$

ARP Problem: We consider a given set of features $F(n)$ with feature values $S(n)$ and $DS(n)$ $(n = 1 \ldots N)$. Among all the plans fulfilling resource constraints, the ARP problem is to find trade-off solutions for concurrently maximizing $TS(x)$ and minimizing $TDS(x)$. That means, ARP is the problem of finding trade-off release plans that are balancing satisfaction and dissatisfaction.

Solution Approach SDO

In its nature, the above ARP formulation is an (bi-objective) Integer Linear Programming (ILP) problem. ILP has been proven successful for solving the (symmetric) next release problem [260]. We propose an approach called *Satisfaction-Dissatisfaction Optimizer* or *SDO*. To address the bi-objective nature of the problem, we transform the integer bi-objective optimization problem ARP into a sequence of single objective problems for maximizing function $G(x, \alpha)$ for varying αvalues:

$$G(x, \alpha) = \alpha \times TS(x) + (1 - \alpha) \times (-1) \times TDS(x) \text{ among all feasible plans x and for all } \alpha \text{ from (0,1)} \qquad (6.20)$$

As the function $G(x, \alpha)$ is to be maximized, the portion refering to minize dissatisfaction is added with the factor $\alpha - 1$. It is known from multi-criteria ILP [46] that:

- All solutions received from the parametric single-objective problem Equation (20) with parameter α varying between $(0, 1)$, represent a trade-off solution for ARP.
- Different values of α may generate the same solutions for ARP. The ranges of α returning the same optimal solution are called *stability interval* of that solution.

- Iterative application of the parametric single-objective problem of Equation (20) cannot guarantee to determine the complete set of all non-dominated solutions (Pareto front).

In Section 3, we have outlined three methods for predicting the impact of offering or missing a feature. The results of each of these methods can serve as input for SDO. The methods are different in nature, in the effort needed and in the reliability of the predictions made. The one-point estimate is the simplest among the three methods. It was used for generating the input of the illustrative example in Section 2. With N features and S stakeholders, $2 \times N \times S$ estimates are required. The pair-wise comparison needs $N \times (N-1)$ estimates and some subsequent eigenvalue computations. With its quadratic effort, this method is applicable only for low to a mid-size number of features. Compared to one-point estimates, its benefit is that evaluations are expected to be more reliable, as they are based on comparisons between all pairs of features.

Kano analysis is the most advanced among the three methods. The continuous method requires $10 \times N \times S$ evaluations. Per feature and per stakeholder, 100 points can be allocated for the five possible answers. The computation of the S(n) and DS(n) scores is straight forward as discussed in Section 3.

SDO tool

We are proposing a method of solving a sequence of single-criterion optimization problems, each of them generating a new or an existing trade-off solution. The step-size for varying the parameter can be selected by the concrete problem but is unlikely to create more meaningful results if step size taken is more fine-grained than 0.01. For the implementation of SDO, we apply the commercial optimizer `Gurobi` [82] `version 6.4` and its interface `MATLAB` to manage data. We call this the *SDO tool*.

`Gurobi` is a set of optimization libraries for linear programming, quadratic programming, and mixed-integer programming. We used the free academic license available for `Gurobi` to develop our application. For problems up to several hundreds of features, the solution gained from running

single-objective optimization are proven to be 100% optimal. The optimality status is displayed by the Gurobi optimizer.

While we rely on Gurobi, the results of our approach in principle do not depend on the underlying optimizer. Depending on the solver, there might be differences in the scalability of the approach, i.e., the size of problems we are able to solve. We measured the time required to determine the optimized plans when using Gurobi on an `Intel(R) core i7-2620M CPU@ 2.70 GHZ` computer. On average, SDO computed a Pareto solution (one iteration of the approach) in 54 ms which seems to be sufficiently fast from an application perspective.

The SDO implementation (code snippets in MATLAB) is available as a complementary material for this paper, and the environment setup and overview of the implementation are described in Appendix II.

6.5 RQ3: Evaluation

To prove the applicability and usefulness of SDO and the quality of its results, we performed a case study with a company developing mobile apps. This case study is descriptive to portray the ARP process. For feature value prediction, we are using the Kano model.

Analysis of the development and usage of mobile apps is an emerging area of research. Properly addressing customer concerns is critical in the highly competitive and dynamic environment. In the context of mobile app stores, we study the ARP process to balance stakeholders' satisfaction and dissatisfaction for proposing (new) mobile apps. For that purpose, we looked at Over-The-Top (OTT) TV services [107] apps offered in the Android app store market and plan for the next release (K = 1) of that product.

Following the guidelines of Runeson and Hoest [231] for conducting case studies, we describe the design, the data collection process, the analysis of data, and the reporting of results for our case study. In RQ3, we evaluate different aspects of improvement from using SDO in comparison to plans generated from random and heuristic search and plans generated from human experts.

Data collection and preparation

Our data collection and preparation was organized towards elicitation of features, effort estimation, and feature evaluation by stakeholders. We describe the steps and their results in more detail:

Stakeholders: The case study company reached out to a large group of direct stakeholders (potential users of the app). Since we did not have access to their data, we invited 24 software engineering graduate students to serve as stakeholders. Even though students were not a direct customer of the company, they were familiar with the domain (OTT services) and were considered to be representative for the purpose of this case study.

Weight of stakeholders: The survey participants provided a self-evaluation in terms of their familiarity with OTT services and mobile applications. At the beginning of the survey, stakeholders stated their domain expertise on a Likert scale ranging from one to nine. We used this value as the weight of stakeholders for the planning process.

Features: The pool of candidate app features was extracted from the description of 261 apps, all of them providing media content over the Internet without the involvement of an operator in the control or distribution of the content (OTT service). A commercial text analysis tool was used to retrieve 42 candidate features. Domain experts evaluated the meaningfulness of extracted features and eliminated the phrases which did not point into any OTT feature. Feature extraction itself was managed by the case study company and resulted in 36 features further investigated.

Feature values: To predict the impact of offering versus missing features, we applied the Kano analysis outlined in Section 3. We performed a survey with a continuous Kano design and asked the two types of questions (functional and dysfunctional) that were introduced in Section 3.3. For each feature, each of the stakeholders expressed the percentages that the feature matches one of the five possible answers per question.

Effort: The effort for developing each feature was estimated by domain experts within the company. A product manager and two senior developers estimated the effort needed (in person hours) to develop each feature. They applied a triangular (three point) effort estimation to estimate

the optimistic, pessimistic and most-likely effort amount needed to deliver a feature. The three estimates were combined using a weighted average with weighting factors of one, one and four, respectively. This weighting technique is originated from the Program Evaluation and Review Technique (PERT) [144].

Capacity: To show the behavior of SDO under different scenarios of tight, medium, and more relaxed resource availability, we ran three concurrent case study scenarios with release capacities Cap(1) of 112.7 (lower bound), 367.4 (most probable), and 625.5 (upper bound) person hours, respectively.

Lists of features, effort estimation, continuous Kano survey, and results are available online[5].

Feature satisfaction and dissatisfaction analysis

Using the data introduced above, we calculated stakeholder scores per features using Equations (5) to (8) and aggregated the scores per feature across stakeholders as shown in Equation (9).

Analysis of solutions generated from SDO

For the purpose of comparison and evaluation, we applied SDO for planning 36 features considering the described constraints and resources. We first look into the portfolio of features generated by SDO. For the three scenarios (different levels of capacity) we study the structure and diversity of them.

Having all the data needed, we now analyze the results from applying SDO to solve the case study problem. This is done for the three varying capacity levels. A set of alternative solutions was generated by SDO for each level. In total, over the three capacity levels, 14 trade-off solutions (plans) were generated. All the release plans along with their objective function values and the effort consumed, are presented in Table 6.4. For simplicity, we use *Fn* instead of *F(n)* to refer to features. With a step size of 0.001, for each α value in the interval (0,1), SDO generated one trade-off solution. Each solution represents one possible way to balance between satisfaction and

[5] `http://www.ucalgary.ca/mnayebi/tools-and-data-sets`

Table 6.4: Results of asymmetric planning: Feature plans, satisfaction and dissatisfaction levels for each plan, stability of results and effort needed to implement optimized plans for three levels of capacity.

Capacity = 112.7

Plan	Satisfaction	Dissatisfaction	Effort
F1 F3 F5 F14 F21 F22 F25	4.713	8.974	111.4
F1 F3 F5 F21 F22 F25 F30 F32	5.031	9.030	111.5
F1 F3 F5 F9 F18 F22 F25 F30 F32	5.236	9.112	112.4

Capacity = 367.4

Plan	Plan	Satisfaction	Dissatisfaction	Effort
Plan 1	F1 F2 F3 F4 F5 F9 F11 F14 F15 F17 F18 F19 F21 F22 F23 F25 F30 F32 F36	10.024	4.301	360.2
Plan 2	F1 F2 F3 F4 F5 F9 F11 F14 F15 F17 F18 F19 F21 F22 F24 F25 F30 F32 F36	10.184	4.369	366.9
Plan 3	F1 F2 F3 F4 F5 F9 F10 F11 F14 F15 F17 F18 F19 F21 F22 F24 F25 F30 F32	10.394	4.469	362
Plan 4	F1 F2 F3 F5 F6 F9 F10 F11 F14 F15 F17 F18 F19 F21 F22 F23 F24 F25 F30 F32	10.597	4.710	366.3

Capacity = 625.5

Plan	Satisfaction	Dissatisfaction	Effort
F1 F2 F3 F4 F5 F7 F8 F9 F10 F11 F13 F14 F15 F17 F18 F19 F21 F22 F23 F25 F26 F28 F30 F32 F35 F36	13.298	1.413	616
F1 F2 F3 F4 F5 F8 F9 F10 F11 F13 F14 F15 F17 F18 F19 F21 F22 F23 F24 F25 F26 F28 F30 F32 F35 F36	13.485	1.439	618.2
F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F14 F15 F17 F18 F19 F21 F22 F23 F24 F25 F26 F28 F30 F31 F32 F36	13.784	1.508	622.5
F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F13 F14 F15 F17 F18 F19 F21 F22 F23 F24 F25 F26 F30 F32 F35 F36	13.786	1.509	625.3
F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F13 F14 F15 F17 F18 F19 F21 F22 F23 F24 F25 F28 F30 F32 F35 F36	13.825	1.629	623.2
F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F13 F14 F15 F17 F18 F19 F21 F22 F23 F24 F25 F28 F30 F31 F32 F35	13.866	1.833	624.6
F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F13 F14 F15 F17 F18 F19 F20 F21 F22 F23 F24 F25 F28 F30 F31 F32	13.876	1.896	623.6

dissatisfaction of stakeholders. Based on the equivalence between parametric and multi-objective optimization [46], the solutions received from automatically running a sequence of single-criterion problems Equation (20) represent Pareto solutions for ARP.

Comparison with random and heuristic search

As formulated in Objective 2, we are interested in the comparison between the quality of SDO solutions in comparison to those obtained from (i) random search and (ii) heuristic search.

Comparison with random search

Using random search as a baseline for comparison has been introduced by Acuri and Briand [12]. It has been applied in various contexts since then, see for example [279]. We used it here for the same purpose as a straw man. Random search is selecting a feature randomly as long as the effort for implementing that feature is less than the available capacity. The procedure is summarized as Algorithm 3. The results showed that SDO generated solutions strongly dominate all the 1,000 solutions generated by random searching. The best random solution is still outperformed by 37% by one of the SDO solutions (having 12% better satisfaction and 25% better dissatisfaction performance).

Comparison with heuristic search

As ARP is in the class of NP-complete problems [72], the question of finding light-weight heuristics arises. Greedy heuristics are widely used to solve combinatorial problems and are used in practice to easily create release plans [228]. The general greedy principle is to select the best local features at each iteration, where the definition of "locally best" varies between the heuristics [48]. With no backtracking, greedy solutions are fast and often "good enough". The quality of the solutions often depend on the problem structure and the instance of the problem. It can be quite far from the optimum in specific instances.

We applied the search for greedy solutions to ARP. In total, we defined and compared eight greedy heuristic approaches. Therein, *locally best* selection for the next feature is defined related to satisfaction, dissatisfaction or a combination of both. As another variation point, definition of *locally best* was done with and without consideration of effort consumed per feature.

Algorithm 3: Random selection of features

```
Set Capacity, Remaining_capacity;
Do
    Select a random feature between F1 to FN;
    Remaining_capacity = Capacity - Effort(n)
While (No Feature with effort <= Remaining_capacity)
```

Table 6.5: Comparison of results between SDO generated plans and the results of eight heuristics for three different effort levels.

ID	Algorithm	FACTOR	Marker	Dominated by SDO plan	Identical to SDO plan	New plan
H1	Algorithm 3	Satisfaction	$*$	0	3	0
H2	Algorithm 3	Dissatisfaction	■	2	1	0
H3	Algorithm 3	Satisfaction/Effort	♦	2	1	0
H4	Algorithm 3	Dissatisfaction /Effort	-	1	2	0
H5	Algorithm 3	Satisfaction + Dissatisfaction	▲	3	0	0
H6	Algorithm 3	(Satisfaction + Dissatisfaction)/Effort	○	2	0	1
H7	Algorithm 4	Satisfaction, Dissatisfaction	△	3	0	0
H8	Algorithm 4	Satisfaction/Effort, Dissatisfaction/Effort	×	3	0	0

All heuristics are instantiations of either Algorithm 4 or Algorithm 5. For Algorithm 4, we selected just one factor for ranking the features (referred to as FACTOR in the algorithm).

The factors used in the greedy heuristics are as follows:

- Satisfaction,
- Dissatisfaction
- Satisfaction/effort
- Dissatisfaction/effort
- Satisfaction + dissatisfaction
- (Satisfaction + dissatisfaction)/effort.

We used Algorithm 5 to define greedy solutions based on the application of two concurrent ranking criteria, applied alternatively between iterations. This is called *Two factors greedy heuristic* and it uses the following pairs of factors:

Algorithm 4: One factor greedy heuristic

```
Rank features in decreasing order of FACTOR;
for n = 1 to N do
    Select f(n) where f(n) is ranked higher than other features and was not selected before; if (Effort(n) <
    Capacity - Total_Effort) then
    Total_Effort + = Effort (f(n));
  end if
  While(Total_Effort <= Capacity)
end for
```

- [Satisfaction, dissatisfaction], and
- [Satisfaction/effort, dissatisfaction/effort].

For example, using Algorithm 4 with [Satisfaction, dissatisfaction] factors, the first feature picked is the best in terms of satisfaction. The second feature selected is the one creating the highest dissatisfaction if it is not selected. Now, the criteria are changed again, and the second best feature in terms of satisfaction is picked. The algorithm terminates when no feature can be added because of the capacity constraint.

From running SDO for three levels of capacity, we received 14 Pareto solutions. By running eight different heuristics, we got in total 24 solutions. 66.7% of the heuristic plans were dominated with at least one of the SDO solutions. Also, one new Pareto solution was retrieved by heuristic H6 which was not found by SDO (the rest were identical to existing SDO solutions). We also noticed that heuristic H2, which is aimed to optimize towards the dissatisfaction criterion, does not even achieve a solution which is best related to that criterion. This demonstrates that heuristics are fast and conceptually easy, but often not good enough. The comprehensiveness of solutions generated from SDO in conjunction with their guaranteed quality is considered a strong argument in favor of the proposed SDO approach.

Comparison between solutions generated from SDO against the eight different heuristics showed that:

- Using stakeholders' satisfaction in Algorithm 2 (H1 in Table 6.5) performed best compared to other heuristics, as it found three solutions identical to the Pareto solutions found by SDO.

Algorithm 5: Two factor greedy heuristic

```
Arr A[] = Rank features in FACTOR decreasing order; Arr B[] = Rank features in FACTOR' decreasing order; Arr
C[] = Merge array A[] with array B[]
for n = 1 to N do
    Select f(n) where f(n) is ranked higher than other features in C[] and was not selected before; if (Effort(n) <
    Capacity - Total_Effort) then
    Total_Effort + = Effort (f(n));
  end if(Total_Effort < Capacity)
end for
```

- Using dissatisfaction/effort (H4 in Table 6.5) also found two solutions which were identical to one of the SDO solutions. One of the solutions found by this heuristic was dominated by SDO plans.

- Among all the heuristics, using Algorithm 3 (H7 and H8 in Table 6.5) had the worst performance. Both heuristics found solutions to ARP which were all dominated by SDO.

The comparison of solutions received from SDO and from the different heuristics are presented in Table 6.5.

Expert evaluation of plans

Besides comparing the results of SDO with other algorithms, we have performed an external evaluation from running two rounds of surveys with actual stakeholders. Industrial evaluation of release planning models is known to be inherently difficult. The results of a systematic literature review by Svahnberg et al. [251] showed a deficit in the industrial evaluation of release planning methods. This was confirmed by a recent survey of software release planning models conducted by Ameller et al. [10]. The authors found that since 2009, only four release planning papers were published that include at least one co-author from industry.

We conducted a survey to understand stakeholders preference among the various plans generated. The survey included 20 stakeholders. Six of the stakeholders were from the case study company, providing additional arguments for justification of their choices. The other 14 participants were from the pool of graduate students having conducted the original Kano feature evaluation before. We invited all the 24 graduate students who participated in the feature evaluation. 22 of them also participated in the second round of evaluation. However, only 14 graduate students answered the questions completely. Our survey was two staged:

Phase 1: We asked all the 20 participants to rank the four SDO plans generated for the capacity level of $cap(1) = 367.4$. They were asked to do this from both satisfaction and dissatisfaction perspective.

Phase 2: We asked the six managers to plan based on the 36 features. They manually planned based on perceived value of features considering $cap(1) = 367.4$.

Phase 1: Ranking of plans

In Phase 1 of the survey we asked stakeholders to rank the four SDO generated plans. The results of ranking are reported in Table 6.6 (satisfaction perspective) and Table 6.7 (dissatisfaction). Using Fleiss Kappa test [246] for measuring inter-rater agreement showed a slight to poor agreement between the 20 participants. Considering the ranks assigned from satisfaction perspective, the Kappa value is 0.0409 ($p - value = 0.05$). Considering dissatisfaction, the Kappa value is 0.0649 ($p - value = 0.002$). Both values indicate a low agreement. This means, that there is no obvious preference among the plans, or stated alternatively, each of the four plans is considered important by at least some of the stakeholders. As a conclusion, this confirms the hypothesis that offering diversified plans is a value for stakeholders to decide about their final plan.

By comparing plans per stakeholder, among stakeholders and between criteria, we found that:

- **One plan does not fit all**: For both planning objectives, there is substantial variation between stakeholders in terms of what they consider their preferred solution,
- **One symmetric criterion is not enough**: Six of the 20 stakeholders have a varying top preference when comparing plans selected from satisfaction and dissatisfaction perspective.

Phase 2: Comparison with manual plans

We asked six project managers to propose their manual plan. They were offered all the 36 candidate features, their effort estimates and the total effort available. The resulting plans are shown in Table 6.8.

From analyzing these manual plans, we observed that the satisfaction and dissatisfaction of each plan are far below the quality of the SDO generated plans (they generate lower satisfaction and higher dissatisfaction). We compared the six manual solutions with the four optimized plans.

Table 6.6: Stakeholder ranking of plans for the four plans of Table 5 calculated for Capacity = 367.4. All plans were evaluated from satisfaction perspective.

Stakeholder	Plan 1	Plan 2	Plan 3	Plan 4
Student 1	4	1	2	3
Student 2	1	2	3	4
Student 3	4	3	1	2
Student 4	1	2	3	4
Student 5	4	3	2	1
Student 6	3	4	2	1
Student 7	1	3	2	4
Student 8	1	2	4	3
Student 9	1	2	4	3
Student 10	3	2	4	1
Student 11	4	2	3	1
Student 12	1	2	3	4
Student 13	2	4	3	1
Student 14	1	2	3	4
Manager 1	3	2	1	4
Manager 2	2	3	4	1
Manager 3	1	3	4	2
Manager 4	2	4	1	3
Manager 5	4	2	3	1
Manager 6	2	4	3	1

On average, SDO solutions perform 59.2% better in terms of satisfaction and 83.4% in terms of (avoided) dissatisfaction. Looking into the case of capacity of 367.4, there are 14 core

features suggested by all optimized plans. These features are *F1, F2, F3, F9, F11, F14, F15, F17, F18, F19, F21, F22, F25, F30*. The manual plans on average offered only 8 of the core features (ranking between 5 and 10). In other words, on average, each of the manual plans are missing six of the core features that have found essential for maximizing satisfaction and minimizing dissatisfaction. This is considered critical as the absence of core features is expected to have a negative impact on the success of the product release.

Threats to validity

The conducted case study was primarily explanatory, with the main goal to provide first evidence confirming the hypotheses that (i) trade-off release plans generated by SDO perform better than

Table 6.7: Stakeholder ranking of plans for the four plans of Table 5 calculated for Capacity = 367.4. All plans were evaluated from dissatisfaction perspective.

Stakeholder	Plan 1	Plan 2	Plan 3	Plan 4
Student 1	4	1	2	3
Student 2	1	2	3	4
Student 3	2	3	1	4
Student 4	1	2	3	4
Student 5	4	3	2	1
Student 6	3	4	2	1
Student 7	1	2	3	4
Student 8	1	2	4	3
Student 9	1	1	1	1
Student 10	1	3	1	4
Student 11	1	2	4	3
Student 12	1	2	3	4
Student 13	3	1	2	4
Student 14	1	2	3	4
Manager 1	2	3	1	4
Manager 2	3	2	4	1
Manager 3	4	3	2	1
Manager 4	1	2	3	4
Manager 5	4	3	2	1
Manager 6	3	1	2	4

Table 6.8: Manual plans generated by company stakeholders M1 to M6, calculated for Capacity = 367.4.

Manager	Feature set	Satisfaction	Dissatisfaction
M1	F2,F4,F5,F8,F9,F10,F13,F15,F17,F19,F29	6.50	8.39
M2	F2,F3,F4,F9,F10,F11,F15,F19,F21,F30,F31,F34,F35	6.16	8.25
M3	F1,F3,F4,F5,F9,F10,F11,F12,F16,F17,F18,F19,F21,F22,F24,F27	7.22	7.78
M4	F1,F3,F4,F5,F9,F10,F11,F15,F19,F21,F28,F31,F34,F35	6.61	8.03
M5	F3,F4,F5,F6,F8,F9,F10,F13,F14,F15,F16,F20,F28	6.53	8.01
M6	F1,F2,F4,F5,F8,F9,F10,F12,F13,F15,F20,F28	5.80	8.72

individual plans generated by random and heuristic search and (ii) are considered valuable by stakeholders.

How trustworthy are the results gained and what threats still exist? Following [231], we discuss four types of threats. To further increase credibility, we have provided access to all case study data. Related to reliability of results, most of them are based on objective measures which have been used as well in other studies in the context of software engineering multi-criteria decision-making (compare e.g., [260], [279]).

Construct validity

How valuable is comparison with random and heuristic search? Random search has been used for various studies in software engineering for the same purpose since it was suggested by Acuri and Briand [12]. Greedy heuristics, in general, are widely used as a light-weight technique in decision-making, and the same is true in release planning [228].

Another construct validity aspect is if stakeholders got the right understanding of the survey questions and in generating manual plans. To mitigate this risks, we gave a 10 minutes instruction and description for filling a survey over the phone. All the candidate features and their effort were made available to them. As they were familiar with the content of the features, their decisions are considered to be well justified.

Internal validity

There are no causal relationship statements made on the planning results studied in the case study. The different plans compared were generated by the various techniques without further impacting (confounding) factor. We used a real-world project for case study evaluation. Also, the survey results were not impacted by any other obvious factor.

External validity

Generalization of findings was not the intention of the case study. Instead, the focus was on a proof-of-concept evaluation. Survey Phase 1 with 20 stakeholders indicates that there are subjective differences among stakeholder perspectives and preferences, which confirms the value of presenting alternative solutions. Ignoring dissatisfaction would imply the risk of stakeholder dissatisfaction. Even though the number of participants in Phase 2 of the survey is small, most stakeholders believe the method is useful and scalable.

The case study had common characteristics in terms of size (36 features) and involved stakeholder (= 20). Even though the computational complexity of the problem is NP-hard, `Gurobi` is

Table 6.9: Manager's level of agreement to Phase 2 survey questions

Question	M1	M2	M3	M4	M5	M6
Usefulness of Kano	5	5	5	5	4	4
Mitigat risk of offering wrong product	5	4	5	5	5	4
Efficiency of Kano	4	3	5	4	4	5
Relevance of ARP	5	4	5	5	5	4
Value of diversity	5	5	5	5	5	5
Optimized solution vs. ad hoc	5	5	5	5	4	5
Optimized solution vs. heuristic	4	5	4	5	5	5
Scalability of SDO	3	5	4	3	5	5

able to solve solving benchmark integer linear programming problems [6] with several thousands of variables with proven optimality. Because of that, we expect that SDO is able to solve most of the practical problems of product release planning.

Reliability validity

Students served as 14 (out of 20 in total) stakeholders evaluating the value and attractiveness of features. Most of them were familiar with OTT services such as Netflix. In-line with the results of [23], we consider them as suitable stakeholders for case study analysis. Furthermore, stakeholders' importance was decided based on their (self-evaluated) familiarity with the context. In other words, if a student did not feel familiar with OTT services, her impact on the actual results was low.

6.6 Discussion

From performing the case study, we can make some conclusions about the proposed SDO approach. We approached the six project managers to perform a more in-depth analysis of SDO plans. We asked these stakeholders to answer eight questions related to different aspects of the usefulness of plans and different aspects of SDO. The questions were defined on a five-point scale with 1 to 5 being *very low* to *very strong* agreement. The categories of questions and the manager's responses are summarized in Table 6.9.

[6] `http://plato.asu.edu/ftp/milpc_log1/benchmark.gurobi.out`

Relevance of ARP

Providing the best set of features for upcoming releases is decisive for product success. There is a risk of providing features that are not relevant as well as missing requested features [228]. The asymmetry between satisfaction and dissatisfaction was confirmed by literature [150]. Looking into the survey results (Tables 7 and 8), it became clear that different plans are preferred between stakeholders. Also, when comparing the two plans preferred per stakeholder (one for satisfaction, one for dissatisfaction), again, there are differences in about one-third of the cases.

We asked the six managers in our case study about the asymmetry between satisfaction and dissatisfaction.

To what extent do you agree that satisfaction and dissatisfaction both are relevant for planning product releases and need to be considered in conjunction? *All participants either agreed or strongly agreed that both criteria are relevant and need to be considered in conjunction, which is the key idea of ARP (See Tables 7 and 8).*

Analyzing the different features from both satisfaction and dissatisfaction perspectives and including all stakeholders in this process helps to not miss essential features.

To what extend do you agree that a systematic planning method which includes stakeholder opinions related to both satisfaction and dissatisfaction reduces the risks of developing the "wrong" product? *Four developers were strongly agreed, and two agreed that joint consideration of satisfaction and dissatisfaction helps mitigate the risk of developing the wrong product.*

Usefulness of Kano for measuring feature satisfaction and dissatisfaction

We introduced three methods for measuring satisfaction and dissatisfaction, and the ARP model works independently of which of them is used. We applied the continuous Kano model for our case study as it has been widely discussed in literature [264]. Offering the results of the case study, we asked the managers about the usefulness of Kano model and its efficiency.

To what extent do you think that Kano evaluation was used to understand users better?

Four managers strongly agreed, and two managers agreed that the Kano was useful for their company.

To what extend do you think the additional effort (from answering ten questions per feature based on Kano) is worthwhile?

Two managers strongly agreed, three agreed, and one manager was neutral about the efficiency of the Kano model.

While the literature confirms and managers agree with the value of the Kano model, the usefulness and efficiency of it are rather context specific and alternative methods should be evaluated.

Value of diversity

The diversity principle formulated in [228] says that *A single solution of a cognitive complex problem is less likely to reflect the real-world problem solving needs when compared to a portfolio of qualified solutions that are structurally diversified*. In its nature, solutions generated from SDO are trade-off balancing satisfaction against dissatisfaction. The plans differ in their structure.

We illustrate this argument by computing the symmetric differences between all pairs of SDO generated plans for the case of $Cap(1) = 367.4$. The symmetric difference between two sets is the set of elements that are in either of the sets, but not in the intersection. We can see that the minimum number of elements in the symmetric difference is two, and the maximum number is five. The results are presented as Table 6.10. With a set of 14 features being offered for all plans, there is variation in the remaining features.

ARP includes the provision of alternative solutions, balancing between satisfaction and dissat-

Table 6.10: Symmetric differences between the four SDO generated plans of Table 5 calculated for Capacity = 367.4.

Plan	**Plan 2**	**Plan 3**	**Plan 4**
Plan 1	F23, F24	F10, F23, F24, F36	F4, F6, F10, F24, F36
Plan 2	*	F10, F36	F4, F6, F10, F23, F36
Plan 3	*	*	F4, F6, F23, F36

isfaction. As can be seen from the inter-rater agreement in terms of ranking plans, there is a slight to poor agreement between the 20 participants of Phase 1 of the survey. The survey results show different preference of plans between stakeholders and between the two planning objectives.

This confirms the value of offering a set of alternative solutions instead of prescribing just one plan. The deeper reason for this is that any problem description can never be complete, and the personal preference naturally varies between stakeholders because of the different degree of tacit knowledge they have about the problem.

How valuable do you consider having several alternative plans as the result of SDO (Pareto solutions) as opposed to offering just one?
All the six managers strongly agreed that diversity of solutions is desirable.

Value of optimality

We compared the results of SDO with (i) random search, (ii) heuristic search, and (iii) manual plans generated by stakeholders of the case study company. All these comparisons showed that the solutions generated by SDO are better. The value of optimized plans in comparison with the heuristic of ad hoc plans was confirmed by all stakeholders, as they almost all strongly agreed on that.

How valuable do you consider optimized solutions in comparison to solutions generated ad hoc, based on gut feeling?
Five managers were strongly agreed, and one agreed that optimized solutions are better than ad hoc solutions based on gut feelings.
How valuable do you consider optimized solutions in comparison to solutions generated heuristically?
Four managers strongly agreed, and two agreed that optimized solutions are better than solutions generated heuristically.

Scalability

Generation of optimized trade-off plans for the advanced model of ARP does not come for free: The method requires feature evaluation and information gathering. In total, the ARP formulation increases the conceptual complexity of the task of the decision-maker. Depending on their needs, they can select from different modeling options for getting the information needed. We have provided alternative techniques to elicit the relative value of features related to satisfaction respectively dissatisfaction of a release.

The computation time for the SDO is comparable with the computation time of the random search and the heuristic solutions. Using an `Intel(R) core i7-2620M CPU@ 2.70 GHZ` computer, using a random search (Algorithm 3) for the problem in our case study took 48ms, while heuristics (Algorithm 4) took 55ms on average to find a solution. SDO finds a solution in 54ms per iteration. We also asked stakeholder on their perspectives.

To what extend the size and complexity of the product impact the performance of the SDO approach?
Three managers strongly agreed, one agreed and two were neutral about the scalability of SDO.

Limitations

The formulation of release planning has aspects of wickedness [192]. That means, as for other design problems, there is no ultimate answer on what is a right model. Robertson and Robertson [224] showed that stakeholders do value gains and losses unequally. A few models were provided relying on the asymmetry between satisfaction and dissatisfaction such as Kano analysis and prospect theory [257].

When SDO should not be applied? The whole investment into advanced planning methods like the proposed method SDO does not make sense when (i) the planning problem is not complex in terms of number of features, number of stakeholders involved, and number of competitors on the market, (ii) strategic perspective of planning (as opposed to short term considerations and/or

continuous delivery) is not valuable as there is too much change in market conditions and data used for performing the planning process, (iii) the organization is not mature enough to provide qualified input for the planning process, and (iv) the differentiation between feature and product satisfaction and dissatisfaction is not important.

6.7 Related Work

In this section we discuss the asymmetry between customers' satisfaction and dissatisfaction referring to other disciplines and evaluate how they were used in software engineering methods. We also give a brief overview on symmetric release planning methods and the use of multi-objective optimization in software engineering.

(Symmetric) Release planning and feature prioritization

The Kano model and analysis

Several models are proposed for understanding customers' satisfaction and dissatisfaction. Satisfaction factors are classified differently mainly followed by the Kano model. These models in general model customers reaction to the product changes in a nonlinear and asymmetric model. Kano et al. [110] suggested a questionnaire which includes two questions related to each feature. The functional question evaluates the customer's reaction to the presence of a particular feature. The dysfunctional question assesses the customer's reaction in the absence of the same feature. Cadotte & Turgeon [36] suggested a model with four different feature considering the potential of features for raising positive or negative feelings. They defined qualitative levels for identifying typical features for each category. Anderson and Mittal [215], Backhause and Bauer [215], Bitner et al. [215], and Brandt [215] and several others proposed methods for modeling the asymmetry between customers satisfaction and dissatisfaction.

The very first Kano model was introduced in the early 1980's [110]. Mikulic [163] classifies the Kano technique as highly reliable for classification of quality attributes at the design stage. The

use of this model for requirement prioritization in non-software products was elaborated [110,237]. The results showed that the requirements have an asymmetric effect on stakeholder satisfaction and dissatisfaction.

The lack of quantitative assessment in the traditional Kano model limits the value of the decision support provided [275]. As a result of that, several Kano extensions have been proposed. Violante et al. [264] found ten extensions of the original Kano method. Berger's quantitative model [26] (As we used in this paper) is the most simplified and earliest proposed quantitative model.

Another critique of the traditional Kano model is its limitation in precisely defining fulfillment and non-fulfillment of stakeholder satisfaction. The traditional Kano model categorizes each feature into exactly one Kano category. This does not sufficiently reflect the complex sentiments of an individual [128, 163]. Lee et al. [128] proposed the fuzzy Kano questionnaire which extends the traditional approach by providing a questionnaire to help stakeholders explain their feelings. The fuzzy Kano model uses a membership function and assigns numeric degrees to stakeholders' feelings. While fuzzy Kano and continuous Kano create the same type of information, the difference in the continuous Kano is that information is also used for subsequent release planning purposes.

Software products and features have numerous value constructs [116]. Khurum et al. [116] provided a comprehensive value map for software. Acquiring values for comprehensive feature prioritization needs careful analysis [116]. For these reasons, feature prioritization methods and, even further, release planning approaches have assumed that priorities are coming from stakeholders.

A spectrum of the prioritization techniques is discussed in literature [2, 25, 67, 220]. Prioritization techniques have been widely discussed in software engineering within release planning methods or independently. An overview of these approaches was given by Berander and Andrews [25]. Achimugu et al. [2] discovered 49 distinct prioritization techniques and reported AHP [234] as the technique with the highest citation and utilization. Feature prioritization is possible based on different criteria. On the other side, the study by Riegel and Doerr [220] classified stakeholder

satisfaction and stakeholder satisfaction among the top ten prioritization criteria [220].

The missing phenomenon of interest for our study within currently available literature reviews [2, 25, 67, 220] is the extent of conjoint consideration of satisfaction and dissatisfaction for feature prioritization and release planning. To gather this information, we took the 73 papers selected by Achimugu et al. [2] for their systematic literature study and the 83 papers analyzed by Riegel and Doerr [220]. These two studies had 15 papers in common. As a result, we analyzed 141 papers to find if and how satisfaction and dissatisfaction were considered jointly in feature prioritization and release planning methods. The results of this analysis are reported in Table 6.11.

We found that 33 studies considered satisfaction or dissatisfaction as their planning criteria. However, only four of these studies jointly considered satisfaction and dissatisfaction. Among these four studies, Fehlmann [65] and Lehtola and Kauppinen [129] refer to the Kano model for feature prioritization, but without entering into release decision making. Robertson and Robertson [224] showed that stakeholder value's gains and losses are unequal and stakeholder satisfaction of receiving a feature is not equal to the dissatisfaction of not receiving that feature. For release planning, EVOLVE was introduced by Ruhe et al. [79, 235] as a method being flexible in the number and selection of planning criteria. In software release planning, it has mostly been assumed that receiving a feature provides specific stakeholder satisfaction which is equal to stakeholder dissatisfaction from not receiving that feature. In other words, release planning methods assumed that satisfaction and dissatisfaction are symmetric. However, the Kano model [110] demonstrated that stakeholders' satisfaction and dissatisfaction occurs conjointly. Our proposed ARP formulation points to the essence of conjoint consideration of satisfaction and dissatisfaction in release planning because using one of these values without the other one will result in biased and incomplete results.

Multi-objective optimization in software engineering

Consideration of multiple objectives is a growing trend in software engineering. One criterion (such as cost, revenue, time-to-market) only provides a partial direction for determining best

strategies. The notion of Pareto-optimality guides searches towards solutions that can only be improved towards one criterion by compromising against another one. Aligned with the trend towards more analytics and subsequent quantitative investigations, multi-objective optimization has been recently applied to the next release problem [260], to software architectures [214], refactoring [147], requirements selection [52], and model merging [146]. A more general release problem with looking at three releases ahead and having revenue and cost as optimization criteria and objectives were studied in [279]. With the exception of [260], all these approaches were based on evolutionary and search-based techniques. In [279], an empirical study of meta- and hyper-heuristic search was performed for a series of ten real-world data sets. The authors found that hyper heuristics [35] were most successful.

For multi-objective release planning, it was shown in [260] that ILP is applicable and competitive with search-based algorithms in terms of computational effort. A similar result was obtained already earlier for the case of single criterion release planning by van den Akker et al. [259]. ILP results are even better with regards to the guaranteed optimality of the solutions obtained. Based on that finding, we have continued that route and have applied ILP on a more general and broader class of bi-objective release planning problems which are based on asymmetric performance of features.

Integer linear programming was used by Veerapen et al. [260] to solve the single and bi-objective Next Release Problem. While there has been a dominance of search-based techniques in the past (starting with the genetic algorithm of Greer and Ruhe [79]), the authors have shown that integer linear programming-based out-performs the NSGA-II genetic approach on large bi-objective instances.

6.8 Summary and Conclusions

Release planning is a cognitively and computationally complex problem. From improving the information used in the planning process, we have a better chance to address the right problem.

We proposed a bi-criteria problem formulation to determine a set of trade-off release planning solutions. From applying the linear integer programming based method SDO, a set of optimized trade-off solutions can be determined. The applicability of the approach was demonstrated by a case study from the app store market. Within this case study, a variety of comparisons between different approaches of solving ARP were discussed.

We consider the proposed research as being a part of a more comprehensive research agenda. We foresee a wide range of future research challenges that need to be addressed. Our proposed ARP model is based on the (continuous) Kano model for feature needs elicitation. Another model which could alternatively be used for asymmetric planning is the prospect theory as proposed by Tversky and Kahneman [257]. Prospect theory is emphasizing the difference between valuing gain and loss and how stakeholders would value decisions that are made based on gain rather than on loss.

A more comprehensive empirical evaluation is required to validate the usefulness of the method over the ”traditional” (symmetric) approaches. In particular, the additional effort needed to solicit more comprehensive stakeholder information and to perform the satisfaction-dissatisfaction optimization needs to be related to the added value gained from having a set of optimized release plan alternatives.

Acknowledgments

We thank all the anonymous reviewers and the Associate editor for their valuable comments and suggestions. We wish to thank Mohsen Ansari, Navid Pourmomtaz, and Maryam Soleimani for their help on the implementation of this research. We are grateful to Des Greer for helpful discussions on a former version of the paper. This research was partially supported by the Natural Sciences and Engineering Research Council of Canada, NSERC Discovery Grant 250343-12.

Table 6.11: Consideration of satisfaction and dissatisfaction as prioritization criteria among the 137 studies systematically reviewed by [2] and [116]. Papers not referenced in the table neither consider satisfaction nor dissatisfaction.

Study	Satisfaction as a criteria	Dissatisfaction as a criteria	Description
Voola and Babu [266]	Considered	-	-
Liu et al. [136]	Considered	-	-
Raharjo et al. [212]	Considered	-	-
Kukreja [126]	Considered	-	-
Voola and Babu [265]	Considered	-	-
Forouzani et al. [68]	Considered	-	-
Babar et al. [17]	Considered	-	-
Otero et al. [196]	Considered	-	-
Gaur et al. [73]	Considered	-	-
Carod and Cechich [39] [38]	Considered	-	-
Daneva and Herrmann [51]	-	Considered	-
Liu et al. [135]	Considered	-	-
Karlsson [112]	Considered	-	-
Fehlmann [65]	Considered	Considered	Uses Kano but doesn't refer to dissatisfaction. Considers satisfaction and technical excellence.
Pitangueira et al. [204]	Considered	-	-
Lehtola and Kauppinen [129]	Considered	Considered	Points to Kano model as a basic model for prioritization.
Berander and Andrews [25]	Considered	-	-
Samer et al. [165]	Considered	-	-
Botta and Bahill [32]	Considered	-	-
Hu et al. [99]	Considered	-	-
Barney et al. [18]	Considered	-	-
Berander [24]	Considered	-	-
Robertson and Robertson [224]	Considered	Considered	Satisfaction of receiving and dissatisfaction of not receiving a feature is not equal.
Ziemer et al. [280]	Considered	-	-
Akker et al. [259]	Considered	-	-
Regnell et al. [217]	Considered	-	-
Benestad et al. [21]	Considered	-	-
Racheva [211]	Considered	-	-
Barney et al. [18]	Considered	-	-
Aurum and Wohlin [14]	Considered	-	-
Logue and Kevin [138]	Considered	-	-
Ruhe et al. [79, 228, 235]	Considered	Considered	Introduced EVOLVE family optimizing a linear combination of planning criteria.

Part IV

Objective 3: Decision Support for Release Functionality and Marketability

Chapter 7

Which Version Should be Released to the App Store?

Authors: Maleknaz Nayebi; Homayoon Farrahi; Guenther Ruhe ESEM 2017

Abstract - Several mobile app releases do not find their way to the end users. Our analysis of 11,514 releases across 917 open source mobile apps revealed that 44.3% of releases created in GitHub never shipped to the app store (market). We introduce "marketability" of open source mobile apps as a new release decision problem. Considering app stores as a complex system with unknown treatments, we evaluate performance of predictive models and analogical reasoning for marketability decisions. We performed a survey with 22 release engineers to identify the importance of marketability release decision. We compared different classifiers to predict release marketability. For guiding the transition of not successfully marketable releases into successful ones, we used analogical reasoning. We evaluated our results both internally (over time) and externally (by developers). Random forest classification performed best with F1 score of 78%. Analyzing 58 releases over time showed that, for 81% of them, analogical reasoning could correctly identify changes in the majority of release attributes. A survey with seven developers showed the usefulness of our method for supporting real world decisions. Marketability decisions of mobile apps can be supported by using predictive analytics and by considering and adopting similar experience from the past.

This paper has been published in Proceedings of the international conference on Empirical Software Engineering and Measurement . This is a joint paper of ***Maleknaz Nayebi****,* ***Homayoon Farrahi****, and* ***Guenther Ruhe***[1].

[1]Authors are affiliated with SEDS lab at university of Calgary

7.1 Introduction

Market and user characteristics of mobile apps make their release management different from proprietary software products and web services. By focusing on mobile apps, we introduce the notion of release marketability. "Marketability" is part of release management for platform-mediated software products [63]. Developers release a version of the app on a platform which was also known as the app store or marketplace. Users can download the app from the platform instead of getting it directly from the software owner as it is done traditionally. We illustrated this fundamental difference in offering software app products in Figure 1.2.

While an increasing number of software products are designed and developed for platform-mediated environments, in this paper we only study releases of open source mobile apps. In our former study [178] we found that a substantial number of releases never reach end users through the app store. A *marketed release* is a release that was introduced in a Git repository as well as in the app store. However, a *not marketed release* is a release that was only introduced in the Git repository of the app [178]. "Marketability" refers to the question if a new release should becomes marketed or not.

Based on our former studies [177, 178], we characterize a release with a set of code, release timing, and market attributes and predict release marketability. Once we find that a release is not marketable, we perform analogical reasoning to retrieve similar not successfully marketable releases in the past. We observe how the attributes changed while a release transitioned into a successfully marketable release. Following the idea of case-based reasoning [244], these *analogical changes* give approximation of effort, nature of changes, and code quality needed to be achieved for a release transition. Offering these analogical changes is not meant to be prescriptive and should be adopted and revised considering the context of the app. However, as a form of software engineering knowledge management [233] this is supposed to support developers' decision-making. This paper has four main contributions:

First, we introduced and confirmed the importance of marketability decision for mobile apps

which has not been investigated so far.

Second, we compared three machine learning methods to predict release marketability.

Third, to transition a not marketable release into a successfully marketable one, we performed analogical reasoning using the experience of the same app and looking across apps.

Fourth, we evaluated results of analogical reasoning with actual changes that transitioned an under question releases (internal validation over time). We also performed a survey with app developers for external validation.

The research conducted in this paper is motivated by a survey with 22 release experts discussed in the next section. In Section 7.3, we formulate the research questions. We provide a motivating example in Section 7.4 and describe our methodology (Section 8.3). We outline the design of the empirical evaluation in Section 8.4, followed by empirical results given in Section 7.7. We discuss threats to validity (Section 7.8), report about the analysis of related work (Section 7.9), and provide conclusions (Section 7.10).

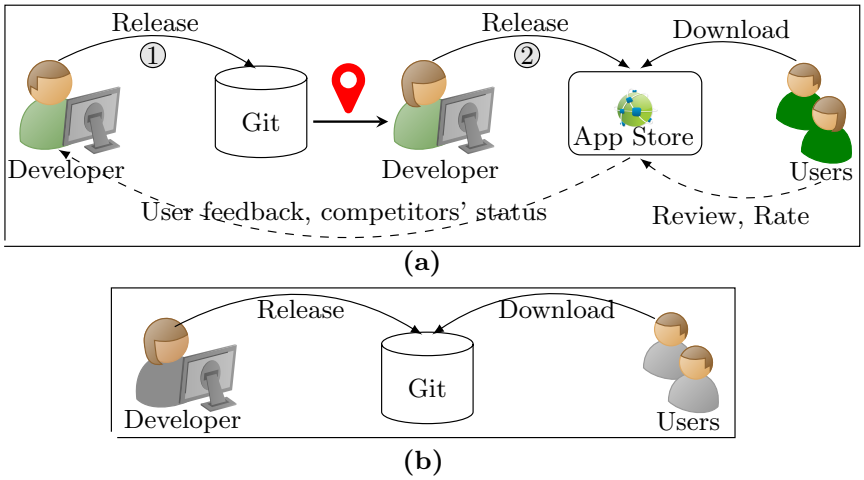


Figure 7.1: Distribution of a product release to the end user for (a) app versus (b) non-app open source software products.

7.2 Significance of mobile app marketability decisions

From analyzing 917 F-Droid open source apps, we found that 35.4% of 11,514 releases were never shipped to the app store. In Figure 1.2 we visualized a simplified release process for open source mobile apps in comparison to traditional open source applications. Considering the process shown in Figure 7.1, a developer releases into a Git repository (① in Figure 7.1(a)). Once the release is in the Git repository, the decision (Figure 7.1 - 📍) to be made is if this release should be shipped to the app store or not. While the release might not be shipped into the app store (② in Figure 7.1 (a)), a user can still install the app from its Git repository. This is different from the analysis of release readiness [6], which is limited to the release on a Git repository as it is shown in Figure 7.1 -(b). Marketability is solely differing between *releasing into a Git repository* or *releasing into the end-user and app store*.

We performed a study with participants of the 4^{th} International Workshop on Software Release Engineering (RELENG 2016) to evaluate the impact of "the market" on release decisions for mobile apps. Participants self-evaluated their level of expertise in release engineering practice, research and release engineering of mobile apps on a 3-point scale (one indicated the lowest and three indicated the highest level of proficiency). The majority of the participants had higher exper-

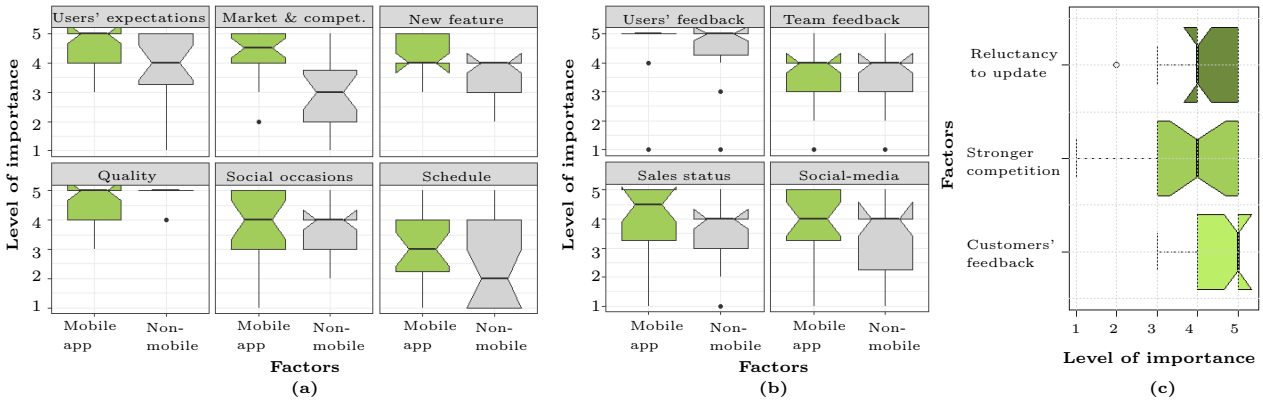


Figure 7.2: Results of survey with 22 release engineers (a) Importance of factors for release planning (b) factors for evaluating success and failure, and (c) reasons of difference in release planning mobile apps versus traditional software.

tise in release engineering practice (with Average 2.1). Our participants had an average expertise of 1.8 in release engineering research and 1.6 in release engineering of mobile apps. To evaluate the importance of marketability, we performed a comparative survey study between mobile apps and non-mobile apps (desktop and web applications) using the factors found by AlAlam et al. [6].

First, participants evaluated the importance of six attributes on planning for mobile versus non-mobile releases. The box plot distributions of the results are shown in Figure 7.2-(a). In this figure, "1" shows the least and "5" shows the highest perceived importance of a factor. The results indicate that *Customers' expectations* and *Market and competitors* are significantly more important for mobile in comparison to non-mobile apps (Mann-Whitney test p-value = 0.032 and 0.001). Also, the results revealed that *Implementing a new feature* has a higher importance in planning for a release of a mobile app compared to other attributes (p-value= 0.002).

Second, the participants evaluated the importance of factors for measuring the success of a release. Our results in Figure 7.2-(b) show that, *customers' feedback* and *sales status* have significantly higher importance for mobile apps in comparison to the non-mobile applications. 20 out of 22 participants (90.9%) believed that *customer feedback* has the highest importance (= 5) for evaluating success and failure of mobile apps.

Third, we asked participants if they think market acceptance is a more important criterion for releasing a mobile app version than it is for other software products.

95.4% *of participants (21 out of 22) believed that market acceptance is more important for mobile apps than it is for any other software product releases.*

Fourth, we asked the survey participants about the perceived reason for this difference by evaluating the importance of three factors extracted by Nayebi et al. [174] on a five-point scale (see Figure 7.2-(c)). On top of that, participants could openly add factors. Survey participants ranked the importance of *customers' feedback* significantly higher than *reluctancy to update* (p-value = 0.032) and *stronger competition* (p-value=0.043). Three participants added below responses:

"Unlike desktop, mobile apps do not seamlessly update. So, this missile some of the effects of public feedback."

"The market is more important for mobile apps as the path between developer and user is much shorter."

"There are more choices in mobile than in web."

The importance of market and the high number of not marketed releases which we observed in open source mobile apps motivated us to investigate on methods for assisting mobile app developers in their decision making. Based on the results showed in Figure 7.2-(b)), we analyze users' feedback to classify releases into successful and non-successful ones.

A marketable release could be successful or not. "Marketable but unsuccessful" is the gray area of marketability decision, saying that similar releases were delivered into the market however they were not successful. In this paper, *transitioning of a release* refers to transitioning a not marketable or marketable but unsuccessful release into a successfully marketable release. We describe the related research questions in the next subsection.

7.3 Research Questions

We investigate on three research questions. We compare classifiers to evaluate if marketability is predictable (**RQ1**) and if so, how can we use past experience to support marketability decisions (**RQ2**). On top of that, we evaluate the usefulness of this support for app developers (**RQ3**):

RQ1 - *(predictability)***:** Which classifier algorithm works better for marketability prediction when comparing Decision Tree, SVM, and Random Forest models?

Why and how: We study how accurately we can predict marketability of a release by analyzing a set of release and app attributes. By identifying a success criteria, we classify each incoming new release of a mobile app as being exactly one of *marketable and successful*, *marketable but unsuccessful*, or *not marketable*. We compare Decision Tree, Random Forest and Support Vector Machine (SVM) classifiers using two sets of attributes.

RQ2 - *(internal validation)***:** How useful is analogical reasoning in suggesting changes for transi-

tioning a release?

Why and how: Learning from experience is an established concept as a form of guidance for decision-makers, not being anyhow prescriptive in its nature. For an incoming new release, we find the most similar releases from the same app and similar apps. Then, we identify changes in release and code which resulted in transitioning a not successful or not marketable release, into a successfully marketable one. Having data over time, we compare analogical changes with the actual changes of a transitioned release.

RQ3 - *(external validation)*: To what extend app developers consider the analogical reasoning useful for making release marketability decisions?

Why and how: Besides the formal and internal validation addressed in RQ2, we performed a survey with actual app developers. This is a very first step in the challenging domain of external validation and design of a decision support tool.

7.4 Motivating Example

We briefly illustrate the main idea of the paper by an example. Through this example, we provide a sneak peek into the prediction and recommendation methods and sample results. For the purpose of illustration, we selected `BatteryXu` which is an app for battery management.

`BatteryXu` has seven releases between *January* 31^{st}, 2014 and *October*, 31^{st}, 2016 as shown in Figure 7.3. Among them, two releases are exclusively kept in the GitHub repository (not on Google Play). Two of the marketed releases were unsuccessful, meaning that they received reviews with

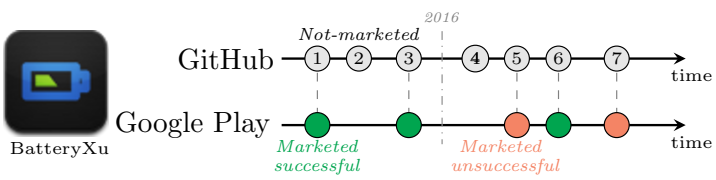
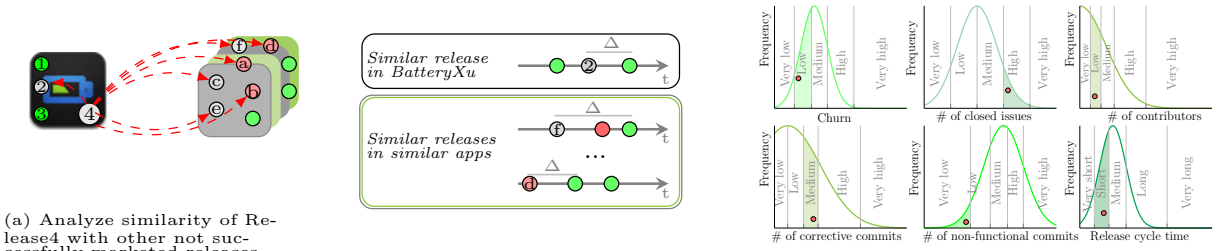


Figure 7.3: Marketed and not marketed release of a sample app "BatteryXu". The color of marketed releases show the success status inferred from review sentiments.



(a) Analyze similarity of Release4 with other not successfully marketed releases (i) within BatteryXu, and (ii) across similar apps.

(b) Find changes in attributes of each similar release that transitioned it

(C) Discretizing relative changes in attributes of similar releases for sample release f

Figure 7.4: Main steps of retrieving past experience for transitioning Release 4 of BatteryXu.

negative sentiments. Beginning of 2016, `BatteryXu` has a *Release* ④ in GitHub. Using release and app attributes (Table 7.1) to build a Random Forest classifier, with 78% precision we predict that *Release* ④ is not marketable and should not be released into the app store (**RQ1**). Looking into the releases later in 2016, we can observe this prediction was correct.

Once we anticipate that the release is not successfully marketable, we retrieve experience to guide developers in future release decisions. In **RQ2**, we search for changes that in the past transitioned a similar not successfully marketed release. Following *case-based reasoning*, by presenting analogical changes to developers, they can adopt, revise, and reuse them [219] to make the release marketable and successful. We summarized the key steps for answering **RQ2** in Figure 7.4.

First, we calculate the Euclidean distance of *Release* ④ of `BatteryXu` from all the other not successfully marketed releases from our sample data set which includes open source mobile apps hosted on F-Droid and GitHub (see Figure 7.4-(a)). We pick the three most similar releases to *Release* ④. As an instance, Release ⓕ of `BatteryBot` was an unsuccessfully marketed release that was similar to *Release* ④ of `BatteryXu`. As demonstrated in Figure 7.4-(b), for each of these three releases we mine which changes were applied to transition the release. These changes are measured as the Δ of the 12 release attributes presented in Table 7.1. To add context and make other releases comparable to *Release* ④, we move from the absolute degree of change in each attribute to relative changes using frequency based discretization. As a simple example, closing six issues in a release of an app which usually closes 20 issues per release is low. However, closing six issues in

a release of an app that usually closes two issues per release is high. Hence, we map the degree of change (Δ) of each attribute relative to the overall change of the attribute across all releases of the app using frequency based discretization. Release ⓕ of `BatteryBot` was not marketed and was the most similar to *Release* ④ of `BatteryXu`. As an instance, we calculated the churn between each two consecutive releases of `BatteryBot`. We calculated the twentieth percentiles of churn for `BatteryBot` and observed that the churn of Release ⓕ is in the 40% percentile, thus considered as "low" (see Figure 7.4-(b)).

As a result, we abstracted three analogical changes that are potentially applicable for *Release* ④ of `BatteryXu`. These changes provided a successful transition of former releases similar to *Release* ④:

Analogical change 1: Low churn ; low # of contributors ; medium # of corrective , very low # of non-functional commits; high # of closed issues ; within a short release cycle .

Analogical change 2: Low churn ; low # of contributors ; high # of perfective, medium # of corrective commits; within very long release cycle.

Analogical change 3: Low churn ; very low # of contributors, high # of corrective, low # of non-functional commits; high # of closed issues ; within a short release cycle .

Analogical change 1 shows that Release ⓕ of `BatteryBot` transitioned into a successful release by involving low churn, low # of contributors, and within short release cycle (proxies for low effort). However, the changes were focused on quality enhancement (nature of changes) as medium # of corrective commits were involved and high # of issues were closed (quality enhancement). These were done along with few non-functional changes.

To internally validate the usefulness of analogical changes, we compared them with the actual changes that transitioned *Release* ④ to successfully marked *Release* ⑥ later in 2016:

Actual change in `BatteryXu`: Low churn ; low # of contributors ; medium # of corrective , low # of non-functional commits; high # of closed issues ; within a short cycle .

Table 7.1: Accuracy of different models for predicting marketability of open source app releases.

Release attributes		
ID	**Attribute**	**Definition**
att_1	Churn	Lines of code that changed (added or deleted) between two releases.
att_2	# of changed files	Number of files that were changed between the former release and the current release.
att_3	# of contributors	Number of people contributed to the current release.
att_4	Release cycle time	The number of days between the former release and current release.
att_5	# of open issues	Number of open issues at the time of release.
att_6	# of issues opened following a release	Number of issues opened between the former release and the current release.
att_7	# of closed issues	Number of issues closed between the former release and the current release.
att_8 - att_{12}	# of corrective, adaptive, perfective, implementation, and non-functional commits	Nature of changes categorized based on categories defined by Hindle et al. [93].
App attributes		
att_{13}	# of app releases	Number of marketed release of the app by the time of current release.
att_{14}	Release cycle variance	Variance of marketed release cycle times by the time of current release.
att_{15}	Average sentiment	average sentiment of all reviews across all releases at the time of the current release.

The maximum # of valid changes (= 5) occurred in *Analogical change 1* which was retrieved from the release most similar to *Release* ④. The minimum # of valid changes (= 3) occurred in Analogical change 2. Overall, two analogical changes, *Analogical change 1* and *Analogical change 3*, were needed to cover all the valid changes.

7.5 Methodology

We discuss the process and different techniques used for our empirical investigation. We used `Python` language and Scikit-learn[2] package throughout the project to gather data, and implement predictive models and perform analogical reasoning.

[2] `http://scikit-learn.org/`

Marketability prediction (RQ1)

A considerable amount of releases fail to attract attention and satisfaction of users. The initial survey in Section 7.2 showed that "customers' feedback" is significantly most important factor to assess success and failure of an app release. In this paper, each release belongs to exactly one category of:

Marketable successful: The release is in a proper status to be released on the app store and likely will be a successful release (attracts positive reviews). This is a *"yes"* answer to the question of marketability.

Marketable but unsuccessful: Similar releases were shipped to the app store however they were unsuccessful. This is a *"maybe"* answer to the question of marketability.

Not marketable: The release is not in a proper status to be shipped to the app store. This is a *"no"* answer to the question of marketability.

To define the success of an app, we relied on users' review sentiments. Among different forms of user feedback in app store, reviews appeared to be the most informative [155, 180]. Each marketed release can be classified as being either successful or unsuccessful:

Release is successful if the average sentiment of reviews after a release and before the next consecutive marketed release is positive;

Release is unsuccessful otherwise.

Following Smedt and Daelemans [248], a user's sentiment in a review could be positive or negative, measured as *polarity* which is a number in the range [-1, 1]. For each marketed release, we used average sentiment of all the reviews following the release and before the next marketed release as the success criteria. We analyzed sentiments' polarity for app store reviews using Python's `Pattern` [248] module. We applied and compared three different machine learning techniques to

classify a new release into one of these categories. To compare the performance of three classifiers Decision Tree, Random Forest, and Support Vector Machine (SVM), we used both 10-fold cross validation and Leave-One-Out (LOOCV) cross-validation. We used two sets of attributes (see Table 7.1):

Release attributes: Code, time, and developer based attributes to reflect status of a release.

App attributes: Release attributes of the app as well as average users' review sentiment across all the releases.

We selected **release attributes** using the results of our former study [174] where we described the characteristics of marketed and not marketed attributes. In that study, we demonstrated the impact of the number and cycle time of releases (*When?*), nature of changes (*What?*), reported issues and change requests (*why?*), and extent and domain of changes (*Which?*). We introduced all the release attributes in Table 7.1. Our previous study [178] showed significant differences between marketed and not marketed releases concerning all the 12 attributes listed in the table. Our initial study showed that not marketed releases have a significantly greater change velocity (att_1, att_2). However, marketed releases have a higher speed in introducing and resolving issues (att_5, att_6, att_7). Our descriptive statistics showed a significant difference between release cycle time for marketed versus not marketed releases (time between each two consecutive releases). With regards to nature of changes (att_8 to att_{12}), it became apparent that not marketed releases have significantly higher number of corrective and implementation commits, while marketed releases put the focus on non-functional changes.

For predicting marketability of releases from Github, we also used **app attributes** and selected them based on the results of our former study [177]. Therein, we investigated which attributes create better entropy between clusters of similar mobile apps. These attributes are also introduced in Table 7.1. Without compromising on entropy we selected a minimum number of possible attributes (three) for building our classifiers [177]. We used average sentiment of the reviews over all releases as the market and user attribute, for simplicity and consistency throughout the paper.

Analogical reasoning to support marketability decisions (RQ2)

In **RQ1** we predicted marketability of mobile app releases. In **RQ2**, we provide analogy-based support for transitioning a not successfully marketable release into a successful one. Analogical reasoning has been successfully used in different fields of software engineering [244] and beyond [219]. Reuse of software knowledge is known to be inherently difficult because, in a strict sense, each software project is different from other in details. However, reuse of experience is essential, and it is intuitive to learn from experience. To assist decision making about release marketability, we retrieve related experience. The input for **RQ2** is a not successfully marketed release. We illustrated the process of analogical reasoning in Figure 7.5 and describe the three steps below:

Step ❶:We find most similar releases to the incoming new release within our knowledge base. The knowledge base is the repository of not marketed releases that have been successfully transitioned into a successfully marketed release. We determine the Euclidean distance between the incoming not marketable release r and all releases in the knowledge base. We pick the m top releases with shortest distance to r.

Step ❷: For each of the m releases similar to r, we analyze their transition into next successfully marketed release. The aim is to guide the transition of r into a successfully marketed release denoted as r^+. We compute the changes in all the 12 release attributes (att_1 to att_{12}) for each pair of a similar release r_j and the transitioned r_j^+ ($j = 1...m$) release. More formally, for an app a, the change $\Delta att_i(a, r_j)$ of a release attribute i is defined by Equation (1):

$$\Delta att_i(a, r_j) = att_i(a, r_j^+) - att_i(a, r_j) \quad \text{for app } a \text{ and release } r \text{ for all similar releases } j = 1...m \tag{7.1}$$

Each value $\Delta att_i(a, r)$ is the absolute value of attributes' change when release r_j transitioned to release r_j^+.

Step ❸: To abstract from detailed numbers, we map the values of $\Delta att_i(a, r)$ into five-point scale

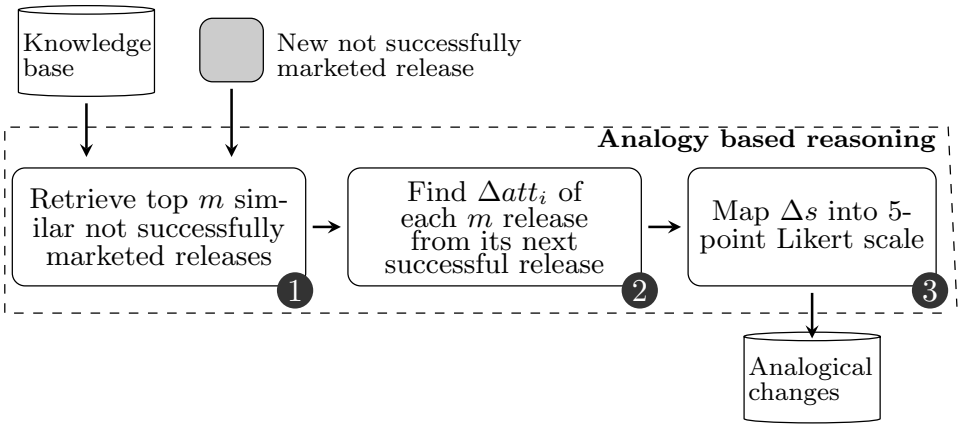


Figure 7.5: The process of creating analogical changes for release transition.

using frequency based discretization. We calculate the change of each attribute during a transition (between the not successfully marketed release under analysis and the first successfully marketed release after that). We descritize $\Delta att_i(a,r)$ considering the frequency distribution of all the changes of att_i in different releases of app a. We calculate five percentiles (at 20^{th}, 40^{th}, 60^{th}, and 80^th levels) of att_i across all releases of app a. This way, each $\Delta att_i(a,r)$ is mapped to exactly one category. We visualized an instance of this process in Figure 7.4.

To evaluate the usefulness of analogical reasoning in this context, we applied analogical reasoning for a release and compared analogical changes with actual ones. To do so, we used all the releases of F-droid apps before 2016 as our knowledge-base. This represent 75% of our whole data set. We demonstrated this process in Figure 7.6. First, we found releases with a successful transition in 2016 for our evaluation purpose (Step 1). For each release with a successful transition (Step 2), we applied the three steps of Figure 7.5 to extract analogical changes. For each release in our test set we found the Δ of all attributes between test release and its actual next successfully marketed release (Step 3). Then we discretized and mapped Δs (Step 4). Finally we compared the actual changes with analogical changes of release attributes (Step 5).

To internally evaluate the usefulness of analogical reasoning in this context (**RQ2**), we compared analogical changes with the actual ones. We used four evaluation criteria:

Coverage: Number of valid attribute changes that were covered by at least one of the analogical

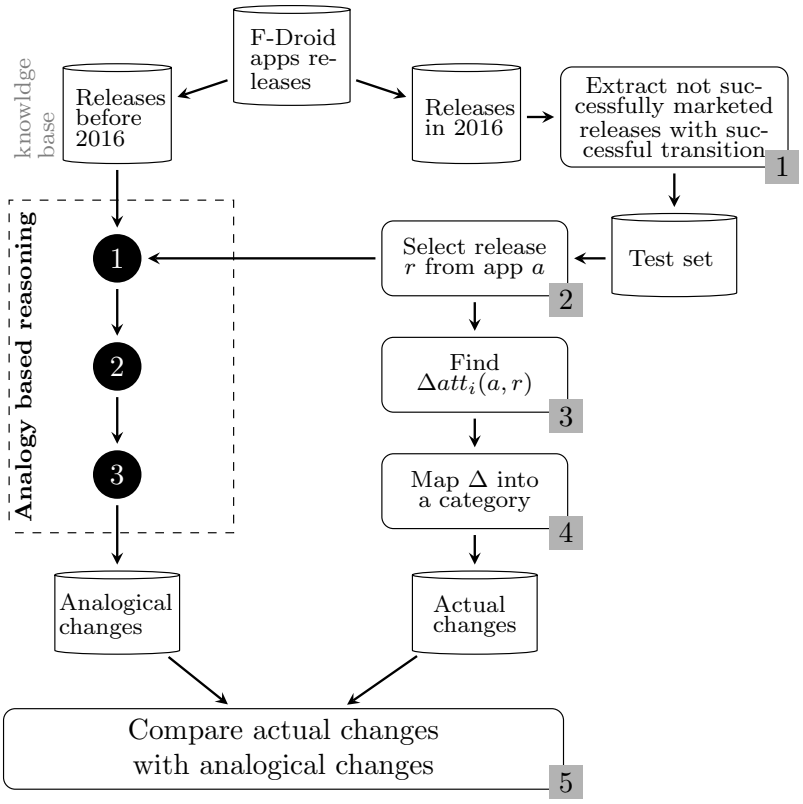


Figure 7.6: The process of analogy based reasoning for transitioning a release and evaluating its results (**RQ2**).

cases.

Distribution: Minimum number of analogical cases needed to cover most number of valid attribute changes.

Max # of valid changes: Maximum number of matches between actual and and analogical changes in a similar case.

Min # of valid changes: Minimum number of matches between actual and analogical changes in a similar case.

External validation of analogical reasoning (RQ3)

In **RQ3**, we performed a survey with mobile app developers to understand usefulness and intuitiveness of the retrieved analogical changes. We asked developers to define the extent (on a 5-point

scale with five being highest) they rely on their own experience, versus the product team experience versus experience from similar products for making marketability decision.

We presented to them "app attributes", a link to apps' GitHub repository, and a link to apps' Google Play page. We showed the attributes of the release under analysis (att_1 to att_{12}) to the survey participants. We presented to them transitions we have mined from similar releases in the past and asked each participant, to classify and rank the usefulness of analogical changes. Each participant could classify extracted transition into **useful** (the retrieved transition is useful to transition the release under question), **maybe** (the retrieved transition may be helpful to transition the release under question), and **useless** (the retrieved transition is not useful to transition the release under question). Participants ranked the transitions in a way that the first transition in class "useful" is the most helpful and the top transition in the class "useless" is the least useful transition.

7.6 Case Study Design

In what follows we describe the design of our case study.

Mining open source apps from F-droid and GitHub

We gathered all the apps from the F-Droid repository of the open source Android apps over a period of 12 months. By monitoring and crawling over that period, we increased our sample size from 1,273 apps at the start to the 1,844 apps as of *October* 2016. Most of them (917 apps) were hosted on GitHub. To get similar data attributes for all the apps (such as release tags), we focused our analysis on the 917 open source Android apps hosted on GitHub. For all 1,844 apps from F-Droid, we parsed the HTML data from their respective F-Droid page and gathered the apps' *package name*, *source control repository*, and *issue tracker* information. We filtered data so that only apps hosted on GitHub open source repositories remained. This process resulted in 917 open source Android apps. We gathered data for each app using their GitHub URL as stated in F-Droid. We analyzed GitHub log of each of these apps to extract release attributes (see Table 7.1).

Collecting data for marketed and not marketed releases

Using the *package names* obtained from F-Droid, we retrieved the Google Play page of each app. We then parsed the HTML data from Google Play (for reviews) and `Searchman.com` (for release dates). `Searchman.com` is a third party analytics platform which gathers data of Google Play over time and hence, keeps all the release dates of an app. We also gathered release identifier and dates from GitHub repository of each app. While mapping releases between GitHub and Google Play, four different cases occurred as shown in Figure 7.7;

Type 1: The GitHub release date and identifier match the Google Play release date and identifier. We call these releases being both in Github and Google Play as *"marketed"*

Type 2: The GitHub release identifier or release date is not an exact match with the Google Play release. We manually inspected several releases to find an approximation schema for mapping two releases. We analyzed release identifiers as well as date of release to map releases between Github and Google Play.

Type 3: The GitHub release is not available in Google Play. This means that neither the identifier nor the release date approximately match with any release in the app store. We call these releases "*not marketed*".

Type 4: The Goole Play release is not available in the GitHub repository of the app. These cases were rarely observed (less than 1% of releases). Often, following this type of releases, open source development on GitHub was discontinued. We excluded these cases from our data.

The most challenging inconsistency was **Type (2)**. In those cases, we called two releases

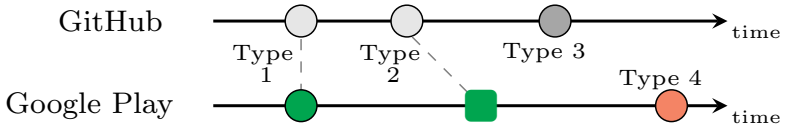


Figure 7.7: Mapping GitHub releases to Google Play releases resulted in four different types of transition.

identical if they had a distance of not more than five days in their release date. We considered two release identifiers the same if one or both had "v" or "version" in the beginning, or one or both had ".0" at the end. For sub-string matching of release identifiers, we used the regular expression "[0-9]+(\.[0-9]+)(\.[0-9]+)?". As an example, if "*V 1.3.0*" were released on March 17^{th} in GitHub and release "*Version 1.3*" released on 18^{th} in Google Play, we considered these two versions the same and called this release "marketed".

As the result of this process, we gathered a pool of 11,514 releases over 917 apps. Among them, 7,435 releases were marketed and 4,079 releases were not marketed. For identifying success of a marketed release, we gathered users' reviews following each release of an app. We analyzed reviews' sentiment [84, 180] by calculating polarity [248]. In total, we analyzed 78,304 reviews. Among the 7,435 marketed releases, we identified 3,734 releases as *successful* and 3,701 releases as not.

Set-up of knowledge base and test set (RQ2)

For the purpose of evaluation in **RQ2**, we mined not marketed or marketed-unsuccessful releases that transitioned into at least one successfully marketed release. For establishing our knowledge base, we limited our search into releases from 2016. The 917 open source mobile apps of our sample set had 2,498 releases on GitHub between *January* 1^{st}, 2016 and *October* 31^{st}, 2016. Among them, 58 releases across 52 apps satisfied the conditions for this evaluation. For a subset of these 58 releases, we could also retrieve similar experience from the same app. This means that for 19 releases of 19 apps, at least one former instance of not successfully marketed releases with a transition to successfully marketed release existed.

7.7 Results

In this section, we present the results of the three stated RQ's sequentially.

Evaluation of classifiers for marketability (RQ1)

We compared Decision Tree, Random Forest, and Support Vector Machine classifiers for predicting marketability. We trained these models first by considering release attributes and second by considering app and release attributes jointly (see Table 7.1 for the attributes). To reduce bias, we evaluated the accuracy of the prediction models by both 10-fold and Leave-One-Out cross-validation (LOOCV).

As the result of data pre-processing in Section 8.4 we ended up with a dataset of 3,734 (32.4%) marketed successful, 3,701 (32.1%) market but unsuccessful, and 4,079 (35.4%) of not marketed releases. We used this data set to build and evaluate our prediction model for **RQ1**. The precision, recall, and F1 score of the three classifiers are shown in Table 7.2. The results of 10-fold cross-validations are the average values of 10 runs of cross-validations. The performance of the three prediction models on our data set is about the same. We used the minimum number of attributes to build our model by excluding correlated variables from our attribute set considering the results of our former study [177].

Decision Tree: We did not weight the decision tree as our sample had approximately the same size of labeled data in each category. We used Gini index to decide on splits of the tree and tuned the parameters to ensure that each leaf node in the tree has the minimum of four samples and prevent over fitting.

Random Forest: For this model, we limited the number of features each tree can use at each split by using square root and 20% threshold. This reduced the F1 score between 4% to 7% comparing to the models without feature restriction.

SVM: To tune parameters we performed an exhaustive search over *Kernel* (function to transform data), *Gamma* and *C* values (for non-linear data transform) in a way to maximize the score of the left out data. This tuning was done using `Scikit`'s `GridSearchCV` package which optimizes parameters by cross-validated grid-search.

Random Forest had a slightly better F1 score. Besides, when using app attributes in conjunction

with release attributes, this results in better recall with precision being about the same.

Internal validation of analogical reasoning results (RQ2)

To internally validate the usefulness of the analogical reasoning following the method described in Section 8.3 and Figure 7.6 we compared the analogical changes with the actual changes and applied the four criteria discussed in Section 7.5. For each release, we retrieved similar releases from the same app (if available) and from five ($m = 5$) similar apps. We used $m = 5$ as we observed most of our 58 releases have five similar releases with normalized Euclidean distance < 0.25. As we discussed in in Section 7.6, we evaluated cross-app analogical reasoning on 58 releases and within app analogical reasoning for 19 releases.

Figure 7.8 shows the evaluation results for 58 releases by only looking into similar apps. Eight release attributes have been changed in the majority of the transitioned releases. The analogical reasoning could cover on average 70% of actual changes, however, these were mostly distributed in three separate pieces of retrieved experience. For 21.1% of transitioned releases, we could correctly retrieve changes in all attributes. For 44.8% of cases, we could correctly retrieve above 90% of attribute changes. The minimum and a maximum amount of change show the most ($median = 50\%$) and least ($median = 18\%$) attribute changes among five similar cases for each of the 58 releases.

Table 7.3 presents the comparison between analogical changes and actual changes, within and

Table 7.2: Accuracy of different machine learning techniques for predicting marketability of open source app releases.

Classification technique	**10-fold cross validation**			**LOOCV**		
	Precision	Recall	F1	Precision	Recall	F1
Use release attributes only						
Decision tree	0.75	0.74	0.74	0.81	0.80	0.81
Random Forrest	0.77	0.78	0.78	0.85	0.86	0.85
SVM	0.68	0.66	0.66	0.83	0.80	0.81
Use app and release attributes						
Decision tree	0.76	0.79	0.74	0.86	0.88	0.81
Random Forrest	*0.78*	*0.79*	*0.78*	0.83	0.89	0.85
SVM	0.71	0.77	0.73	0.73	0.77	0.74

Table 7.3: Comparison of cross app and within app experience for open source mobile apps.

	Reasoning based on similar apps					Reasoning based on same app releases					
ID	**Coverage**	**Distribution**	**# of changes**	**Min. Valid changes**	**Max. valid changes**	**# of cases**	**Coverage**	**Distribution**	**# of changes**	**Min. Valid changes**	**Max. valid changes**
R1	100%	3	8	12.50%	50%	2	100%	9	2	22.20%	33.30%
R2	100%	4	8	25.0%	50.0%	1	75.0%	4	1	75.0%	75.0%
R3	100%	5	11	18.2%	45.5%	1	55.6%	9	1	55.6%	55.6%
R4	100%	3	9	11.1%	55.6%	2	50.0%	4	1	50.0%	50.0%
R5	100%	3	15	26.7%	46.7%	1	41.7%	12	1	41.7%	41.7%
R6	100%	5	6	33.3%	50.0%	1	40.0%	5	1	40.0%	40.0%
R7	100%	2	5	40%	80%	1	25%	8	1	25%	25%
R8	92.80%	4	14	14.20%	64.20%	2	36.30%	11	1	9%	36.30%
R9	85.7%	3	5	20.0%	40.0%	1	57.1%	5	1	57.1%	57.1%
R10	85.7%	4	8	25.0%	62.5%	2	42.9%	7	1	42.9%	42.9%
R11	84.6%	5	15	13.3%	46.7%	1	53.8%	7	1	53.8%	53.8%
R12	80.0%	3	14	21.4%	28.6%	1	40.0%	11	1	40.0%	40.0%
R13	71.4%	2	7	28.6%	85.7%	1	57.1%	5	1	57.1%	57.1%
R14	62.5%	4	11	18.2%	27.3%	2	75.0%	14	2	14.3%	41.7%
R15	60.0%	4	7	14.3%	42.9%	1	40.0%	5	1	40.0%	40.0%
R16	42.9%	3	11	9.1%	27.3%	1	41.7%	12	1	16.7%	25.0%
R17	41.7%	3	6	16.7%	50.0%	1	33.3%	6	1	33.3%	33.3%
R18	41.7%	4	12	16.7%	33.3%	2	23.1%	13	2	7.7%	16.7%
R19	36.4%	4	13	9.1%	27.3%	2	25.0%	12	2	16.7%	16.7%

across apps. Only for one release (R14), the within app analogy reasoning outperformed the cross app analysis. However, it appeared that reasoning within and across apps are complementary. In the majority of releases (63.1% of 19 releases), looking into the within app reasoning increases the coverage of the cross app analysis by 20% to 37.5%. In other words, we can correctly infer changes in some attributes analyzing former releases of the same app.

The results showed that analogical reasoning is useful for guiding marketability decisions. However, the valid changes are distributed across several analogical changes. This is aligned with the general methodology, as in this context, there is no expectation that experience could be reused

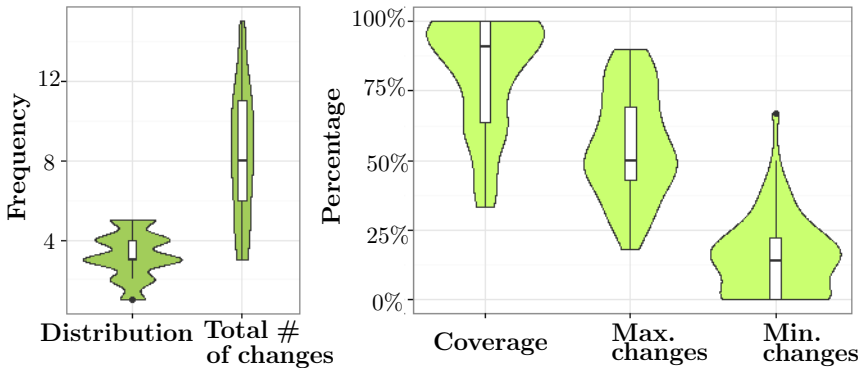


Figure 7.8: The distribution, min and max actions across 58 similar cases.

without adaptation by the domain expert.

External validation of analogical reasoning (RQ3)

We identified apps with frequent marketed and not marketed releases in 2016 (more than 2 of each) and identified developers with a considerable number of commits (more than 5) from each project and invited them to participate in our survey (16 developers in total). Seven developers from four different open source apps participated in our survey for external evaluation of analogical reasoning.

First, we asked the developers to what extent (on a five-point Likert scale) they rely on their "own experience", "status of the apps' former releases", and "lessons learned from other apps" to make release decisions. Only two developers rely on their personal experience stronger than former experience and lessons learned. Reuse of experience within app context were slightly more important to developers comparing to cross-app experience. it appears that reuse of former experience is acceptable along with personal knowledge for surveyed developers. The developers valued their own experience and within- and across- app experiences all on average or above.

Second, for each of the 19 releases we discussed in Table 7.3, a developer defined if the retrieved case is *useful*, it *might be* helpful or is *useless*. Overall, 121 analogical changes for 19 releases were evaluated by seven developers. Each release was evaluated by one developer and no developer evaluated analogical cases of more than four releases. The results showed that 48.8% of all the retrieved cases were considered as "useful" by developers, while 42.9% were considered as "might be useful", and 8.3% as "not useful". Also, 84.2% of *the most similar* cases to each of the 19 releases are "useful" and the rest (three of them) were evaluated as "might be useful". Developers ranked the retrieved experience based on the usefulness. We compared developers' ranking with the ranking based on similarity between analogical releases and the under question release. The positive 0.63 Spearman's rank-order correlation between these two showed that there is a positive relation between releases' similarity and usefulness of them for developers.

Third, we asked developers if and to what extent they would like to reuse experience from other

apps to decide on release marketability. Six of the seven developers stated their definite willingness to know about similar cases while making marketability decision. One developer were uncertain about ease of using this information. Four of the developers suggested more structure, context, visualization and a tool support to make the results more usable:

"Availability of multiple choices is the best. But, looking into the app repository with URL eats my time."

"The choices are not extremely different. Visualizing them or making them side by side or color them."

"With a tool I will use it better. This was not easy to follow."

"I like more organization of info like differences, category, why they are similar to my release, etc. Also, app store had millions of apps not only five."

7.8 Threats to validity

As of any emperical study, we need to consider several threats to judge the validity of the results.

RQ1 - Prediction models for marketability: Parameter tuning (e.g., the number of trees, the size of the trees) affect the performance and accuracy of machine learning techniques. The same applies to the definition of the training and test sets as used in the experiment. We used exhaustive search for tuning SVM in conjunction with extra benchmarking for other two classifiers to tune parameters. While the variation of these parameters and time component is of potential interest, the detailed results of this type of analysis were not considered essential for this study. We also evaluated the accuracy of classifiers using ten times of 10-fold cross-validation as well as Leave-One-Out technique. Mapping releases between GitHub and Google Play for one of the four types (Type (2)) are uncertain. We used manual inspection of multiple releases to infer time and textual filters to mitigate the risk of mapping. The same is true for analyzing the success of marketed releases by analyzing the sentiment of reviews. Multiple factors in app store could be used as a success factor. However, reviews have been used more widely in former research.

RQ2- Usefulness of analogical reasoning: We validated the results of analogical reasoning by comparing the changes retrieved with actual changes over time to reflect how good was our reasoning. The measures we used to evaluate the conformance between former experience in the context of a release and actual changes applied in the under question release are proxy measures. We abstracted attribute changes by mapping them into discretized categories. While this added context to our reasoning, it might create construct validity in the process. We also could evaluate 58 releases using cross-app data and 19 releases using the same app information. We used almost $4/5$ of our data as the knowledge base and only used data of 2016 for evaluation purposes. We can be sure that the results of **RQ2** reflect the performance of our reasoning in most recent data. However, a more comprehensive evaluation by changing the size of the knowledge base and test set might give more insight. We also looked only into top 5 most similar releases considering the distribution of Euclidean distance but tuning this parameter for other contexts should be considered.

RQ3- Perceived usefulness of analogical reasoning by developers: We evaluated the results of analogical reasoning with relatively small number of developers. This is comparable with similar studies evaluating the initial results [262]. We interpret these finding as being initial, considering them as the first step of a more comprehensive evaluation. The ultimate goal is to develop a tool to provide recommendations for developers and evaluate it within real world projects. We have made the first effort in this direction by asking them via a survey. Also, the survey results reflect the subjective opinion of the participants, which may be different from reality. We used convenience sampling to obtain responses which increase the risk of bias.

7.9 Related work

Mobile app release planning: Release planning and feature prioritization is a decision-centric process and part of incremental and iterative software evolution. Release planning and management of mobile apps were studied. Nayebi et al. [174] performed a survey with app developers and users and found that almost half of the developers have a clear strategy for releasing mobile

apps and the strategy impacts the end user. Villarroel et al. [262] introduced CLAP to assist prioritization of users' need using the app reviews. Three app developers evaluated positive usefulness of CLAP. Xia et al. [273] predicted crashing releases of mobile apps by in depth analysis of 10 open source applications with total of 2,638 releases. Beside a successful predictive model, they found that textual data of commit logs are the best predictors of crashing releases. Also, Martin et al. [154] analyzed releases which significantly changed user ratings and performed causal analysis over time.

Release readiness: Release readiness is a composite attribute of software products. In a recent study, AlAlam et al. [6] found that the definition of release readiness is not consistent among different publications. Ware et al. [267] defined release readiness as the attribute for systems' stability and maturity. Port and Wilf [207] related release readiness as the process of measuring uncertainty about the quality of the software to evaluate its fitness for distribution. Shahnewaz and Ruhe [243] also considered release readiness as the readiness of software to be released at a specific point in time. In contrast to the previous work, release marketability is solely focused on the market and customer acceptance using analogy with former releases.

7.10 Conclusions and Future Work

Marketability of mobile app releases is a new decision problem and we showed the practical importance of it. For a set of open source mobile apps and their releases, we applied three machine learning techniques to predict marketability of a release and the results showed the better performance of Random Forest performed best. For transitioning of a not successfully marketed release, we used analogical reasoning to retrieve experience of similar releases and extracted analogical changes. We internally validated applicability of analogical reasoning for guiding marketability decision by comparing analogical changes with actual changes. Moreover, our external evaluation of analogical reasoning with open source app developers showed the usefulness of this method. While results look promising but future research is needed to address some of the discussed threats

to validity. Designing a support tool for more comprehensive empirical evaluation over time and by developers is of interest. The integration of this tool with phases of Case-based Reasoning for reusing, adapting and retaining experience to enhance the reasoning is highly valuable. In addition, identifying higher level proxies such as *effort* and *cost* to abstract release attributes such as *churn* or number of developers will provide tangible insight to plan for the app evolution.

ACKNOWLEDGEMENT

We like to thank the attendees of RELENG'16 workshop for their comments on the initial study of this research and for participating in our survey. Many thanks to the app developers who helped us evaluating the results. This research was partially supported by the Natural Sciences and Engineering Research Council of Canada, NSERC Discovery Grant 250343-12 and Alberta Innovates Technology Futures.

Chapter 8

Crowdsourced Exploration of Mobile App Features

Authors: Maleknaz Nayebi et al. ICSE 2016

Abstract - The ubiquity of mobile devices has led to unprecedented growth in not only the usage of apps, but also their capacity to meet people's needs. Smart phones take on a heightened role in emergency situations, as they may suddenly be among their owner's only possessions and resources. The 2016 wildfire in Fort McMurray, Canada, intrigued us to study the functionality of the existing apps by analyzing social media information. We investigated a method to suggest features that are useful for emergency apps. Our proposed method called MAPFEAT, combines various machine learning techniques to analyze tweets in conjunction with crowdsourcing and guides an extended search in app stores to find currently missing features in emergency apps based on the needs stated in social media. MAPFEAT is evaluated by a real-world case study of the Fort McMurray wildfire, where we analyzed 69,680 unique tweets recorded over a period from May 2^{nd} to May 7^{th}, 2016. We found that (i) existing wildfire apps covered a range of 28 features with not all of them being considered helpful or essential, (ii) a large range of needs articulated in tweets can be mapped to features existing in non-emergency related apps, and (iii) MAPFEAT's suggested feature set is better aligned with the needs expressed by general public. Only six of the features existing in wildfire apps is among top 40 crowdsourced features explored by MAPFEAT, with the most important one just ranked 13^{th}. By using MAPFEAT, we proactively understand victims' needs and suggest mobile software support to the people impacted. MAPFEAT looks beyond the current functionality of apps in the same domain and extracts features using variety of crowdsourced data.

This paper has been published in Proceedings of the International Conference on Software Engineering (ICSE). This is a joint paper of ***Maleknaz Nayebi****, **Mahshid Marbouti** , **Rachel Quapp****, **Frank Maurer** , and **Guenther Ruhe****[1].*

[1] *First, third and fifth authors are affiliated with SEDS lab at University of Calgary, Second and forth authors are

8.1 Introduction

There is evidence that social media platforms, especially Twitter, are widely used during emergency situations [102, 277]. In times like this, when staying connected is not a convenience but rather a necessity, mobile devices are a lifeline. During the 2016 Fort McMurray wildfire in Alberta, Canada, 80,000 people were evacuated and unable to return for over a month. As local radio stations went off the air and websites failed, social media became the crisis' unofficial emergency broadcast system.

Aware of the importance of mobile functionality during emergency situations, several applications were designed by NGOs (non government organizations), official state departments, and international humanitarian organizations. In particular, mobile apps for wildfire emergencies have been produced by the Red Cross and International Association of Fire Fighters, as well as the Canadian, United States, and Australian governments. However, our analysis of the apps available in Apple iTunes (Apple's app store) revealed that current wildfire app functionality is mainly limited to pushing notifications, sharing news, and providing access to fire maps. Our results show that the apps presently available to people in need are lacking relevant and useful features.

We investigate whether, and to what extend, existing wildfire apps reflect people's needs in emergency situations. We also suggest features to enhance these apps' functionality by harnessing the power of Twitter. To do so, we propose a method called MAPFEAT[2] and apply it to a case study of the Fort McMurray wildfire. MAPFEAT is a method for systematic tweet analysis and extended app store search. It explores software features that meet the needs of people in emergency situations.

MAPFEAT extracts needs from Twitter and maps them to features of apps already existing in the market (not exclusive to emergency apps). App stores provide a wealth of information about software features, as each app in app store is described by its technical functionality. MAPFEAT performs an extended search to support a better match between users' needs and actual function-

with ASE lab at University of Calgary

[2]**M**ining **AP**p **FEA**tures from **T**weets: MAPFEAT

ality offered. This automated process and the use of crowdsourced data enables emergency app developers and owners to provide mobile software support that more effectively helps impacted people and facilitates the management of the crisis.

Suggesting functionality enhancements to emergency apps by mining tweets would primarily benefit ordinary people (also known as the general public) that were affected by the crisis. This support is intended to (i) facilitate the connection of impacted people to the emergency authorities and service providers, and (ii) enable people to connect to each other and help in possible cases. The method is evaluated by a case study where we report on the results of mining tweets published during the Fort McMurray wildfire.

In this paper, we answer the following research questions:

RQ1: How can tweets be mapped into mobile app features automatically?

Why and how? Past research has shown that mining social media services, particularly

Table 8.1: Features of the top ten apps retrieved by search queries from the Apple iTunes app store. The *italic* features were common to at least two apps retrieved by a search query.

(i) "gas", "availability","lineup"		(ii) "gas" AND "availability"		(iii) "gas" AND "lineup"	
App	**Feature**	**App**	**Feature**	**App**	**Feature**
(i)-1	*Find cheapest gas based on your gas type*	(ii)-1	*Find cheapest gas* based on your gas type.	(iii)-1	Shows *gas prices* with easy to read maps.
(i)-2	*Share gas prices* across borders	(ii)-2	*Choose the nearest gas station*, bank and hospital based on your position.	(iii)-2	Crowd-sourced *gas prices* across borders
(i)-3	*Get the best Fuel Prices.*	(ii)-3	Gas delivered to your car - anytime, anywhere.	(iii)-3	Gas/diesel taxes 55 pieces of travel relevant information.
(i)-4	Gas/diesel taxes 55 pieces of travel relevant information for each of the 50 states.	(ii)-4	GPS station locator that provides real-time maps, directions. *Map a route to the selected station or enter a custom address.*	(iii)-4	Live routing based on community driven, *real-time traffic*. Find the *cheapest gas station* route with *lowest lineup*.
(i)-5	*Near real-time prices on your mobile device.* 10 Month Futures Chain. *Report gas prices*. More Crude Oil tracking: Sweet Light, Brent.	(ii)-5	*Find the cheapest gas* near. *Find gas stations* by distance/price. Report gas prices to help others find cheap gas and earn points. Filter stations by amenities and brands.	(iii)-5	Find route to gas station with *real time traffic, alerts and lineups*. Share location, route and ETA. Make a pit stop on your route and find the *cheapest gas* and other local amenities.

Twitter, is an effective way to manage emergency situations [96]. We expect that mining the needs of the general public and supporting them with software functionality would have a considerable impact on app design. We propose a method called MAPFEAT to perform this process systematically.

RQ2: For the Fort McMurray case study: How do features mined from MAPFEAT compare to the ones provided by existing wildfire apps?

Why and how? We compared the MAPFEAT results with features in existing wildfire apps by exploring which features in existing wildfire apps matched with needs stated in social media. We mined features from existing app descriptions following the method proposed by Harman et al. [88].

RQ3: For the Fort McMurray case study: How will the features generated by MAPFEAT be perceived by the general public?

Why and how? With the purpose of understanding the value of the MAPFEAT results, we crowdsourced a survey of 500 respondents. Respondents were given the list of features and asked to evaluate each feature as either essential, worthwhile, unimportant, or unwise.

To give an outline of the key idea and outcome of the paper, we provide a motivating example in the next section. In Section 8.3, we offer a solution to **RQ1** by introducing MAPFEAT. Then, we describe the data set and design of our case study in Section 8.4. Using the case study, we answer **RQ2** in Section 8.5. In section 8.6, we respond to **RQ3** by reporting the results of a survey of the general public. We conclude the paper by discussing threats to validity (Section 8.7), related work (Section 8.8), and the relevancy and applicability of the results (Section 8.9).

8.2 Motivating example

We briefly illustrate the main idea of the paper with an example. Through this example, we provide a sneak peak into our method, MAPFEAT, and the results of its application. For the purpose of illustration, we selected a sample of tweets about the Fort McMurray wildfire, which can be seen in the text box below.

May4 4:30pm: Moose Haven Lodge has rooms, food, gas avail. HWY 881 2 KM 217. Call 780-935-2372.
Line for fuel - four pumps- snakes in Conklin.
Any gas available between Fort Mac and Grasslands?
YMMfire gasoline supplies to stations south along HWY 63 stable but lines ups from exodus causing some stations to not honour credit cards.
Is there any gas at Wondering River?Line up?
I'm in Boyle and have some gas, granola bars, and water if people are still stranded. Text (780)504-***8.

To automatically extract the topic of discussion from these tweets, we first lemmatized them by mapping the words into their dictionary form [106]. After applying topic modeling, we received the topic "gas, availability, lineup." Then, following the next step of MAPFEAT, we used this topic to run a thorough search in the app store and find all relevant apps. To search the app store, every combination of words in the topic with a length greater than one were sent to the Apple iTunes API:

(i) "gas" AND "availability" AND "lineup"

(ii) "gas" AND "availability"

(iii) "gas" AND "lineup"

(iv) "availability" AND "lineup"

We selected the top 10 apps retrieved by each search query, with the exception of (iv), which did not receive any results from Apple iTunes (none of the apps matched this search query). The features of the apps found by queries (i) to (iii) are presented in Table 8.1. In some cases, separate search queries retrieved the same apps. Then, we collected and mined features from the app descriptions [88]. Some features are shared between different apps, such as (i)-4 and (iii)-3 in Table 8.1. The features in italics are the ones common to at least two apps retrieved by a single search query. Only these features are of interest to the MAPFEAT process because they are mutually shared between apps, and thus, have a higher chance of matching the original topic. As a result of this process, we ended up with the following features:

- Find real-time gas prices
- Find cheapest gas
- Find nearest gas station
- Share gas prices
- View real-time traffic
- View lineups

In accordance with the last step of MAPFEAT, we finalized our mapping between tweets and app features by using crowdsourced evaluation to see if our extracted features were related to the tweet topic. We submitted the following question to `Amazon Mechanical Turk`[3], ensuring that exactly five workers answered it:

[3] `https://www.mturk.com/mturk`

Select all features related to the topic ”gas, availability, lineups”:

☐ 1- Find real-time gas prices

☐ 2- Find cheapest gas

☐ 3- Find nearest gas station

☐ 4- Share gas prices

☐ 5- View real-time traffic

☐ 6- View lineups

The results of this crowd evaluation showed that all five of the workers agreed that the tweet topic “gas, availability, lineup” was related to features (1), (2), (3), (4), and (6). However, *four* of the workers believed that the topic is not related to feature (5), “view real-time traffic”. Consequently, we mapped the tweet topic “gas, availability, lineups” to the following app features:

- Find real-time gas prices
- Find cheapest gas
- Find nearest gas station
- Share gas prices
- View lineups

Our study of the 26 existing wildfire apps showed that none of these features were currently implemented in any of these apps. However, people affected by such a crisis can benefit from having integrated information about the local price, availability, and convenience of gas stations within one app. Going a step further, we surveyed the crowd via `Amazon Mechanical Turk` and found that 53.2% of those surveyed consider these features to be essential in a wildfire app (average between five features).

So, while a feature like ”view lineups” already exists in the app store as part of a navigation app, an emergency app with that feature would be better suited to address the needs of those involved in a wildfire emergency.

8.3 MAPFEAT: Mapping tweets to app features

The purpose of MAPFEAT is to introduce new features to emergency apps by offering a systematic method that transforms needs communicated over social media into software features. We propose a combination of techniques that automate the mining of needs from general purpose tweets and their translation into real app features using app store as the pool of possible features. The process of our proposed method is presented in Figure 8.1 and its steps are described in the following subsections.

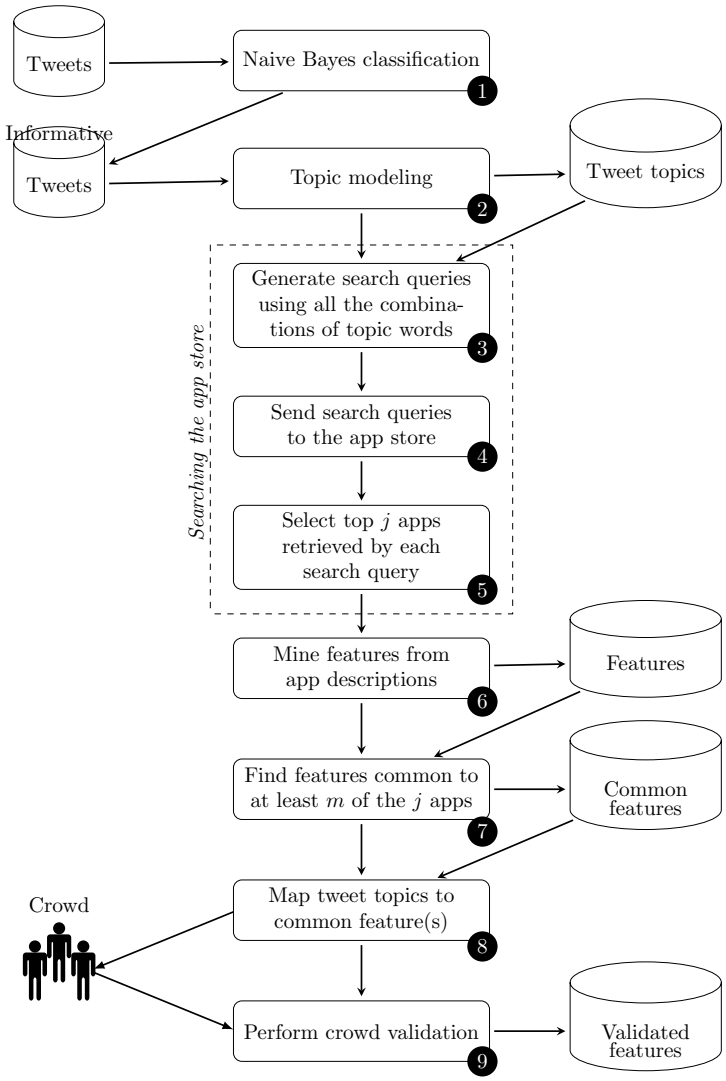


Figure 8.1: MAPFEAT process of mapping tweets to app features

Automatic classification of tweets

Focusing on a specific event gave us the chance to collect related tweets by following the trending keywords and hashtags of that event. This is represented by Step ❶ in Figure 8.1. Many tweets do not convey a need or requirement. Consider the following tweets:

"I don't care if you're right or wrong - seeing #ymmfire as an opportunity to talk politics or anything else is wrong."

"All my thoughts are with all the people who are affected by the devastation of the fire. #ymmfire #FortMacFire"

We defined the class of *non-informative tweets* as tweets lacking an explicit potential to be mapped into a software feature (like the above instances). *Informative tweets* are the ones that either express a requirement or offer support for fulfilling a requirement (see the tweets in the motivating example).

We used the Naive Bayes classifier, as it has been suggested by other studies for tweet classification [127]. Naive Bayes is based on the strong assumption that the existence of a word is independent of the existence of another word. For example, if the tweet contains the word "map", this gives us no further information about "traffic". The strength of Naive Bayes is that it is simple and has proved to perform well with a small training set [124].

Topic modeling

In the next step we used topic modeling to extract the in common topics between the needs stated in general purpose tweets (Step ❷ in Figure 8.1). The topic modeling of tweets was studied by Hong et al. [94]. LDA, known to work well on tweets [30], is an established method used to find common topics in a set of natural language text documents. Topics are created out of a combination of words that tend to appear together frequently in the documents of the corpus. LDA assumes there is a fixed number of K topics. It assigns a probability distribution over topics to each document. To

decide which number of topics K made the most sense, two measures were suggested:

- the perplexity (log-likelihood of a held-out test set), where low perplexity indicates better generalizability of a topic [30], and
- intuitive meaningfulness of the results [42].

Each topic consists of a set (cluster) of w words that describe the topic. For example, if we decided to describe each topic by *three* words ($w = 3$), then LDA would group "gas", "traffic", and "map" together under one topic, and "accommodation", "room", and "hotel" under another topic.

Searching the app store

At this point, we have extracted topics from tweets and each topic is represented by a set of words. A software analyst can potentially map a tweet topic into software features. By systematically searching the app store, we not only automate this process, but also overcome the subjectivity involved in the human-based mapping process.

To this end, we generated multiple search queries using the combination of words in each topic (Step ❸ in Figure 8.1). We selected all combinations of words having a length ≥ 2 (Step ❹). Search queries with the $length = 1$ (Searching one word only) results in many general apps not really related to our purpose. We crawled the description of the top j apps retrieved by each search query (Step ❺).

Mining features from app description

Using the app descriptions gathered in the previous step, we then mined features (Step ❻ of Figure 8.1). To accomplish this, we followed the method proposed by Harman et al. [88, 236] to extract *featurelets* from app descriptions. They defined featurelets as "a set of commonly occurring co-located words, identified using NLTK's N-gram CollocationFinder package" [88, 236]. Following their proposed approach, we used a greedy hierarchical clustering to aggregate similar featurelets

(having similarity $\geq 60\%$) [88]. As a result, we were able to extract features from the app descriptions, each consisting of two or three words (bitri-grams, i.e. 2-grams or 3-grams), which are considered to be the features provided by an app [236].

Mapping tweet topics to app features

So far, we described the process in which a tweet topic resulted in several search queries. Then, we extracted features from the description of the top apps retrieved for each search query. At this step, we mapped each search query to the features that were shared between at least *m* of the apps (Step ❼ in Figure 8.1). We did this because a specific keyword search retrieves a variety of apps, and thus, the commonality between their features is of special interest to us (we consider these functionalities as the reason that the apps were retrieved together by a specific search query). For the precise mapping, we present each tweet topic along with its corresponding features to the crowd and asked five workers to select all the features that are related to the tweet topic (see the motivation example). Considering the majority of votes, the mismatched features are excluded from the tweet topic.

Lastly, we mapped the original tweet topic to a set of features (Step ❽). The corresponding features of a tweet topic are defined as set of features received from every search query that originated from it. This way, we performed an extensive search of the app store and retrieved the union of all features found by search queries that originated from a specific tweet topic.

Evaluation

At this point, tweets have been mapped to app features. In order to validate the accuracy and value of the results, we performed crowdsourcing to evaluate the semantic relationship between each topic and its resulting features (Step ❾ in Figure 8.1). We automated this step by using the API of `Amazon Mechanical Turk`. We provided the extracted features along with their corresponding tweet topic to the crowd and asked them to select all features relevant to the topic (see the motivating example). If the majority consider a feature not belonging to the topic, the respective feature is

eliminated from the topic mapping list, otherwise there would be no change in the proposed feature set.

8.4 Case Study Design

We recorded 69,680 unique tweets submitted in the context of the Fort McMurray wildfire over a period of May $2^n d$ to May $7^t h$, 2016. We applied MAPFEAT on these tweets as the proof of concept. In the following subsections, we describe the design of our case study. The results are presented in Sections 8.5 and 8.6 (addressing RQ2 and RQ3, respectively).

Mining tweets about the Fort McMurray wildfire

Studying Twitter communication during emergency events is challenging. Access to the Twitter Search API is limited so we needed to decide what to collect at a stage when we only had limited information about the crisis.

To form the tweet database for our case study, we used the `Twitter Search API` to obtain publicly available tweets about Fort McMurray wildfire. The `Twitter search API` is restricted to 180 queries every 15 minutes and only returns tweets from the past 14 days. We used the trending hashtags #ymmfire, #FortMacfire, and #ymm to pull tweets connected to the Fort McMurray wildfire. We chose these hashtags through an initial investigation of the public Twitter stream.

We continuously gathered tweets related to these hashtags over the course of 6 days. The accumulation of this data resulted in 69,680 unique tweets.

Then, we randomly selected 2% of the tweets and performed manual analysis. First, we explored the usefulness of the data for our study. Second, we used this subset to train the Naive Bayes classifier to separate informative tweets from the non-informative ones.

Text pre-processing and lemmatization

Observing the common practice in text mining, we pre-processed tweets to make them ready for applying machine learning techniques such as Naive Bayes and LDA. We took the following steps:

1. Eliminated retweets,
2. Eliminated hashtags, emojis and special characters,
3. Eliminated URLs,
4. Eliminated duplicate tweets, and
5. Lemmatized tweets to map words into their dictionary form while retaining the context of the word [106, 141].

For pre-processing, we used an open source module called `Pattern` [248], which is built on top of the comprehensive `NLTK` python package [139]. We also used a python library, `Gensim`, to apply LDA and extract topics from the pre-processed tweets.

Survey design for evaluation of existing wildfire app features

All features were evaluated through crowdsourcing as the last step of MAPFEAT. We considered the crowd workers to be representative of the general public, with the assumption that they potentially might be involved in a fire emergency (often this probability provoke us to buy a fire insurance). We used a Kano-inspired survey [20] to evaluate the proposed features.

We asked the crowd to imagine that their hometown or place they were traveling was being destroyed by a wildfire and that they have their smart phone to help them in this situation. Then, we presented the features to them and asked:

"*In your opinion, how important is it to have this feature as part of a wildfire emergency app?*"

- It is essential
- It is worthwhile
- It is unimportant
- It is unwise
- I don't understand

To increase the validity of the results, we filtered out incomplete and low quality responses. A response was considered low quality if the respondent spent less than 20 seconds answering a question or selected the same answer for more than 90% of the questions. From the responses, we calculated the percentage of answers in each of the five categories for every feature. Then, we assigned the category with the highest percentage to the features and call it *an essential feature*, *a worthwhile feature*, *an unimportant features*, or *an unwise feature*. We used this type of survey to evaluate both the features in the existing apps and the features mined by MAPFEAT.

8.5 Comparison of features: generated by MAPFEAT vs. existing apps (RQ2)

We compared the features extracted by MAPFEAT with the ones in pre-existing apps for wildfire emergencies. For each feature, three scenarios were possible:

Scenario 1. The feature mined by MAPFEAT exists in the current wildfire apps.

Scenario 2. The feature mined by MAPFEAT does not exist in the current wildfire apps. In this case, MAPFEAT could provide a suggestion on a feature needed by general public. This can be taken by developers as a useful insight to design more desirable apps.

Scenario 3. The feature that exists in a current wildfire app was not mined by MAPFEAT. We interpret this either as a lack of justification for having such a feature, because it was not requested by the crowd, or incompleteness of the data. We suggest that app developers and software engineers inspect the usefulness of such features.

To answer **RQ2** and demonstrate the added value of our proposed method, we took three steps which are further described in the subsections below:

Step 1: Extract features from existing wildfire apps,

Step 2: Apply MAPFEAT to the Fort McMurray wildfire tweets, and

Step 3: Contrast the results of MAPFEAT with the existing app features.

Step 1: Extracting features from existing apps

In our study we focused on the Apple iTunes app store as it provides a free API for searching. We searched with "fire" and "wildfire" as keywords to find the apps available in the market. Our initial search resulted in 86 apps. We then read the app descriptions and filtered irrelevant apps from this set (for example, game and music player applications). We ended up with 26 apps that were related to wildfire emergencies.

Among the 26, one app was developed by the government of Alberta (the province in which Fort McMurray is located). This app was first released in 2013 and had three updates at the time this paper was written. Two other Canadian wildfires apps were found, one from British Colombia and the other from Saskatchewan, as well as three Australian apps from Tasmania, Victoria, and Queensland which all were developed by government and emergency department. Also, nine apps showed wildfire information for different cities of the United States, in addition to an official Red Cross app titled "Wildfires by American Red Cross." Lastly, there was an app from the International Association of Fire Fighters (IAFF). These 26 apps were published across seven different app store categories: lifestyle, news, weather, navigation, utilities, business, and education.

To extract features, we used a method proposed and evaluated by Harman et al. [88] as described in Section 8.3. Using this method, we found a total of 28 unique app features distributed

across all 26 apps.

Step 2: Applying MAPFEAT to Fort McMurray tweets

In this subsection, we describe the concrete data and results by applying the steps of MAPFEAT, as described above on Fort McMurray data. We first separated informative and non-informative tweets by applying the Naive Bayes classifier (Step ❶ in Figure 8.1). Applying 10-fold cross validation showed that the classifier has an accuracy of 75.8%, precision of 63.4%, and recall of 62.2%.

We then performed topic modeling on the informative tweets. Considering the perplexity and intuitiveness of the topics (see description of **RQ1**), we extracted 193 topics, each being described by *seven* words ($w = 7$). From using all the different combination of words (with combination of at least two words) as described in Step ❸ of Figure 8.1, we generated 23,400 search queries. We used the `Apple iTunes Search API` to retrieve apps for each search query. Each of these searches took 0.53 *seconds* on average to get the results. We selected the top 10 apps from each search result ($j = 10$ in Step ❺ of Figure 8.1). 10,950 search queries did not return any apps. The results of all the search queries originating from a single topic were then aggregated. In total, our search queries resulted in 14,626 unique apps.

The motivation for conducting this extended search was to find software support (in form of apps' functionality) using the app store as a rich set of software features. For that purpose, we extracted features from all app descriptions that matched any of the search queries and selected features shared by at least two of the 10 apps retrieved per query ($m = 2$ in Step ❼ of Figure 8.1). To determine the similarity between two feature we set the threshold of cosine similarity measure to 60%. Doing this for all the retrieved apps of each search query, we found 188 features. We submitted these features along with their originating tweet topic into a crowd to verify the correct semantic relation between them. Crowd detected 13.2% semantic mismatch between the topics and extracted features. Excluding the mismatched feature and topics, we ended up in the final number of 163 features that were mapped into the tweets of Fort McMurray wildfire using MAPFEAT. The

partial list of mined features is presented in Table 8.2 while the complete list of features is available online[4].

Step 3: Contrast MAPFEAT results with existing wildfire app features

MAPFEAT extracted 163 features considering the tweets about Fort McMurray wildfire. Having the features extracted by MAPFEAT, we compared them with features existing in the current wildfire apps. MAPFEAT could extract 139 features matching the needs of general public which were not considered within existing apps.

Our analysis showed that applying MAPFEAT on Fort McMurray tweets can find 85.7% of the features (i.e. 24 out of 28 features) available in existing wildfire apps. Using MAPFEAT, we mined 90% of the features that exist in more than one app. However, MAPFEAT did not mine the feature "follow location" that occurred twice in the existing wildfire apps. This might be because of data incompleteness or may indicate that general public does not need this feature(Scenario 3). The results of survey (**RQ3**) will shed some light on this by evaluating the importance of features for the crowd.

8.6 RQ3: Evaluation of the wildfire app features

We performed another round of crowdsourced evaluation and asked the crowd how they perceive the value of the features mined by MAPFEAT. In other words, **RQ3** asks whether our proposed method generates app features that are deemed valuable by the general public in an emergency wildfire situation.

Having all the features extracted by MAPFEAT in addition to the *four* features in the existing apps that were not mined by MAPFEAT, we performed crowdsourcing. We asked 500 master (highly ranked) workers using `Amazon Mechanical Turk` to evaluate the importance of having

[4] `http://www.ucalgary.ca/mnayebi/tools-and-data-sets`

each specific feature in the app. Using the results of the questionnaire (see Section 8.4), we provide in Table 8.2 the percentage of subjects who believe that a feature is essential, worthwhile, unimportant or unwise.

Table 8.2: Evaluation profile of top 40 ranked features extracted by MAPFEAT in comparison to features of existing wildfire apps

Rank	**Feature**	**MAPFEAT**	**Baseline**	**% selected by survey participants**			
				Essential	Worthwhile	Unimportant	Unwise
R1	Fire alarm notification	✓	-	69.80%	22.04%	5.31%	1.63%
R2	Food and water requests and resources	✓	-	67.76%	22.04%	6.94%	2.86%
R3	Emergency maintenance service	✓	-	65.71%	24.49%	6.53%	2.04%
R4	Send emergency text messages	✓	-	65.31%	28.16%	4.90%	1.22%
R5	Safety guidelines	✓	-	64.98%	26.35%	6.14%	1.81%
R6	Fire and safeness warning	✓	-	64.90%	24.49%	8.16%	1.63%
R7	Request ambulance on a tap	✓	-	64.62%	22.74%	7.58%	4.33%
R8	Find nearest gas station	✓	-	63.90%	22.02%	8.30%	3.97%
R9	Emergency zones maps	✓	-	63.54%	25.27%	6.86%	3.97%
R10	Find a medical center	✓	-	61.01%	25.27%	10.11%	2.53%
R11	Subscribe for real time alerts	✓	-	60.82%	28.98%	6.94%	2.86%
R12	View gas lineups	✓	-	60.65%	26.71%	7.22%	4.69%
R13	Real-time fire information	✓	✓	60.55%	28.60%	6.31%	2.56%
R14	Fire education	✓	-	60.29%	26.71%	9.03%	3.25%
R15	Report incident	✓	-	60.29%	28.52%	7.22%	3.25%
R16	List of medical centers	✓	-	59.93%	27.80%	7.58%	3.61%
R17	Emergency kit list	✓	-	59.93%	28.88%	5.42%	4.69%
R18	Real time news notification	✓	✓	59.76%	28.60%	6.11%	4.54%
R19	Real time maps	✓	-	59.21%	25.99%	8.66%	4.69%
R20	Location-based fire information	✓	✓	58.78%	31.36%	5.72%	2.96%
R21	Gas availability & supply	✓	-	58.48%	28.16%	9.75%	3.61%
R22	List of fires	✓	-	57.76%	25.99%	11.19%	3.97%
R23	Direct police call	✓	-	57.04%	27.08%	10.47%	5.05%
R24	Instructions for before/during/after a wildfire	✓	✓	57.00%	31.95%	7.50%	2.17%
R25	Wildfire map	✓	✓	56.02%	31.76%	6.71%	3.75%
R26	Choose emergency types	✓	-	55.92%	34.69%	3.67%	3.27
R27	Real time update synchronization	✓	-	54.69%	32.24%	10.20%	1.22%
R28	Interactive learning procedure about fire	✓	-	52.71%	31.77%	9.03%	5.42%
R29	Device battery saving mode	✓	-	52.65%	31.84%	11.02%	4.08%
F30	Chat with aid respondents	✓	-	52.65%	36.73%	8.16%	2.45%
R31	Urgent health care provider request on tap	✓	-	51.99%	30.69%	13.36%	3.25%
R32	Patient transport request	✓	-	51.62%	33.21%	9.03%	5.05%
R33	Friends emergency contact	✓	-	50.90%	36.10%	8.30%	3.97%
R34	Live police chat	✓	-	50.54%	32.85%	10.11%	6.14%
R35	Up-to-date weather condition	✓	-	50.18%	32.85%	12.27%	3.97%
R36	Test fire vision distance and safety quality	✓	-	50.18%	33.21%	9.75%	5.42%
R37	Post a service request	✓	-	49.80%	35.51%	9.39%	2.86%
R38	Real-time news feed	✓	✓	49.70%	34.32%	11.24%	3.94%
R39	Evacuees residence and site list	✓	-	49.39%	30.20%	14.29	4.90%
R40	Emergency electricity maintenance request	✓	-	48.74%	35.38%	10.11%	5.05%

Figure 8.2: Boxplots of crowdsourced feature evaluation

Scenario 1. **Features that were mined by MAPFEAT and currently exist in wildfire apps**
The results showed that the respondents found 25% of the 26 features of the existing wildfire apps as *essential*, while 35.7% of these features were evaluated as *worthwhile*. On the other side, 28.5% of the features were considered *unimportant* and 10.7% of them were evaluated as *unwise* ("flashlight" and "Generating and drawing wildfire maps" were considered unwise to be part of a wildfire app).

Comparing the crowd stated importance of these 24 features with the rest of the 139 features mined by MAPFEAT, shows the extensive mismatch between what the general public needs and what is currently provided in the wildfire apps. Looking into the results (partially reflected in Table 8.2), shows the feature with highest "essential" degree that exists in the current wildfire apps, is ranked as #13. However, MAPFEAT could mine 12 features that had higher priority for public.

Scenario 2. **Features that were mined by MAPFEAT and do not exist in wildfire apps**
By crowdsourced evaluation of the 139 features mined by MAPFEAT, we found that:

- 28% of the features were classified as *essential*,
- 56.1% of the features were classified as *worthwhile*,
- 14.3% of the features were classified as *unimportant*, and
- 1.4% of the features were classified as *unwise*.

That means that about 84% of the features detected are worthwhile or even essential. The violin boxplots in Figure 8.2 show the frequency distribution for all four categories. For example, for the *worthwhile* category, the highest concentration is around 40%.

MAPFEAT introduced 139 features that were not available in the currently existing wildfire apps. These features are not new to the app market in general, however they were not offered as part of existing wildfire apps. MAPFEAT mined 39 essential features that not only covers all the already available features in the wildfire apps but also is 83.9% more features. In terms of quality, among the top 40 wildfire features mined from articulated needs, just six of them were present in the baseline feature set, with the most urgent one of them ranked 13^th.

Scenario 3. **Features that were not mined by MAPFEAT**

Using MAPFEAT, we could not find one *unimportant feature* (51.08% of the respondents called it *unimportant*) and two unwise (39.05% and 42.41% called them unwise) features. So while MAPFEAT resulted in 144 essential and important features, it failed to find one worthwhile feature that already exists in the wildfire apps.

Three of the features that were missed by MAPFEAT were evaluated as *unimportant* and even *unwise*. This indicates that existence of some of the features in wildfire apps are not well justified as these were not communicated in the tweets nor being evaluated as *essential* or *worthwhile*. Also, MAPFEAT could not mine one worthwhile feature. This happened as we only used one data set for a specific wildfire. Other wildfire tweet data set, would likely lead us to explore more features for wildfire emergencies.

8.7 Threats to validity

We reported the results of a case study to show the process and effectiveness of MAPFEAT. In the following, we discuss threats to validity.

Construct validity: Considering tweets as a representative source for the needs of the general public might be of concerns for designing an app. However, we limited the application of MAPFEAT to emergency apps, as a huge body of research and analysis showed that tweet analysis is indeed an efficient aid to manage a wildfire.

We used crowdworkers for validation purposes. Another threat here is that subjects might not have first-hand experience with a wildfire emergency. As the context of the questions is easy to understand and fire protection is part of insurance plans and education, we consider the crowd as representative respondents. In addition, we had a substantial number of responses, from people directly impacted by the Fort McMurray emergency.

Conclusion validity: The number of crowd workers (500) we used is considered large enough to exclude randomness. The reliability of the crowd responses is considered high as we excluded responses made in less than 20 seconds and eliminated subjects selecting the same answer for more than 90% of the questions.

Internal validity: By searching for specific hashtags, we have ensured that all tweets are connected to the Fort McMurray wildfire. We chose these hashtags through an initial investigation of this emergency.

Features extracted by MAPFEAT and features from existing apps can be imprecise as this is largely investigated in app store mining research. However, we selected the method [88] that were checked for sanity in several research studies. We would also argue that the crowdsourced evaluation mitigates the influence of this phenomenon.

MAPFEAT does not claim to find all features necessary in an emergency app. The completeness is dependent on data as well as the process. A needed feature in a wildfire emergency might not have been communicated in the tweets of Fort McMurray. Also, during the nine step process, there are several assumptions made about parameters. The number of words used for defining a topic could be done in a different way which potentially enlarges the search space. However, we do not argue for completeness, but just for correctness and added value. We also used crowdsourcing to back up the process validation.

External validity: We evaluated MAPFEAT through a comprehensive case study. From there, we do not claim generalization of results to other social media or to other events. We argue that the application of MAPFEAT could provide similar support in other situations as long as the tweet data set is representative of that event. Use of tweets for managing emergency events that involve

masses of people were proved to be informative.

8.8 Related work

Twitter analysis for emergency response

Social media services like Twitter have emerged as a popular medium for communication. Twitter data offers a rich mechanism for exploring events before, during, and after emergencies. Before an emergency, data can be monitored and analyzed to identify events [57]. Various research studies examined the use of Twitter data during emergency events [96]. Vieweg et al. [261] analyzed Twitter communication during the 2009 Red River Floods and Oklahoma grass fires. Hughes et al. [101] also analyzed online public communication with the police and fire departments during hurricane Sandy in USA. The study of Takahashi et al. [252] during Typhoon Haiyan in Philippines focused on the general public and organizational usage of Twitter. Also, Power et al. [209] developed a notification system to identify fire events in Australia.

Different data mining techniques, such as anomaly detection by clustering [40, 254] and topic modeling using textual data [56, 276], have been applied to tweets in order to bring insight during emergency events. Using classifiers to separate informative from non-informative tweets is also established in this context [168]. The use of Twitter data for managing emergencies is evolving rapidly for different types of events [96]. However, to the best of our knowledge this paper is the first study that focuses on requirements elicitation of Twitter data for mobile app mining.

Mining software features from App stores and product descriptions

App store mining and analysis has been widely studied in the context of software engineering [155]. Mining features from apps has been studied by analyzing the app description [78, 88], as described in this paper, or by mining app reviews [141]. Both approaches were proven to extract comprehensive yet incomplete sets of app features, as neither the description nor reviews mention

all features. Although, knowing this to be a threat, many research studies base their analysis and decision-making for app development on the features being mined from the app descriptions [155]. Looking into app features as a source for developing new ideas was also discussed by Maalej et al. [142].

On the other hand, mining the public description of software products to reduce effort and help the domain analysis process for tradition software products, such as Anti-Virus products has been discussed [87]. In the same context, a recommendation system for modeling product features in a specific domain have been designed and proposed [59]. MAPFEAT differs from these methods as it does not necessarily look into a specific domain, rather, it globally searches the app store.

Crowdsourcing in Software Engineering

The wisdom of the crowd is increasingly used across different disciplines of software engineering [149]. Crowdsourcing for the purpose of requirements elicitation and categorization has been used [241], and tweets themselves are a form of crowdsourced data [34]. We used tweet data from the crowd to elicit needs and map them into software requirements, as well as, verifying the mapping between tweet topics and the retrieved corresponding features. Crowdsourced human-assisted verification was first studied by Li et al. [171]. We also used crowdsourcing to validate the importance of features. Sustainability of crowdsourcing for software products was studied by [203] and mechanisms to enhance sustainability of crowsourced software were suggested in [203, 238].

8.9 Relevance and Applicability

Information about societal problems is typically distributed over different locations and systems. This is particularly true for information needed in various kinds of emergency situations. The challenge tackled by this paper is to define and execute a systematic process to combine different sources of data. Intrigued by the initial evidence that existing apps for handling wildfire emergency situations were lacking valuable functionality, we designed a process looking beyond contextual

boundaries and searched the app store holistically for the user needs.

The systematic method of understanding user needs and searching them to map it to the existing features of non-emergency app products has the potential for application beyond wildfire. The application of our proposed method relies on the:

- Mass impact of the event,
- Wide usage of mobile apps in the event context, and
- Usage of social media to stay connected, communicate, and share experience and opinions about the event.

The above conditions are prerequisite for applying MAPFEAT and are all satisfied in mass emergencies such as earthquakes, flooding and wildfires. Another common feature of emergency situations is the need to explore a wide range of dimensions by thinking "outside the box." This is difficult to accomplish solely in a proactive manner. Yet, MAPFEAT is not even limited to emergency situations and can be used for finding new features to provide better software support in day to day life of people addressing scenarios in the economic, political, environmental, social and technical aspects of society and urban life.

8.10 Conclusions

MAPFEAT is an automated method designed for the purpose of mapping general public needs that were communicated via Twitter into mobile app features. It uniquely combines different forms of crowdsourcing with existing techniques for classification, natural language processing, and topic modeling. The key idea is to map topics extracted from a set of event-related tweets to features that already exist in the apps apart from the app category or scope of functionality. This way, MAPFEAT enhances the design of an app by transferring people's stated needs in social media into technological functionality.

The feasibility of the method was demonstrated by a real-world case study. Based on the textual analysis of 69,680 unique tweets, we could specify a significant amount of new functionality (139 new features) demonstrated to be valuable for general public users in wildfire emergency events. The main conclusion of the case study is that MAPFEAT is able to find a significant number of additional features which were not in the currently existing wildfire apps. 87.1% of these features were confirmed useful by general public. The additional features that we elicited, was evaluated as clearly more comprehensive and more helpful functionality comparing to what was offered in the existing apps. For app providers, these findings are considered of substantial value to better support victims in future emergency events.

Acknowledgments

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada, NSERC Discovery Grant 250343-12 and Alberta Innovates Technology Futures.

Part V

Vision

Chapter 9

The Vision: Requirements Engineering in Society

Authors: Maleknaz Nayebi; Guenther Ruhe; and Christof Ebert RE 2017

Abstract - Industry and society are facing radical changes due to fast growing digital technologies and its ubiquity. Products and services will increasingly augment and integrate the real world with the digital world. This digital transformation has reached all business areas. Companies and consumers expect to obtain innovation, market penetration, cost reductions and more flexibility. We no longer are just fascinated by a RE technology without looking "behind the scenes". An increasing number of research studies look into impacts on society and industries. In the 21^{st} century, computational power, storage (memory), and communication capacity are in the hands of every online person in the society. While this brings the opportunity of using crowd and cloud computations, it also implies the responsibility of improving *the quality of life* in our society, and not limiting the discipline to address exclusively business-driven problems.
The relationship between RE and society is bi-directional. In this talk, we discuss the evolving role of RE by referring to a quarter century of impressive research. We discuss the increasing scope and responsibility of our discipline, serving as the bridge between the general public and technical teams and providing a response to the dramatic changes in our society. The vision is that we not only study possible options to perform requirements engineering in socio-technical systems, but get closer to make the results happen and evaluate their actual impact.

This paper has been published in the proceeding of International conference on Requirements Engineering (RE 2017) by ***Maleknaz Nayebi****,* ***Christof Ebert****, and* ***Guenther Ruhe*** [1].

[1]Authors are with SEDS lab at University of Calgary.

9.1 25 Years of RE: Reflecting a Changing World

Industry and society are facing radical changes due to fast growing digital technologies and its ubiquity [60, 61]. Products and services will increasingly augment and integrate the real world with the digital world. This digital transformation has reached all business areas. Companies and consumers expect to obtain innovation, market penetration, cost reductions and more flexibility.

The first Requirement Engineering (RE) paper in IEEE Software journal was published on verifying and validating software requirements by Barry Boehm in 1984 [31]. Starting from 1990, *requirements* attracted increasing attention and the term was very frequent among IEEE Software papers.

Besides quantity, the content and emphasis of RE papers have changed over years. We no longer are just fascinated by a RE technology without looking "behind the scenes". An increasing number of research studies look into impacts on society and industries [58]. In the 21^{st} century, computational power, storage (memory), and communication capacity are in the hands of every online person in the society. While this brings the opportunity of using crowd and cloud computations, it also implies the responsibility of improving *the quality of life* in our society, and not limiting the discipline to address exclusively business-driven problems.

The relationship between RE and society is bi-directional. In this talk, we discuss the evolving role of RE by referring to a quarter century of impressive research. We discuss the increasing scope and responsibility of our discipline, serving as the bridge between the general public and technical teams and providing a response to the dramatic changes in our society. The vision is that we not only study possible options to perform requirements engineering in socio-technical systems, but get closer to make the results happen and evaluate their actual impact.

9.2 Changing Requirements Decisions

RE is a decision-centric discipline [13] with consumers, users, organizations being key stakeholders in it. New technologies such as mobile and wearable devices create a satellite data. Analyzing user communities, forums [45], social media [84], and app stores [141], provides a broad range of information that supports all types of requirements decisions. Not considering this satellite data for requirement decision making makes our developed software applications less helpful and desirable for general public. Extracting information from tweets about a wildfire emergency situation showed that existing wildfire mobile apps cover only 15% of essential features requested by the general public [182].

What's being decided? Requirements engineering will remain a decision-centric process, but the way decisions are made, the information they rely upon, and the people and stakeholder involved in the process will further change [142]. Analyzing social media to elicit the software requirements is a contemporary example. MAPFEAT [182] is a method to automatically transfer general purpose tweets (for example about wildfire) into software features by searching for user needs in mobile app stores.

How the decisions are made? We are moving from intuition to evidence. Daneva et al. considered the pathway of empirical RE research analyzed from a series of EmpiRE workshops [50]. RE involves numerous decision problems that now should be extended toward involving the general public. There is increasing emphasis on evidence as studied in empirical RE. More powerful embedded systems and mobile devices provide situational and personal data from the general public [119]. These all enable software engineers to move from intuition to evidence. The analysis of app store reviews and social media is a prominent example of relying on extracted facts rather than gut feeling. In a recent study, we proved that information combined from social media and app stores provides essential and complementary support for RE decision making [180].

Who makes the decisions? The importance of user and stakeholder involvement for project success has been analyzed by various authors [1]. Crowdsourcing is increasingly being discussed

to enlarge the set of stakeholders and to elicit and manage requirements. Workshops like CrowdRE were designed to address the role of the crowd in RE. Social media and other communication channels allow almost unlimited access participating in the decision-making processes.

9.3 RE for Society: The Road Ahead

Software has a tremendous impact on society, and so does RE. Not understanding needs, markets and trends will ruin companies, but even entire countries, as we currently see when looking at the world map. Requirements engineering is the key lever to keep focus on what matters. We have asked decision makers worldwide to identify such trends [62]. In the sequel, we briefly outline selected trends, intended to show the breadth and diversity of society themes impacted by RE.

RE for safety-critical systems: There is no business if IT systems are perceived as insecure or even unsafe. Governments and companies are equally worried about today poor state of IT protection. Autonomous vehicles as well as connected medical systems are not trusted as long as there is no proven safety – which of course depends on cyber-security.

Security RE, and more widely speaking correctly addressing quality requirements is pivotal across industries [60, 62, 156].

RE for digital health and aging society: Traditional requirement analysis is now replaced by careful understanding of users in social media. 19% of Twitter users and 56% of internet users older than 65 years old use Facebook. Understanding and investigating on their attitude and stated needs in a form of social media status by mining social media over time will support decisions for technology design [69, 134].

RE for smart things and cities: The Internet of Things, cyber-physical systems, and the trend towards digitalization have become the main source of new business opportunities. Robots cooperate with human workers; high-speed trains are flexibly configured according to volatile mobility demands, and smart grids self-manage demand and response of energy. Requirements to such systems are very different from what we are used to in the – limited – worlds of Apps, IT

systems and embedded systems, as they connect these three areas. Future RE has to specify and model connectivity, distribution, flexibility, self-adaptation, and the usage of massive amounts of data [62, 64, 156].

RE for collaboration: Development and operations of large and ultra-large software-driven systems converges and needs a continuous RE. Continuous evolution of such systems demands modeling dependencies and risk on performance, safety, and availability. Failure is not an option for such systems. Resilience thus is a core requirement, often with decades of continuous operation. Collaborative tools facilitate the necessary flexibility. They also allow new eco-systems such as for pen innovation [62, 95].

RE for innovation: RE together with product management helps to balance cost and effort and thus maintain a sustainable business. Innovation drives all companies and social economy. RE provides the framework for innovative products and the innovation pipeline. A good example is the current convergence of IT with many traditional business models. This digital transformation is moving products to connected services [62, 95, 130].

9.4 Discussion

A little rebellion now and then is a good thing, as Thomas Jefferson once remarked. Do we need a new RE? Not to our point of view. But we need to better position RE at the center of all engineering disciplines and application domains. RE not only has to be addressed across study programs but also needs a stronger emphasis in companies and society. Times are gone when RE was only about specifications, tools, and modeling. RE in society creates new challenges along the value, human engagement and enabling processes.

This talk is intended to intrigue discussion about how RE as a discipline is impacted by the digital transformation – and how requirements engineering will help societies succeed in their digital innovations and transformations. Considering the changes in RE decision process we discuss (but not limit the discussion) to **value** (elicitation of value to evoke knowledge, user needs, and

business rationales), **human interactions** (visualization, usability and human factors in RE), **resilience** (business continuity and risk mitigation, e.g., in case of cyber-attacks), and **dependencies** (user experience across consistent services) for better RE for better society.

Bibliography

[1] Ulrike Abelein and Barbara Paech. Understanding the influence of user participation and involvement on system success–a systematic mapping study. *Empirical Software Engineering*, 20(1):28–81, 2015.

[2] Philip Achimugu, Ali Selamat, Roliana Ibrahim, and Mohd Naz'ri Mahrin. A systematic literature review of software requirements prioritization research. *Information and Software Technology*, 56(6):568–585, 2014.

[3] Bram Adams, Stephany Bellomo, Christian Bird, Tamara Marshall-Keim, Foutse Khomh, and Kim Moir. The practice and future of release engineering: A roundtable with three release engineers. *IEEE Software*, 32(2):42–49, 2015.

[4] Apoorv Agarwal, Boyi Xie, Ilia Vovsha, Owen Rambow, and Rebecca Passonneau. Sentiment analysis of twitter data. In *Proceedings of the workshop on languages in social media*, pages 30–38. Association for Computational Linguistics, 2011.

[5] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 3–14. IEEE, 1995.

[6] S. M. Didar Al Alam, Maleknaz Nayebi, Dietmar Pfahl, and Guenther Ruhe. A two-staged survey on release readiness. In *EASE*, page To appear. IEEE, 2017.

[7] Al Alam, SM Didar, Maleknaz Nayebi, Dietmar Pfahl, and Guenther Ruhe. A two-staged survey on release readiness. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 374–383. ACM, 2017.

[8] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 468–471. IEEE, 2016.

[9] D. Ameller, C. Farré, X. Franch, and G. Rufian. A survey on software release planning models. In *Proc. PROFES*. Springer LNCS, 2016.

[10] David Ameller, Carles Farré, Xavier Franch, and Guillem Rufian. A survey on software release planning models. In *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings 17*, pages 48–65. Springer, 2016.

[11] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[12] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings 33rd International Conference on Software Engineering*, pages 1–10. IEEE, 2011.

[13] Aybüke Aurum and Claes Wohlin. The fundamental nature of requirements engineering activities as a decision-making process. *Information and Software Technology*, 45(14):945–954, 2003.

[14] Aybüke Aurum and Claes Wohlin. A value-based approach in requirements engineering: explaining some of the fundamental concepts. In *Proceedings Conference on Requirements Engineering: Foundation for Software Quality*, pages 109–115. Springer, 2007.

[15] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 426–436. IEEE Press, 2015.

[16] Jay Ayres, Jason Flannick, Johannes Gehrke, and Tomi Yiu. Sequential pattern mining using a bitmap representation. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 429–435. ACM, 2002.

[17] Muhammad Imran Babar, Muhammad Ramzan, and Shahbaz AK Ghayyur. Challenges and future trends in software requirements prioritization. In *Proceedings Computer Networks and Information Technology*, pages 319–324. IEEE, 2011.

[18] Sebastian Barney, Ganglan Hu, Aybüke Aurum, and Claes Wohlin. Creating software product value in china. *IEEE Software*, 26(4):84–90, 2009.

[19] Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. The impact of api change-and fault-proneness on the user ratings of android apps. *IEEE Transactions on Software Engineering*, 41(4):384–407, 2015.

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

[21] Hans Christian Benestad and Jo E Hannay. A comparison of model-based and judgment-based release planning in incremental software projects. In *Proceedings 33rd International Conference on Software Engineering*, pages 766–775. ACM, 2011.

[22] Fabricio Benevenuto, Gabriel Magno, Tiago Rodrigues, and Virgilio Almeida. Detecting spammers on twitter. In *Collaboration, electronic messaging, anti-abuse and spam conference (CEAS)*, volume 6, page 12, 2010.

[23] Patrik Berander. Using students as subjects in requirements prioritization. In *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*, pages 167–176. IEEE, 2004.

[24] Patrik Berander. Evolving prioritization for software product management. *Doctoral thesis, Blekinge Institute of Technology*, pages 1–250, 2007.

[25] Patrik Berander and Anneliese Andrews. Requirements prioritization. *Engineering and managing software requirements*, 11(1):79–101, 2005.

[26] Charles Berger, Robert Blauth, David Boger, Christopher Bolster, Gary Burchill, William DuMouchel, Fred Pouliot, Reinhart Richter, Allan Rubinoff, Diane Shen, et al. Kano's methods for understanding customer-defined quality. *Center for Quality Management Journal*, 2(4):3–35, 1993.

[27] Christian Bird, Tim Menzies, and Thomas Zimmermann. *The Art and Science of Analyzing Software Data*. Elsevier, 2015.

[28] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14. ACM, 2011.

[29] Nicole J-M Blackman and John J Koval. Interval estimation for cohen's kappa as a measure of agreement. *Statistics in medicine*, 19(5):723–741, 2000.

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

[31] Barry W. Boehm. Verifying and validating software requirements and design specifications. *IEEE software*, 1(1):75, 1984.

[32] Rick Botta and A Terry Bahill. A prioritization process. *Engineering Management Journal*, 19(4):20–27, 2007.

[33] Gargi Bougie, Jamie Starke, Margaret-Anne Storey, and Daniel M German. Towards understanding twitter use in software engineering: preliminary findings, ongoing challenges and future questions. In *Proceedings of the 2nd international workshop on Web 2.0 for software engineering*, pages 31–36. ACM, 2011.

[34] Danah Boyd, Scott Golder, and Gilad Lotan. Tweet, tweet, retweet: Conversational aspects of retweeting on twitter. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, pages 1–10. IEEE, 2010.

[35] Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, 2013.

[36] Ernest R Cadotte and Normand Turgeon. Key factors in guest satisfaction. *Cornell Hotel and Restaurant Administration Quarterly*, 28(4):44–51, 1988.

[37] T Calinski and LCA Corsten. Clustering means in anova by simultaneous testing. *Biometrics*, pages 39–48, 1985.

[38] Nadina Carod and Alejandra Cechich. Requirements prioritization techniques. *Encyclopedia of Information Science and Technology*, pages 3283–3291, 2009.

[39] Nadina Martinez Carod and Alejandra Cechich. Cognitive profiles in understanding and prioritizing requirements: a case study. In *Proceedings Software Engineering Advances*, pages 341–346. IEEE, 2010.

[40] J. Chae, D. Thom, H. Bosch, Y. Jang, R. Maciejewski, D. S. Ebert, and T. Ertl. Spatiotemporal social media analytics for abnormal event detection and examination using seasonal-

trend decomposition. In *IEEE Conference on Visual Analytics Science and Technology 2012, VAST 2012 - Proceedings*, pages 143–152, 2012.

[41] Jonathan Chang, Sean Gerrish, Chong Wang, Jordan L Boyd-Graber, and David M Blei. Reading tea leaves: How humans interpret topic models. In *Advances in neural information processing systems*, pages 288–296, 2009.

[42] Ning Chen, Jialiu Lin, Steven CH Hoi, Xiaokui Xiao, and Boshen Zhang. Ar-miner: mining informative reviews for developers from mobile app marketplace. In *Proceedings of the 36th International Conference on Software Engineering*, pages 767–778. ACM, 2014.

[43] Adelina Ciurumelea, Andreas Schaufelbühl, Sebastiano Panichella, and Harald Gall. Analyzing reviews and code of mobile apps for better release planning. In *SANER*, volume 1, pages 12–22. IEEE, 2017.

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

[45] Jane Cleland-Huang, Horatiu Dumitru, Chuan Duan, and Carlos Castro-Herrera. Automated support for managing feature requests in open forums. *Communications of the ACM*, 52(10):68–74, 2009.

[46] João Clímaco, Carlos Ferreira, and M Eugénia Captivo. Multi-criteria integer programming: An overview of the different algorithmic approaches. In *Multi-criteria analysis*, pages 248–258. Springer, 1997.

[47] Louis Columbus. Roundup of mobile apps & app store forecasts, 2013. *Forbes*, 2013.

[48] Thomas H.. Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*, volume 6. MIT Press Cambridge, 2001.

[49] Amy Cravens. A demographic and business model analysis of today's app developer. *GigaOM Pro, September*, 2012.

[50] Maya Daneva, Daniela Damian, Alessandro Marchetto, and Oscar Pastor. Empirical research methodologies and studies in requirements engineering: How far did we come? *Journal of systems and software*, 95:1–9, 2014.

[51] Maya Daneva and Andrea Herrmann. Requirements prioritization based on benefit and cost prediction: A method classification framework. In *Proceedings 34th Euromicro Conference Software Engineering and Advanced Applications*, pages 240–247. IEEE, 2008.

[52] José del Sagrado, Isabel M del Águila, and Francisco J Orellana. Multi-objective ant colony optimization for requirements selection. *Empirical Software Engineering*, 20(3):577–610, 2015.

[53] Andrea Di Sorbo, Sebastiano Panichella, Carol V Alexandru, Junji Shimagaki, Corrado A Visaggio, Gerardo Canfora, and Harald C Gall. What would users change in my app? summarizing app reviews for recommending software changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 499–510. ACM, 2016.

[54] Andrea Di Sorbo, Sebastiano Panichella, Carol V Alexandru, Corrado A Visaggio, and Gerardo Canfora. Surf: Summarizer of user reviews feedback. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 55–58. IEEE Press, 2017.

[55] Yi Ding and Kah Hin Chai. Emotions and continued usage of mobile applications. *Industrial Management & Data Systems*, 115(5):833–852, 2015.

[56] Zhang Dong and Xie Zhicheng. Analysis and research on microblogging network model based on crawler data. In *Computer Science and Network Technology (ICCSNT), 2011 International Conference on*, volume 2, pages 653–656. IEEE, 2011.

[57] W. Dou, X. Wang, D. Skau, W. Ribarsky, and M. X. Zhou. Leadline: Interactive visual analysis of text data through event identification and exploration. In *IEEE Conference on Visual Analytics Science and Technology 2012, VAST 2012 - Proceedings*, pages 93–102, 2012.

[58] C.H.C. Duarte and T. Gorschek. Technology transfer - requirements engineering research to industrial practice: An open (ended) debate. *RE 2015*, pages 414–415, 2015.

[59] Horatiu Dumitru, Marek Gibiec, Negar Hariri, Jane Cleland-Huang, Bamshad Mobasher, Carlos Castro-Herrera, and Mehdi Mirakhorli. On-demand feature recommendations derived from mining public product descriptions. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 181–190. IEEE, 2011.

[60] Christof Ebert. Looking into the future. *IEEE Software*, 32(6):92–97, 2015.

[61] Christof Ebert and Carlos Henrique C Duarte. Requirements engineering for the digital transformation: Industry panel. In *Requirements Engineering Conference (RE), 2016 IEEE 24th International*, pages 4–5. IEEE, 2016.

[62] Christof Ebert and Kris Shankar. Industry trends 2017. *IEEE Software*, 34(2):112–116, 2017.

[63] Thomas R Eisenmann. Platform-mediated networks: definitions and core concepts. In *Harvard Business School Module Note 807-049*, 2006.

[64] J.-P. Exner. The espresso-project-a european approach for smart city standards. *Lecture Notes in Computer Science*, 9788:483–490, 2016.

[65] Thomas M Fehlmann. New lanchester theory for requirements prioritization. In *Second International Workshop on Software Product Management*, pages 35–40. IEEE, 2008.

[66] D Mendez Fernandez, Stefan Wagner, Marcos Kalinowski, Michael Felderer, Priscilla Mafra, Antonio Vetrò, Tayana Conte, M-T Christiansson, Desmond Greer, Casper Lasse-

nius, et al. Naming the pain in requirements engineering. *Empirical Software Engineering*, pages 1–41, 2016.

[67] Donald Firesmith. Prioritizing requirements. *Journal of Object Technology*, 3(8):35–48, 2004.

[68] Soroush Forouzani, Rodina Ahmad, Sepehr Forouzani, and Nasim Gazerani. Design of a teaching framework for software requirement prioritization. In *Proceedings Computing Technology and Information Management*, volume 2, pages 787–793. IEEE, 2012.

[69] S.A. Fricker, C. ThÃ¼mmler, and A. Gavras. *Requirements engineering for digital health*. Springer, 2015.

[70] Johann Füller and Kurt Matzler. Customer delight and market segmentation: An application of the three-factor theory of customer satisfaction on life style groups. *Tourism management*, 29(1):116–126, 2008.

[71] Minos N Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Spirit: Sequential pattern mining with regular expression constraints. In *VLDB*, volume 99, pages 7–10, 1999.

[72] Michael R Gary and David S Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman and Company, New York, 1979.

[73] Vibha Gaur, Anuja Soni, and Punam Bedi. An agent-oriented approach to requirements engineering. In *Proceedings Advance Computing Conference*, pages 449–454. IEEE, 2010.

[74] Jean Dickinson Gibbons and Subhabrata Chakraborti. *Nonparametric statistical inference*. Springer, 2011.

[75] Tudor Girba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 113–122. IEEE, 2005.

[76] Robert L. Glass, Iris Vessey, and Venkataraman Ramesh. Research in software engineering: an analysis of the literature. *Information and Software technology*, 44(8):491–506, 2002.

[77] Maria Gomez, Matias Martinez, Martin Monperrus, and Romain Rouvoy. When app stores listen to the crowd to fight bugs in the wild. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)-Volume 2*, pages 567–570. IEEE Press, 2015.

[78] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1025–1035. ACM, 2014.

[79] Des Greer and Guenther Ruhe. Software release planning: an evolutionary and iterative approach. *Information and software technology*, 46(4):243–253, 2004.

[80] Xiaodong Gu and Sunghun Kim. " what parts of your apps are loved by users?"(t). In *Automated Software Engineering (ASE)*, pages 760–770. IEEE, 2015.

[81] Renu Gupta and M. C. Puri. Bicriteria integer quadratic programming problems. *Journal of Interdisciplinary Mathematics*, 3(2-3):133–148, 2000.

[82] Gurobi. Gurobi optimizer reference manual, 2012.

[83] Emitza Guzman, Rana Alkadhi, and Norbert Seyff. A needle in a haystack: What do twitter users say about software? In *Requirements Engineering Conference (RE)*, pages 96–105. IEEE, 2016.

[84] Emitza Guzman and Walid Maalej. How do users like this feature? a fine grained sentiment analysis of app reviews. In *IEEE 22nd international requirements engineering conference (RE)*, pages 153–162. IEEE, 2014.

[85] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Freespan: frequent pattern-projected sequential pattern mining. In *Proceedings*

of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, pages 355–359. ACM, 2000.

[86] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and MC Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *proceedings of the 17th international conference on data engineering*, pages 215–224, 2001.

[87] Negar Hariri, Carlos Castro-Herrera, Mehdi Mirakhorli, Jane Cleland-Huang, and Bamshad Mobasher. Supporting domain analysis through mining and recommending features from online product listings. *IEEE Transactions on Software Engineering*, 39(12):1736–1752, 2013.

[88] Mark Harman, Yue Jia, and Yuanyuan Zhang. App store mining and analysis: Msr for app stores. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 108–111. IEEE Press, 2012.

[89] Ahmed E Hassan and Tao Xie. Software intelligence: the future of mining software engineering data. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 161–166. ACM, 2010.

[90] Ed Heierman, M Youngblood, and Diane J Cook. Mining temporal sequences to discover interesting patterns. In *KDD Workshop on mining temporal and sequential data*, pages 104–111. Citeseer, 2004.

[91] Ashkan Hemmati, Chad Saunders, Chris Carlson, Guenther Ruhe, and Maleknaz Nayebi. Analysis of software service usage in health care communication services,. In *IEEE International Conference on Software Security and Reliability*, page in press. IEEE, 2017.

[92] Kim Herzig and Andreas Zeller. Mining cause-effect-chains from version histories. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pages 60–69. IEEE, 2011.

[93] Abram Hindle, Daniel M German, and Ric Holt. What do large commits tell us?: a taxonomical study of large commits. In *MSR*, pages 99–108. ACM, 2008.

[94] Liangjie Hong and Brian D Davison. Empirical study of topic modeling in twitter. In *Proceedings of the first workshop on social media analytics*, pages 80–88. ACM, 2010.

[95] J. Horkoff, N. Maiden, and J. Lockerbie. Creativity and goal modeling for software requirements engineering. *Conference on Creativity and Cognition*, pages 165–168, 2015.

[96] J. Brian Houston, Joshua Hawthorne, Mildred F. Perreault, Eun Hae Park, Marlo Goldstein Hode, Michael R. Halliwell, Sarah E. Turner McGowen, Rachel Davis, Shivani Vaid, Jonathan A. McElderry, and Stanford A. Griffith. Social media and disasters: a functional framework for social media use in disaster planning, response, and research. *Disasters*, 39(1):1–22, 2015.

[97] Chun-Hua Hsiao, Jung-Jung Chang, and Kai-Yu Tang. Exploring the influential factors in continuance usage of mobile social apps: Satisfaction, habit, and customer value perspectives. *Telematics and Informatics*, 33(2):342–355, 2016.

[98] Chin-Lung Hsu and Judy Chuan-Chuan Lin. What drives purchase intention for paid mobile apps?–an expectation confirmation model with perceived value. *Electronic Commerce Research and Applications*, 14(1):46–57, 2015.

[99] Ganglan Hu, Aybüke Aurum, and Claes Wohlin. Adding value to software requirements: An empirical study in the chinese software industry. In *Proceedings 17th Australasian Conference on Information Systems*, page Paper 7, 2006.

[100] William Hudson. Card sorting. *The Encyclopedia of Human-Computer Interaction, 2nd Ed.*, 2013.

[101] Amanda L Hughes, Lise AA St Denis, Leysia Palen, and Kenneth M Anderson. Online public communications by police & fire services during the 2012 hurricane sandy. In *Pro-*

ceedings of the 32nd annual ACM conference on Human factors in computing systems, pages 1505–1514. ACM, 2014.

[102] Amanda Lee Hughes and Leysia Palen. Twitter adoption and use in mass convergence and emergency events. *International Journal of Emergency Management*, 6(3-4):248–260, 2009.

[103] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010.

[104] Clayton J Hutto and Eric Gilbert. Vader: A parsimonious rule-based model for sentiment analysis of social media text. In *Eighth international AAAI conference on weblogs and social media*, pages 50–60, 2014.

[105] Claudia Iacob and Rachel Harrison. Retrieving and analyzing mobile apps feature requests from online reviews. In *Mining Software Repositories (MSR), 10th IEEE Working Conference on*, pages 41–44. IEEE, 2013.

[106] Anjali Ganesh Jivani et al. A comparative study of stemming algorithms. *Int. J. Comp. Tech. Appl*, 2(6):1930–1938, 2011.

[107] Keith Johnson. *The Future of Video Content Convergence: Consumer Engagement Opportunities in Multi-platform Video and Over-the-top TV*. Business Insights Limited, 2010.

[108] Robbert Jongeling, Subhajit Datta, and Alexander Serebrenik. Choosing your weapons: On sentiment analysis tools for software engineering research. In *Software maintenance and evolution (ICSME)*, pages 531–535. IEEE, 2015.

[109] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. Real challenges in mobile app development. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 15–24. IEEE, 2013.

[110] Noriaki Kano, Nobuhiku Seraku, Fumio Takahashi, and Shinichi Tsuji. Attractive quality and must-be quality. *Journal of the Japanese Society for Quality Control*, 14(2):39–48, 1984.

[111] Muhammad Rezaul Karim and Guenther Ruhe. Bi-objective genetic search for release planning in support of themes. In *Proceedings Symposium on Search Based Software Engineering*, pages 123–137. Springer, 2014.

[112] Joachim Karlsson. Software requirements prioritizing. In *Proceedings 2nd Conference on Requirements Engineering*, pages 110–116. IEEE, 1996.

[113] Noureddine Kerzazi and Foutse Khomh. Factors impacting rapid releases: an industrial case study. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 61. ACM, 2014.

[114] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E Hassan. What do mobile app users complain about? *IEEE Software*, 32(3):70–77, 2015.

[115] Foutse Khomh, Tejinder Dhaliwal, Ying Zou, and Bram Adams. Do faster releases improve software quality? an empirical case study of mozilla firefox. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 179–188. IEEE, 2012.

[116] Mahvish Khurum, Tony Gorschek, and Magnus Wilson. The software value map—an exhaustive collection of value aspects for the development of software intensive products. *Journal of Software: Evolution and Process*, 25(7):711–741, 2013.

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

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

[119] Barbara A Kitchenham, Tore Dyba, and Magne Jorgensen. Evidence-based software engineering. In *ICSE*, pages 273–281. IEEE Computer Society, 2004.

[120] Barbara A Kitchenham and Shari Lawrence Pfleeger. Principles of survey research: part 3: constructing a survey instrument. *ACM SIGSOFT Software Engineering Notes*, 27(2):20–24, 2002.

[121] Barbara A Kitchenham, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on software engineering*, 28(8):721–734, 2002.

[122] Barbara Ann Kitchenham, David Budgen, and Pearl Brereton. *Evidence-Based Software engineering and systematic reviews*, volume 4. CRC Press, 2015.

[123] Aniket Kittur, Ed H Chi, and Bongwon Suh. Crowdsourcing user studies with mechanical turk. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 453–456. ACM, 2008.

[124] S. B. Kotsiantis. Supervised machine learning: A review of classification techniques. In *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*, pages 3–24, Amsterdam, The Netherlands, The Netherlands, 2007. IOS Press.

[125] Efthymios Kouloumpis, Theresa Wilson, and Johanna D Moore. Twitter sentiment analysis: The good the bad and the omg! *Icwsm*, 11:538–541, 2011.

[126] Nupul Kukreja. Decision theoretic requirements prioritization: a two-step approach for sliding towards value realization. In *Proceedings Conference on Software Engineering*, pages 1465–1467. IEEE Press, 2013.

[127] Kathy Lee, Diana Palsetia, Ramanathan Narayanan, Md Mostofa Ali Patwary, Ankit Agrawal, and Alok Choudhary. Twitter trending topic classification. In *2011 IEEE 11th International Conference on Data Mining Workshops*, pages 251–258. IEEE, 2011.

[128] Yu-Cheng Lee and Sheng-Yen Huang. A new fuzzy concept approach for Kano's model. *Expert Systems with Applications*, 36(3):4479–4484, 2009.

[129] Laura Lehtola and Marjo Kauppinen. Suitability of requirements prioritization methods for market-driven software product development. *Software Process: Improvement and Practice*, 11(1):7–19, 2006.

[130] Johan Linåker, Björn Regnell, and Hussan Munir. Requirements engineering in open innovation: a research agenda. In *Proceedings of the 2015 International Conference on Software and System Process*, pages 208–212. ACM, 2015.

[131] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. Api change and fault proneness: A threat to the success of android apps. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 477–487. ACM, 2013.

[132] Mario Linares-Vásquez, Andrew Holtzhauer, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Revisiting android reuse studies in the context of code obfuscation and library usages. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 242–251. ACM, 2014.

[133] Bing Liu. Sentiment analysis and subjectivity. *Handbook of natural language processing*, 2:627–666, 2010.

[134] L. Liu, L. Feng, Z. Cao, and J. Li. Requirements engineering for health data analytics: Challenges and possible directions. *RE*, pages 266–275, 2016.

[135] Xiaoqing Liu, Chandra Sekhar Veera, Yan Sun, Kunio Noguchi, and Yuji Kyoya. Priority assessment of software requirements from multiple perspectives. In *Proceedings 28th COMPSAC Conference*, pages 410–415. IEEE, 2004.

[136] Xiaoqing Frank Liu, Yan Sun, Chandra Sekhar Veera, Yuji Kyoya, and Kunio Noguchi. Priority assessment of software process requirements from multiple perspectives. *Journal of Systems and Software*, 79(11):1649–1660, 2006.

[137] David Lo, Siau-Cheng Khoo, and Chao Liu. Efficient mining of iterative patterns for software specification discovery. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 460–469. ACM, 2007.

[138] Kevin Logue and Kevin McDaid. Handling uncertainty in agile requirement prioritization and scheduling using statistical simulation. In *Proceedings Agile Conference*, pages 73–82. IEEE, 2008.

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

[140] Tainyi Luor, Hsi-Peng Lu, Kang-Min Chien, and Tzong-Chen Wu. Contribution to quality research: A literature review of Kano's model from 1998 to 2012. *Total Quality Management & Business Excellence*, 26(3-4):234–247, 2015.

[141] Walid Maalej and Hadeer Nabil. Bug report, feature request, or simply praise? on automatically classifying app reviews. In *IEEE 23rd international requirements engineering conference (RE)*, pages 116–125. IEEE, 2015.

[142] Walid Maalej, Maleknaz Nayebi, Timo Johann, and Guenther Ruhe. Toward data-driven requirements engineering. *Software, IEEE*, 33(1):48–54, 2016.

[143] Nizar R Mabroukeh and Christie I Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys (CSUR)*, 43(1):3, 2010.

[144] Donald G Malcolm, John H Roseboom, Charles E Clark, and Willard Fazar. Application of a technique for research and development program evaluation. *Operations Research*, 7(5):646–669, 1959.

[145] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *ACL (System Demonstrations)*, pages 55–60, 2014.

[146] Usman Mansoor, Marouane Kessentini, Philip Langer, Manuel Wimmer, Slim Bechikh, and Kalyanmoy Deb. MOMM: Multi-objective model merging. *Journal of Systems and Software*, 103:423–439, 2015.

[147] Usman Mansoor, Marouane Kessentini, Manuel Wimmer, and Kalyanmoy Deb. Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm. *Software Quality Journal*, pages 1–29, 2017.

[148] Mika V Mäntylä, Bram Adams, Foutse Khomh, Emelie Engström, and Kai Petersen. On rapid releases and software testing: a case study and a semi-systematic literature review. *Empirical Software Engineering*, 20(5):1384–1425, 2015.

[149] Ke Mao, Licia Capra, Mark Harman, and Yue Jia. A survey of the use of crowdsourcing in software engineering. *RN*, 15(01), 2015.

[150] Joaquín Alegre Marin and Jaume Garau Taberner. Satisfaction and dissatisfaction with destination attributes: Influence on overall satisfaction and the intention to return. *Retrieved December*, 18:2011, 2008.

[151] Rob Markey, Fred Reichheld, and Andreas Dullweber. Closing the customer feedback loop. *Harvard Business Review*, 87(12):43–47, 2009.

[152] Matthias Marschall. Transforming a six month release cycle to continuous flow. In *Agile Conference (AGILE), 2007*, pages 395–400. IEEE, 2007.

[153] William Martin, Mark Harman, Yue Jia, Federica Sarro, and Yuanyuan Zhang. The app sampling problem for app store mining. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 123–133. IEEE, 2015.

[154] William Martin, Federica Sarro, and Mark Harman. Causal impact analysis for app releases in google play. In *FSE*, pages 435–446. ACM, 2016.

[155] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. *RN*, 16:02, 2016.

[156] L.E.G. Martins and T. Gorschek. Requirements engineering for safety-critical systems: A systematic literature review. *Information and Software Technology*, 75:71–89, 2016.

[157] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 131–140. IEEE, 2009.

[158] Stuart McIlroy, Nasir Ali, and Ahmed E Hassan. Fresh apps: an empirical study of frequently-updated mobile apps in the google play store. *Empirical Software Engineering*, 21(3):1346–1370, 2016.

[159] Tim Menzies, Ekrem Kocaguneli, Burak Turhan, Leandro Minku, and Fayola Peters. *Sharing data and models in software engineering*. Morgan Kaufmann, 2014.

[160] Tim Menzies and Thomas Zimmermann. Software analytics: so what? *IEEE Software*, 30(4):31–37, 2013.

[161] Amir Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 22nd international conference on Software engineering*, pages 167–176. ACM, 2000.

[162] Martin Michlmayr, Brian Fitzgerald, and Klaas-Jan Stol. Why and how should open source projects adopt time-based releases? *IEEE Software*, 32(2):55–63, 2015.

[163] Josip Mikulic and Darko Prebezac. A critical review of techniques for classifying quality attributes in the Kano model. *Managing Service Quality: An International Journal*, 21(1):46–66, 2011.

[164] Leandro Lei Minku. The wisdom of the crowds in software engineering predictive modelling. *Perspectives on Data Science for Software Engineering*, 2016.

[165] Samer I Mohamed, Islam AM El-Maddah, and Ayman M Wahba. Criteria-based requirements prioritization for software product management. In *Proceedings Software Engineering Research and Practice*, pages 587–593, 2008.

[166] Saif M Mohammad, Svetlana Kiritchenko, and Xiaodan Zhu. Nrc-canada: Building the state-of-the-art in sentiment analysis of tweets. *arXiv preprint arXiv:1308.6242*, 2013.

[167] Israel J Mojica, Bram Adams, Meiyappan Nagappan, Steffen Dienst, Thorsten Berger, and Ahmed E Hassan. A large-scale empirical study on software reuse in mobile apps. *IEEE software*, 31(2):78–86, 2014.

[168] Sung Pil Moon, Yikun Liu, Steven O Entezari, Afarin Pirzadeh, Andrew Pappas, and Mark S Pfaff. Top health trends: An information visualization tool for awareness of local health trends. In *ISCRAM*, pages 177–187, 2013.

[169] Carl H Mooney and John F Roddick. Sequential pattern mining–approaches and algorithms. *ACM Computing Surveys (CSUR)*, 45(2):19, 2013.

[170] Brendan Murphy, Jacek Czerwonka, and Laurie Williams. Branching taxonomy. *Microsoft Research, Cambridge*, 2014.

[171] Robert Musson, Jacqueline Richards, Danyel Fisher, Christian Bird, Brian Bussone, and Sandipan Ganguly. Leveraging the crowd: how 48,000 users helped improve lync performance. *IEEE software*, 30(4):38–45, 2013.

[172] Mor Naaman, Jeffrey Boase, and Chih-Hui Lai. Is it really about me?: message content in social awareness streams. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, pages 189–192. ACM, 2010.

[173] M Nayebi and G Ruhe. Analytical product release planning. pages 550–580. Morgan Kaufmann, 2015.

[174] Maleknaz Nayebi, Bram Adams, and Guenther Ruhe. Release practices for mobile apps–what do users and developers think? In *Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 552–562. IEEE, 2016.

[175] Maleknaz Nayebi, Bram Adams, and Guenther Ruhe. From perception to evidence: Release patterns in mobile app stores. In *Emprical Software Engineering Journal (EMSE)*, page Under submission. Springer, 2017.

[176] Maleknaz Nayebi, Henry Cho, and Guenther Ruhe. App store mining is not enough. In *Emprical Software Engineering Journal (EMSE)*, page Under submission. Springer, 2017.

[177] Maleknaz Nayebi, Homayoon Farrahi, Ada Lee, Henry Cho, and Guenther Ruhe. More insight from being more focused: analysis of clustered market apps. In *WAMA*, pages 30–36. ACM, 2016.

[178] Maleknaz Nayebi, Homayoon Farrahi, and Guenther Ruhe. Analysis of marketed versus not-marketed mobile app releases. In *RELENG 2016*, pages 1–4. ACM, 2016.

[179] Maleknaz Nayebi, Homayoon Farrahi, and Guenther Ruhe. Which app should be released to the market? In *Emprical Software Engineering and Measurement (ESEM)*, page in press. IEEE, 2017.

[180] Maleknaz Nayebi, Homayoon Farrahi, Guenther Ruhe, and Henry Cho. App store mining is not enough. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 152–154. IEEE Press, 2017.

[181] Maleknaz Nayebi, Rachel Quapp, Guenther Ruhe, Mahshid Marbouti, and Frank Maurer. Crowdsourced exploration of mobile app features: a case study of the fort mcmurray wildfire. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Society Track*, pages 57–66. IEEE Press, 2017.

[182] Maleknaz Nayebi, Rachel Quapp, Guenther Ruhe, Mahshid Marbouti, and Frank Maurer. Crowdsourced exploration of mobile app features: a case study of the fort mcmurray wildfire. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Society Track*, pages 57–66. IEEE Press, 2017.

[183] Maleknaz Nayebi and Guenther Ruhe. An open innovation approach towards feature elicitation for product release planning. In *The Consortium for Software Engineering Research (CSER)*, page poster, 2013.

[184] Maleknaz Nayebi and Guenther Ruhe. Analytical open innovation for value-optimized service portfolio planning. In *International Conference of Software Business*, pages 273–288. Springer, 2014.

[185] Maleknaz Nayebi and Guenther Ruhe. An open innovation approach in support of product release decisions. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 64–71. ACM, 2014.

[186] Maleknaz Nayebi and Guenther Ruhe. Analytical product release planning. In *The Art and Science of Analyzing Software Data*, pages 550–580. Morgan Kaufmann, 2015.

[187] Maleknaz Nayebi and Guenther Ruhe. Asymmetric release planning, compromising customers' satisfaction and dissatisfaction. In *Transaction of Software Engineering (TSE)*, page Under submission. IEEE, 2017.

[188] Maleknaz Nayebi and Guenther Ruhe. Optimized functionality for super mobile apps. In *Requirements Engineering Conference (RE)*, pages 388–393. IEEE, 2017.

[189] Maleknaz Nayebi, Guenther Ruhe, and Thomas Zimmermann. Mining treatment-outcome constructs from sequential software data. In *Transaction of Software Engineering (TSE)*, page Under submission. IEEE, 2017.

[190] Maleknaz Nayebi, Kabir S. Jeeshan, Guenther Ruhe, Chris Carlson, and Francis Chew. Hybrid labels are the new measure. *IEEE Software*, 44(1):to appear, 2017.

[191] Maleknaz Nayebi and Krzysztof Wnuk. Summary of the 1st international workshop on open innovation in software engineering (oise 2015). In *Proceedings of the 2015 International Conference on Software and System Process*, pages 183–184. ACM, 2015.

[192] An Ngo-The and Guenther Ruhe. A systematic approach for solving the wicked problem of software release planning. *Soft Computing*, 12(1):95–108, 2008.

[193] Finn Årup Nielsen. A new anew: Evaluation of a word list for sentiment analysis in microblogs. *arXiv preprint arXiv:1103.2903*, 2011.

[194] Toshiaki Nishihara and Yasuhiko Minamide. Depth first search. *The Archive of Formal Proofs. http://afp. sf. net/entries/Depth-First-Search. shtml*, 2004.

[195] Brendan O'Connor, Michel Krieger, and David Ahn. Tweetmotif: Exploratory search and topic summarization for twitter. In *ICWSM*, 2010.

[196] Carlos E Otero, Erica Dell, Abrar Qureshi, and Luis D Otero. A quality-based requirement prioritization framework using binary inputs. In *Proceedings 2010 Fourth Asia International Conference on Mathematical/Analytical Modelling and Computer Simulation*, pages 187–192. IEEE, 2010.

[197] Fabio Palomba, Mario Linares-Vásquez, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. User reviews matter! tracking crowd-

sourced reviews to support evolution of successful apps. In *Software Maintenance and Evolution (ICSME)*, pages 291–300. IEEE, 2015.

[198] Gabriele Paolacci, Jesse Chandler, and Panagiotis G Ipeirotis. Running experiments on amazon mechanical turk. *Judgment and Decision making*, 5(5):411–419, 2010.

[199] Manan Parikh, Bharat Chaudhari, and Chetna Chand. A comparative study of sequential pattern mining algorithms. *International Journal of Application or Innovation in Engineering and Management*, 2(2):103–109, 2013.

[200] Thanasis Petsas, Antonis Papadogiannakis, Michalis Polychronakis, Evangelos P Markatos, and Thomas Karagiannis. Rise of the planet of the apps: A systematic study of the mobile app ecosystem. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 277–290. ACM, 2013.

[201] Shari Lawrence Pfleeger and Barbara A Kitchenham. Principles of survey research: part 1: turning lemons into lemonade. *ACM SIGSOFT Software Engineering Notes*, 26(6):16–18, 2001.

[202] Shari Lawrence Pfleeger and Barbara A Kitchenham. Principles of survey research part 2: designing a survey. *Software Engineering Notes*, 27(1):18–20, 2002.

[203] Dennis Pilz and Heiko Gewald. Does money matter? motivational factors for participation in paid-and non-profit-crowdsourcing communities. In *Wirtschaftsinformatik*, page 37, 2013.

[204] Antônio Mauricio Pitangueira, Rita Suzana P Maciel, Márcio de Oliveira Barros, and Aline Santos Andrade. A systematic review of software requirements selection and prioritization using sbse approaches. In *Proceedings Symposium on Search Based Software Engineering*, pages 188–208. Springer, 2013.

[205] Antonio Mauricio Pitangueira, Paolo Tonella, Angelo Susi, Rita Suzana Maciel, and Marcio Barros. Risk-aware multi-stakeholder next release planning using multi-objective optimiza-

tion. In *Proceedings Conference on Requirements Engineering: Foundation for Software Quality*, pages 3–18. Springer, 2016.

[206] Project Management Institute (PMI). *MSoftware Extension to the PMBOK Guide*. IEEE Computer Scociety, 2013.

[207] Daniel Port and Joel Wilf. The value of certifying software release readiness: an exploratory study of certification for a critical system at jpl. In *ESEM*, pages 373–382. IEEE, 2013.

[208] Martin F Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[209] Robert Power, Bella Robinson, and David Ratcliffe. Finding fires with twitter. In *Australasian language technology association workshop*, volume 80, pages 80–89, 2013.

[210] Philips K Prasetyo, David Lo, Palakorn Achananuparp, Yuan Tian, and Ee-Peng Lim. Automatic classification of software related microblogs. In *Software Maintenance (ICSM)*, pages 596–599. IEEE, 2012.

[211] Zornitza Racheva, Maya Daneva, Klaas Sikkel, Andrea Herrmann, and Roel Wieringa. Do we know enough about requirements prioritization in agile projects: insights from a case study. In *Proceedings 18th Requirements Engineering Conference*, pages 147–156. IEEE, 2010.

[212] H Raharjo, M Xie, and AC Brombacher. Prioritizing quality characteristics in dynamic quality function deployment. *International Journal of Production Research*, 44(23):5005–5018, 2006.

[213] Daniel Ramage, Susan T Dumais, and Daniel J Liebling. Characterizing microblogs with topic models. *ICWSM*, 10:1–1, 2010.

[214] Aurora Ramírez, José Raúl Romero, and Sebastián Ventura. A comparative study of many-objective evolutionary algorithms for the discovery of software architectures. *Empirical Software Engineering*, pages 1–55, 2015.

[215] D Randall Brandt. How service marketers can identify value-enhancing service elements. *Journal of Services Marketing*, 2(3):35–41, 1988.

[216] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165. ACM, 2014.

[217] Björn Regnell, Richard Berntsson Svensson, and Thomas Olsson. Supporting roadmapping of quality requirements. *IEEE Software*, 25(2):42–47, 2008.

[218] Maria Riaz, Travis Breaux, and Laurie Williams. How have we evaluated software pattern application? a systematic mapping study of research design practices. *Information and Software Technology*, 65:14–38, 2015.

[219] Michael M Richter and Rosina Weber. *Case-based reasoning: a textbook*. Springer Science & Business Media, 2013.

[220] Norman Riegel and Joerg Doerr. A systematic literature review of requirements prioritization criteria. In *Proceedings Conference on Requirements Engineering: Foundation for Software Quality*, pages 300–317. Springer, 2015.

[221] Peter C Rigby, Daniel M German, Laura Cowen, and Margaret-Anne Storey. Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):35, 2014.

[222] Peter C Rigby, Daniel M German, and Margaret-Anne Storey. Open source software peer review practices: a case study of the apache server. In *Proceedings of the 30th international conference on Software engineering*, pages 541–550. ACM, 2008.

[223] James Robertson and Suzanne Robertson. Volere. *Requirements Specification Templates*, 2000.

[224] Suzanne Robertson and James Robertson. *Mastering the requirements process: Getting requirements right*. Addison-Wesley, 2012.

[225] Sara Rosenthal, Preslav Nakov, Svetlana Kiritchenko, Saif M Mohammad, Alan Ritter, and Veselin Stoyanov. Semeval-2015 task 10: Sentiment analysis in twitter. *Proceedings of SemEval-2015*, 2015.

[226] Guenther Ruhe and Maleknaz Nayebi. Twhat counts is decisions, not numbers – towards an analytics design sheet. *Perspectives on Data Science for Software Engineering*, 1(1):48–54, 2016.

[227] Guenther Ruhe, Maleknaz Nayebi, and Christoph Ebert. The vision: Requirements engineering in society. In *Requirements Engineering Conference (RE)*, pages 478–479. IEEE, 2017.

[228] Günther Ruhe. *Product release planning: methods, tools and applications*. CRC Press, 2010.

[229] Günther Ruhe and Omolade Saliu. Art and science of system release planning. In *International Conference on Product Focused Software Process Improvement*, pages 458–461. Springer, 2006.

[230] Günther Ruhe and Claes Wohlin. *Software Project Management in a Changing World*. Springer, 2014.

[231] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131, 2009.

[232] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.

[233] Ioana Rus and Mikael Lindvall. Knowledge management in software engineering. *IEEE software*, 19(3):26, 2002.

[234] Thomas L Saaty. *Fundamentals of decision making and priority theory with the analytic hierarchy process*, volume 6. RWS Publications, 2000.

[235] Omolade Saliu and Guenther Ruhe. Supporting software release planning decisions for evolving systems. In *29th Annual IEEE/NASA Software Engineering Workshop*, pages 14–26. IEEE, 2005.

[236] Federica Sarro, Afnan A Al-Subaihin, Mark Harman, Yue Jia, William Martin, and Yuanyuan Zhang. Feature lifecycles as they spread, migrate, remain, and die in app stores. In *2015 IEEE 23rd International Requirements Engineering Conference (RE)*, pages 76–85. IEEE, 2015.

[237] Elmar Sauerwein, Franz Bailom, Kurt Matzler, and Hans H Hinterhuber. The kano model: How to delight your customers. *International Working Seminar on Production Economics*, 1(4):313–327, 1996.

[238] Eric Schenk and Claude Guittard. Crowdsourcing: What can be outsourced to the crowd, and why. In *Workshop on Open Source Innovation, Strasbourg, France*, page 72, 2009.

[239] ANDREW JHON Scott and M Knott. A cluster analysis method for grouping means in the analysis of variance. *Biometrics*, pages 507–512, 1974.

[240] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

[241] Norbert Seyff, Florian Graf, and Neil Maiden. Using mobile re tools to give end-users their own voice. In *2010 18th IEEE International Requirements Engineering Conference*, pages 37–46. IEEE, 2010.

[242] Norbert Seyff, Irina Todoran, Kevin Caluser, Leif Singer, and Martin Glinz. Using popular social network sites to support requirements elicitation, prioritization and negotiation. *Journal of Internet Services and Applications*, 6(1):1, 2015.

[243] SM Shahnewaz and Guenther Ruhe. Relrea-an analytical approach for evaluating release readiness. In *SEKE*, pages 437–442, 2014.

[244] Martin Shepperd. Case-based reasoning and software engineering. In *Managing Software Engineering Knowledge*, pages 181–198. Springer, 2003.

[245] Martin Shepperd. Software project economics: A roadmap. In *Proc. FOSE'07*, pages 304–315. IEEE, 2007.

[246] David J Sheskin. *Handbook of parametric and nonparametric statistical procedures*. crc Press, 2003.

[247] Mini Shridhar, Bram Adams, and Foutse Khomh. A qualitative analysis of software build system changes and build ownership styles. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 29. ACM, 2014.

[248] Tom De Smedt and Walter Daelemans. Pattern for python. *Journal of Machine Learning Research*, 13(Jun):2063–2067, 2012.

[249] Qinbao Song, Martin Shepperd, Michelle Cartwright, and Carolyn Mair. Software defect association mining and defect correction effort prediction. *IEEE Transactions on Software Engineering*, 32(2):69–82, 2006.

[250] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *International Conference on Extending Database Technology*, pages 1–17. Springer, 1996.

[251] Mikael Svahnberg, Tony Gorschek, Robert Feldt, Richard Torkar, Saad Bin Saleem, and Muhammad Usman Shafique. A systematic review on strategic release planning models. *Information and software technology*, 52(3):237–248, 2010.

[252] Bruno Takahashi, Edson C. Tandoc, and Christine Carmichael. Communicating on twitter during a disaster: An analysis of tweets during typhoon haiyan in the philippines. *Computers in Human Behavior*, 50:392–398 Elsevier.

[253] Mike Thelwall, Kevan Buckley, and Georgios Paltoglou. Sentiment in twitter events. *Journal of the American Society for Information Science and Technology*, 62(2):406–418, 2011.

[254] D. Thom, H. Bosch, S. Koch, M. Wörner, and T. Ertl. Spatiotemporal anomaly detection through visual analysis of geolocated twitter messages. *Pacific Visualization 2012*, 2012.

[255] Yuan Tian and David Lo. An exploratory study on software microblogger behaviors. In *Mining Unstructured Data (MUD), 2014 IEEE 4th Workshop on*, pages 1–5. IEEE, 2014.

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

[257] Amos Tversky and Daniel Kahneman. Advances in prospect theory: Cumulative representation of uncertainty. *Journal of Risk and Uncertainty*, 5(4):297–323, 1992.

[258] Gias Uddin, Barthélémy Dagenais, and Martin P Robillard. Temporal analysis of api usage concepts. In *Proceedings of the 34th International Conference on Software Engineering*, pages 804–814. IEEE Press, 2012.

[259] Marjan van den Akker, Sjaak Brinkkemper, Guido Diepen, and Johan Versendaal. Software product release planning through optimization and what-if analysis. *Information and Software Technology*, 50(1):101–111, 2008.

[260] Nadarajen Veerapen, Gabriela Ochoa, Mark Harman, and Edmund K Burke. An integer linear programming approach to the single and bi-objective next release problem. *Information and Software Technology*, 65:1–13, 2015.

[261] Sarah Vieweg, Amanda L. Hughes, Kate Starbird, and Leysia Palen. Microblogging during two natural hazards events: What twitter may contribute to situational awareness. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1079–1088, New York, NY, USA, 2010. ACM.

[262] L. Villarroel, G. Bavota, B. Russo, R. Oliveto, and M. Di Penta. Release planning of mobile apps based on user reviews. In *ICSE*, pages 14–24, 2016.

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

[264] Maria Grazia Violante and Enrico Vezzetti. Kano qualitative vs quantitative approaches: An assessment framework for products attributes analysis. *Computers in Industry*, 86:15 – 25, 2017.

[265] Persis Voola and A Vinaya Babu. Interval evidential reasoning algorithm for requirements prioritization. In *Proceedings Conference on Information Systems Design and Intelligent Applications*, pages 915–922. Springer, 2012.

[266] Persis Voola and A Vinaya Babu. Requirements uncertainty prioritization approach: a novel approach for requirements prioritization. *Software Engineering*, 2:37–49, 2012.

[267] MP Ware, F George Wilkie, and Mary Shapcott. The use of intra-release product measures in predicting release readiness. In *ICST*, pages 230–237, 2008.

[268] Andrzej Wasylkowski and Andreas Zeller. Mining temporal specifications from object usage. *Automated Software Engineering*, 18(3-4):263–292, 2011.

[269] Karl Wiegers and Joy Beatty. *Software requirements*. Pearson Education, 2013.

[270] Igor Scaliante Wiese, José Teodoro da Silva, Igor Steinmacher, Christoph Treude, and Marco Aurélio Gerosa. Who is who in the mailing list? comparing six disambiguation

heuristics to identify multiple addresses of a participant. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 345–355. IEEE, 2016.

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

[272] Shin-Yi Wu and Yen-Liang Chen. Discovering hybrid temporal patterns from sequences consisting of point-and interval-based events. *Data & Knowledge Engineering*, 68(11):1309–1330, 2009.

[273] Xin Xia, Emad Shihab, Yasutaka Kamei, David Lo, and Xinyu Wang. Predicting crashing releases of mobile applications. In *ESEM*, page 29. ACM, 2016.

[274] Tao Xie, Suresh Thummalapenta, David Lo, and Chao Liu. Data mining for software engineering. *Computer*, 42(8), 2009.

[275] Qianli Xu, Roger J Jiao, Xi Yang, Martin Helander, Halimahtun M Khalid, and Anders Opperud. An analytical Kano model for customer need analysis. *Design Studies*, 30(1):87–110, 2009.

[276] Seungwon Yang, Haeyong Chung, Xiao Lin, Sunshin Lee, Liangzhe Chen, Andrew Wood, Andrea L Kavanaugh, Steven D Sheetz, Donald J Shoemaker, and Edward A Fox. Phasevis1: What, when, where, and who in visualizing the four phases of emergency management through the lens of social media. In *ISCRAM*, 2013.

[277] Jie Yin, Andrew Lampert, Mark Cameron, Bella Robinson, and Robert Power. Using social media to enhance emergency situation awareness. *IEEE Intelligent Systems*, 27(6):52–59, 2012.

[278] Mohammed J Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine learning*, 42(1-2):31–60, 2001.

[279] Yuanyuan Zhang, Mark Harman, Gabriela Ochoa, Guenther Ruhe, and Sjaak Brinkkemper. An empirical study of meta-and hyper-heuristic search for multi-objective release planning. *Research Notes City University London, Dept. of Computer Science*, 14(07), 2014.

[280] Sven Ziemer, Pedro R Falcone Sampaio, and Tor Stålhane. A decision modelling approach for analysing requirements configuration trade-offs in timeconstrained web application development. In *Proceedings Conference on Software Engineering and Knowledge Engineering*, pages 144–149, 2006.