Software Engineering in the Age of App Stores

Feature-Based Analyses to Guide Mobile Software Engineers

Afnan A. Al-Subaihin

A dissertation submitted in fulfilment
of the requirements for the degree of
Doctor of Philosophy
of
University College London

University College London
Department of Computer Science
Centre for Research on Evolution, Search and Testing

Student funded by King Saud University, Saudi Arabia.

20 Dec. 2018

Declaration

I, Afnan A. Al-Subaihin confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis. Parts of this document have been published in the following papers:

- Chapter 3 is published as: A. A. Al-Subaihin, F. Sarro, S. Black, L. Capra and M. Harman, “App Store Effects on Software Engineering Practices," in IEEE Transactions on Software Engineering (TSE), 2019. (*Early Access*). DOI: 10.1109/TSE.2019.2891715.
- Chapter 4 is published in: A. A. Al-Subaihin, F. Sarro, S. Black, L. Capra, M. Harman, Y. Jia, and Y. Zhang, “Clustering mobile apps based on mined textual features," in Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ser. ESEM’16. New York, NY, USA: ACM, 2016, pp. 38:1-38:10. DOI: 10.1145/2961111.2962600.
- Chapter 5 is to appear as: A. A. Al-Subaihin, F. Sarro, S. Black and L. Capra, “Empirical Comparison of Text-Based Mobile Apps Similarity Measurement Techniques," in Empirical Software Engineering (EMSE). (*To appear*).
- Chapter 6 is part of this publication: F. Sarro, A. Al-Subaihin, M. Harman, Y. Jia, W. Martin, and Y. Zhang, “Feature Lifecycles as They Spread, Migrate, Remain and Die in App Stores," in 2015 23rd IEEE International Requirements Engineering Conference. IEEE, Aug. 2015. DOI: 10.1109/RE.2015.7320410.

Signature: **Date:**

Acknowledgements

This PhD thesis: the culmination of years of continuous learning and hard work, barely merits its name without the tremendous effort of my supervisors, mentors, family and friends, to whom I would like to express my sincerest gratitude.

Supervisors guided me through this journey with generous support, gracious mentorship and extreme patience. A favourite writer of mine, Maria Popova, writes: “Few things reveal your intellect and your generosity of spirit - the parallel powers of your heart and mind - better than how you give feedback," and I have been immensely lucky to have four mentors who possessed both. They have been role-models to whom I look up and whose academic path I aspire to follow: Federica Sarro, Mark Harman, Sue Black and Licia Capra. Thank you.

In times when this journey seemed more challenging, my family’s steadfast support kept me afloat. Their constant faith in me, even when mine was faltering, has been a rock on which I invariably relied in times of self-doubt. My parents and my siblings: Thank you.

Finally, I would not have been able to do this PhD without the funding granted by King Saud University in Riyadh, Saudi Arabia. In addition to the the efforts of the staff at the Saudi Arabian Cultural Bureau in the UK who liaised with the financial sponsor and offered continuous aid to students. Thank you.

Abstract

Mobile app stores are becoming the dominating distribution platform of mobile applications. Due to their rapid growth, their impact on software engineering practices is not yet well understood. There has been no comprehensive study that explores the mobile app store ecosystem's effect on software engineering practices. Therefore, this thesis, as its first contribution, empirically studies the app store as a phenomenon from the developers' perspective to investigate the extent to which app stores affect software engineering tasks.

The study highlights the importance of a mobile application's *features* as a deliverable unit from developers to users. The study uncovers the involvement of app stores in eliciting requirements, perfective maintenance and domain analysis in the form of discoverable features written in text form in descriptions and user reviews. Developers discover possible features to include by searching the app store. Developers, through interviews, revealed the cost of such tasks given a highly prolific user base, which major app stores exhibit.

Therefore, the thesis, in its second contribution, uses techniques to extract features from unstructured natural language artefacts.

This is motivated by the indication that developers monitor similar applications, in terms of provided features, to understand user expectations in a certain application domain. This thesis then devises a semantic-aware technique of mobile application representation using textual functionality descriptions. This representation is then shown to successfully cluster mobile applications to uncover a finer-grained and functionality-based grouping of mobile apps. The thesis, furthermore, provides a comparison of baseline techniques of feature extraction from textual artefacts based on three main criteria: silhouette width measure, human judgement and execution time.

Finally, this thesis, in its final contribution shows that features do indeed migrate in the app store beyond category boundaries and discovers a set of migratory characteristics and their relationship to price, rating and popularity in the app stores studied.

Impact Statement

This research inspects mobile app stores as a source of data mining to aid software engineering tasks. It furthermore focuses on software *features* as a unit of analysis when mining from app stores.

First, we bridge the gap between the software engineering research community and mobile software engineering with regards to their interaction with mobile app stores. This has been accomplished using a survey-based exploratory study employing first and second-degree data collection methods from mobile app developers. We summarise the findings under three main themes: enhanced communication among developers and users, increased market transparency and altered mobile release strategies. This understanding enriches the empirical software engineering research literature with better understanding of the ecosystem's involvement in mobile software engineers' practices. This can help guide future research in mining software repositories, requirements engineering and the business community as it provides a roadmap of practitioners' needs from data found in app stores throughout the evolution of their mobile apps. The findings of the survey may also inform managers of app development projects and their developers.

Based on certain results of the previous exploratory study, this research employs a feature extraction technique that can be carried out over natural language and does not require access to the source code, as it is hard to be obtained from mobile app stores. The extracted features were then employed to carry out several software engineering research tasks guided by the needs of developers found in the aforementioned study.

Firstly, these features were used in app representation in order to find underlying, functionality-based clustering of mobile apps using a novel semantically-aware technique that enhances over the typical baseline. The devised algorithm provides both business and technical value as developers may use it to find similar applications, from which they can explore the application domain and investigate desired supporting feature sets. Several feature extraction algorithms are further investigated to observe their applicability for detecting the similarity of mobile applications.

Secondly, These features' migratory behaviour over app stores categories is then analysed to guide developers in making decisions of whether features carry any transitive value and therefore can be adopted.

Contents

1 Introduction **13**

2 Literature Review **15**

2.1 Introduction . . . 15
2.2 Software Features . . . 15
2.2.1 Feature Modelling . . . 17
2.2.2 Feature Location . . . 17
2.2.3 Feature Extraction from Natural Language . . . 19
2.3 Automatic Software Categorization . . . 24
2.3.1 Categorization of Software . . . 25
2.3.2 Categorization of Mobile Applications . . . 27
2.4 App Store Analysis . . . 33

3 App Store Effects on Software Engineering Practices **35**

3.1 Introduction . . . 35
3.2 Related Work . . . 38
3.3 Methodology . . . 40
3.4 Study Design . . . 41
3.4.1 Research Questions . . . 42
3.4.2 Interviews . . . 43
3.4.2.1 Protocol . . . 43
3.4.2.2 Participants . . . 44
3.4.2.3 Data Analysis . . . 46
3.4.3 Questionnaire . . . 47
3.4.3.1 Design . . . 48
3.4.3.2 Participants . . . 49
3.4.3.3 Data Analysis . . . 49
3.5 Findings . . . 51
3.5.1 Interview Analysis Results . . . 52
3.5.2 RQ1: Lifecycle Processes . . . 53
3.5.2.1 Requirements Elicitation . . . 53
3.5.2.2 Testing . . . 54
3.5.2.3 Maintenance . . . 56

3.5.2.4 Release Management . . . 59
3.5.3 RQ2: Emerging Skill-sets and Best Practices . . . 60
3.5.4 RQ3: New Success Criteria and Performance Measures . . . 61
3.6 Discussion . . . 62
3.6.1 Developer-User Interaction . . . 62
3.6.2 Market Transparency . . . 63
3.6.3 Release Planning and Quality . . . 64
3.7 Threats to validity . . . 65
3.7.1 Construct Validity . . . 65
3.7.2 Internal Validity . . . 65
3.7.3 External Validity . . . 65
3.8 Conclusions . . . 66

4 Feature-Based Mobile App Categorization 67
4.1 Introduction . . . 67
4.2 Approach . . . 68
4.2.1 Feature Extraction . . . 69
4.2.2 Feature Clustering . . . 69
4.2.2.1 Feature Representation . . . 70
4.2.2.2 Selecting K . . . 71
4.2.2.3 Final Clustering . . . 71
4.2.3 App Clustering . . . 72
4.2.3.1 App Representation . . . 72
4.2.3.2 Selecting Clustering Technique . . . 72
4.3 Empirical Study Design . . . 72
4.3.1 Research Questions . . . 72
4.3.2 Dataset . . . 75
4.3.3 Evaluation Criteria . . . 75
4.4 Results Analysis . . . 76
4.5 Threats to Validity . . . 84
4.6 Conclusions . . . 84

5 Comparison of Feature Extraction Techniques for App Clustering 88
5.1 Introduction . . . 88
5.2 Empirical Study Design . . . 89
5.2.1 Text Representation Techniques . . . 89
5.2.1.1 Vector Space Model . . . 90
5.2.1.2 Latent Dirichlet Allocation . . . 90
5.2.1.3 Feature Vector Space . . . 91
5.2.2 Research Questions . . . 93
5.2.3 Dataset . . . 94
5.2.4 Evaluation Criteria . . . 94

5.3 Empirical Study Results . . . 95
5.4 Discussion . . . 100
5.5 Threats to Validity . . . 101
5.6 Conclusions . . . 101

6 Feature Migration in the Samsung App Store 103
6.1 Introduction . . . 103
6.2 Overview of Feature Migratory Behaviour . . . 104
6.3 Empirical Study Design . . . 105
6.3.1 Dataset . . . 105
6.3.2 Research Questions . . . 106
6.3.3 Methodology . . . 106
6.4 Results Analysis and Discussion . . . 107
6.4.1 RQ0. Feature Evolution . . . 107
6.4.2 RQ1. Feature Migration . . . 107
6.4.3 RQ2. Differences in Migratory Behaviours . . . 107
6.4.4 RQ3. Correlations among Price, Popularity and Rating . . . 110
6.5 Threats to Validity . . . 110
6.6 Conclusions . . . 116

7 Conclusion and Future Work 118

Appendices 120

A Questionnaire - App Store Ecosystems Effects 120

Bibliography 132

List of Figures

3.1 The various stages of the study. 36

3.2 Interview design: The topics planned for discussion and the questions answered within. The dotted line resembles an example of topic flow in one interview. No structure is enforced and the questions were discussed as the respondent moved freely from one topic to another. 44

3.3 Transcript raw data codes used to tag responses: These codes represent recurring topics and certain responses of interest. This is the first stage of interview analysis. The codes and their content are then used to deduce themes in Figure 3.6 46

3.4 Histograms depicting the distribution of the various demographical information of the questionnaire's respondents. 50

3.5 RQ1. Responses to questions regarding the initial phases of development. 51

3.6 Thematic analysis findings in the form of a theme map. The theme map summarizes the data patterns found in the interview transcripts relating to the research questions. 53

3.7 RQ1. Responses to questions regarding the testing phase. 55

3.8 RQ1. Responses to questions regarding the usefulness of user feedback. 55

3.9 RQ1. Responses to questions regarding the usefulness of user feedback for corrective maintenance. 56

3.10 RQ1. Responses to questions regarding the usefulness of user feedback for perfective maintenance. 58

3.11 RQ1. Responses to questions regarding the effect of app store to release strategy. . 59

3.12 RQ2. Responses to questions regarding the type of activities and skills required in a development team. 60

3.13 RQ3. Responses to questions regarding the knowledge of success factors in the app store. 62

4.1 Feature Extraction and the Two-Phase Clustering System Architecture 69

4.2 Dendrogram of the resulting agglomerative hierarchical clustering using cosine dissimilarity and Ward's criterion. 73

4.3 RQ 1. The silhouette width for each category in the existing categorisation of the BlackBerry and Google datasets. 77

4.4 RQ2.1 The average silhouette width for the different number of segments in BlackBerry and Google dataset. 80

4.5 (Middle) The average silhouette score for each existing commercial category in the BlackBerry app store when measured using our app representation technique (RQ1). As an example, we zoom in the Utilities category (which exhibits a low current avg. silhouette) and the News & Magazines category (which has a relatively high current avg. silhouette). The left- and right-hand sub-figures illustrate how our approach can be used to uncover the clustering within each category (RQ2.2). 82
4.6 RQ2.2. Refining the existing BlackBerry app store categorisation by sub-clustering each category and observing how the average silhouette scores behaves as the granularity increases. The x axis is the granularity whereas the y axis is the achieved average silhouette for that granularity. For each category, the dotted line is the current average silhouette score for the category's members in relation of the entire app store. 85
4.7 RQ2.2. Refining the existing Google app store categorisation by sub-clustering each category and observing how the average silhouette scores behaves as the granularity increases. The x axis is the granularity whereas the y axis is the achieved average silhouette for that granularity. For each category, the dotted line is the current average silhouette score for the category's members in relation of the entire app store. 86

5.1 RQ2.1. Average silhouette score as the granularity (k) increases for each technique. . 98

6.1 The Feature Migration Subsumption Hierarchy. 105
6.2 RQ1. Observed Number of Features for each Migratory Behaviour for the Samsung App Store. 108
6.3 RQ2. Boxplots of Mean Price, Rating and Popularity (Rank of Downloads) for each of the non-migratory behaviours. 109
6.4 RQ2. Boxplots of Median Price, Rating and Popularity (Rank of Downloads) for each of the non-migratory behaviours. 109
6.5 RQ3. Scatterplot of Mean Price (P), Rank of Downloads (D) and Rating (R) for the migratory behaviours (W)eak (M)igration and (N)o (M)igration 112
6.6 RQ3. Scatterplot of Mean Price (P), Rank of Downloads (D) and Rating (R) for the migratory behaviours (I)ntransitive and (S)trong (M)igration. 113
6.7 RQ3. Scatterplot of Mean Price (P), Rank of Downloads (D) and Rating (R) for the migratory behaviours (W)eak e(X)tinction and (S)trong e(X)tinction. 114
6.8 RQ3. Scatterplot of Median Price (P), Rank of Downloads (D) and Rating (R) for all the features. Please, note that we grouped the points based on their median values. 114
6.9 RQ3. Scatterplot of Median Price (P), Rank of Downloads (D) and Rating (R) for the non-migratory behaviours I, WX, SX. Please, note that we grouped the points based on their median values. 115

List of Tables

3.1 The set of interview questions. 45
3.2 Demographical data of the developers interviewed. 46

4.1 Examples of extracted featurelets representing each feature and the number of times these features appear in the dataset (number of apps that boast the feature) for both Blackberry and Google datasets. 70
4.2 RQ1. Summary measures (Min, Max, Mean and Median) of the silhouette widths of existing categories and those achieved when applying our clustering approach using as granularity the number of existing categories in the stores considered. 78
4.3 RQ2.1. Comparing the individual element's silhouette widths of: The current quality of the existing categorisation, our clustering reusing existing coarse granularity, and our clustering using the maximum viable granularity. 78
4.4 RQ2.2. Categories, their size, and granularity level that provides the highest silhouette width for each app store category when sub-clustered. 79
4.5 RQ3. App pair examples and the feature terms that they share selected from the feature cluster prototype. 81
4.6 RQ3. The Spearman rank correlation coefficient (ρ) between each of the raters similarity scores of pairs of apps and the finest granularity that these apps remain clustered together before they separate. All reported coefficients have $p - values < 0.001$. 83
4.7 RQ4. Computation time of tasks in our framework. The first two tasks are up-front costs. 83

5.1 Examples of extracted topics (represented by 4 terms) and the number of apps associated with each topic (occurrences). 91
5.2 Examples of extracted featurelets representing each feature and the number of times these features appear in the dataset (number of apps that boast the feature) for collocation-based and dependency-based parsing. 93
5.3 RQ0. Jaccard similarity index between each of the clustering solutions of the studied techniques for 24 clusters (the number of app store categories). 96
5.4 RQ1. Summary of silhouette width scores for each of the techniques when considering app store category as a cluster assignment (existing categorisation) and when selecting $k = 24$ (same number of categories in the app store). 97
5.5 RQ2.1. For each technique, the maximum viable granularity and the generated maximum silhouette score. 97

5.6 RQ3. Inter-rater agreement of the obtained goldset (8 raters) on the three rating criteria using intra-class correlation. 99
5.7 RQ3. Spearman Rank correlation scores (p-value in brackets) between hierarchical sampling level (technique-assigned similarity) and human-assigned similarity scores. Scores are deemed statistically significant if p-value < 0.01. 99
5.8 RQ4. Efficiency of each of the studied techniques. The technique's run-time was measured on a standard laptop with an Intel Core i7 3.1 GHz and 16 GB RAM; d=days, h=hours, m= minutes, s=seconds. 100

6.1 Summary Data for the Samsung Apps Studied Between Two Time Intervals. 105
6.2 RQ0. Number of features contained in a given category in the Samsung app store; and Jaccard Similarity (JS) of the initial and final categories over the time period. . . . 108
6.3 Wilcoxon Test Results: mean price. 110
6.4 Wilcoxon Test Results: mean rating 111
6.5 Wilcoxon Test Results: mean rank of downloads comparison among the migratory behaviours. 111
6.6 Wilcoxon Test Results: Median Price. 111
6.7 Wilcoxon Test Results: median rating. 111
6.8 Wilcoxon Test Results: median rank of downloads. 112
6.9 RQ3. Raw Value Correlations. 113
6.10 RQ3. Median Price Point Correlations. 116
6.11 RQ3. Mean Price Point Correlations. 116

Chapter 1

Introduction

In the quest to increase the value and usability of software (i.e. operating systems, browsers and office suites), vendors detected the value of establishing software ecosystems around the software platform [1]. A software ecosystem is an environment that facilitates the extension and customisation of said platform by external and/or internal developers to attract and satisfy the requirements of a wide user-base [2][3]. Mobile operating system-based ecosystems are currently the major portals for acquiring mobile software [4] manifesting in their online deployment portals, referred to as 'app stores' in relevant literature and hereafter.

App stores are now a common channel that facilitates app deployment for developers and discovery for users. They house a vast amount of applications belonging to diverse application domains; consolidating business, customer and technical information. Additionally, app stores provide an environment for user-user in addition to user-developer interaction. App stores, therefore, hold the potential of being a new software repository with large and various types of software meta-data and related artefacts that can aid empirical software engineering research.

App stores' dominance over the mobile application deployment market, gives grounds to conjecture that they may influence software engineering practices of mobile app developers. App stores' effect on software development activities is not yet well understood due to their relative novelty. An understanding of app developers' practices and goals when dealing with app stores, and how they influence decision making throughout the development process, promises to guide further analysis and mining of information contained therein. This prompted the study reported in Chapter 3 as an initial exploration of app stores as a new avenue of software repository mining. This empirical study views app stores as a global phenomena motivating the study of its effects using first and second-degree data collection methods. By interviewing and surveying mobile app developers, then analysing the gathered data, the study reports how app stores are affecting software engineering practices. Among the many findings reported, the study highlights how important developers deem monitoring other similar apps to support the evolution of theirs, especially in terms of the *features* they include.

This is not surprising as the concept of ***feature*** in software engineering research, as well as practice, is central to many of the software engineer's activities. Features are the unit of a software system that connect the problem domain to the solution domain. It is in the successful transformation of user requirements into software features that an application fulfils its purpose. Features enable the modular extension and customisation of software and therefore garner much of the mobile software

engineer's attention throughout out the software development cycle as found by this thesis' first study.

The study finds that 56% of surveyed developers elicit requirements from the features of other similar apps in the app store (more so than other web/desktop applications or user surveys and focus groups). Furthermore, 68% of those respondents locate features by reading the app's description. In the stage of perfective maintenance, the study reveals that 51% of respondents discover new features and enhancements by looking into similar apps. This highlights the integral role that features play in the elicitation and design phases, in addition to the interaction between users and developers in app stores. Subsequently, this thesis focuses on the analysis of features, extracted from natural language, to aid in several software engineering tasks; namely, app clustering based on those shared features (Chapter 4), how different feature extraction techniques perform for app representation in the task of clustering (Chapter 5) and finally studying their migratory behaviours (Chapter 6).

This thesis, therefore, empirically studies mobile app stores and reveals their involvement in software engineering practices throughout the software life cycle, emerging new skill-sets and success metrics. Then, upon revealing how developers elicit requirements and perform perfective maintenance from information available on the app store, this research embarks on quantifying the similarity between mobile apps for the task of clustering mobile application based on application features expressed in natural language and studying feature movement behaviour in the app store.

The remainder of this document first presents a review of the literature pertaining to various aspects of this thesis (Chapter 2). Chapter 3 shall report an empirical study that observes the extent to which mobile app stores affect the practices of involved software engineers. Afterwards, Chapter 4 reports the design and evaluation of an investigation of whether apps can be clustered, based on their textual features, to uncover hidden categorization of apps in app stores. Chapter 5 compares different common baseline feature extraction techniques from natural language and evaluate their adequacy for app similarity detection and clustering. Chapter 6 reports a study that classifies and detects feature migration behaviour in app store. Finally, Chapter 7 concludes the thesis.

Chapter 2

Literature Review

2.1 Introduction

This chapter presents a report of the literature deemed related to various aspects of this thesis. Features (Section 2.2) appear in two main lines of research in software engineering. The first is their major role in capturing components of similarity and variability in domain analysis for product line engineering (discussed in Section 2.2.1). Second is the feature location problem in software comprehension where features represent the unit of functionality in code that developers or system maintainers need to locate in order to enhance or fix the functionality (Section 2.2.2). After a brief overview of seminal work in these lines of research, a dedicated section (2.2.3) will revisit work that performed automatic feature extraction from various sources. Afterwards, a section is dedicated to research on the problem of categorising software (Section 2.3). This research is related to this thesis as it involves a certain degree of comprehension of the underlying software or its artefacts to achieve an accurate classification and serves as a precursor to chapters 4 and 5. The final section briefly discusses app store analysis research with special emphasis on work that investigates software features (Section 2.4).

Conducting the literature review for each sub-section began with a few seminal papers relating with the respective sub-field. Then a snowballing technique was followed to identify related and similar research works. The author only selected the papers that extend, improve, and/or transfer the proposed technique in a way deemed relevant to this thesis; in addition to publications that proposed novel techniques to solve a similar problem. This literature review, aims to provide sufficient review of related works to all several aspects of this thesis, though they may not strictly relate to one another. The literature review prioritises breath coverage over in-depth and therefore, the following sections are not comprehensive. However, Section 2.3 is exempt as the author aimed to review all literature concerned with categorising software. This has been conducted by using related look up terms on Google Scholar coupled with snowballing.

2.2 Software Features

A software feature is a concept understood by engineers, customers and end-users. This ubiquity of understanding caused a substantial attention in software engineering research.

However, the term *feature* itself is multi-faceted and may refer to slightly different things depending on the research context. In the reviews of definitions mentioned in relevant literature presented below, all definitions agree that a feature is a software attribute ultimately present in the solution

domain that represents a cohesive set of system functionality. The disagreement is observed with regard to its origin, whether it encapsulates non-functional requirements, the level of granularity of a feature towards functionalities, and the level of visibility to the user in the end product. These ultimately are implicitly agreed-upon depending on the software engineering line of work. For example, in feature modelling, a feature is usually user-centric functionality that originates from the problem domain; whereas in feature location, a feature is exhibited in the solution domain and is typically a functional requirement that has a specific collection of statements that implement it.

One aspect of conflict in the definition of the term *feature* is the origin and location in which it is observed. The origin of a feature maybe either in the problem domain or the solution domain [5]. Different lines of research adopt either of these views. In the line of work concerned with feature-oriented software engineering, features are a subset of system requirements. On the other hand, in the literature concerned with feature identification and location, features are assumed to be a subset of system implementation.

In defining the relationship between features and requirements, we observe two different definitions as to whether a feature also represents non-functional requirements. The Institute of Electrical and Electronics Engineers (IEEE) defines the term feature in their *829 Standard for Software and System Test Documentation*[6] as “A distinguishing characteristic of a software item (e.g., performance, portability, or functionality)." Which draws a one-to-one mapping between a feature and both a functional or a non-functional requirement. On the other hand, the *Guide to the Software Engineering Body of Knowledge* (SWEBOK) [7] synonymises it with ***functional requirements*** as they both represent a software capability that can be tested and verified. This particular definition is used in feature identification and location line of work, as it aims to identify the set of statements that implement a certain feature which can be impossible for non-functional requirements.

Another possible implicit constraint on the usage of the term ‘feature’ is its user-centrality. Turner et al., surmises that features, as opposed to functionalities, are visible properties of a software system: They represent the set of functionalities that are specifically user-centric [5]. This particular aspect of the definition of features is observed when reading works pertaining to feature modelling. On the other hand, work on feature identification, not only disregards this aspect, but adds its own: a feature is a unit of functionality that is optional or incremental to the system [8].

This thesis views features as a unit of an application’s behaviour, expressed in natural language, that is both understood by developers and end users; regardless of the terminology used to communicate that feature.

Many advocate the use of features as first-class objects of the software development process. This can be found in *domain engineering* research where feature models are used to represent the collective functionalities of applications in a certain application domain. Features also are the unit of analysis in the *concept assignment and identification* line of research which aims to reverse engineer code segments into their semantic feature counterparts. A point of interest to this research, in both these lines of work, is how features are extracted from existing software artefacts, automatically or otherwise.

In addition to a brief overview of publications in both previous research paradigms, this section will look into various feature extraction methodologies done in previous work.

2.2.1 Feature Modelling

Features play a main role in domain analysis and engineering. The use of features as first class objects in software development was first promoted by Kang et al. when they proposed Feature Oriented Domain Analysis (FODA) and Feature Oriented Reuse Method (FORM) [9][10]. In their work, they view features as “distinctively identifiable functional abstractions that must be implemented, tested, delivered and maintained". They argue the need to give features more emphasis when analysing and designing domain specific applications. Their proposed method: FORM, is a method used to facilitate code re-use and minimize time to market in software product lines via the construction of feature models.

The goal of Feature modelling is to discover and model software similarity and variability in an application domain. It is the activity of constructing a hierarchical graph representing feature AND/OR relationships. Where an AND relationship conveys a mandatory feature and OR is for alternative features [11].

This method has been developed and extended resulting in Feature-Oriented Product Line Engineering [12] and Feature-Oriented Software Development (FOSD) [13]. Where defining an application via a unique collection of features is not only apparent in the design phase, but also carried out in subsequent system development stages [14]. In FOSD, software features are the first class entities when analysing, designing, implementing, or maintaining a software system. With the maturity of these methods, the term ‘feature’ reaches a consensus in definition [13][15], where it is viewed to carry three distinctive characteristics:

1. A feature is a logical grouping of a set of requirements,
2. it is prominent and user-visible, and
3. has an incremental quality and usually used to extend a system’s functionality.

Constructing feature models, although highly rewarding, proved to be a difficult and costly task. Due to its relation with the domain at large, it is highly time consuming. Additionally, research shows that it is susceptible to the engineer’s experience and background. Consequently, subsequent efforts shifted towards facilitating the formation of feature models from existing software. These efforts looked into automatic extraction of features and their relationships from different software artefacts. Extraction sources include requirements documentation [16][17][18][19], use cases [20] and product descriptions [21][22][23][24].

2.2.2 Feature Location

Feature location (also known as concept location) is the activity of identifying the subset of code that implements a certain functionality. Essentially, it is a problem of linking system modules, expressed in code, to human-oriented, context-rich form expressed in natural language (usually known in advance) [25]. Naturally, this line of work views a feature as user-viewable realization of a functional requirement; excluding non-functional requirements due to the nature of the problem. Feature location is an important step for code maintenance as every code modification starts with locating the code that needs to be changed. Establishing this mapping aims to enhance practitioner’s understanding of the system which greatly facilitates subsequent system maintenance and evolution tasks [26].

Feature identification and location is closely related to *reverse engineering* which aims to analyse a system to uncover its components and their interdependence to finally synthesize alternate forms of representations [27]. Another closely related research area is *requirements traceability*, which aims to link features in several design artefacts. Feature location, the focus of this section, is only concerned with linking source code sections with the functionality they implement. It mainly enhances *software comprehension* by providing faster and more reliable way of finding code than basic text searching techniques such as *grep*; which has been found by Sim et al. [28] to be very commonly used for searching code.

Methods proposed, thus far, to carry out feature location employ static or dynamic code analysis. Other methods look into historical data such as mining version control systems. These methods also vary in terms of the type of user input required to carry out the task. This can be: a natural language query describing the feature, an execution scenario proposed by the developer to run a feature, or a source code point that is used as a starting point for the location process [8]. The returned result also varies in terms of granularity. Approaches available return location either at the file/class, method/function or statement level.

One of the earliest efforts towards feature location is Software Reconnaissance by Wilde et al. [29][30]. Their work provide a **dynamic analysis** of code to identify features using two sets of test cases: Test cases that exercise this feature, and test cases that do not. Then they formally define sets of components according to their relationship to the feature to be located. The sets include: common components, potentially involved components, indispensably involved components and unique components. Common components are those components invoked by most features. Potentially involved components are components that were executed in at least one test case involving that feature. An indispensably involved component is a more specific type where the component is called in every test case that covers the feature. Lastly, a unique component is a component that is exercised only in test cases that exhibit the feature. This method has many shortcomings, mainly its failure to identify main features that are present in all system procedures since it can be impossible to generate test cases that do not include that feature. Furthermore, this method cannot guarantee the identification of all locations associated with the feature. However, it succeeds in reducing the search space for the developer. This work sets the foundation of using execution traces from test scenarios which is heavily extended in subsequent research [31][32].

The majority of dynamic analysis techniques takes as input a *scenario*. A scenario is the set of steps taken by the user to trigger a certain feature [33].

Feature location was also attempted using **static analysis** methods. These methods do not require the execution of any of the code, making it possible to perform feature location with code that is not entirely compilable. Manual or automatic analysis of the code is used to generate system structure graphs, augmented with other system artefacts, these are used to locate feature implementation locations. Attempts employing static analysis of the code used System Dependence Graphs [34][35], concern graphs [36], method call graphs [37] and static data-flow analysis [38]. In terms of input, most of these techniques require a program element from the user, it uses it as a starting point from which the analysis starts of the structure graph of choice.

Other feature location techniques used **textual analysis** of the code and possibly other software artefacts. This approach relies on user's input as a natural language query then establishing the

connection using information retrieval (IR) or natural language processing (NLP) techniques.

Among the first to employ **information retrieval** to solve the concept assignment problem is Marcus et al. [39] where they use Latent Semantic Indexing (LSI) to map feature queries to code components that address those features. In their work, they leverage comments and identifier names found in source code to calculate a similarity metric between any two segments of the code. In the same way, the user query is considered by their tool as a segment whose similarity can be calculated and compared to any of the system's segments. The results are then returned ranked according to their calculated relevance to the query. This method, performs faster and more reliably than basic regular expression search techniques (e.g. *grep*). It also boasts no programming language dependency. This work was later extended in [40] to include formal concept analysis (FCA) which helps the user in sorting through the returned results by clustering them according to common topic. FCA performs grouping of methods according to their shared attributes (i.e. words in source code) such that every group forms a certain concept. This technique proved to enhance regular IR method and produced good results. Also using LSI, Kuhn et al. [41][42] propose enhancing program comprehension by building semantic clusters using LSI over text found in source code and other artefacts. A distribution map is drawn to show prevalence of certain concepts over regions of source code thus revealing its intention. They apply their approach over 7 open source projects showing the tool can increase insightful analysis and understandability of code.

A drawback of LSI is that it does not factor synonymity into the similarity metric. For example, the phrases *cell phone* and *mobile device* would not have special similarity. This places emphasis on query formation by the user. Cleary and Exton [43] overcame this shortcoming by finding external relationships to concepts from non-source code artefacts. Using their approach, code fragments can be returned although they do not contain any of the query words thus matching (if not enhancing) the performance of previous approaches.

Natural language processing was also incorporated to solve the concept assignment problem. These approaches add the extra step of analysing linguistic attributes such as part-of-speech (noun, verb, etc.) and syntactic roles (i.e. subject, object, etc.) in addition to looking into comments and identifier names. In [44] they specifically exploit the verb-object pair found in source code where they collect all verb and their direct object pairs while maintaining the mapping with code fragments in which they were found. The user then may start a query using one verb, the tool would suggest a list of possible direct objects which the user selects to refine the search query. Another approaches incorporating NLP techniques used different type of phrases (verb, noun and prepositional phrases) to cluster code segments into concepts and all different linguistic representations [45]. Other approaches employ ontologies to capture phrases and their code location found in: [46] and [47].

Many other feature location techniques employ a hybrid of previously discussed approaches. An example is tools that incorporate a mix of static and dynamic analysis. This mixture proved powerful and highly effective. Dit et al. provide a comprehensive survey of these approaches [8].

2.2.3 Feature Extraction from Natural Language

This concerns the line of work attempting to conceptualise features from various software engineering artefacts. These artefacts may include source code in addition to other textual repositories such as release notes, bug databases, comments, and product description. Automatic feature concep-

tualization aids in reverse engineering the system to facilitate bug prediction and fixing, building feature models for domain analysis and traceability link recovery, among many other uses. This section focuses on a subset of this problem tackling feature extraction from natural language software artefacts as it is most pertinent to this thesis (Chapters 4, 5 and 6).

One of the earliest publications in extracting software engineering concepts from natural language is done by Maarek et al. [48] where they aim to automatically extract concepts from natural language code-related artefacts to build re-usable libraries. In their work, they devise a novel information retrieval method similar to Lexical Affinity (LA)[49]. The method finds words whose frequency correlates with each other. Lexical Affinity in large text proved to convey both lexical and semantic significance. Using LA to index code comments and manuals, they carry out indexing and classifying code to re-usable components. The LA method seems to be a predecessor of the now widely used collocation extraction. However, in this work they only extract collocations that convey a modifier-modified relationship; then ranking them by their frequency to attain more meaningful collocating words. Interestingly, these extracted collocations do resemble the concept of 'feature' that forms the main unit of extraction in later work concerned with app store analysis. One limitation of this work is that it does not give special importance to domain-specific concepts appearing in text. Especially when these concepts consist of more than 2 words.

Subsequent research by Niu et al. [17][50] expands on the LA technique augmenting it with further clustering with the goal to aid the design of software product lines. To address the limitation of using LA, they augment the analysis with a list of domain-specific concepts in text form. The concepts are then identified in the source text and replaced with single word identifier prior to running the LA algorithm. This proved to overcome its insensitivity to domain-specific terms and inability to capture 3 word collocations.

Cleland-Huang et al. [51] sought to facilitate feature request consolidation found in user-forums, due to their large redundancy and noise. They use a straight-forward vector space model (with TF-IDF weighting), and a variant of spherical k-means clustering technique to cluster user text requesting the same feature. They then tested their technique on feature requests extracted from user forums of three open-source software products, one of which has been manually annotated to create a truth-set clustering solution. They compare the output with the truth-set using a metric called Normalised Mutual Information (NMI) that ranges from 0 (entirely different) to 1 (identical), in which their proposed technique scores 0.57. In [52], Rahimi and Cleland-Huang introduce the notion of a 'personas' representing the profiles of possible users and around which certain groups of features are centred. These profiles are designed by extracting and clustering feature requests from user forums (using incremental diffusive clustering[23]) coupled with association rule mining to find commonly co-occurring feature requests resulting in a specific user profile.

Similar to feature requests in user forums, **issue tracking systems** have elicited the need for feature identification and consolidation, as presented and tackled using NLP in the seminal work by Runeson et al. [53]. This work provides evidence that NLP (namely, vector space model) can successfully detect similarity of software issue reports (up to two thirds) to an acceptable degree of accuracy, paving the way for the usage of NLP in feature identification and conceptualisation. This work also shows that using the vector space model with cosine similarity perform better recall than dice and Jaccard metrics. Wang et al. [54] extends this technique to add bug/issue technical

report (e.g. execution trace) also represented using the vector space model to the issue's natural language description. They were able to improve the correct detection rate from 74% to 93% at best. Sun et al. [55] replaces the stack trace information due to their lack of availability with other issue meta-data (e.g. software component, version and priority) to aid in identifying duplicates. They use BM25F which is an information retrieval framework that ranks similarity of documents based on a search query and is similar to using TF-IDF weighting. They improve over the baseline by 27% and 23% in recall and precision respectively. Alipour et al. [56], extend that work by introducing further contextual information of the bug using topic modelling (LDA) and domain knowledge (later extended in [57]), improving the detection accuracy by 16% over the baseline. On the other hand, research done by Rakha et al. [58] found that the majority of duplicate issue reports were identified with little effort. They further extend the research with thorough evaluation frameworks and enhancements over the previously proposed methods [59].

Merten et al.[60] extends this problem domain by incorporating the capability to distinguish a request from other natural language mentioning a feature, in addition to a clarification of what the feature is and any explanation of how it could be implemented, therefore uses supervised techniques rather than clustering. They benchmark the performance of 7 different machine learning approaches with 6 different features (ways of representing the data for the model): bag-of-words, subject-action-object patterns, bi- and tri-grams, keywords, issue and data field meta data. They test their model on a truth set of 150 issues extracted from the issue tracking systems of 4 different software projects. The findings reveal that incorporating the meta-data of the issue does indeed improve classification results. Furthermore, they find that incorporating sentence lexical patterns (subjection-action-object) and bi- and tri-grams does not improve over bag-of-words significantly.

Automatic feature extraction from various system-related sources is frequently attempted for **feature model synthesis**. This line of work aims to aid in building domain-specific feature models from existing software and software artefacts. Additionally, research in this area focuses on feature extraction from textual documentation. This is due to the reality of existing system documentation as having different structures (or lack thereof) and different levels of abstractions. In this section, for the sake of brevity, approaches that extract features from structured input (RDL documents, modularized use cases, etc.) are not discussed since they do not directly contribute to the thesis problem as these kinds of resources are not readily available from app stores.

Since the term *feature* in feature modelling refers to a cohesive set of requirements, these approaches usually employ a clustering technique of uncovered requirements into features. In the collection of works that aim to extract features from natural text, a previous step to clustering would be extracting textual representation of features from text. These approaches are mostly done using information retrieval techniques including Latent Semantic Indexing and Topic Modelling.

An early work in mining product description to uncover feature descriptors for feature recommendations is done by Dumitru et al. [21][23]. In this work, they crawl *softpedia.com* for product raw descriptions spanning 20 product categories. These descriptions are structured in a consistent way facilitating the identification of feature descriptors. Feature descriptors are represented using vector containing TF-IDF measures for all possible terms in the text. Then, they use incremental diffusive clustering (IDC) algorithm to cluster terms found in descriptors into features concepts. IDC is an iterative clustering algorithm devised by the authors and claimed to have best results for this

context. In each iteration of the clustering algorithm, the best cluster is retained. In the next iteration, only dominant terms belonging to the retained cluster is removed from the search space. The goal of the clustering is to generate clusters (i.e. features) containing terms that may overlap. This work was later adapted to build feature models [24].

Bakar et al. performed a systematic literature review of research tackling feature extraction from natural language software artefacts to be used in software product lines[61].

In the context of **App Stores**, which are ripe with unstructured textual information regarding offered software, feature identification and location serves many purposes including analysing feature adoption behaviour, detecting malicious software, and feedback/bug report summarisation and analysis.

In the line of work done by the UCLAppA[1] team a feature is defined as "a claimed functionality offered by an app, captured by a set of collocated words in the app description and shared by a set of apps in the same category [62]." Harman et al. have been the first to carry out feature extraction from app descriptions by identifying feature list locations in the description that follows a conventional pattern [62][63]. Using NLP collocation finder, they identify bi- and tri-grams that consists of words that commonly occur together. A third and final step is carried out to cluster features according to the number of words they share, minimizing the redundancy of features. These features have been successfully used to study the relationships between features and rating/popularity/price [62][63] to predict customers' reaction to app features [64], and to study cohesive clusters of features that provide maximum value to the app [65]. Chapter 6 builds on this technique to study features migratory patterns [66]. Drawbacks of this approach is its reliance on feature listing conventions followed by developer, and its disregard of semantics (i.e. different words that have similar meaning are treated differently). To overcomes these limitations, semantics are included by employing the WordNet similarity score into account in a technique proposed in Chapter 4 which is then used to cluster mobile apps [67]. Chapter 5 compares the performance of this technique with several others for the task of clustering.

Martin et al. [68, 69] use app's description and what's new content (i.e., release text) of over 26,000 app releases from Google Play and Windows Phone stores in order to investigate the relationship between most prevalent terms/topics (extracted by using TF-IDF and topic modelling) and impactful releases revealed by causal impact analysis. The results highlight that releases significantly affecting app success have more descriptive release text and also make prevalent mentions of bug fixes and new features.

Feature discovery from app descriptions has been used to detect anomalous apps. Using topic modelling, Gorla et al. [70] clustered applications based on their advertised functionalities apparent in their descriptions. These clusters were then used to detect common APIs used within a functionality cluster. Apps that belonged to a certain cluster based on their textual functionality descriptions that used non-common APIs for that cluster were then flagged as potentially dangerous. While they report that existing app store categories are too coarse grained to carry-out this task, they do not provide evaluative measure regarding the resulting clustering before evaluating the anomaly detection system. Similarly, Kuznetsov et al. [71] leverage app description to assess their safety

[1] http://www0.cs.ucl.ac.uk/staff/F.Sarro/projects/UCLappA/home.html

using topic modelling. They compare mobile applications' advertised features (from descriptions) and their interface text to detect possible anomalies and misbehaviour. To this end, they generate an LDA topic model over the dataset's app description, then they use the model to assign topics to the apps using their interface text. If a mismatch is detected between an app's description topic distribution, and the topic distribution of its interface text, then it is flagged as a risk.

Extracting features is also useful for the task of tackling and analysing user reviews. This aims to guide developers in sorting app reviews according to the features they evaluate, request or complain about (or a variation of this taxonomy). Research also looked into collating user feedback into specific issues/features to help prioritise and summarise the large amount of feedback. Whereas a large amount of research was dedicated to classifying reviews according to their general topic or purpose (e.g. [72][73],[74],[75] and [76]), we focus here on research that aim to extract software-specific features.

Guzman and Maalej [77], tackle the problem of extracting features from user reviews by identifying collocations. The goal is to provide fine-grained feature-level user rating by conducting sentiment analysis over high level grouping of related extracted features. After identifying bi-grams of words that co-occur using the likelihood ratio test. Synonyms are factored into the feature extraction by employing WordNet which also helps in identifying misspelled words. Finally, they group features into supersets of similar functionality using Latent Dirichlet Allocation (LDA). They evaluate their technique by comparing the results with human coded dataset. This technique's measured precision is 0.601 while recall and F-measure are 0.506 and 0.549 respectively. Bakiu and Guzman [78] use this technique, coupled with sentiment analysis, in order to specifically detect users' degree of satisfaction regarding usability and user experience. This showed that this algorithm can be adapted to generate features of a specific domain in cases where search by keyword might not be accurate enough as done in [79] where a search for all words relating to advertisements (i.e. ad and advert) successfully returned all user reviews discussing the app's advertisements.

Research has found that identifying requirements from natural language using syntactical sentence patterns can produce good accuracy (though low recall) when applied on software quality concerns (non-functional requirements) [80]. Therefore, the following papers employ a variation of identifying sentence templates or syntactical patterns.

Johann et al. [81] extracted features from mobile app descriptions and user reviews using a set of manually compiled linguistic part-of-speech and sentence patterns. Their approach, dubbed SAFE, then matches features mentioned in user reviews to those in the app's description relying on shared terms and WordNet semantic similarity of terms, for ease of review retrieval and management. Evaluating their approach over a manually labelled set, their feature extraction algorithm achieves 0.56 precision and 0.43 recall improving about 50% of the semantic-agnostic state-of-the-art [63]. As to feature extraction from app reviews, they report 0.24, 0.71 and 0.36 for precision, recall and F-measure respectively greatly improving the recall of state-of-the-art by [77] in terms of recall but not much else (from 0.28 recall).

A closely related endeavour is done by Iacob and Harrison [82] where they mine feature requests from user reviews. Feature requests are first identified as following a common linguistic rules containing a pre-defined set of key words (e.g. add, allow and if only) resulting in a total of 237 rules. This approach performs very well with 0.85 precision and 0.87 recall. Then LDA was conducted

over the entire set of feature requests to identify general topics of user requests. Although their work succeeds in automatically extracting feature requests, LDA does not work well in identifying feature topics with fine granularity.

Panichella et al. [76][83] also employ linguistic patterns that were manually compiled by analysing a sample of 500 user reviews. They identify 246 recurring sentence patterns that were used to request features thus enabling building an automatic NLP module that can automatically extract sentences following any of the patterns. Similarly, Gu and Kim [84] base a user review summariser (SUR-Miner) on linguistic patterns comprising of certain orders of part-of-speech tags in the sentence. In their approach, they isolate features 'aspects' and the user opinion with regards to it. Their feature/opinion extraction model achieves 0.85 F1-score when tested over the user reviews of 17 Android applications.

In the work presented by Scalabrino et al. [85], not only is useful information extracted from user feedback, but an attempt to cluster feedback reporting the same problem together for prioritisation and summarisation purposes. To this end, the framework, after successfully categorising the type of review, represents the feedback using the vector space model coupled with DBSCAN as the clustering algorithm. They evaluate the resulting clusters using MoJoFM (Move Join Effectiveness Measure). Their algorithm scores an average of 75% and 83% in clustering bug reports and feature requests respectively.

These search endeavours do indeed point to the importance, transferability and timeliness of the feature conceptualisation problem especially in modern deployment and online based software management portals where collaboration and textual-based exchange of information among the software stakeholders is common. App stores are of particular relevance to this problem as they open a line of communication among developers and users, as shall be shown in Chapter 3.

2.3 Automatic Software Categorization

Categorization is the activity of grouping data points according to their similarity. Cluster analysis is concerned with studying effective unsupervised techniques of detecting similarity and subsequently grouping of data such that data points in the same group are more similar to each other than members in other groups. On the other hand, classification is a supervised method that uses a training set to find the set of features that better describe the similarity/difference between data points and then perform the classification based on the features exhibited by these data points. Cluster analysis is an exploratory endeavour that helps uncover underlying, seemingly unknown, segmentation of data. Classification on the other hand, assumes that final classes are not only known, but that a labelling of the data can be obtained to train a prediction model to carry out the classification.

Clustering and classification are employed in many research endeavours to help solve software engineering problems. It helps detecting anomalies, grouping of requirements into coherent sets, detecting semantically related code chunks, and classifying systems into application domains.

This section shall review works pertaining to clustering and classification of software systems and related artefacts in the context of software engineering. This area of work is closely related to this thesis as discussed in Chapters 4 and 5. This section is organized as follows, at first, a scan of early works investigating techniques to categorize software systems in Chapter 2.3.1. In this line of work, the software system, or a software library is the unit of classification. The second section 2.3.2

focuses on categorizing mobile applications and the employment of classification and clustering in mobile-related software engineering issues.

2.3.1 Categorization of Software

One of the earliest works investigating the detection of similarity in a software system context was published in 1991 by Maarek et al. [48]. Theirs is among the first published works that uses information retrieval techniques conducted over source code to help comprehension. This work paved the road for further investigation of using such techniques to aid in software engineering tasks.

In 2002, Ugurel et al. [86] trained a support vector machine (SVM) classifier to classify open source projects into their application topics. The goal of such classifier is to aid the, then manual, task of categorizing open source projects in online repositories. The classification features were tokens extracted from source code, comments, read-me files and header names. The training set was obtained by extracting the topic domain from the online source code repository. Their classifier, at best case, achieves an accuracy of 67%.

MUDABlue is a tool proposed by Kawaguchi et al. [87][88]. The aim of the tool is to categorize projects found in online open source repositories for ease of search and access. Providing this classification of software according to its application domain facilitates to developers ease of access to related projects for learning best practices and code reuse. The tool relies on three principles for software categorization: automatic determination of categories, non-mutual-exclusiveness of categories, and finally, sole reliance on source code for performing the categorization. The tool first extracts a set of prevalent concepts from identifiers in code. Then, extracted identifiers are clustered using Latent Semantic Analysis similarity. These clusters will later correspond to categories. Then software is categorized into a certain category if it contains one or more identifier in the corresponding identifier cluster. To determine whether MUDABlue succeeds in categorizing software systems, they test it to categorize 41 projects with over 2 million lines of code. Then, they compare the tool's assigned category with a manually assigned one. They report the calculated F-measure of 0.72.

A variation of this technique that replaces LSA with Latent Dirichlet Allocation (LDA) was proposed by Tian et al. [89]. LDA is a probabilistic model that assigns mixtures of topics to documents (software systems in this case). Each topic is defined as a certain distribution of words. In their work, they calculate LDA topics over the software corpus extracted from SourceForge with their actual categories as ideal set, then they cluster topics according to cosine similarity, finally, a system is said to belong to a certain category if one of the category's topics belongs to that system with a probability above a pre-defined threshold. When using 40 topics generated by LDA, the techniques clusters them into 33 categories. Of these categories, 19 are directly related to actual categories in SourceForge, 7 are based on architectures or libraries and the remaining 7 do not seem to carry meaningful concepts. This algorithm was then evaluated over the same data set used by MUDABlue in order to directly compare the two tools. Their tool was able to categorize software with 74% precision and 72% recall. When comparing the results with MUDABlue, they both perform similarly.

Härtel et al. [90] tested classifying Java APIs only using class and method names represented in a vector space model using cosine similarity and a hierarchical clustering algorithm. They compare the performance of using TFIDF, LSI and LDA finding that TFIDF performs best at identifying API similarities, whereas LSI does a good job at producing API tags; LDA, on the other hand, has

the largest amount of uncertainty. Altarawy et al. [91] use topic modelling (LDA) with hierarchical clustering over the source code to build a programming language-agnostic classification tool. They achieve 67% precision, 85% recall and 75% F-measure. They also compare their results to those produced by [89] reporting improved performance.

While all of the previously discussed methods rely on the openness of the code, classifying closed-code software remains a problem. In addressing this problem, McMillan et al. [92] extends Ugurel et al. [86] to replicate the classification of open source repositories only using API calls in the form of class and package names in conjunction with SVM, Naive Bayes and Decision Tree classifiers. They show that the SVM classifier, using significantly less features that that of Ugurel et al.'s, can achieve comparable performance. This work is then extended to use Latent Semantic Analysis that shows improvement of categorization results [93]. Similarly, Linares-Vaśquez et al. [94] propose mining a system's third party API calls to automatically detect and classify the system's application domain. In their work, they investigate whether using API calls is similar to or better than using entire source code, and if so, which machine learning technique would yield better classification results and at what level of granularity should API calls be used (API package, class or method level). First, they use Expected Entropy Loss (EEL) to identify the attributes (API calls) that will be used for classification. EEL is an algorithm to calculate which attribute better describes a certain category. Then they used supervised classification technique. The reason of this choice is the fact that categories are known in advance, hence a training set is available. They train and test their algorithm using 745 closed coded Java project and 3,286 open-source Java projects. After testing five different machine learning techniques they find that Support Vector Machines (SVM) is the best performer. Their algorithm achieved 40.24% F-measure for closed coded project and 55.31 F-measure for open source ones. They also report that API method names and packages perform better features than classes.

Wang et al. [95] propose the ability to categorize software solely based on the software profile page found in the repository which enhances the scalability of the categorization technique. While all previously discussed methods consider limited number of groups for categorization they propose a fine-grained Hierarchical categorization. In their work, they use both collaborative tags and application description to perform the categorization. They perform the analysis over 18,032 systems crawled from SourceForge, Ohloh and Freecode. They extract important classifying words from the software profile page using TF-IDF over both software description and tags. They then use SVM to categorize systems into a hierarchical category tree that was manually adapted from the one provided by SourceForge. They find that best way of classifying is by incorporating both software tags and description which yielded a precision of 0.68 and an F-measure of 0.62. However, they find that the precision is greatly improved if the classification dis-regards the hierarchy and includes only one level of categorization resulting in 0.85 precision and 0.77 F-measure. They also compare using only data found in description and tagging with using API calls to see which one provides better features for categorization. They show that both these provide similar F-measure; using API calls achieves an F-measure of 0.6464 while using software descriptors achieves 0.6696 F-measure. Similar to their work is that of Escobar-Avila et al. [96] where they carry out the automatic assignment of category to Apache libraries using the vector space model weighted using term frequency - inverse document frequency (TF-IDF). They extract features from names of classes, attributes, methods and

local variables from the software bytecode. They show that this technique correctly categorizes 86% of the dataset.

2.3.2 Categorization of Mobile Applications

Mobile applications market, as one the largest online applications repositories, has received much attention in research in order to automate and regulate its categorization. A large number of these studies used unsupervised classification techniques.

Research, for the most part, aimed to automate the existing categorization of mobile application markets. This kind of research deem the categorization pre-known thus, most use supervised classification. Berardi et al.[97] implement a SVM classifier to aid users in app discovery. Chen et al. [98] tackle the problem using an online kernel learning approach that extracts features from app titles, descriptions, categories, permissions, images, rating, size and reviews. Vakulenko and Müller [99] also attempted to replicate app store categorization of mobile apps automatically using topic modelling over their description. Surian et al. [100] use a variation of topic modelling, with vector normalisation, of the app's descriptions to test whether a sample will generate an unsupervised classification immitating that of the app store. To evaluate their technique by calculating the overlap between actual category and topic model distribution.

On the other hand, research used unsupervised classification (clustering) to group mobile application either based on different criteria or to achieve a far finer granularity than the taxonomy provided by the app repository. Kim et al.[101] used cluster analysis to study and analyse mobile application service networks showing the relationships between software capabilities and categories. Zhu et al.[102] also proposed a solution to automatically classify mobile application in which they leverage contextual information mined from usage logs in addition to textual description of apps. Lulu and Kuflik [103] carry out this same task while incorporating a method of detecting word semantic similarity using WordNet. Mokarizadeh et al.[104] used topic modelling to extract features that were fed into a k-means clusterer to generate a clustering of applications based on their functionality. All previous works were not evaluated based on human judgement but rather internal cluster quality measures (except for Zhu et al [102] whose clustering was limited to a pre-defined taxonomy).

As part of their large study of the entirety of Google Play, Viennot et al.[105] introduced a simple approach that uses MD5 hashes to identify similar applications that detects clones and apps with duplicate content. Linares-Vásquez et al. [106] use the approach of McMillan et al. [107] to automatically detect similar mobile apps. Their approach employs Android-specific semantic features such as intents, user permissions and hardware sensors uses. Unsupervised clustering of the mobile app market has also been proposed to enhance sampling applications for research purposes by Nayebi et al. [108]. They propose an approach that uses DBSCAN clustering technique carried out over features extracted from app specific metadata such as topic models from descriptions, number of downloads, ratings and reviews. Gorla et al. [70] used topic modelling to cluster applications based on their general functionality. This clustering was then used to establish a baseline of the type of user-granted permissions required by the app for each particular application domain. This facilitated detecting outliers that require permission not required by other similar apps, thus deemed suspicious. They report in their study that clusters acquired using an unsupervised technique performs better than using the app store's existing categorization in defining groups of apps that share similar

functionality. All of the above endeavours, regardless of the goal, do not specifically cluster mobile apps based on their functionality at a finer granularity of specific provided features, as proposed in section 4.

Several of the work conducted over mobile applications used other types of apps' metadata, especially user-granted permissions that the app requires, to detect anomalous or malicious apps (e.g. [109] and [110]).

Table 2.1 contains a comprehensive table depicting the research done, to the best of my knowledge and until the time of writing this thesis, that conducts software classification and clustering in chronological order. The table shows a total of 31 papers. Of these, 10 papers use supervised technique while 21 use unsupervised methods. Of the 21 that use unsupervised, 10 are using mobile application dataset.

Table 2.1: List of papers investigating classification of software and/or related artefacts.

#	Authors	Title	Venue	Technique	Dataset	Evaluation Method
1	Maarek et al.	An Information Retrieval Approach for Automatically Constructing Software Libraries	TSE'91	Built a search engine based on attributes automatically extracted from natural language documentation using lexical affinities that were used to conduct hierarchical clustering.	AIX 3 (IBM RISC operating system) dataset consists of 1100 components and 800,000 words of documentation. Truth set consists of 30 queries and the best set of retrieved documents for each query based on expert judgement.	Measured precision and recall. Evaluation criteria were user effort, maintenance effort, efficiency, and retrieval effectiveness. In addition, conducted a comparison with another tool (INFOEXPLORER an IBM RISC system/6000 CD-ROM hypertext information base library).
2	Ugurel et al.	What's the code? automatic classification of source code archives	KDD'02	Used Support Vector Machines (SVM) to classify archived source code into eleven application topics and ten programming languages. The features used for SVM were tokens in the source code and/or words in the comments. For topical classification, they generated features from words, bigrams and lexical phrases extracted from comments, readme files and header file names extracted from source code. Features were selected using Expected Entropy Loss.	Open source projects from online repositories. Training dataset: 100 source code files. Test dataset: 330 source code files from 11 categories.	Evaluated using 5-fold cross validation. The goldset is the topical categories from the original software archive. They reported the performance of their categorization technique using the accuracy, false positive (FP) and true positive (TP) scores.
3	Kawaguchi et al.	MUDABlue: An automatic categorization system for Open Source repositories	APSEC'04	Automated the categorization process including defining the taxonomy using their technique 'Unifiable Cluster Map'. Used Latent Semantic Analysis with cosine similarity where projects represent documents and identifiers within the source code are tokens (words).	Sample data from SourceForge consisting of 41 C programs from 5 categories. A manually labelled goldset.	Measured precision, recall and f-value against manually labelled goldset. Additionally, compared resulting categories with the categories in SourceForge in addition to the newly emerging categories from the tool (40 total categories). They compared the results of the tool with Maarek et al.'s GURU (see number 1) and Ugurel et al.'s SVM.
4	Kuhn et al.	Enriching Reverse Engineering with Semantic Clustering	WCRE'05	Clustered a system's entities (classes, methods) using LSA as a similarity measure (cosine). Clustering was done using hierarchical clustering (average linkage).	Core classes and plug-ins of Moose (a reengineering environment) and JEdit.	Applied the clustering at different levels of abstractions on case studies written in different languages. Then discussed the results of applying the clustering to the case studies.
5	Niu et al.	On-Demand Cluster Analysis for Product Line Functional Requirements	SPLC'08	A semi-automatic technique for extracting SPL's functional requirements profiles (FRPs) from natural language documents as verb-direct Object pairs. Which were clustered to form features using 'overlapping partitioning cluster algorithm (OPC). They devise their own similarity measure that capitalizes on the common attributes between every two FRPs.	Proof-of-concept was done for two sample applications, both have textual descriptions explaining system functionalities.	Tested on a Library management system example consisting of 10 FRPs (data points) and clustered them into 6 clusters three of which have more than one FRP with further discussion. Proof-of-concept applied to two larger application with an in-depth discussion of how the resulting clustering is going to help two stakeholders: users and designers.
6	Duan et al.	A consensus based approach to constrained clustering of software requirements	CIKM'08	A semi-supervised framework based on a combination of consensus-based and constrained clustering techniques. This, they report, performs better than k-means and hierarchical approaches.	Six TREC (Text Retrieval Conference) datasets and two sets of feature requests (STUDENT and SUGAR).	Used the Normalized Mutual Information measure (NMI) to measure how well the new approach agrees with a reference (ideal) clustering that was manually created.
7	Tian et al.	Using Latent Dirichlet Allocation for automatic categorization of software	MSR'09	Categorised open source repositories using Latent Dirichlet Allocation (LDA). The corpus used is the set of identifiers and comments in the source code. Clustered resulting topics using cosine similarity. Once the categories were populated with topics, the software systems were assigned to the categories if they exhibit a topic above a certain probability threshold, then it is said to belong to the category containing that topic.	A set of 41 open source systems written in C and a set of 43 open source software systems written in other programming languages.	Compared the tool (LACT) with MUDABlue (number 3) using a set of 41 open source systems written in C. Then studied LACT's performance when categorizing software systems written in different languages using a set of 43 open source software systems. They computed the Precision and recall using the manual categorization SourceForge as an ideal set.
8	Shabtai et al.	Automated Static Code Analysis for Classifying Android Applications Using Machine Learning	CIS'10	Used 8 different supervised machine learning classifiers, various numbers of features and two different feature selection methods.	A dataset of 2285 free apps from the Android app store in the Tools and Games categories.	Performed 10-fold cross validation repeated 5 times for all the combinations of the algorithm. They calculated the False Positive Rate (FPR), Accuracy and Area Under the Curve (AUC) considering the actual app store categorization as the true labels of the dataset.

#	Authors	Title	Venue	Technique	Dataset	Evaluation Method
9	McMillan et al.	Categorizing software applications for maintenance	ICSM'11	Used API calls for categorization. They performed supervised classification using three different machine learning algorithms: Decision Trees, Naïve Bayes and Support Vector Machines. Features considered: terms, classes and packages using Expected Entropy Loss to select highly effective attributes.	A dataset of 3286 open-source projects from SourceForge and 745 closed-source applications from ShareJar.	Performed 5-fold cross validation tested on two software repositories one open sourced and one closed sourced. They also compared the performance of their approach with that of Ugurel et al. (see number 2) on 330 application from SourceForge and 1353 from IBiblio. They used TPR and TNR to measure the performance of the system using the existing categorization as the goldset.
10	Sanz et al.	On the automatic categorisation of android applications	CCNC'12	They used Machine learning (Bayesan Networks, Decision Trees, K-Nearest Neighbour and Support Vector Machines) to classify the data. They extract features from app permissions, strings contained in the application source (w/ tf-idf), and app meta data from the app store: rating, number of reviews, size of application. Features are selected using Informatio Gain.	820 apps from 7 different categories from the Android App store.	They perform 10-fold cross validation of the system. To measure the performance, they calculate the AUC, TPR and FPR. They assume that the original categorization is the true label.
11	Linares-Vásquez et al.	On using machine learning to automatically classify software applications into domain categories	ESE'12	[this is journal extension of number 9 McMillan et al. please see number 9 for details]		
12	McMillan et al.	Recommending source code for use in rapid software prototypes	ICSE'12	Performed clustering of software online description to refine sets of extracted features. Use Incremental diffusive clustering (IDC) that is based on spherical K-Means.	Online pages of 117,265 products categorized under 21 categories and 159 sub-categories from SoftPedia.	Evaluated the entire system as a feature recommendation system. The resulting feature clusters sanity is not evaluated.
13	McMillan et al.	Detecting similar software applications	ICSE'12	Identified a set of semantic anchors, mainly API calls, exploiting their inheritance hierarchies to calculate the similarities. They claimed would perform better than identifier names/comments since it avoids the synonymy/polysemy problems. They used LSA to reduce dimensionality with tf-idf weighting. Upon building a similarity matrix, and given a query, the search engine can retrieve 10 results ranked based on similarity to the query.	8,310 Java Applications from SourceForge.	Evaluated using manual relevance judgments by experts. Experimented with 33 Java programmers (students) to evaluate the system. Each participant rated the retrieved result assigning it a level of confidence using a four level Likert scale. The results show that users find more relevant applications with higher precision with their tool (CLAN) than those based on MUDABlue (number 3)
14	Kim et al.	Mobile application service networks: Apple's App Store	Service Business 2013	Built a keyword vector space, then an association matrix using cosine similarity. Then conducted network analysis of the service network resulting from the association matrix. They used UCINET (a networking analysis program). Cluster analysis was implemented to group the app categories based on pattern of network characteristics (using k-means)	The metadata of a set of 100,830 apps from all 20 categories in Apple's app store.	For the clustering part, they provide the centre of each cluster and the p-values showing the significance of ANOVA statistical analysis, as well as a suggested naming and members of each cluster.
15	Wang et al.	Mining Software Profile across Multiple Repositories for Hierarchical Categorization	ICSM'13	Constructed a category hierarchy containing more than 120 categories organized in four levels based on the predefined categories in SourceForge. Then used an SVM-based categorization approach that classifies software hierarchically based on the software online profile mined from 3 different sources for the same software project.	The software descriptions and collaborative tags of 18,032, 9,813 and 10,357 software projects from SourceForge, Ohloh and Freecode, respectively.	Performed 5-fold cross validation. Used a modified version of precision, recall and F-measure to better suit the hierarchical nature of the classification. They used SourceForge categorization as the baseline. Compared the performance of their classification vs. using API calls only, they find very similar F-measure scores.
16	Zhu et al.	Mobile App Classification with Enriched Contextual Information	IEEE Transactions on Mobile Computing 2013	Trained a Maximum Entropy (MaxEnt) classifier using a novel set of classification features involving app contextual information mined from user logs (pattern mining) and description snippets from the web represented in a Vector Space Model using Cosine similarity and LDA.	Usage logs from the Nokia phones of 443 volunteers using 680 unique mobile apps.	Manually defined a two-level app taxonomy based on the categories found in the Nokia store containing 9 level-1 categories and 27 level-2 categories. Three human labellers to manually labelled the 680 apps in the dataset applying a voting scheme. The evaluation metrics used are precision, recall and F-Measure.

#	Authors	Title	Venue	Technique	Dataset	Evaluation Method
17	Lulu and Kuflik	Functionality-based clustering using short textual description	IUI'13	Build vector space weighted using TF-IDF of verb phrases extracted from app descriptions. Synonyms lookup done using Verbnet and Wordnet. Clustering was conducted using Lingo, Suffix Tree Clustering and Bisecting k-means.	A dataset of 120 fast-growing high-ranking Google Play apps.	The clustering algorithms were compared in terms of clusters overlap, cluster diversity, quality of cluster labels and number of unlabelled apps.
18	Mokarizadeh et al.	Mining and Analysis of Apps in Google Play	WEBIST'13	Cluster apps based on functionality using topic modelling (LDA). K-means was used to cluster feature vectors in conjunction with cosine similarity.	Two datasets of 21,064 apps from 24 categories, and 450,933 apps mined from Google Play and Android Market.	Quality of clustering effort was measured using the harmonic mean of the internal similarity and external similarity.
19	Vakulenko et al.	Enriching iTunes App Store Categories via Topic Modelling	ICIS'14	Used LDA to find the probabilistic distribution of topics assigned to each app. Built clusters by setting the probability threshold to 0.2, thus one or two topics were assigned per app.	The Apple's app store metadata of 600,000 apps.	Measured the agreement of their classification with the existing one in Apple app store by measuring an overlap coefficient.
20	Viennot et al.	A measurement study of google play	ACM SIGMETRICS'14	Introduced a simple approach to identify similar applications in Google Play to detect clones and duplicate content. They built a feature set of resource names and asset signatures generated by hashing each asset in the app (using MD5). They detected clones by measuring Jaccard similarity between the two apps' features sets.	The metadata and binaries of 610,000 free applications in the Android app store.	Evaluated the accuracy of their approach by randomly sampling 400 applications flagged as similar, and manually inspecting it. They found that %5 were false positives (FPR).
21	Gorla et al.	Checking app behaviour against app descriptions	ICSE'14	Used Topic modelling to extract main topics for each application then build a vector space of apps and topics which was used to cluster using k-means. They used silhouette to identify the best number of clusters.	The metadata and binaries of 32,136 from the Android app store (top 150 free applications in each category).	No evaluation of the sanity of clustering results.
22	Berardi et al.	Multi-store metadata-based supervised mobile app classification	SAC'15	Used a Support Vector Machine classifier (SVM) with a set of features derived from the app metadata: app category, name, rating, number of reviews and size. Features are selected and weighted according to IG (information Gain)	A dataset of 5,792 apps from both Apple and Android app stores.	Assessed the performance of the classifier using Precision, recall and F-measure. They used the actual classification of the dataset (from the original app store) as baseline.
23	Chen et al.	SimApp: A Framework for Detecting Similar Mobile Applications by Online Kernel Learning	WSDM'15	Used an online kernel learning approach that fuses multi-modal data in app markets by learning the optimal combination weights from streams of training data. The kernels extract features from apps title, description, category, developer, update text, permissions, images content rating, size and reviews.	21,624 apps from 42 different categories in Google Play.	Comparison against gold set: Rank based metrics: Precision@K and mean Average Precision (mAP) The ground truth of the similar apps of any query apps is collected by building an 'app-app relevance matrix' which is built using the list of 'similar' apps crawled from the app web-page. Human evaluation: Used 32 query apps and for each, a list of 10 'similar' apps as detected by Google Play store and another 10 similar apps detected by their technique. The authors labelled the similarity independently. They find that their technique achieves better results.
24	Escobar-Avila et al.	Unsupervised Software Categorization using Bytecode	ICPC'15	extracted the names of classes, attributes, method and local variables from bytecode. Then compiled into a single text file describing the corresponding library. Libraries were clustered based on the semantic similarity of their corresponding documents using a vector space model with TF-IDF weighting. Used software profiles (description, categories and tags) to label resulting clusters.	A dataset of 158 libraries from the Apache Software Foundation repository and 15 non-ASF libraries.	Manual evaluation: 17 developers evaluated the relevance of the assigned categories. It suggested at least one relevant category for non ASF libraries. The tool (BUCS) correctly categorized 86% of the java software libraries in the Apache repo.
25	Linares-Vasquez et al.	On Automatically Detecting Similar Android Apps	ICPC'16	Extends McMillan et al. (see number 13) to cluster Android apps. Extracted Android-specific semantic anchors: intents, user permissions and sensors. Each semantic anchor and its frequencies are represented as a separate Term-Document-Matrix (TDM) at app level. Used LSI on each matrix to extract latent concepts for each semantic anchor.	A dataset of 14,450 free apps from the Android App store.	Manual evaluation: Asked 27 participants to rank the similarity between a target app and a set of potentially similar apps. The participants evaluated 12 apps belonging to different domain categories, and their top-3 similar apps (in the same domain category) retrieved by each one of the tool's instances.

#	Authors	Title	Venue	Technique	Dataset	Evaluation Method
26	Nayebi et al.	More insight from being more focused: analysis of clustered market apps	WAMA'16	Used unsupervised clustering technique DBSCAN. Extracted features include: descriptions topic model (LDA), number of downloads, rating, number of reviews, number of five star raters, number of one star raters and size. Included development attributes (from Github): Release cycle time, number of releases, lines of code, number of commits, number of contributors per release, churn per release, number of post release issues and number of addressed issues per release.	940 apps from F-Droid (open source Android apps repository)	Use variance improvement to evaluate the quality of clustering.
27	Lulu and Kuflik	Wise Mobile Icons Organization: Apps Taxonomy Classification Using Functionality Mining to Ease Apps Finding	Journal of Mobile Information Systems 2016	[this is journal extension of number 17 Lulu and Kuflik. please see number 17 for details]		
28	Lu and Liang	Automatic Classification of Non-Functional Requirements from	EASE'17	Classify user reviews into four types of non functional requirements: reliability, usability, portability and performance using bag-of-words, TF-IDF, word2vec, CHI squared and AUR-BoW. Supervised classification carried out using Naïve Bayes, J48 and Bagging.	User reviews of two popular apps (iBooks and WhatsApp) from Google Play. The dataset consists of 21,969 sentences obtained from 14369 user reviews.	Randomly sampled and classified 4000 reviews to act as ground truth. Best result yielded using combination of AUR-BoW and Bagging. This ensemble achieves a 71.4% precision, 72.3% recall and 71.8% F-measure
29	Surian et al.	App Miscategorization Detection: A Case Study on Google Play	IEEE Transactions on Knowledge and Data Engineering (TKDE)	Categorise mobile applications (to detect misplaced apps) using topic modelling normalised using von Mises-Fisher distribution. Number of topics is empirically selected as the one that generates the highest silhouette score.	Synthetic (to test the proper selection of topics), semi-synthetic and real data. Read world dataset consists of 5,546 game apps mined from Google Play in 2014.	Qualitative evaluation by comparing the generated topics with the existing app store sub-categories of the 'games' category. They find that 9 out of the 18 generated topic clusters are highly similar to pre-existing categories in the app store.
30	Härtel et al.	Classification of APIs by Hierarchical Clustering	ICPC'18	Use hierarchical clustering with cosine and Jaccard similarity to cluster Java APIs. Class and method names are represented using the vector space model. They use LSI, TFIDF and LDA or a variation thereof to test effectiveness.	Curated Java APIs that are used frequently in open source projects (hosted on GitHub). Gold set is 60 APIs classified by authors, in addition of a curated list of 100 APIs and their versions.	Compare the results with a manually labelled sample, in addition to comparison to online classification provided by the Maven Central Repository of Java APIs. They find that TFIDF performs better than LDA.
31	Altarawy et al.	LASCAD: Language-Agnostic Software Categorisation and Similar Application Detection	Journal of Systems and Software	Use topic modelling (LDA) over source code and a hierarchical clustering algorithm to categorise software. To measure the similarity between two topic vectors, they use Jensen-Shannon Divergence metric.	Three labelled datasets: Baseline used in MUDABlue (See 3), baseline used in LACT (See 7), 103 applications in 19 different programming languages. Additionally, 5,220 unlabelled applications for topic extraction.	Calculate precision, recall and F-score in comparison to the labelled dataset. Over the 103 unobserved labelled projects, the algorithm achieves 67% precision, 85% recall and 75% F-score.

2.4 App Store Analysis

App store analysis is a term coined by Harman et al. [63] to denote applying mining technique to the wealth of information found in app stores. The goal is to uncover actionable and interesting information in the paradigm of developing, deploying and maintaining mobile apps. App stores promise interesting results due to the huge amount of data they contain in addition to its amalgamation of business (price, rank), customer (rating, reviews) and technical (description, permissions, release notes) information. This work since its publication has paved the way for further app store analysis as a software repository. Researchers endeavoured to: use app descriptions, permissions and ratings to identify malicious apps [70][111][112][113][114], automatically categorize apps [110][99], identify and analyse user feedback to facilitate software maintenance [82][77][115][116][117][75][118][119][120][121][122][123], prioritize software testing tasks [124][125], and other analyses incorporating software engineering activities with user assigned quality of the app [126][127]. A comprehensive review of the literature pertaining to app store analysis is provided by Martin et al. [128]. This section shall focus on reviewing app store analysis research that analyses natural language artefacts as it is most relevant to the topic at hand.

App stores provide a new frontier for ***feature-related analysis***. Features can appear in product description, release notes and user feedback (reviews), augmented with other metrics such as rating, price and rank of downloads, feature extraction and analysis from app stores proved to yield interesting results that are sensitive to technical aspects of app deployment.

In their work, Harman et al. [63] extracted features from 32,108 apps in the Blackberry app store. Extracted features span more than one app, so every features is assigned price, rating and rank of downloads that is the average of those pertaining to the apps that exhibit this feature. The aim is to uncover correlations that hold on the feature level not only on the app level. They report a strong positive correlation between the feature rating and its downloads rank.

A follow-up of this work is done by Sarro et al. [66] where features are extracted in two different time points of the app stores. Then feature behaviour is analysed in terms of its adoption in categories other than the ones it originates from.

Research has focused on tackling user reviews in order to aid maintenance tasks and release planning. In their paper, Palomba et al. [119] show that accommodating user request coincides increase in app market performance by building traceability links between commits and user reviews. They analysed 100 mobile apps finding that there is a positive correlation between apps that have high response rate to informative reviews and their subsequent increase in rating. Software and hardware fragmentations are one of the main issues faced by mobile developers. Ecosystem vendors enforce different compatibility guidelines (commonly relying on a certain mobile operating system); furthermore, mobile hardware fragmentation can complicate the development process. Han et al. [129] show that by analysing bug reports in user reviews, they were able to detect hardware and software fragmentation in the Android app store. Their findings aid in planning testing tasks in addition to informing possible vendors regarding fragmentation issues.

Iacob and Harrison [82] perform feature request extraction from user reviews to facilitate review tracking. They devise 237 linguistic rules that they show greatly cover feature requests found in app reviews. Khalid et al. [117][75] provide a roadmap for future user complaint analysis from reviews in

app stores by providing a taxonomy covering 12 types of user complaints. Villaroel et al. [121] build a machine learning tool that classifies user reviews into bug reports, feature requests and other. They further cluster the various bugs reported by users into a list of unique bugs in order to prioritise the bugs based on the volume of complaints for each bug. They use a set of 200 apps and 1000 reviews for classification; 5 apps, and 200 reviews for clustering into unique requests, and a set of 5 apps and 463 reviews for bug prioritisation. They conduct the classification tasks using random forest technique and the clustering (unsupervised classification) using DBSCAN. They report the results in terms of precision and recall scores; scoring an overall accuracy of 86% for the classification task.

Tackling the possibility of user reviews containing more than one request, Guzman and Maalej [77] bring into light the need for finely grained user feedback. When users give star rating, they are rating the entire app, however, a rating of individual features would be more useful to developers. In their paper, they propose a framework for extracting features from reviews and addition to the sentiment expressed in association with these features. They use NLP to extract features, sentiment analysis to assign sentiment to features, and finally, they use Latent Dirichlet Allocation (LDA) to group features into coherent collections of closely related features. They then evaluate their result using a manually constructed truth set. Their data set contains 7 apps and their reviews (32210 in total). The truth set a manually analysed by 9 human coders and consists of 2800 randomly sampled reviews which included 2928 features. They evaluate their approach using precision, recall and F-measure (60% precision, 51% recall and 55% F-measure).

The extraction of information from multiple similar apps promises a unified view of requirements in a specific application domain. The work done by Shah et al. [120] extends Guzman and Maalej's [77] to use, along with sentiment analysis, frequent item-set mining to fine-tune the extracted features by clustering them together along with extracting feature-based reviews from competing apps in the app store for the benefit of the developers. Lu and Liang. [122] investigate similar apps to extract four main types of non-functional requirements (reliability, usability, portability, and performance) pertaining to a specific application domain. They use machine learning techniques to classify user reviews in conjunction to a novel feature representation technique they dub: Augmented User Reviews - Bag of Words (AUR-BoW). They find that theirs, in conjunction with Bagging classification performs best with 71.4% precision, 72.3% recall, and 71.8% F-measure. On the other hand, Scoccia et al. [130] were able to successfully classify user reviews isolating ones that evaluate app permission issues using bag-of-words representation and a Naive-Bayes classifier (0.86 recall, 0.68 precision and 0.75 F-Score). Using the same technique, Pascarella [131] classify 5,000 commit messages of 9,478 mobile apps from Github after manually devising a taxonomy representing the activities of mobile software developers. The random forest classifier achieves the highest precision (75%) while logistic regression achieves the highest recall (68%).

This brief overview of apps that incorporate textual feature analysis from app stores show its scarcity. True, the concept of mining apps stores for extra information to aid software development is new. The true potential of leveraging features found in app description, user reviews and release notes is yet to be realized.

Chapter 3

App Store Effects on Software Engineering Practices

3.1 Introduction

In recent years, software distribution channels have evolved immensely. Before the internet reached the current upload and download speeds it boasts today, turnkey software was mainly distributed in a tangible storage device (e.g. as firmware or in a CD). In a time where software was almost exclusively packaged and shipped physically, the line separating customer service tasks and software engineering ones was clearer.

The rise in internet speeds and the enabling of secure online payments contributed to the advent of online software marketplaces. These portals emerged as two-sided markets [132][133] in which the developers are the content creators on one side, and end users are the consumers on the other. In the case of mobile application markets, as they are an environment for distributing operating-system specific software, they are mainly oligopolised by the operating system's vendor [4]. As in typical two-sided markets, the wealth of an application store of a particular operating system attests to its quality and grows its desirability to end users (i.e. growth of one side of the market enables the growth of the other)[133]. Consequently, platform intermediaries aimed to cultivate both sides of the market by policing the quality and safety of the displayed products for the benefit of end users and by lowering the barrier to entry for content creators. This facilitation, together with the growth of smart phone software user base and the uptake of agile development practices contributed to the growth of the mobile application development market. This caused a rise of an engineer-entrepreneur culture where engineers envision a market opportunity fulfilled by software and build a business around it with minimal initial monetary investment.

This can be observed in the large number of mobile apps conceived, coded, released, maintained and marketed by small teams of people (mostly engineers) if not one. This trend gave the rise to the culture of the 'new school engineer' who, in addition to being technologically adept, is business savvy [134].

Previous work has mainly focused on mobile developers' perspective regarding engineering aspects and implementation challenges introduced by the mobile platform [133][135][136] [137], briefly alluding to a few app-store-specific findings. The body of literature reviewed in Section 3.2 confirms that indeed developing mobile applications requires a new set of best practices in addition to, and in some cases instead of, the one cultivated for traditional software systems engineering. For

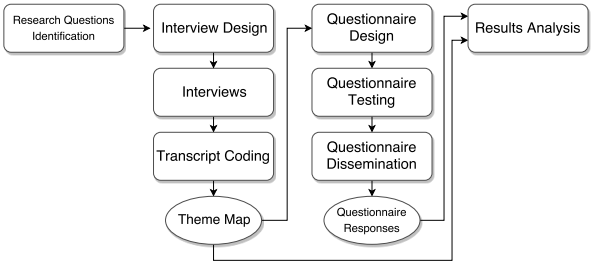


Figure 3.1: The various stages of the study.

example, Nayebi et al. [137] found that developers are aware of how the app appears to potential users in the app store based on certain release strategies. Lim et al. [4] surveyed app store users reporting the importance of packaging decisions for mobile app success. A previous study by Rosen and Shihab [138] about the questions asked by app developers on Stack Overflow revealed that the most popular topic asked was app distribution, i.e. the requirements imposed by the app store owner for publishing apps.

In this study, we interview and survey app developers, regarding their interactions with app stores. Our aim is to better understand developers' practices when making apps. This understanding will help us determine the extent to which information from app stores affects developers' decision making and observe how the app store ecosystem influences engineering tasks during the app's development process. Moreover, our findings may guide future software engineering research in app development, maintenance and evolution.

This chapter reports the most comprehensive and large scale attempt to date that surveys app developers' software engineering practices through a standard interview-and-questionnaire approach, thus highlighting open issues and challenges for the growing App Store Software Engineering community with a focus on relationships between app stores and software engineering research in several research areas including requirements, testing and software repository mining.

The used methodology combines an empirical study technique with a thematic analysis approach [139][140], which is commonly used in behavioural sciences to analyse qualitative data [141][142]. The stages of our methodology are illustrated in Figure 3.1. After formulating the research questions, data collection was conducted in two main stages. The first stage was a series of interviews with mobile development team managers and members. The interviews were then analysed and coded using deductive thematic analysis [142] and the results were used to design a questionnaire that was subsequently disseminated (stage two) to a wider audience in order to collect further quantitative data. Both the qualitative results of the interviews' thematic analysis (i.e., theme map) and the quantitative results of the questionnaire were used to explore and deduce the findings reported herein.

The findings of this study make several contributions that give evidence to support the perceived differences between app store development and more traditional software development. The principal findings about app stores themselves are:

1. **Closed loop:** The gap between developers and their users is closed by the facilities app stores provide. They denote a channel of communication that directly connects users to developers. For example, our study findings indicate that 51% of respondents frequently perform perfective

maintenance based on user's public feedback in the app store. This may be introducing new communication channels and approaches through which requirements are gathered and acted upon.

2. **Transparent market:** The ability for developers to, not only experiment with competitors' products, but also to be able to witness, in real time, the performance of these products in the marketplace, constitutes a considerably more transparent market for deployment of software systems. For example, our study reveals that 56% of respondents frequently elicit requirements by browsing similar apps in the app store. This is one of the motivations for app store mining and analysis: to better understand the marketplace, and emerging trends from competitors and overall user and developer behaviour within app stores.

3. **Tailored release strategy:** The gap between releases is shorter in app stores than for more traditional software systems deployed such as shrink-wrapped software [143]. It is also governed and constrained by a third party: the app store. Our results report that 54% of respondents adopt a release strategy that is influenced by the app store's regulations. This has implications for innovation and rapid response to technical and market developments.

One of the interesting properties of app stores is the way in which these stores cut across different software engineering concerns, raising inter-related questions and research problems for different software engineering activities [128]. More specifically, the findings of the survey have actionable conclusions for researchers and practitioners from several software engineering subfields, including:

1. **Requirements engineering research:** We report evidence that suggests that developers are almost as concerned about reported features (both wanted and unwanted) by their users, as they are with, for example, bug reports. This finding highlights the way in which app stores provide a direct communication channel between developers and their users. It is also particularly interesting to note that feedback is used by developers, to also identify *un*wanted features, hinting at the growing prevalence of the need to remove/modify features as well as the continuous need to identify new features to add.

2. **Mining software repositories:** Our study reveals that app store developers place importance on screenshots (in order to gather features for their apps). Therefore, although existing work on mining textual information from feedback and reviews and ratings is valuable (and also used by developers according to our survey), the current lack of studies on the use of images as a source of information needs to be addressed. Mining user interfaces is typically undertaken in communities such as the Computer-Human Interaction (CHI)[144] and User Interface Software and Technology (UIST) [145] communities, which may also find novel research challenges in this new area of application. We envisage that 'user interface mining' may find new applications in app store mining and analysis.

3. **Other software engineering disciplines:** We find a strong belief among developers that app stores contain information that help developers maximise the chance of success for their products, thereby motivating and partly validating app store analysis and mining as a research area.

However, the quality of code is not identified as being the strongest influencing success factor; other aspects such as user experience, visibility, novelty and brand are accorded notably higher importance by developers. This indicates, for example, that work on code 'smells' in app store code, while important, needs to be complemented by, and combined with, work on user experience and other human- and business-facing aspects.

4. **Business Community:** Our study has important findings for those working at the interface between software engineering and business considerations. That is, while developers claim it is important to have someone in their team concerned with marketing and business intelligence, they also report that this person tends to be self-taught and relies on experience rather than formal training.

Our study also has other findings for these sub-fields, and several findings concerning app testing of relevance to the wider software testing branch of research [146]. The observation that a study on app store developers can have actionable conclusions for so many different software engineering communities, highlights the crosscutting nature of this relatively recent phenomenon in software development and deployment. Clearly, as app stores develop further, there will be a need for further studies and surveys. We hope that the findings from this survey will provide a useful reference point for such further studies and analyses.

3.2 Related Work

Smartphone adoption grew rapidly in the past years. The number of smartphone users grew about 290 million users from 2014 and 2015 totalling 1.86 billion users in 2015 with projections estimating the number to grow another billion by 2020 [147]. Such popularity in adoption, gave rise to advancing the computing capabilities and hardware features of these devices. Though Smartphones are portable devices, they boast relatively advanced operating systems and great customizability using operating system specific applications. Concurrent with the rising popularity of these devices, is the thriving market of their software applications. In 2016, mobile application markets clocked approximately 150 billion application downloads and that number is estimated to triple by 2021 [148]. Hence, the end-user software development market witnessed a large shift towards mobility. As mobile operating systems and underlying hardware form-factor differ from their desktop counterparts, it naturally follows that mobile applications are also distinctive [149]. In investigating mobile app development from a software engineering perspective, research generally took one of two themes: (1) works investigating **how mobile app development differs from classical software development** and (2) uncovering **software engineering challenges rising from the mobile development paradigm**.

In investigating the distinction between mobile application and classical software development, Wasserman summarises the differences in 8 areas including the hybrid nature of applications, platform fragmentation and new user interface requirements. Minelli and Lanza [150] show that open source F-Droid mobile applications are distinct from classical software system in terms of size, ease of comprehension and degree of reliance on external libraries. This is confirmed by Syer et al. [151][143] as they report that mobile apps tend to have less lines of code, smaller development teams, and rely more heavily on the underlying platform. Additionally, they find that, regardless of

the size of the project, mobile developers tend to fix bugs faster than desktop/server software teams. In terms of code re-use, Ruiz et al. [152][153] found that 27% of the classes within the Android apps investigated inherit from another class that is domain-specific to that app's category. More surprisingly, they find that 61% of classes in apps in each category appear in more than one app. This finding contradicts those of Minelli and Lanza [150] that reported very little inheritance in investigated Android applications. This suggests a potential divide in practices between open source apps and the closed source ones.

Other aspects in which mobile application software engineering have been found to differ from classical one is in testing [154][155], release management [137] and size/effort estimation [156][157][158] [159] [160][161].

Looking into the challenges that are introduced by mobile software development, Joorabchi et al [135] followed a grounded theory approach in interviewing 12 app developers followed by a survey of 188 respondents. Among the challenges they find is platform fragmentation, lack of testing tools, closed source underlying platforms, data management intensity, frequent changes of underlying platform and third party libraries, hybrid nature of mobile apps, limited hardware capabilities, difficulty of code re-use from other platforms and strict HCI guidelines. These findings were confirmed by survey study conducted by Flora et al. [162] in addition to suggesting new challenges: The high quality expectations of users augmented with big competition and the insufficiency and uncertainty of requirements gathering for such markets. These last two challenges were also observed by Lim et al. [4]. They identify the need for newly emerging packaging requirements with price sensitivity and managing a large space of potential features even when domain-specific. Rosen and Shihab [138], by employing topic modelling over StackOverflow data, report a set of 32 main topics concerning distribution, third party APIs, data management, sensors, tools and user interfaces.

The qualitative study by Francese et al. [136] gathered information by interviewing 4 technology managers in addition to surveying 82 mobile app professionals. They report that their surveyed developers perceive mobile platform fragmentation and inadequate testing support as the two main difficulties. They also report that mobile developers concede that developing software for mobile devices is different than that of other type of software development.

In testing, Kochhar et al. [154] investigated the adequacy of mobile app testing practices. They report that 86% of open source Android applications do not contain test cases, and those that do have poor line coverage (median 9.33%). To that end, they survey 127 Microsoft developers and found that the majority (114 out of 127) use manual testing rather than automated testing tools. They compile a list of challenges that prevents developers from adopting said tools, including time constraints, poor documentation and emphasis on development. These limitations are confirmed in a study by Linares-Vàsquez et al. [155] comprising 102 respondents of open-source mobile developers.

In maintenance, Linares-Vàsquez et al. [163] investigated how mobile app developers detect and fix performance issues. They collected the responses of 485 open-source Android developers and analysed their Github repositories. They found that, in order to detect performance bottlenecks, developers are aided by user reviews and mostly rely on manual execution. Salza et al. [164] found that developers do not promptly update third-party APIs (especially non GUI-related ones) and that 89% of apps with up-to-date APIs are highly rated.

Nayebi et al. [137] shed light on possible changes in release practices of mobile applications due to the OS vendor's quality assurance process. Surveying 674 mobile app users and 36 developers, they find that only about half of the surveyed developers follow a rational release strategy. Those who do, are more likely to be experienced developers with higher success. More interestingly, it seems that how the app is perceived by users when considering downloading it affects how developers make release strategy decisions. About 44% of the surveyed developers believe that their release strategy affect the perception of users regarding that app. On the other hand, they find that end users do indeed prefer apps that are more recently updated. Based on these results, they subsequently investigate open-source app versions that are not shipped into the app store[165], introducing the concept of release 'marketability'. To this end, they surveyed 22 developers, the majority of which (95%) state that market acceptability of a mobile app release is more important than that of traditional software.

Villarroel et al. [121] and Scalabrino et al. [85] conducted a semi-structured interview with 3 project managers of software companies developing mobile apps in order to evaluate the usefulness of their tool (which extracts and clusters user reviews from the app store into bug report or new feature request). The tool was first demonstrated to the managers, then they were asked about the usefulness of reviews (do you analyse user reviews when planning a new release?), to which they answer yes. Our study confirms this prior finding with a wider basis of scientific evidence (186 respondents), it also extends it by reporting the frequency with which developers receive bug reports from user feedback in app stores and how this affects its priority (compared to other channels). This study additionally investigates further stages at which developers refer to the app store (idea conception/validation, requirement elicitation, GUI design inspiration and feedback of competing/similar apps).

While these studies partially address the changes introduced by app stores, research studies fully addressing the potential effects of the mobile application distribution model are few and far-apart. Holzer et al. [133] identify the app store as a two-sided market model consisting of developers-users where the increase of one side attracts the increase of the other and thus the market is in growth loop. They discuss the various trends such a market may introduce and their implication on developers. The centralised sales portal model, for example, carries the implication that developers have immediate access to the entire consumer base and lowers distribution costs; but on the other hand, imposes limits on the freedom of the developers. This study addresses this gap by taking software development life cycle phases as the point of analysis when surveying industry practitioners to uncover the involvement of app store in developers' practices. We believe conducting this type of research is important as more platform-mediated application markets rise in popularity (e.g. wearable apps, voice assistant skills) introducing the need to investigate whether and how deployment portals for platform-specific software affect software development practices.

3.3 Methodology

To study developers' practices when developing for mobile app stores, we followed a mixed method drawing from **survey and case study empirical research methodologies** [166].

There are two reasons supporting this choice of methodology. Firstly, case study research is a way of analysing contemporary phenomena that are difficult to separate from their natural context

[167], which is the case for app stores. However, as this is a global phenomena affecting the majority of the population (app developers), it is not strictly a case study, so we also followed a survey technique to collect data from a sample of the affected population using two data collection methods (namely, interviews and questionnaires).

This particular research will aim to be both an exploratory qualitative empirical study as well as a descriptive one [168][167]. Since it is unclear whether and how app stores are affecting developers' views and practices, it is only logical to conduct exploration activities first. This study is designed following the case study research guidelines by Runeson et al. [167] and survey research guideline by Kitchenham and Pfleeger[169].

We coupled our methodology with **deductive thematic analysis** to analyse qualitative data [139][140][142].

Thematic analysis is a method of analysing textual content and deriving patterns of thematised meaning from it. Similar to grounded theory [170], thematic analysis originated from the social sciences and has since been utilised in computer science empirical research involving human subjects. Wohlin and Aurum [171] report it as one of the qualitative analysis methodologies in their decision making structure for empirical software engineering research; Cruzes and Dyba [172] formalize an extension of thematic analysis to thematic synthesis in software engineering research.

We have selected thematic analysis due to its flexibility, ease of understanding and independence from theory. While grounded theory allows for theory-agnostic analysis of data, thematic analysis can be conducted within a theoretical framework [142]. In this study, we operate under the software engineering life cycle stages (as per the software engineering body of knowledge areas 1-5 [7]) as our theoretical framework, hence we follow deductive thematic analysis [139][142] (as opposed to inductive analysis or grounded theory[170]), using semantic themes as developers are expected to have sufficient domain knowledge eliminating the need for latent theme inference. Furthermore, our thematic analysis method follows the essentialist/realist method and not a constructionist one as we assume a simple relationship between participants' answers and meaning [139][142]. In conducting thematic analysis for this study, we follow the guidelines provided by Braun and Clarke [142].

Data collection was conducted by surveying main stakeholders of this research who are mobile app owners and developers. Similar to the scientific method of gathering information via surveys, gathering initial insights was done using interviews. Then, based on the interviews' initial findings, we designed a questionnaire and disseminated it in developers social circles. The questionnaire is important in order to validate the findings with larger consensus. Through analysing the interviews we identified areas of interest on which the chapter focuses and sheds more light. Certain patterns of responses (whether with more consensus or disagreement) that pertain to the research questions and promise valuable and deeper understanding of software engineering practices were highlighted when writing the survey questions. The survey was designed in order to investigate in more detailed and systematic way all that relates to the specified research questions.

3.4 Study Design

The stages of our study are depicted in Figure 3.1. As both empirical research and thematic analysis studies rely on proper identification of research questions, the first stage is setting the questions to

be emphasized and answered. Phase two is dedicated to the exploration and preliminary gathering of information. This is done by interviewing app developers and discussing their views and current practices. During this phase the interview structure is designed with a set of potential topics and questions to be explored in light of the research questions; then the transcripts of the interviews are analysed and coded using deductive thematic analysis resulting in a theme map. The third phase consists of collecting data by disseminating a questionnaire (designed in light of both the research questions and the insights gathered from the interviews and the theme map) to communities of interest. In the following subsections we discuss in greater details each of the phases depicted in Figure 3.1.

3.4.1 Research Questions

The research questions we aim to answer in this study cover two main areas of interest: app store's effects on software engineering processes and possible success criteria and skill sets that emerge due to app stores.

App stores are now major application deployment portals that drive the user's application discovery process. A large scale study of mobile users' tendencies by Lim et al. [4] unveiled that the majority of users rely on the app store to discover new apps: 73% of more than 10,000 respondents visited an app store at least once a month; whereas only 9% did not rely on an app store to download apps. Another major aspect of the app store is users' ability to rate the quality of apps, post feedback, comments and reviews; thus effectively opening a channel of communication between developers and users. Therefore, we believe user feedback in the app store may go beyond its recognised benefit in general markets in establishing trust of the seller's ability to deliver on their product's promise [173]. Furthermore, app stores are designed to collate similar apps together. Developers and managers are able to find apps in the same application domain including their specifications and performance in the app store environment.

This gives us ground to suspect that the app store's configuration may have an effect on the evolution of apps. which motivates our first research question:

RQ1. How does the app store ecosystem affect the software development life cycle processes? For this research question, and to set the scope of this study, we consider the Software Engineering Body of Knowledge (SWEBOK)[7] areas 1 through 5 as the software engineering life cycle stages; namely software requirements, design, construction, testing and maintenance. The Software Engineering research community has indeed highlighted the opportunities and challenges introduced by such an ecosystem [146][174]. Software engineering researchers sought to leverage information found in the app store to guide mobile developers during requirements engineering [175][176][63][177][66][178][179], testing [129][124][180][181], maintenance [92][82][115][73][182][183] and release management [121][85][69][165][184]. In posing this research question, we aim to observe the current involvement of information extracted from the app store in guiding the software engineers' effort in each of the aforementioned stages.

The study by Lim et al.[4] also reported that the market is dominated by a handful of app stores, chiefly Google Play and the iOS App Store. This accounts for high density of potential users, exposure and total downloads for apps. Furthermore, the ecosystem offers low barrier to entry giving rise to the number of offered apps making it an increasingly competitive marketplace. We investigate

whether such high competitiveness and emphasis on user acquisition and retention increases the types of considerations that the development team takes. Hence, we ask this research question:

RQ2. What new sets of best practices and skill sets emerge due to app stores, if any? The low barrier to entry also facilitated smaller teams of developers (2-5 developers) to publish apps that are still deemed viable and competitive [136][154]. We are interested to investigate the types of activities that mobile development teams carry out and the required skills that are influenced by the app store ecosystem and are outside of the recognised software engineering life cycle activities considered in RQ1.

App stores do not only provide browsing and search capabilities to users, but also employ quality measurements to provide curated content and refined 'lists' to users. Lim et al. highlighted that, in order to discover new apps, 37.6% of respondents browse the app store randomly, 34.5% check the top downloads charts and 25.8% look at featured apps. This highlights the major role that app stores play in driving success to mobile applications. To observe the involvement of app stores in success and its measurement, we ask the following research question:

RQ3. How is success perceived and measured by developers in the app store environment? Previous research seems to regard app rating as a proxy for quality (and therefore success), thus investigating the relationships between user rating scores and apps' user reviews content [127][185], release plan [69][186][187], features [179], software metrics [188], security [113], code churn [189], faultiness [190][126], underlying hardware/architecture [191][192][193] and many other software factors [194][153]. By contrast, we shift the focus to app developers and owners' view on what defines 'success', thereby uncovering other app-store-specific metrics which developers monitor. Furthermore, we aim to uncover the relationship between success and the developers' perceived quality of the app. This research question does not look into the role of the market for success, but rather investigate whether the market introduced new metrics through which developers perceive the success and quality of their app.

3.4.2 Interviews

Interviews were conducted to initially explore developers' interaction with app stores before and after release. The interview protocol is described in the upcoming section followed by a description of the participating sample and data analysis method.

3.4.2.1 Protocol

The interviews were semi-structured and followed a funnel model where questions are generic at the beginning and become more specific as the interview progresses. The funnel approach was selected to permit the conversation to flow naturally instead of controlled question-answer cycles. This allows the interviewees to be put at ease thus talking freely about what they deem important and pertinent with regards to the general topic. Then, taken from the current topic of conversation, the interviewer refocuses the conversation to a more precise subject of interest (as depicted in Figure 2). This method suited the exploratory and observational goals we required of the interviewing process.

The interview questions were brainstormed by the researchers. They, we believe, cover most aspects of contact between developers and app stores. All interviews were conducted by the one interviewer (the author), except for one which was attended by the first supervisor.

The interview plan contained 40 questions that the interviewer, ideally, sought to cover. The set

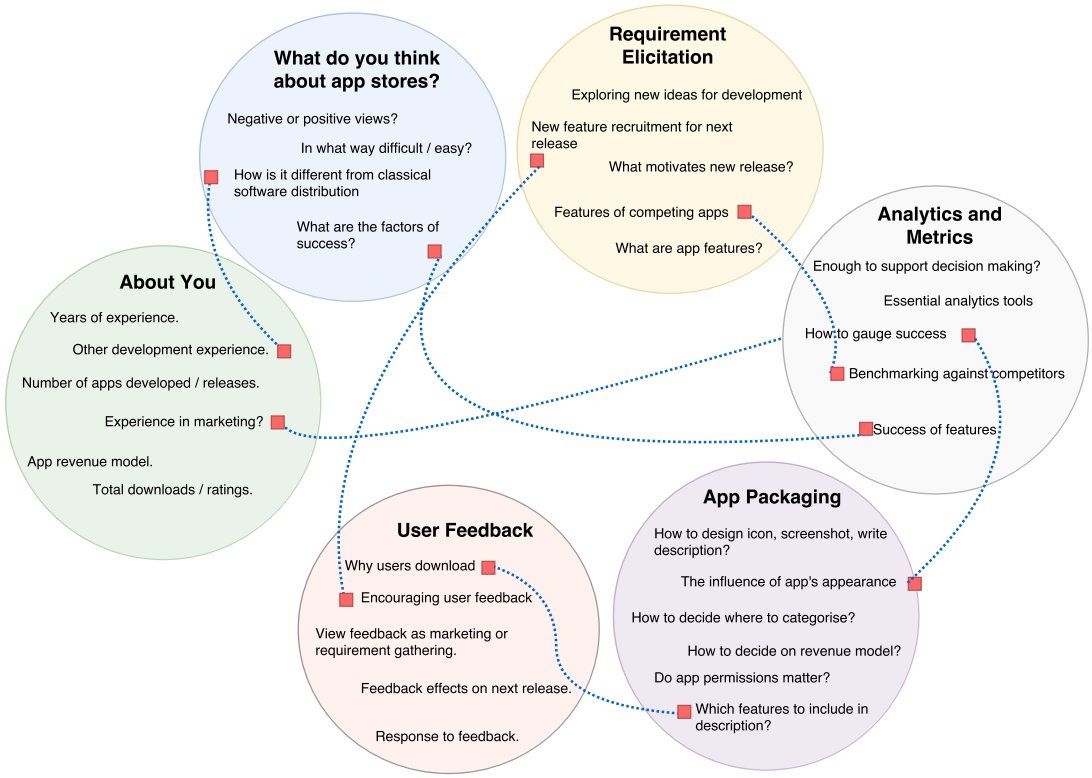


Figure 3.2: Interview design: The topics planned for discussion and the questions answered within. The dotted line resembles an example of topic flow in one interview. No structure is enforced and the questions were discussed as the respondent moved freely from one topic to another.

of drafted questions are in Table 3.1. Since the interviews were semi-structured, this plan served only as a reference for the interviewer and was not enforced. The plan highlighted some of these questions as suggested conversation starters within broad topics. Using this way of conducting the interview, interviews typically flowed smoothly and the developer answered most questions without interrupting the flow of the conversation. Figure 3.2 shows the sets of topics and questions within each topic showing an example flow of conversation within the broad topics.

3.4.2.2 Participants

The selection of interview participants relied on purposive sampling where participants had to be individuals involved in the production of an app in the app store. In selecting participants, we sought a broad set of sources for opinions with regards to team roles including developers, managers and app owners. Since this is an exploratory step, we are not aiming to make any generalizable discoveries and, therefore, relied on convenience recruitment[1] of participants.

We have interviewed a total of 10 app development team members. The interviewees were recruited through UCL Advances[2] and via social contacts. From there, a snowballing recruitment technique was carried out in which the developer was asked to recommend other colleagues and

[1] Convenience sampling is the most common sampling technique especially in laboratory psychology research where participants are mostly volunteers [195]. It is a non-probabilistic sampling method that is used for preliminary exploration of a phenomenon since it is less costly than probabilistic methods.

[2] UCL Advances is a project by the UCL Economic Challenge Fund that contains a large contact base of entrepreneurs and app owners through its UCL testing app lab.

Table 3.1: The set of interview questions.

***First* Background and demographics**	***Fourth* App Features**
Mobile platforms / app stores	What do you think are app features?
Other development experience (desktop, web, etc.)	Do you think it is easy to find something to implement?
Years of Experience (Development and Mobile)	How do you decide which features to include at the beginning?
Application Domain	How do you decide which features to add/remove later on?
Independent or corporate?	How do you gauge the success of a certain feature?
Number of developed apps (How many of these app were released in an app store?)	How do you decide which app features to include in the app description?
Dedicated marketing team (or person?)	Do you look into competitors features to identify technical trends?
Team Size	***Fifth* User Feedback**
Number of total downloads, ratings, feedback..	How do you know why users downloaded your app?
Revenue model	(Android): How do you know why users uninstalled your app?
***Second* Generic Views**	How/why do you encourage users to rate/review/share your app?
How would you describe your experience in dealing with app stores?	Do you actively respond to user reviews and Feedback? How? Why?
How are app stores different from any other deployment platform/method?	How do you respond to user reviews and feedback? What are the reasons as to why?
How does it make development easy how does it make it difficult?	To what extent does user feedback affect next releases?
What are the important factors of success in app stores?	***Sixth* Tools and Metrics**
***Third* App Packaging**	How do you measure your success over competitors (metrics)?
What do you think the most important criteria when selecting screenshots/ description/ tag line?	Do you find analytical tools provided by the app store enough to support your decisions (previously discussed)?
(Android) Do you think app permissions matter? Why?	What extra tools do you use, if any?
What do you think causes users to download your app? (same: to uninstall your app?)	What procedures do you take to advance your competitive advantage?
How do you decide on a revenue model?	..and what metrics do you think useful to enhance the app?
Did you ever have to change your app's price/revenue model? Reasons behind the change?	Do you think you need analysis and statistics that encompass the entirety of the app store?
How do you decide in which categories to release your apps? Do you think it matters?	

Table 3.2: Demographical data of the developers interviewed.

Participant	Formal Education	Years of Experience	Team Size	Team Role	Number of Apps	Success Metric
P1	Technical	4	1	All	6	-
P2	Non-Technical	-	6	Owner/Manager	1	2,000 Ratings
P3	Non-Technical	2	6	Owner/Manager	1	100 - 500 Downloads
P4	Non-Technical	7	9	Owner/Manager	1	32 Ratings
P5	Technical	6	1	All	3	200 - 250 Downloads
P6	Technical	17	6	Owner/Manager	2	140 Ratings
P7	Technical	27	17	Developer	1	1,000 Ratings
P8	Technical	10	5	Owner/Manager/Marketing	20-30	800,000 Downloads
P9	Technical	-	4	Owner/Tester	3	10,000 Downloads
P10	Technical	-	3	Developer	3	15,000 Downloads

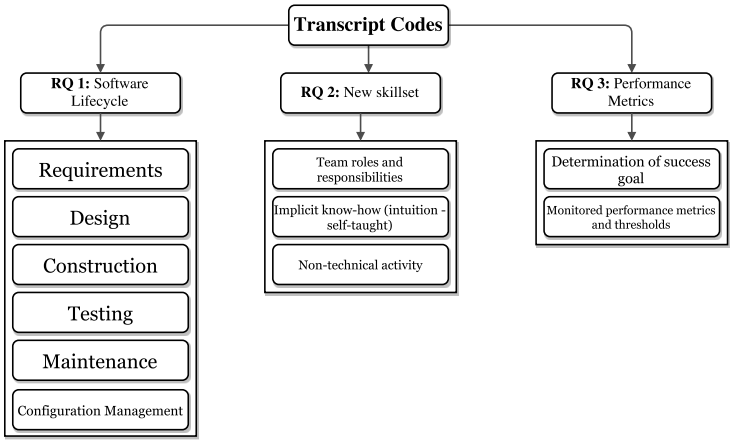


Figure 3.3: Transcript raw data codes used to tag responses: These codes represent recurring topics and certain responses of interest. This is the first stage of interview analysis. The codes and their content are then used to deduce themes in Figure 3.6

connections for the interview.

Table 3.2 reports the interviewed sample along with their respective experience demographics (strokes denote where information was not collected). Among the 10 interviewees, 7 had formal formal education in an engineering/computer science related field. Fields that are outside of the faculty of engineering were dubbed non-technical. The team sizes of participants were between 1 and 17 developers. The interviewed sample had between 4 and 27 years of experience in software development. The number of apps they have developed spans from one app to 20 apps. The degree of exposure of the sample's apps also ranges between apps that have been downloaded 100 times to apps downloaded 800,000 times.

3.4.2.3 Data Analysis

After transcribing recorded interviews, data analysis was carried out to identify emerging concepts from the corpus. This was conducted using Thematic Analysis [142]. Thematic analysis, as the name suggests, employs the concept of themes when analysing textual data. Thematic analysis requires reading the scripts intensively before coding the responses in light of the research questions. The codes are tags that interpret certain responses and help identify their topic. The codes are then clustered to form a theme map that serves as the visual representation of the main findings of the interviews.

Transcript coding: In light of each research question, the data is scanned in order to be assigned a code. Interview codes represent a certain area and tags certain attitude, opinion or knowledge expressed by the respondent regarding that area. Due to the extensive length of the

interviews, the coding process helps tag only relevant responses with regards to the research questions. The author initially tagged the interview transcripts, then compiled a list of all the codes and example instances of each of the codes. The list was then revised by two of the thesis supervisors (collaborators) to ensure their representativeness with regards to the research questions with consensus. After the revision, the author revised the tagged corpus, this has been done in two iterations. Due to the nature of the codes, they were allowed to overlap and merge/divide throughout the revision process. Figure 3.3 shows the final list of codes for each research question.

For the **first research question**, the codes are the typical software development phases according to the Software Engineering Body of Knowledge (SWEBOK) [7] as the research question investigates practices related to the software lifecycle processes. This practice of using the processes as interview codes is part of the deductive nature of the thematic analysis methodology in which the codes follow a certain pre-known taxonomy. It is also a practice done by other similar software engineering qualitative research [136][196][197].

The **second research question** centres around new emerging skill sets and roles required of mobile app development team members. The codes selected for RQ2 pertain to the possible assigned tasks for team members and to what degree do they deviate from those of a classical software team roles. The second code (implicit know-how) highlights exhibiting knowledge or certain app store-specific best practice that was not formally learned or part of the respondent's education. The final code (non-technical activity) is for highlighting any skills that the respondent is exhibiting or discussing that are not engineering-related.

The **third and final research question** pertains to performance measurements that are particularly important for mobile apps distributed through app stores. The first code (determination of success goal) highlights the clarity and determination of a success goal for the release of the app and whether that goal is app-store-specific such as being featured in the app store's main page. Other codes tag the various app store analytics and set thresholds.

Deducing Themes: Codes are then collated into a group of potential themes. A theme represents a unit of an emerging pattern of responses with regards to a certain topic and/or research question. Themes do not certainly perform a one-to-one mapping to research questions and they go through rigorous revisions as the researcher goes through the data in several passes. From analysis, thematic coding results in a theme map. The theme map reflects the main findings observed from coded responses and their relationships with one another. This process produces a rich, detailed description of the data without hindrance by data that are not relevant to the research questions. This has been carried out by the first author and then revised by three more authors in a collaborative session until a consensus has been reached (over three iterations).

3.4.3 Questionnaire

Based on the emergent topics of interest extracted from analysing the interviews (Figure 3.6), and in light of the research questions, a questionnaire was used to ascertain findings, explore new ideas and measure the prevalence of some practices. The following subsections describe the questionnaire, its design and the participating sample.

3.4.3.1 Design

In designing this survey, we followed the guidelines provided by Kitchenahm for personal opinion surveys[198]. The initial design comprised of 118 statements and questions that have been drafted in several collaborative sessions with the study's collaborators (i.e. supervisors). The survey is divided into subsections representing themes of activities in addition to the demographics section. The survey draft went through several revisions where we have removed questions that we deemed to be open to interpretation based on the respondent's experience and may be ambiguous or misunderstood. An example of an unclear question was: 'I prefer releasing an alpha/beta version of my app on the actual app store rather than one specific for testing.' (developers may not understand what is meant by 'one specific for testing' and may interpret it differently). Furthermore, we have prioritised questions with higher relevance to the software engineering community and so did not include ones such as: 'When I find an app that has a similar main functionality, I still can have a competitive advantage." After eliminating repetitive, unclear and questions deemed irrelevant (22 in total), the questionnaire ended up with 96 questions. The questionnaire is divided into these sections: Demographics, Software lifecycle (Idea conception and requirements gathering, design, construction, testing and maintenance), Emerging new skill sets and finally, Metrics. We have first conducted a pilot study where we invited developers to fill the survey in read-aloud sessions in which they read questions out loud as well as externalized their thinking process. A total of six developers reviewed the questionnaire questions and gave feedback. First of which was their complaints regarding the length of the questionnaire. Based on that, we removed a few more questions that were lower in priority; in addition to merging the last section of the questionnaire with previous sections. Additionally, we arranged the questions such that demographics only appear at the end of the questionnaire except for two easy questions that serve as a warm-up. Another valuable insight from how developers filled the questionnaire was their consistently mistaking 'design' in a software engineering process sense with the process of graphic design and building user interfaces. This and other inconsistencies in the meaning of certain terms were observed in the read-aloud sessions and were thus corrected in the survey. Two questions were deemed totally unclear and were therefore rephrased. The final questionnaire contains 11 short sections and 79 statements grouped into 42 questions[3]. The questions' answers are 5 Likert items on the Likert scale that represent degrees of agreement, frequency, interest or importance. The survey also includes multiple choice and open ended questions. We have elected to make all the questions optional in order to mitigate the challenge of the length of the survey. This means that each question has its own sample that can be a subset of the surveyed sample. By making the questions optional, we ensure the certainty of the response since no respondent has to reply in order to progress further in the survey. Another approach we used to mitigate the length of the questionnaire was branching: Based on the respondent's answers to certain questions, the control flow will skip questions that are irrelevant. For example, we ask the participant if they ever released more than one version of their app, if the answer is no, we skip questions relating to release management and perfective maintenance and proceed to the subsequent section. The questionnaire experienced response fatigue where number of respondents for questions decreases as more respondents abandon the questionnaire as they

[3]The final survey is attached in Appendix A. The online the questionnaire, as well as the complete results, can be accessed via `https://afnan-s.github.io/appa/survey.html`

advance further into it. First question garnered 185 respondents, whereas the fewest responses for a non-branching question was 107.

3.4.3.2 Participants

The survey was disseminated via posters and flyers around UCL campus, email to interest groups as well as social media. The flyers were also passed around 2 research conferences. Cold calls were also posted to several mobile developer groups in the professional social network LinkedIn[4]. The total number of individuals who have responded is 186. However, since all questions are optional, each question has its own sub-sample of respondents. Of the 186 respondents, 103 have completed the questionnaire. The maximum number of respondents for a question is 185 and minimum is 107, average number of respondents over all questions is 133 with a median of 119 (barring open-ended questions and those in a branch). Of all those who entered the survey, 57% answered 100% of the questions.

The survey responses came from developers based in 36 different countries. The majority of the respondents are aged between 25 and 34 (50%). We consider the responses of any of the mobile application development team members regardless of their role. The majority of the responses originated from developers (57%), remaining roles include: managers (14%), marketers (8%), and those who assumed multiple team roles (18%). The years of experience in software development ranged from less than a year to 20 years, with an average of 7 years and a median of 5 years. The respondents reported an average of 4.2 years of experience in developing mobile apps specifically; with a median of 4 years, a maximum of 15 years and minimum of 1 year. The majority (84% of a total of 101 respondents) reported having a formal education in a technical/engineering field whereas 21% had a business-related formal education. The average size of teams reported was 5 working full time with as low as 1 and as high as 66 team members and a median of 2. A total of 103 respondents informed us of the platforms they develop for: 72% publish in iOS app store, 75% in the Android app store (Google Play), 12% in Windows Phone store and 11% published to other platforms: Amazon, Blackberry and Samsung stores, Apple TV and others. The respondents' apps had varying degrees of exposure, the largest had 10 million active users and the lowest had 14. The sample had a median of 1,500 active users and a mean of approximately 300,000 active users. Figure 3.4 shows the distribution of the questionnaire respondents in terms of their years of experience (in development in general and mobile development), team sizes, number of active users, generated reviews and the rating of their most prominent mobile application.

3.4.3.3 Data Analysis

In reporting the results of the questionnaire, we merge the number of respondents of the two extreme Likert items to simplify interpretation. For example, in the Likert agreement scale, we merge the number of those who agree and strongly agree to report the overall agreement rate and also merge the number of those who disagree and strongly disagree to report the overall disagreement rate. Moreover, we report the weighted average response in order to differentiate higher agreement/disagreement (or its equivalent in other scales), especially when ranking popularity of answers. The weighted average response of each statement is the average of scores assigned where strong agreement (or its equivalent) is scored 5, agreement is 4, neutral is 3, disagreement is 2 and strong

[4] LinkedIn: `http://www.linkedin.com`

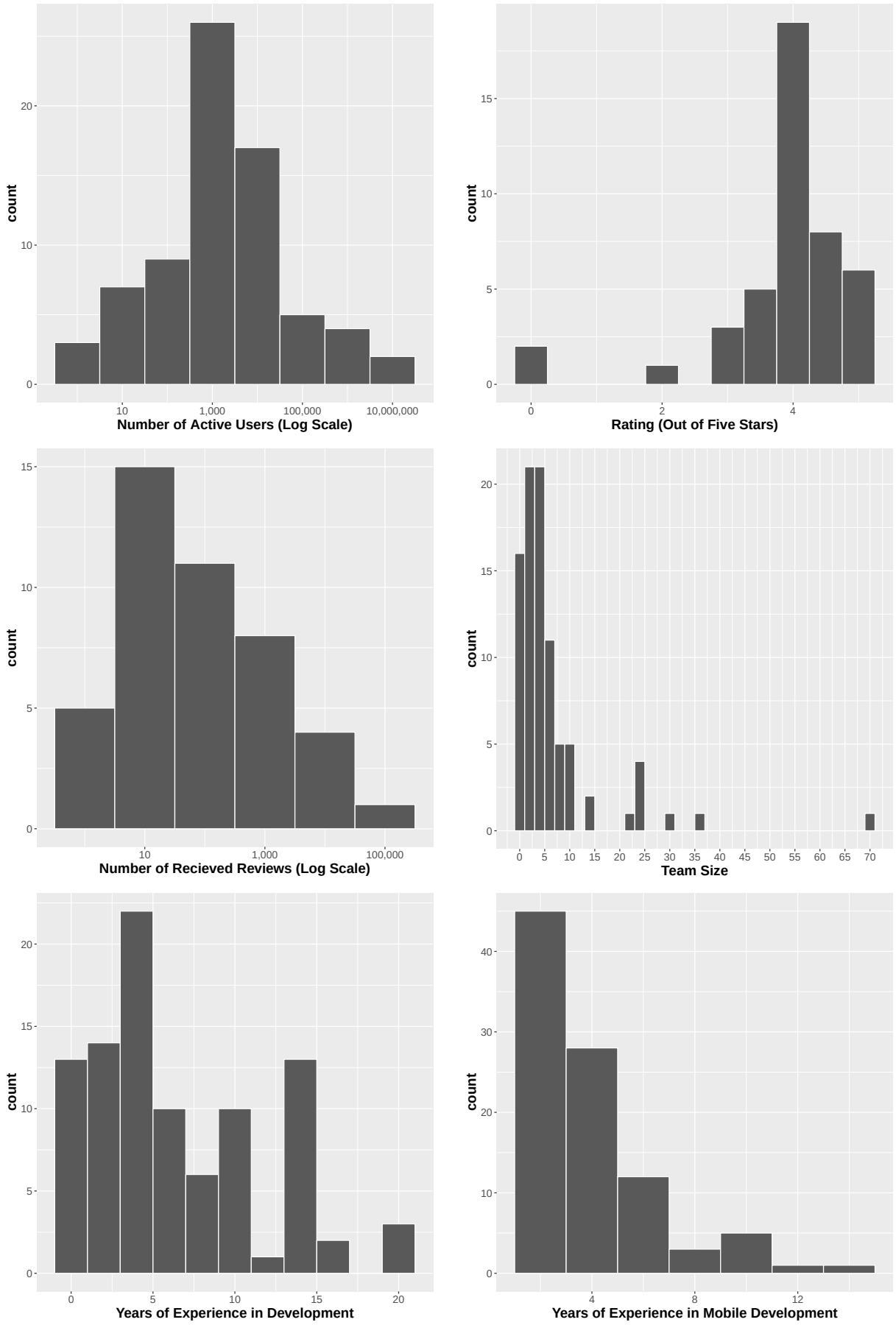


Figure 3.4: Histograms depicting the distribution of the various demographical information of the questionnaire's respondents.

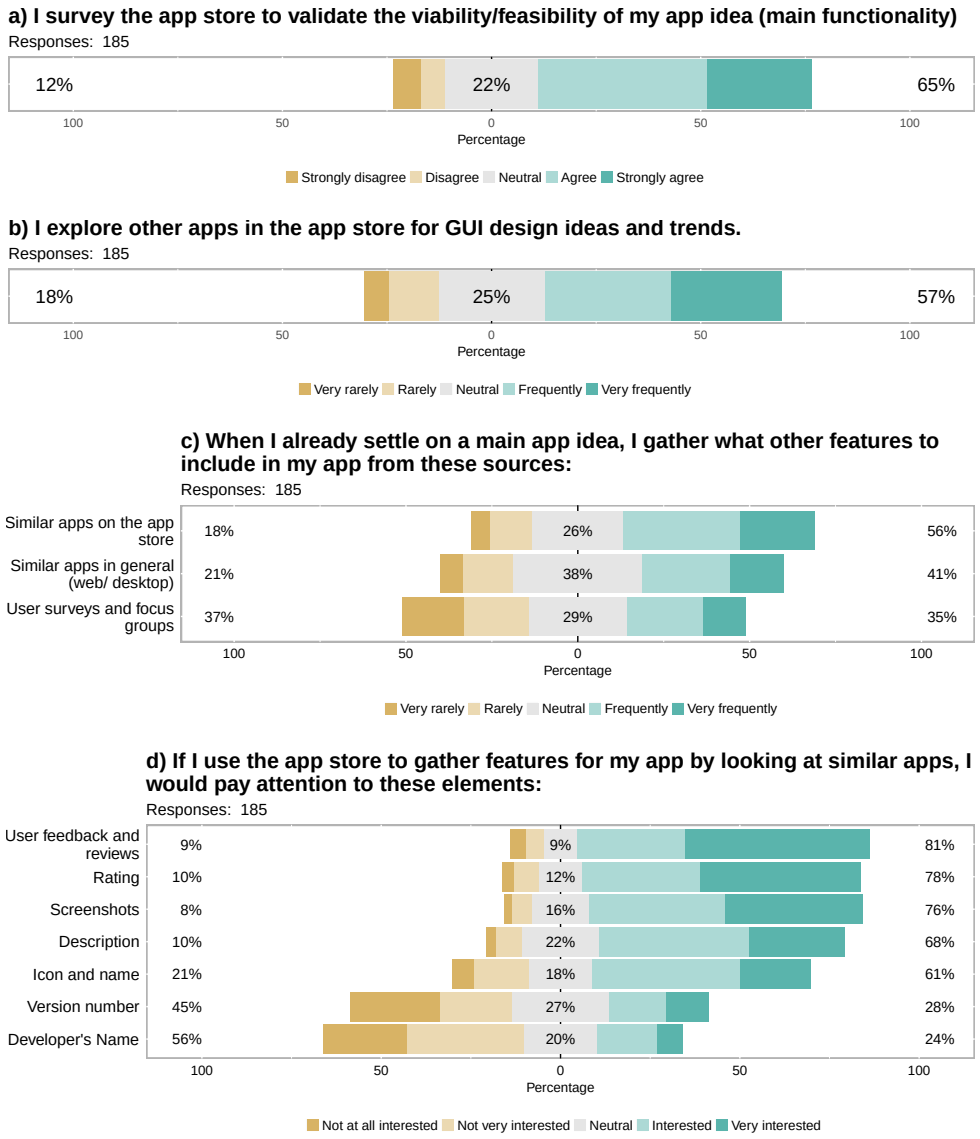


Figure 3.5: RQ1. Responses to questions regarding the initial phases of development.

disagreement is 1.

In summary, the agreement percentage gives the ratio of respondents who agree/strongly agree (or their equivalent) with a statement; whereas the weighted average score gives insight on how strongly respondents agree with this statement.

The questionnaire's quantitative findings are reported augmented with relevant qualitative ones extracted from interview transcripts to help aid the reader in understanding some developers' point-of-view regarding certain patterns of responses or opposing opinions. The quotes were selected by backtracking through pertinent themes and their codes.

3.5 Findings

This section reports the findings of the empirical study. The findings from the interview phase are presented in Subsection 3.5.1 in the form of the resulting theme map, which guided the design of the survey. Since the main goal of conducting interviews was to guide the survey design and due

to the limits of the interviewed sample, we do not discuss the results of the interviewing process in isolation but discuss them across Sections 4.2–4.4 in conjunction with the quantitative results of the questionnaire[5] to augment it with qualitative insights.

3.5.1 Interview Analysis Results

Figure 3.6 shows the results of the theme map deduced from the interview analysis. This theme map helped us construct an insight into the state of interaction between developers and app stores. Its main purpose was to pave the way towards designing the questionnaire.

The first theme map component is **Software Process**. In terms of the **requirements gathering** phase, the app store has been proposed as a method of exploring an application domain, validating ideas, checking ideas against redundancy and exploring the possibility of reuse. User expectations of features required of apps in a certain domain seem to be particularly of interest.

In terms of **design**, designing user interfaces is important as it will translate to a screenshot in the app's page. Screenshots are viewed as a big determinant of whether the user decides to download an app. Throughout discussions regarding designing user experiences, developers expressed exasperation regarding following strict OS vendor and app store owner's guidelines especially since the changes are often out of the team's control and interfere with the team's plan.

At the **construction** phase, developers include specific pieces of code that ask the user to rate the app and redirect them to the app store for that purpose. App permissions are a worrying factor during development as importing unnecessary APIs might bloat the permissions list thus making an app less desirable. Furthermore, during the construction phase, developers settle on tracking strategy in order to implant tracking code within the app.

During alpha and beta **testing** developers sometimes choose to distribute testing versions via the app store which gives them more feedback and exposure. Developers expressed interest in gauging users' interaction with the app within the app store ecosystem as part of the beta testing phase.

Maintenance has been found to be the most affected by the app store ecosystem. Developers expressed interest in users' feedback and rating as a major driving factor for new releases. Many of the interviewed developers mentioned that the app store ecosystem has enforced a certain release plan for their apps.

One aspect of interest was the practice of monitoring similar competing apps, especially during gathering requirements and perfective maintenance. Interviewed developers responses were divided regarding that particular practice. Those who declared it dangerous quoted addiction towards constant comparison and the eventual uselessness of having an app that is a copy of another. Opposing those views are developers who said that keeping an eye on competitors is necessary in such a competitive environment as the app store. However, they said that it is important to monitor in order to differentiate the app from similar apps and gather certain features from similar apps for perfective maintenance.

The second theme is related to the interviewees' app store know-how, aptitude, general **best practices** and other activities outside of the well known software lifecycle. These are patterns

[5]The questionnaire responses can be viewed online at: `https://www.surveymonkey.com/results/SM-NJX8DMHDV/`

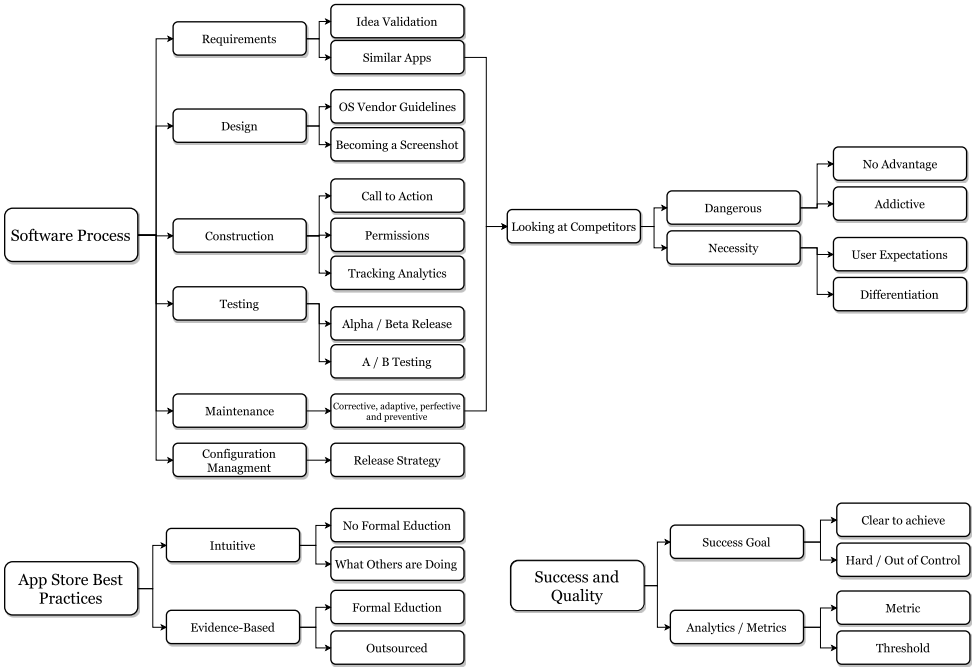


Figure 3.6: Thematic analysis findings in the form of a theme map. The theme map summarizes the data patterns found in the interview transcripts relating to the research questions.

of knowledge that are not evidence- or theory-based. This knowledge appeared as **intuitive** and not the result of formal training and in some cases heavily relied on observation of other apps in the app store. When a respondent shows propensity for **evidence-based** knowledge of app store management it was either the effect of formal training or the outsourcing of such tasks.

Finally, the last theme is app's **success and performance monitoring** in the app store environment. We have detected variation in terms of perceived success of an app in the app store. A large number of developers did not emphasize the quality of code, documentation, or overall architectural design for building a good software product. On the other hand, app store analytics are an often mentioned topic in our interviews. The respondents quoted many metrics they deem important to monitor to gauge the success of the app and its perceived quality by users. There were no global threshold for any of the metrics but an upward trend is certainly desirable.

3.5.2 RQ1: Lifecycle Processes

App stores as they reach an almost-monopolization of mobile app distribution with regards to a particular operating system, are prone to introduce some changes to how developers carry out software engineering tasks. For example, we anticipate, due to the app store regulation, for it to change the way developers plan releases. Additionally, as app stores provide a rich environment in which users leave feedback, including reporting bugs and requesting features, for it to affect developers requirements elicitation activities. The following sections go through our findings affecting requirement elicitation, testing, maintenance and release management.

3.5.2.1 Requirements Elicitation

To developers, not only can the app store serve as a distribution channel, it is also a large repository of apps. In this repository, access to similar and competing apps have never been easier. Not only can developers see how are other apps presented, but users' reaction to them. By sifting

through user comments, they can identify common bugs, appreciated features, requests and usage scenarios of apps in an application domain of interest.

We hypothesized that, naturally, developers follow and observe similar and competing apps. However, during the interview process, we observed polarised results regarding this particular activity. Certain developers expressed negative connotations with such practices *"You'll never win if you are stuck playing catch-up"* one developer expressed. On the other hand, others stated it as a necessity for survival. Among those one who said: *"I think it's vital to know what else is out there. You have to get a sense not just of what you are competing with but how it is delivered. Looking at [competitors'] reviews is something that we did to see if the features we included were appreciated by people or whether they were just not mentioned or actually thought to be waste of time. So, the app store provide a rich stream of information about what works and what people think of the app itself."*

In the questionnaire, more than half respondents surveyed the app store at the initial phases of development for both validating the app's idea (65% answered agree/strongly agree) or for user interface inspiration (57% answered frequently/very frequently). Of those gathering requirements for their app, the most frequent source has been other similar apps (56% answered frequently/very frequently scoring a weighted mean of 3.55) followed by similar desktop and web apps (41%, 3.30) and user surveys and focus groups (35%, 2.92). Figure 3.5 shows a breakdown of answers.

When asked about which elements of other similar apps are investigated, the three most popular were: user feedback (81% answered interested/very interested scoring a weighted mean of 4.19), rating (78%, 4.09), app's screenshots (76%, 4.05) and description (68%, 3.83). On the other hand, over half of respondents did not find the developer's identity of interest (56% not interested, 2.51) and %45 were not interested in how many versions competing apps released (2.69) (Figure 3.5-d).

Some developers clarified that this is not done just for the purpose of comparison with other apps, but for understanding a specific market and the user's expectations for a particular application domain. One developer clarifies: *"I focus on understanding the experience of the users and customer development more than comparing my idea to other apps. If I'm browsing other apps I'm either looking for inspiration in design or other ways to solve my problem."*; A survey respondent further clarifies: *"I found that app-users (especially social media) have been accustomed to a bunch of features that become de facto a must for a new project."*

> *For **requirement elicitation**, app stores provide a large stream of information and historical data to software engineers. The majority of surveyed developers use it to explore apps related to their application domain to gain an understanding of the expected user experience and anticipate features.*

3.5.2.2 Testing

App stores provide developers with a rich channel to conduct pre-release testing. Additionally, the rating and comment/review sections can give developers much to process. In this section we review developer responses regarding intent when pre-releasing the app in the store.

When a sample of 171 developers were asked if they indeed release alpha and/or beta versions to the app store, 59% answered yes. Among those who answered yes, we further investigated what

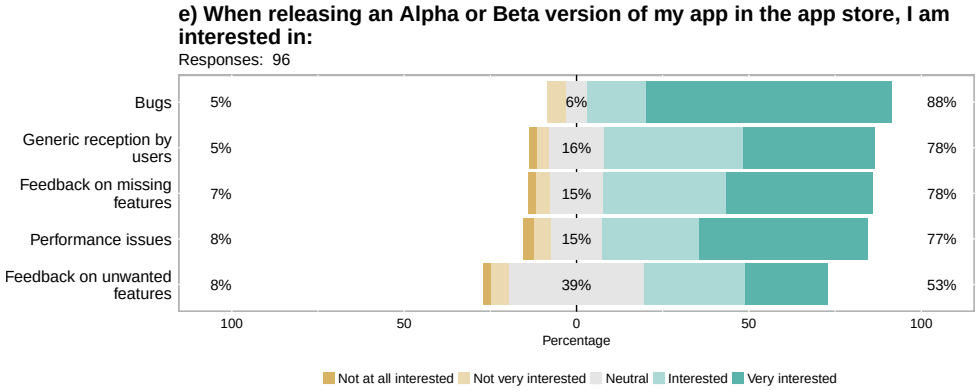


Figure 3.7: RQ1. Responses to questions regarding the testing phase.

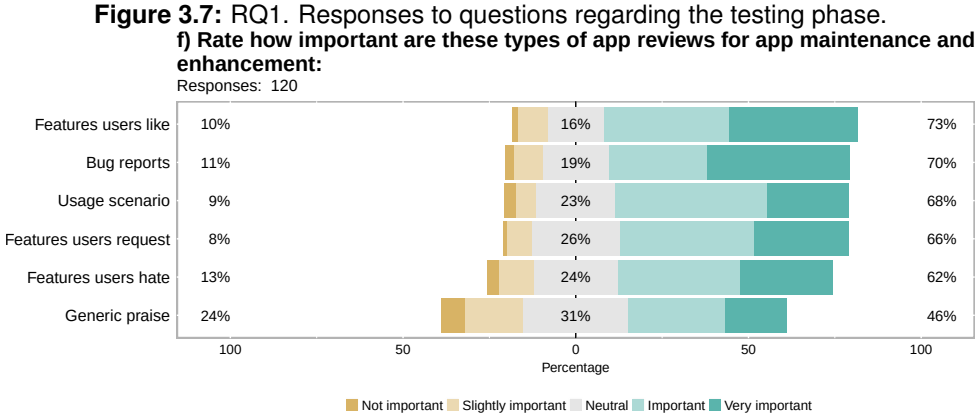


Figure 3.8: RQ1. Responses to questions regarding the usefulness of user feedback.

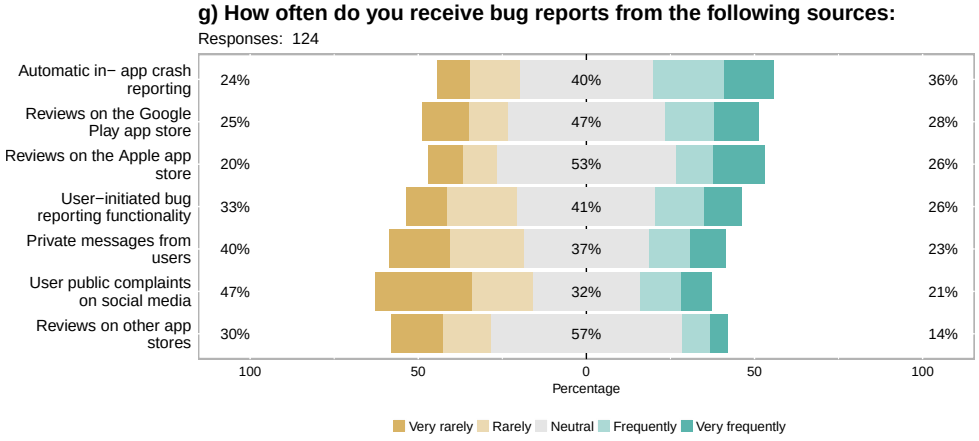
they hope to uncover by pre-releasing the app. The distribution of the answers is depicted in Figure 3.7.

Perhaps unsurprisingly, finding bugs garnered the highest interest. The finding with least interest was unwanted features; however, a previous study [199] reports that 78% of 106 surveyed developers rated functionality deletion as important and/or more important than adding new features. This may suggest that, while developers deem the removal of functionalities important, they might not necessarily discover which features to remove during alpha/beta testing.

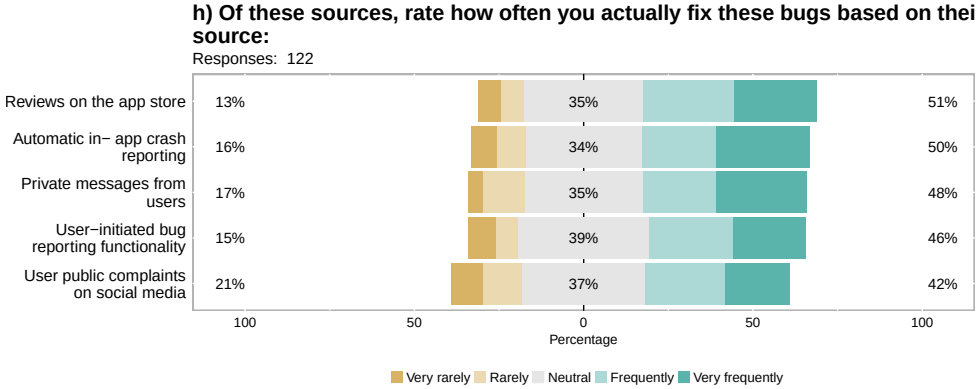
More interestingly, we observe that 78% of developers who release alpha/beta versions of their apps in the app store, are also interested in the generic reception of the app and the type of ratings, reviews and social hype it would garner (4.11 weighted average).

While the large amount of users that find and download a pre-release of the app is a good thing, some developers warned that over-exposure of the app might negatively impact the app's image if it has major issues. One developer wrote: *"We release the app in a staggered way so that a subset tests it and if something goes wrong we can early roll back to a stable version and fix any major bugs."*. A survey respondents also concurs: *"Premature social hype could doom the project."*

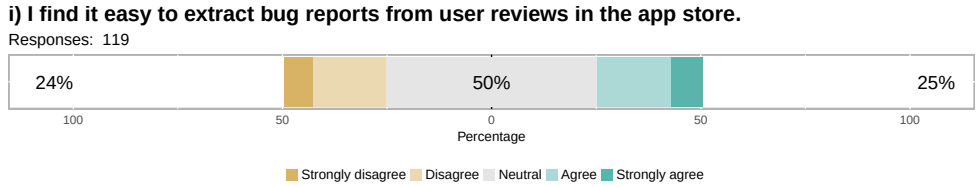
*For **testing**, many of the developers use the app store to publish pre-releases. In addition to finding bugs and discovering enhancements, 78% of those developers also stated that they release the alpha/beta version to test the general reaction of users in the form of ratings and social hype.*



(a) Figure A



(b) Figure A



(c) Figure A

Figure 3.9: RQ1. Responses to questions regarding the usefulness of user feedback for corrective maintenance.

3.5.2.3 Maintenance

When the app is published in the app store, developers come to maximum contact with users. The ratings, reviews and recommendations start coming in. We investigate the extent to which developers incorporate user input from the app store into their maintenance strategy.

During the interview process, we have detected that developers regard user reviews posted in app stores as a bug reporting and feedback collection tool in addition to a marketing tool. Several developers informed us that having a healthy proportion of negative feedback is an important nudge in the right direction *"[Positive Feedback] doesn't really help me. It should contain some information to help me improve the app, either something is wrong, something is missing, something they want,"* one developer expressed. Due to the rapid iterations typical of mobile apps release plans, one developer informs us that *"those bad reviews is what makes a really successful product."* To some developers, the app store is just another bug reporting and user engagement channel, albeit a

prolific, public one: *"We see what is being asked the most, regardless of the channel, we get the feedback from the different channels and aggregate them."*

Another developer highlights the importance of feedback coming through the app store rather than any other channel: *"Because whenever you're frustrated you want to voice your frustration immediately. And the only way to communicate with the developer, people think, is the app store."*

Since user reviews in app stores contain large diversity of information including complaints, praise, usage scenarios, feedback on features and bug reports (a taxonomy devised by Guzman et al. [200]), we asked developers about the types of feedback that they deem particularly important for app maintenance and enhancement. Developers rated high all of the suggested types as seen in Figure 3.8. Scoring highest (according to weighted mean) are bug reports (70% agreed/strongly agreed scoring 4.01 weighted mean), features users like (73%, 3.99) was next, followed by feature requests (66%, 3.86), usage scenarios (68%, 3.81), features they hate (62%, 3.74), and generic praise (46%, 3.35).

Corrective Maintenance: Developers were asked to rate the frequency of receiving bug reports based on the channel. Figure 3.9-g, depicts the results. In general, it shows an equal distribution with no channel prevalent in frequency. The highest in agreement, in terms of frequency is automatic in-app crash reporting, followed by the app store user reviews. User public complaints on social media was rated the least frequent (47% of respondents rated rarely/very rarely, 2.5 weighted average) followed by private messages from users (e.g. via email) which was rated rarely/very rarely by 40% of respondents scoring a weighted mean of 2.74.

On the other hand, when developers were asked which issues are frequently prioritised based on these sources, there is a trend towards favouring user reviews in the app store (51% of respondents prioritise it frequently/very frequently scoring a weighted mean of 3.61) tied with user's private messages (48%, 3.61) followed by automatic in-app crash reporting (50%, 3.6). We noticed that although private messages were less common, they were prioritised more frequently. This has been expressed during the interviewing process; especially vehemently by one developer: *"There's something more direct about an email [opposed to user public reviews]. A person has gone through the trouble of writing an email. It's more in-depth about it as well, I appreciate that."*

When it comes to prioritising user feedback coming from the app store, 51% of respondents reported that they frequently/highly frequently fix issues coming via that channel; whereas only 13% rarely/very rarely did it, as depicted in Figure 3.9-h. In that regard, we were interested to gauge whether developers found it challenging to extract actionable feedback from the app store. Figure 3.9-i shows that, 25% of respondents agreed/strongly agreed that it's an easy task, while 24% professed to finding it hard.

By analysing interview content, we find three main obstacles preventing developers from fully leveraging user feedback in app stores, despite its perceived importance. First is the frequency with which users post into the app store can make it challenging to catch up with those comments. Second, reviews can be largely repetitive and mixed with noise obscuring finding a distinctive list of requested fixes and enhancements. As one developer puts it: *"The problem is we get 4-5 reviews a day. And because they're largely similar and positive we don't read them in any depth. It would be really useful to have a way of aggregating the things that people most often asked for and the things that they said annoyed them the most. I certainly know what the highest things are as they*

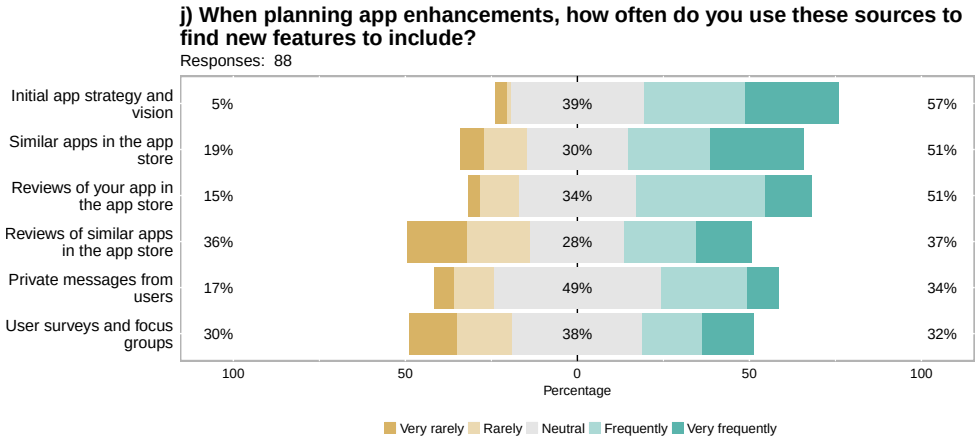
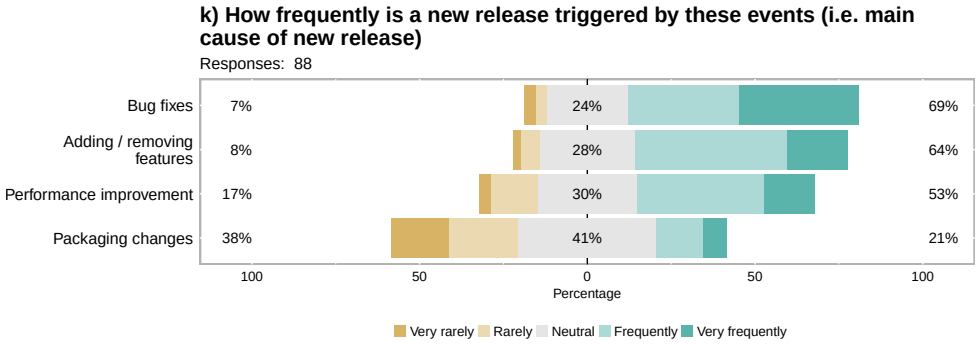


Figure 3.10: RQ1. Responses to questions regarding the usefulness of user feedback for perfective maintenance.

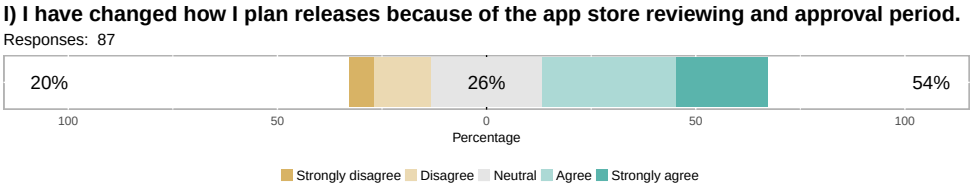
get repeated often. But within there are sort of 'second tier' stuff that I'm not clear about what we should prioritize. so we have to choose and understand what makes users happy is the thing that would be useful." The third challenge is a general distrust over the content found in app store reviews. A developer informs us *"I don't rely on comments coming from app store] because the comment system on the app store is completely broken. It's full of fake reviews, people leaving reviews because they are working for the competition and people leaving bad reviews because they're angry they didn't get the point of your app."*

Perfective Maintenance: In an app's journey, developers seek to grow the app by providing more value to users in the form of functionality and performance enhancements. This type of perfective maintenance is typically planned around user engagement in test sessions and focus groups in addition to the application's vision and roadmap. App stores provide rich communication channels in which users are able to submit their requests for new features and possible enhancements. Developers believe that delivering on those requests carry large marketing value for the app. Research by Martin et al. [69] showed that 33% of releases from a sample of 26,339 had an impact on user rating, and that impact is likely to be positive in free apps (59%), most importantly, they report that these significant releases are bug fixes and new features. While a study by Palomba et al. [119] showed that on average, developers include feedback from 49% of informative reviews into the new release, they also report that responding to user reviews has a positive effect on subsequent app rating ($\rho = 0.59, p - value < 0.01$). This highlights the important role app stores play as a communication channel and a source for planning app evolution.

To gauge the role user feedback play in perfective maintenance, we asked developers to rate how frequently do they use feedback from app stores as opposed to other sources. Figure 3.10 shows the tendency of the results. The results reveal that the most popular one (ranked by weighted mean) is initial app strategy and vision (57% use it frequently/very frequently scoring 3.79 weighted mean) while viewing the features of similar apps in the app store comes in second in frequency (51%, 3.53), next is user feedback of the app itself (51%, 3.48); 34% of respondents agreed to viewing private messages of users such as email and direct messages on social media (3.21 weighted mean) whereas 32% frequently/very frequently looked at user surveys and focus groups (3.04 weighted mean). On the other hand, rated least frequent was user reviews of similar apps in the app store



(a) Figure A



(b) Figure A

Figure 3.11: RQ1. Responses to questions regarding the effect of app store to release strategy.

(37%, 3.0 weighted mean).

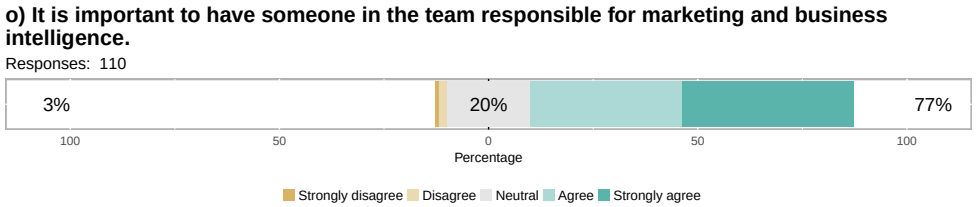
This indeed agrees with our interview observations. As one developer informed us regarding their practice when trying to find new features to enhance the app: *"[I keep] an eye on competitors and eye on my customers and community."*

*In **maintenance**, classical channels for user engagement and bug detection endure. However, developers seldom ignore those enhancement requests posted by users in the app store. For perfective maintenance, developers employ user reviews of their app and the features of similar apps for enhancements and possible reuse.*

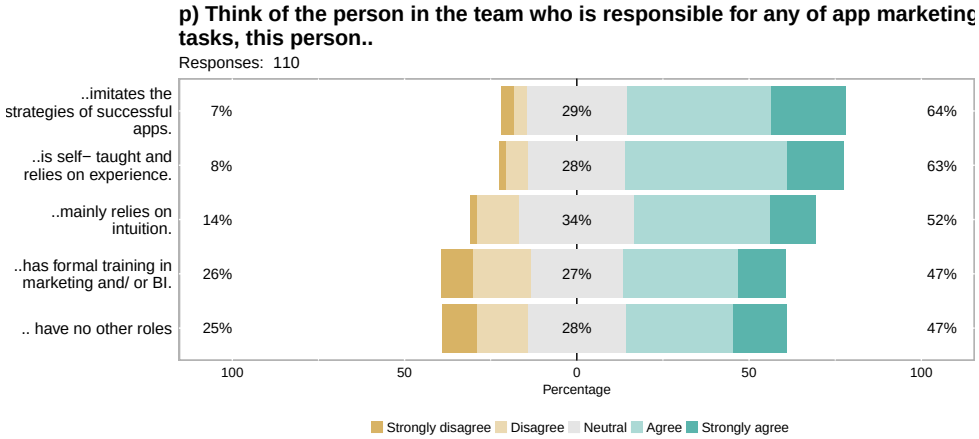
3.5.2.4 Release Management

App stores are managed by large firms who are usually the ones managing the platform and/or operating system. These organizations tend to prioritise raising the quality of apps marketed in their stores and thus enforce certain criteria on apps prior to granting them access to the store. This review procedure introduces delays that mobile developers usually plan around. Mobile developers expressed exasperation at losing a certain degree of control when it comes to release planning. *"And what that means is, you try to get rid of all the bugs before you launch. And this slows things down"*, a developer complains, *"So you try to avoid this horrible situation, which we've been in a few times, where you release something and then it breaks and then you have 11 days of letting your users down and getting negative reviews. You can't do anything about it because Apple takes a long time."*

This is also mirrored in the questionnaire results as 54% of respondents agreed that they changed the way they plan releases because of the app store review and approval period. And this indeed is a worrying concern when our results reveal that the major reason motivating a new release is a bug fix as apparent in Figure 3.11.



(a) Figure A



(b) Figure A

Figure 3.12: RQ2. Responses to questions regarding the type of activities and skills required in a development team.

*In **release management**, more than half of the developers reveal that they indeed change how they plan releases based on the app store's approval period. In general, developers expressed a need to conduct more rigorous testing as the gap between submitting a bug fix and it being published increases.*

3.5.3 RQ2: Emerging Skill-sets and Best Practices

We investigated if the app store ecosystem introduced new types of activities carried out by the development team. We conjecture that due to the way app stores lowered barriers to entry, smaller development teams had to carry out non-technical tasks and demonstrate app-store specific know-how for their app to thrive.

During our interviews, developers highlighted that it is paramount to the success of an app the way it is presented in the app store. In app stores, the competition is very high as a great deal of apps compete for the user's attention. To developers, quality of the product does not only come down to good software, other factors regarding presentation in the app store environment come into play. *"I can be very cynical and say that it's the only thing that matters. From experience I say that great communication and normal app works better than great app with bad communication. And for communication I mean everything: packaging, marketing, PR, etc. So it's crucial."* a developer informs us.

We have observed throughout the interviewing process that respondents exhibited confidence in their best practices knowledge: Application strategy, implementation, and mostly, app store culture and know-how. For example, one developer elaborated on best ways to post a screenshot in the app store *"Do not put a boring screenshot that's not wrapped in a phone: wrap it in a phone and put some text above it."* when asked how did they know this technique is effective, they said it was by

looking at other apps. Another developers informs us: *"It's just trial and error, looking at what other people are doing, what I like and what works."* Something that prevails many of their practices.

This is reflected in the questionnaire responses as the majority of surveyed developers acknowledged that it is important to have a team member who is responsible for carrying out marketing and business intelligence tasks (77% agreement, 4.13). However, 63% report that their marketing team member is self-taught and relies on experience (3.75 weighted average). Of those surveyed, 64% agree that whomever is carrying out these tasks imitates the strategies of successful apps (3.8) whereas 25% report that the team member responsible for marketing decisions is not dedicated to that role (3.31). Figure 3.12 depicts a breakdown of the answers.

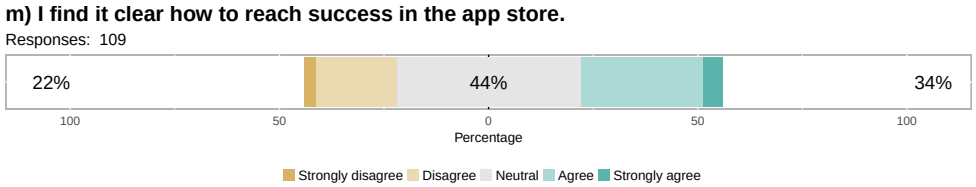
We observed a number of developers who needed to address many non-technical issues. The ***new skill sets*** *required by engineers developing for app stores include facilitating app discovery for users in addition to understanding the competitive environment and user expectation when selecting core functionality and supporting features, custom release strategy for mobile app stores, and several practices leading to a better brand for the app.*

3.5.4 RQ3: New Success Criteria and Performance Measures

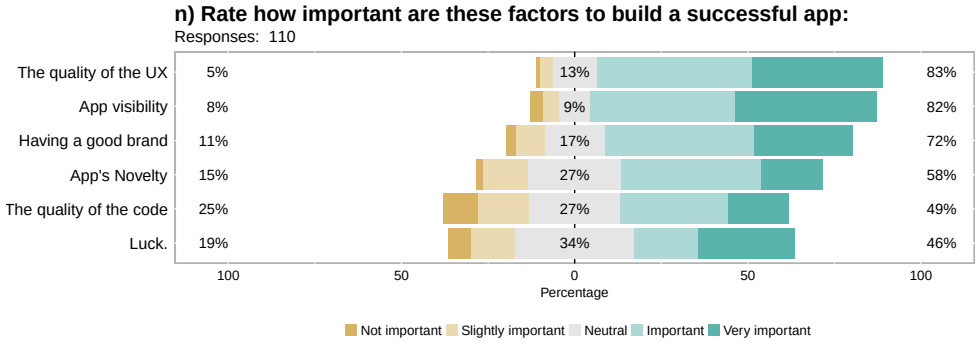
As the app is released into perfective and corrective maintenance cycles, developers have access to immediate feedback regarding the quality and performance of the app. This feedback takes many forms. In addition to user rating and reviews, developers have access to a large number of metrics including app downloads (rank), user retention rate, revenue and number of reviews. We investigated the extent to which developers monitor these metrics and the role they play in decision making.

To gain a better idea of perception of success, we asked respondents to write what they define as success in the app store. Several developers restricted their definition of success to the app correctly delivering its functionality. *"When the user is able to do the core features of the application quickly and without much trouble,"* a developer wrote. Other developers answered similarly: *"[When app solves a real problem."* and *"Providing a real great solution to an existing problem."* However, and more interestingly, the majority of their answers quoted a measurable, app store metric. Of our sample, 52 informed us of the metrics they observe to evaluate the success of the app. The most popular was the number of downloads/installs (37%), followed by rating (28%), active users/retention rate (27%), revenue (15%), then application's ranking (3.8%). Few respondents (6%) mentioned application's validity/verification (i.e. the app delivering the needed functionality without faults).

Through interviewing developers, we detected some uncertainty and lack of control towards reaching success in the app store. This is confirmed by the questionnaire responses. Respondents, when asked if the path to achieving that success is clear and easy to follow, showed reluctance with only 34% of respondents agree that, indeed, they find it clear (3.14 weighted mean) as seen in Figure 3.13-m. We further explored their opinion regarding the most important factors to build a successful app and were surprised to see that the lowest rated was the quality of code and documentation while the one rated highest in importance was the quality of the user experience (UX) as shown in Figure 3.13-n.



(a) Figure A



(b) Figure A

Figure 3.13: RQ3. Responses to questions regarding the knowledge of success factors in the app store.

*Surveyed developers reported a unique perception of **quality measurement** giving low ranking to code quality and documentation in determining an application's success. Developers tend to quote more app-store-specific **quality measurements** than classical software engineering ones with number of downloads surpassing user's rating.*

3.6 Discussion

The results presented reveal implications that can inform relevant research spanning several scientific sub-fields. Through our findings, we summarise implications under three top-level categories: developer-user interaction, market transparency and application release cycles.

3.6.1 Developer-User Interaction

App stores, in their current format, have further **bridged the gap** between potential users and developers. We detect that developers view the app store as a channel of communication. Though not the only one, this channel has two distinct properties: It increases the prolificacy of users and can affect the success of the app. Pagano and Maalej [116] found that free apps received an average of 36.87 daily reviews in 2013; and more recently, Mcilroy et al. [201] analysed the reviews of 12,000 apps in the app store and reported that free apps receive 7 reviews per day on average; in addition, McIlroy et al. [202] and Palomba et al. [119] report that responding to user reviews has a positive effect on subsequent app rating ($\rho = 0.59, p - value < 0.01$); whereas Lee and Raghu [203] find that continuous updates increase the applications success. Developers seem to be aware of this to some extent. This is reflected by our survey respondents professing interest in gauging public reception and social hype of the app in alpha or beta testing stages (78% agreement), this is particularly of importance in light of the findings of Ruiz et al. [204] in which they find that mobile app stores rating fails to adapt to the actual current satisfaction levels and are resilient once they reach a certain number of user base.

While automatic in-app crash reporting is the most prolific channel of reporting bugs, the one

mostly prioritised by our respondents is user reviews in app stores. Additionally, 51% of respondents frequently use user reviews for app features enhancement. Our results reveal that, while developers and researchers point to the benefits of using reviews for app evolution, 24% reported experiencing difficulty in extracting bug reports from user reviews. Reasons hindering proper utilisation of user reviews include its noise and volume.

Requirements engineering research have directed their attention to solving the problem of analysing user reviews for the benefit of app evolution. Research has been carried out to classify user reviews according to their type and actionability to the developer [82][115][121][85][200][72][122], review summarisation [73] [205][206] as well as feature-specific analysis [66][179][77][81]. Further research employed user reviews and the app's extracted features to localise change requests within code and to couple natural language with source code patterns [182][207]. A systematic review of the literature relating to opinion mining from user comments in the app store is provided by Genc-Nayebi and Abran [208]. However, research remains scarce on the problem of detecting fraudulent reviews. Such reviews, not only increase the amount of noise when extracting useful information from user reviews, they also introduce errors regarding app ratings and subsequently in any analyses that incorporate the app's rating score (e.g. [127][113]). While the field of "opinion spam" detection advances in other areas of research, its transference to app store analysis is necessary. Xie and Zhu [209] and Li et al. [210] reveal the existence of a fake rating black market and provide in-depth analysis of their characteristics and its effect on the app store.

3.6.2 Market Transparency

One of the major contributions of a centralized mobile application marketplace is the significant increase in **transparency and availability of information** for content creators. The applications' price, features, reviews, ranking and release strategy are publicly available. Our results reveal that over half respondents monitor similar and competing apps at the stage of requirement elicitation (56% frequently view similar apps). Requirements engineering research can help further investigate the effect of this practice and facilitate it further.

In performing perfective maintenance, frequently investigating features of similar apps is as common as considering the feedback of the developer's own app (51% frequently/very frequently for both). This insight can help guide further software repository mining work that collates information from various applications that share functionalities or are in the same application domain. This insight also attests to the viability of app store performance predictions based on the past evolution of other similar applications.

To this end, Vu et al. [125] provided a keyword-based approach to mining reviews of apps and Shah et al. [120] detected similar apps based on feature overlap and merge reviews of those app for feature-specific sentiment analysis. Sarro et al. [179] showed that similarity of app features/descriptions can successfully lead to accurate prediction of app success (i.e. rating). We believe this vein of research can be further extended to incorporate automatic detection of similar useful apps while using the evolution of these apps (and their reviews) to recommend possible feature inclusion and other strategic decisions for developers.

However, we draw the attention of the community to possible pitfalls when analysing reviews and ratings of mobile apps as a 'rating call-to-action' gains popularity among developers. A rating

call-to-action operates within the app to request users to rate the app and subsequently directs them to the app store. Of our respondents, 38% embedded a rating call-to-action into their apps. The majority of those (56%) admitted to ensuring a call-to-action is activated when the user appears sufficiently engaged and having a positive experience with 35% ensuring the app first prompts the user for their rating, then only directing them to the app store when their rating is high enough. This may carry certain implications towards the bias of app rating and reviews. In their empirical study, Pagano and Maalej [116] analyse 1,126,453 reviews from 1,100 applications from the Apple app store, half of which were free, and reported that the overall average rating of all reviews is 4.13 with 61.96% of reviews having a 5 star rating, while such a high average rating value is not observed analysing the content of app stores in 2011 when such call-to-action may not have been as popular [175].

Among similar applications' attributes that are made available for developers to observe, user feedback garner the most attention (81% are interested/very interested scoring 4.19 weighted mean) closely followed by ratings (78%, 4.09) and screenshots (76% , 4.05). This promises significant contribution to developers were researchers to employ image processing to mine applications users interfaces to extract actionable information as is done with applying natural language processing over applications descriptions [66][177], UI text [211] and user feedback [128].

3.6.3 Release Planning and Quality

Our questionnaire reveals that the app store regulations and approval periods affected 54% of respondents' **release strategy**. The research by Nayebi et al. [137] reveals that a majority of their sample (36 developers) adopt a time-based strategy (scheduled releases) (80%) with 45% releasing weekly or bi-weekly. Furthermore, they find that 36% of respondents will change their release plan to accommodate user feedback and that 61% agree that a time-based release strategy affects the application's success in terms of feedback and user rating. Several studies reported that frequent releases cause an increase in user engagement (i.e. reviews and ratings) [116][201] and that certain types of releases have significant impact on user ratings [69][186]. This supports the idea that release strategies in app stores may not only be influenced by vendors' guidelines but also by users' public reaction in the form of downloads, reviews and ratings. Adams and McIntosh [212] emphasize the need for software engineering research to further investigate the implications of the industry's recent trends in adopting certain release engineering practices including rapid delivery and mobile app release cycles.

These release practices enable mobile development to adopt rapid adaptation and fast route to market that closely resembles that of web application development. This adaptation model may inform further research in the software engineering community to extract and transfer experiences and techniques from similar platforms such as web development. One example is the ease of adoption of A/B testing which thus far has been predominantly applied in web based applications. Of our respondents, 39% already perform A/B testing.

In addition to change in release practices, the perceived quality criteria of mobile apps seem to shift. Surveyed developers deemed user experience design of higher importance than code quality. This is in line with the findings of Nayebi et al. [165] in which they surveyed 22 mobile developers and found that 'customers expectations' and 'market and competitors' were deemed more important

in mobile development compared to other platforms, whereas 'quality' was rated higher with more consensus for other platforms than mobile apps. This is confirmed by the study of Minelli and Lanza [150] where they report that open source Android apps showed high complexity with smaller sizes, large reliance on third party libraries and overall neglect of development guidelines.

3.7 Threats to validity

During the design of this study, potential threats to its validity have been addressed in an effort to minimise their risk.

3.7.1 Construct Validity

Construct validity in qualitative studies mainly pertains to a unified understanding between what the researcher has in mind and what the respondent eventually understands [167]. Prior to building the questionnaire, the interviewing process with 10 app developers served to orient the researcher towards the culture and type of knowledge to which the mobile app community adheres. Several books were suggested by the developers that the researcher has read to familiarise themselves with the terminology and the process (a good example is The Lean Startup by Eric Ries [213]). Due to the nature of the interviews, misunderstandings were detected and cleared up. The questionnaire was built upon the insight provided by the interviewing process in addition to transcript analyses. The read-aloud pilot study of the questionnaire ensured the detection and elimination of incompatible terminology and other misunderstandings.

3.7.2 Internal Validity

Internal validity is at risk when causal factors are examined and reported. As this study is mainly data-driven with first and second degree collection methods (interviews and questionnaire), we present the results as observed. In interpreting the data, we make clear our conjectures are aligned with those recorded during the interviews. Causal analysis is limited as to this type of study. Another aspect that is a threat to internal validity is proper analysis by the researchers of the interview transcripts. While one researcher (the author) has done the thematic analysis, it has been revised and validated by the three supervisors (research collaborators) in more than one collaborative session till consensus was reached. We limit the threat by augmenting our findings with questionnaire responses.

3.7.3 External Validity

Although initial information gathering technique only aims to interview a low number of developers, the interviewing process terminated when responses to all questions were pre-observed in previous interviews. The developers selected for interviews, though with varying backgrounds and sizes of teams, represent a limited sample. However, it is common for interviews to limit the sample as they only serve as an exploration device rather than seeking generalizable answers. Afterwards, all possible findings are augmented by disseminating a questionnaire to developers in order to measure the extent to which developers adhere with the findings. Through the questionnaire, we were able to reach an even more diverse set of mobile developers overseas. Having both methods of collecting data, we employ triangulation that can help us in affirming the validity of the results.

The questionnaire garnered 186 responses, this number, thought fairly large and in line with several similar research as shown in Section 3.2, cannot be claimed to be representative of all types

of development teams, applications domains or characteristics other than the ones reported in our sample.

3.8 Conclusions

This study investigated aspects of app store developers' software engineering activities revealing overarching themes of importance to app store software development. The three main themes that emerged were market transparency, user-developer gap reduction, and release cycles.

App stores exemplify market transparency in which app description, features, price, rating and user feedback are public. Our survey found that developers do, indeed, refer to similar apps when designing their own. Our results reveal that developers are interested in monitoring similar apps for maintenance and evolution. We also found that other apps' user feedback, rating and screenshots and are the three most important aspects of information gleaned from the open market by developers. Our results highlight the way in which app stores have become a communication channel between users and developers. Our findings confirm that developers seldom neglect user feedback posted on app stores; user feedback was the third strongly agreed-on source of app improvement after the initial strategy of the app and monitoring similar apps on the app store. Our survey respondents also rated user reviews as the second most prolific channel of bug reporting after automatic in-app crash reports but regardless was scored highest in prioritisation. User feedback, in addition to being informative to developers, also determines the overall rating of the app. Previous research on this subject showed that release frequency correlates with increased user feedback[116][201]. More than half of our respondents reported changing their release plan in accordance with perceived constraints imposed by the app store ecosystem. These findings have actionable conclusions for software engineering practitioners and researchers, including requirements engineering, testing and mining software repositories research communities, and also business communities.

Chapter 4

Feature-Based Mobile App Categorization

4.1 Introduction

An effective categorisation of software according to its functionalities offers advantages to both users and developers. In exploring developers' interaction with the app store, the study presented in Chapter 3 finds that more than half of the surveyed developers elicit requirements by browsing similar and competing apps in the app store, more often than web, desktop apps or user surveys. Furthermore, half of respondents repeat the same discovery patterns via searching the app store when performing perfective maintenance.

Therefore we conclude that categorisation can help app developers by facilitating code-reuse, locating desirable features and technical trends within domains of interest [21][87][94]. For app store users, effective categorisation may facilitate better application discovery and more exposure to newly emerging apps [214][215].

Unfortunately, as several other researchers have noted [70] [95], existing categorisations are ineffective because the clustering is simply too coarse-grained. This is particularly pernicious in the case of app stores, where there are typically 100,000s of apps, yet the existing commercial categorisations of app stores such as Google Play contain only 10s of different app categories. These categorisation approaches are *theme*-based which may fail to explain an app's functionality, and do not cluster apps according to the features they exhibit. As a result of this coarse granularity, apps within the same category bear only an unhelpfully broad sense of 'similarity'. For example, Elevate - Brain Training [1] , an app that claims to improve the user's critical cognitive skills (through a series of games), is found in the same category as Blackboard's Mobile Learn™[2] , a learning management system client that claims to facilitate academic course management tasks. The reason they are in the same category derives from the fact that both apps pertain to `Education`. However, this is an overly broad categorisation, and it fails to respect the fact that they have entirely different functionality and supporting features.

Current app store categorisation approaches are not only hampered by their coarse granularity, they are also inherently unresponsive and unadaptive, yet they seek to categorise a rapidly-changing software deployment landscape, in which apps can release with high frequency [75]. That is, the current approach to commercial app store categorisation uses a manual assessment of broad themes,

[1] Elevate - Brain Training. `https://play.google.com/store/apps/details?id=com.wonder`

[2] Blackboard's Mobile Learn. `https://play.google.com/store/apps/details?id=com.blackboard.android`

and therefore cannot speedily adapt to shifting developer behaviours and market preferences, nor can it help to identify emerging technical trends.

Our approach seeks to overcome these twin limitations of coarse granularity and the lack of adaptivity, to provide dynamic, automated, finer-grained categorisations of app stores based on their claimed functionality. Our approach builds on recent research on app categorisation approaches [70][110], which have sought to better understand app behaviour in order to automatically identify harmful, spam, and/or misclassified apps. We therefore cluster apps based on their claimed behaviour. Specifically, we use the evidence of feature claims present in developers' description of their apps, which we extract using the feature mining framework proposed by Harman et al. [63]. Therefore, a feature in the context of this chapter refers to a claimed functionality (i.e., software capability) that has been mined from the app description and it is represented by a collection of terms. Using the claimed behaviour means that we do not need to access the source code of the app which is often unavailable. Moreover, since we use hierarchical clustering, one can choose the granularity of the clustering by selecting a suitable point in the hierarchy, thereby providing multiple (feature claim) views of the app store at different granularities. These 'feature claim space' views of the app store offer a unique perspective to developers and users (and to those who manage the app store). This claim space has been found useful in other software engineering domains, such as feature modelling [24], but has not previously been used in app store analysis. Furthermore, our approach is automated, so it can also be re-run, periodically, to adapt as novel apps are deployed and/or as existing apps evolve, thereby responding to emergent feature claims.

Our use of feature claims means that our clustering focuses on those technical aspects that developers deem to be sufficiently important to be mentioned in the descriptions they offer to their users. However, we certainly do not claim that this is the *only* categorisation view of interest, and believe that there is very interesting future work to be conducted on the comparison of different clustering-based app store 'views'. See Chapter 5 for more details.

The categorisation 'views' we construct within the feature claim space also do not replace the existing coarse-grained commercial app store categorisation, nor do they seek to imitate it. Rather, we seek to provide an alternative, feature-claim based categorisation. We can assess the degree of improvement in the quality of clustering achieved by our finer granularity, using standard clustering assessment metrics, such as the silhouette width method [216] applied on two real-world datasets extracted from the Google Play (Android) and BlackBerry app stores in 2014[3].

4.2 Approach

To achieve an app clustering solution, the approach we developed consists of three main stages: (i) feature extraction from the textual description of the mobile apps, (ii) feature clustering to reduce the granularity of the features used to describe each app, and finally, (iii) clustering is conducted over the apps themselves. Figure 4.1 shows the overall architecture of the system. In order to uncover the implicit (latent) categorisation in an app store data set, we first extract claimed features from descriptions, using the feature-extraction algorithm proposed by Harman et al. [63]. Then, we introduce a novel two-step clustering technique that first reduces the granularity of the extracted

[3]The data is publicly available on the companion website: `https://afnan-s.github.io/appa/clustering.html`

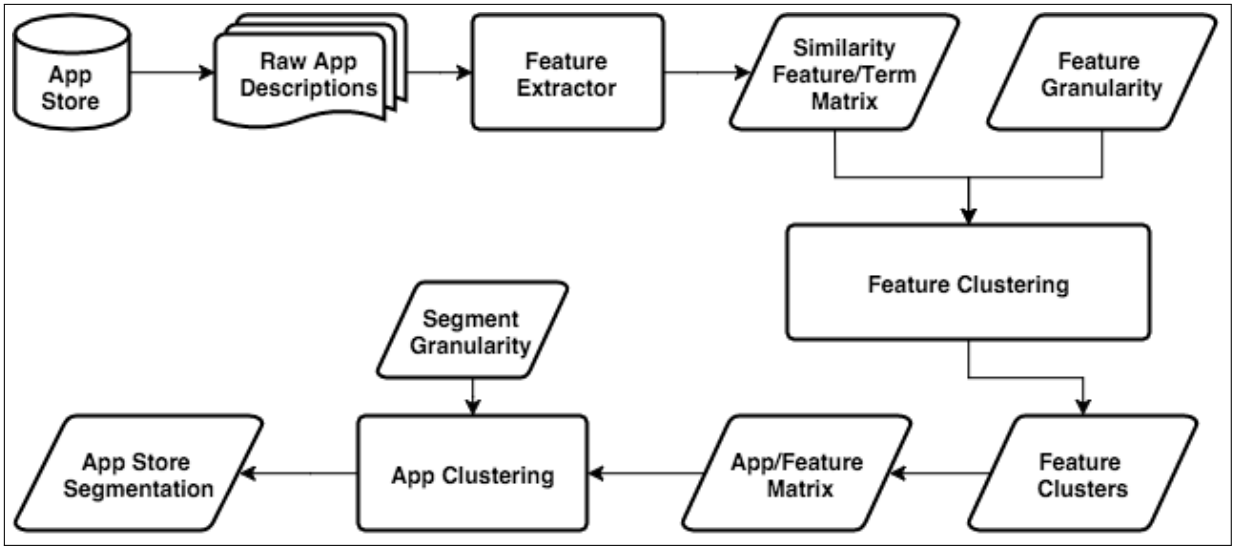


Figure 4.1: Feature Extraction and the Two-Phase Clustering System Architecture

features and subsequently uses the feature clusters to describe the relationships between apps. We represent these relationships using an App-Feature Matrix (AFM), in which rows are apps and columns are Feature Clusters (FC) exhibited by the corresponding app. The Feature Clusters are groups of features, computed using a Feature-Term Matrix (FTM) to reduce the dimensionality of the AFM. The FTM captures the relationship between each feature and the linguistic terms it contains. In order to abstract away any superficial syntactic variations in the extracted features (that do not affect semantics), we build the FTM using ontological analysis. In the following sections, we provide a detailed description of our framework and give further details of the choices and configurations of the clustering algorithm.

4.2.1 Feature Extraction

We use the framework proposed by Harman et al. [63] to mine claimed features from raw app descriptions. Firstly, feature list patterns are identified to highlight and segment the coarse features from the textual description of the app. Then the features are refined by removing non-English and stop words and by converting the remaining words to their lemma form. Secondly, NLTK's N-gram CollocationFinder is used to extract what we call 'featurelets' which are lists of bi- or tri-grams of commonly collocating words. Lastly, a greedy hierarchical clustering algorithm is employed to aggregate similar featurelets. The result of this process is a collection of featurelets where each featurelet represents a certain feature. Table 5.2 shows examples of extracted featurelets and their prevalence in the dataset. Throughout this process, links are maintained between each featurelet and apps containing features that contributed to that featurelet. More details can be found in [217][63][66].

4.2.2 Feature Clustering

For the purpose of clustering apps based on the extracted features they share, each app is represented as a data point described using the features its description exhibits. However, the featurelets extracted using the previous phase may be of a too fine granularity for this purpose. To further abstract features from the language used to express them, and to reduce the dimensionality of the AFM, we first cluster the featurelets where semantic similarity is factored into the clustering algorithm.

Table 4.1: Examples of extracted featurelets representing each feature and the number of times these features appear in the dataset (number of apps that boast the feature) for both Blackberry and Google datasets.

Dataset	Featurelet terms	Occurrences
Blackberry	[easy,use]	1242 (most common)
	[latest, news]	603
	[share, friend]	462
	[kid, friendly]	5
	[choose, output, folder]	1
Google	[game, play]	476 (most common)
	[challenge, friend]	139
	[music, play]	43
	[weather, forecast]	26
	[photo, share]	9

4.2.2.1 Feature Representation

To achieve this clustering, we use the vector space model [218] as a representation of the features. Given that each featurelet is a set of terms $f = \{t_1, t_2, .., t_k\}$, we construct the set of all unique terms in the corpus $T = \{t_1, t_2, .., t_N\}$. Then, we convert each featurelet to a vector in which each element corresponds to a term in the set T. The element's value corresponds to whether the feature contains that term. We later transform the value of the element to be a weight calculated using the standard term frequency-inverse document frequency (TF-IDF). This gives less importance to common words used to express software features (e.g., create, view,..) and more importance to less common words (e.g., wallpaper, voice-over,..). Meaning that features that share the word 'voice-over' are deemed more similar than features that share the word 'view'.

At this stage, the feature vector does not convey any semantic similarity with other features. For example, the featurelet ('view', 'image') shares no similarity with the featurelet ('show', 'photo'). To amend this problem, we replace the notion of *term frequency* with *term similarity*. Due to the nature of our featurelets, term frequency will never exceed 1. That is, a featurelet will never contain the same term twice. So, there is no loss of data when removing the term frequency aspect from the weight-calculation formula. On the other hand, term similarity carries information about the relatedness of each term in the dictionary to the words contained in the featurelet.

To calculate the similarity between each word in the featurelet and each term, we use Wordnet [4], since it provides an adequate way of quantifying the similarity among general English terms. The Wordnet similarity score is stored in the feature's vector in the corresponding term's element. Wordnet's similarity calculator returns a score representing the shortest path between the two words in the English language ontology [219]. Finally, the resulting vector space F is defined as follows:

[4] http://wordnet.princeton.edu/

$$F = \begin{array}{c} \\ f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_M \end{array} \begin{array}{c} \begin{array}{cccc} t_1 & t_2 & ... & t_N \end{array} \\ \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1N} \\ w_{21} & w_{22} & \dots & w_{2N} \\ w_{31} & w_{32} & \dots & w_{3N} \\ \vdots & \vdots & \ddots & \vdots \\ w_{M1} & w_{M2} & \dots & w_{MN} \end{bmatrix} \end{array}$$

where M is the total number of features (denoted with f) in the dataset and N is the length of the dictionary of unique terms (denoted with t) found in the set of all features. Each element stores the weight w_{ij}. The weights are calculated as follows:

$$w_{ij} = s_{ij} \times idf_j$$

Where idf_j is the inverse document frequency of the term t_j defined as the logarithm of the total number of features divided by the number of features that contain the term t_j:

$$idf_j = \log \frac{M}{|\{f : t_j \in f\}|}$$

s_{ij} is the maximum of the similarities between each word in f_i and the term t_j. Thus, it is defined as follows:

$$s_{ij} = max_{w \in f_i}\{sim(w, t_j)\}$$

4.2.2.2 Selecting K

Selecting the optimal number of clusters remains a problem in unsupervised machine learning with no optimal universal solution [220]. In this context, K represents the number of features that will be the variables that describe the app in the next phase of clustering. To determine the optimal number of clusters we used the modification of Can's metric [221] proposed by Dumitru et al. [21]. This metric is based on the degree of difference between each feature vector and another. We set the threshold of term frequency to be $0.00075M$ (where M is the total number of features) to distinguish terms that have more contribution to representing the features. This threshold have been empirically tested to ensure its suitability with our dataset. Using this metric, K is calculated as follows:

$$k = \sum_{i=1}^{M} \frac{1}{|f_i|} \sum_{j=1}^{N} \frac{w_{ij}^2}{|\{f : t_j \in f\}|}$$

4.2.2.3 Final Clustering

To cluster the resulting FTM, we use the prototype-based *spherical k-means* (skmeans) technique. Skmeans is built upon the vector space model and hence, uses *cosine similarity* to measure the similarity between vectors. It has shown to exploit the sparsity of the FTM, and generates disjoint clusters the centroids of which carry semantic information about the members of the clusters that serve as high-level concepts of clustered text [222]. For example, the featurelets (1:1, aspect), (aspect, ratio, option), (aspect, ratio), (multiple, aspect) were all grouped into one feature cluster; another example is the feature cluster containing featurelets: [search, extension], [previous, search], [enable, search], [easy, search, torrent], [full-text, search], [global-text, search, book].

4.2.3 App Clustering

The final stage is clustering the apps in the dataset to uncover its segmentation. To achieve this, apps are described using the features they share. First, we design the app representation technique which is then used to cluster the apps.

4.2.3.1 App Representation

We use the resulting feature clusters to construct the AFM, an app-feature matrix where the rows are vectors corresponding to apps. The AFM columns are by now greatly reduced in dimension due to them corresponding only to feature clusters resulting from the previous step. Each element in the AFM is a Boolean value to indicate whether the app exhibits a feature within the corresponding feature cluster.

4.2.3.2 Selecting Clustering Technique

We cluster the apps using agglomerative hierarchical clustering technique [223]. We opted for a hierarchical technique due to its efficiency when studying the effects of selecting different granularity levels. Once the dendrogram is generated, any cut-off point can be applied at low cost. This facilitates further analysis and provides a wider range of options for possible users of the technique without the need to re-execute the clustering procedure when a different granularity level is required. The hierarchical clustering was done in conjunction with cosine dissimilarity as a distance metric. We have found that cosine dissimilarity results in a better clustering over Euclidean distance. We have also selected Ward's linkage criterion [224][225] since we found it performs better than single, average and complete criteria based on our empirical observation. Figure 4.2 shows a dendrogram of the resulting clustering for both datasets.

4.3 Empirical Study Design

4.3.1 Research Questions

We investigate the three research questions below in order to assess the effectiveness and efficiency of our proposed feature-claim based clustering technique.

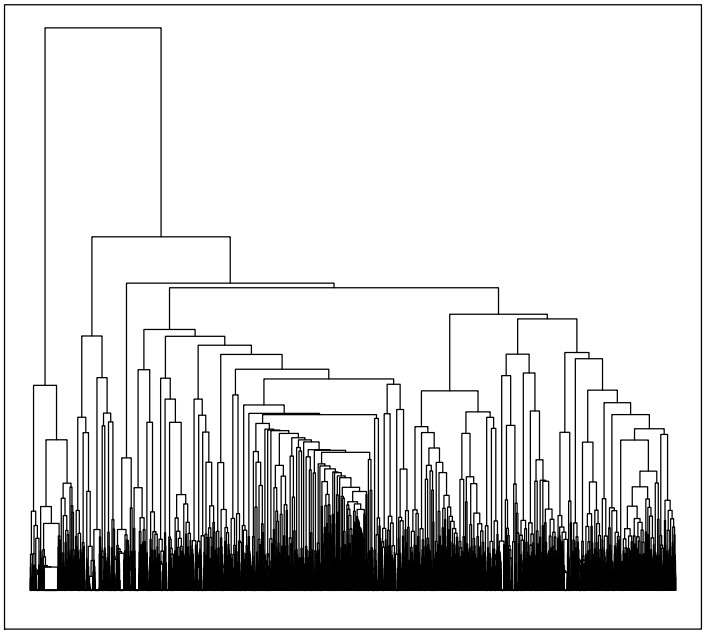
RQ1. Sanity Check: What is the baseline cluster quality of the commercially given app categories?

There is no ground truth for app store categorisation. Nevertheless, we can apply a 'sanity check' as an internal validity check on the categorisation we produce using our technique. Although we do not seek to replicate nor replace the existing commercial categorisation, it would be somewhat perverse if we would find that categorising according to claimed features produces a worse cluster quality than that of the existing app store categories. Our clustering is based on the features we extract, and has a finer granularity. Therefore it should perform better than the given commercial categories (which may not group apps according to their claimed features, and which are constructed at a coarser level of granularity). We therefore use the silhouette width (explained in Section 4.3.3) to assess the quality of the given clustering denoted by the current commercial app categories. This forms a baseline for comparison of clustering quality of the clusterings of the app store that our technique produces.

RQ2. Granularity: What is the clustering performance at different granularity levels?

The 'granularity' of a categorisation is determined by the number of different categories (i.e. clusters)

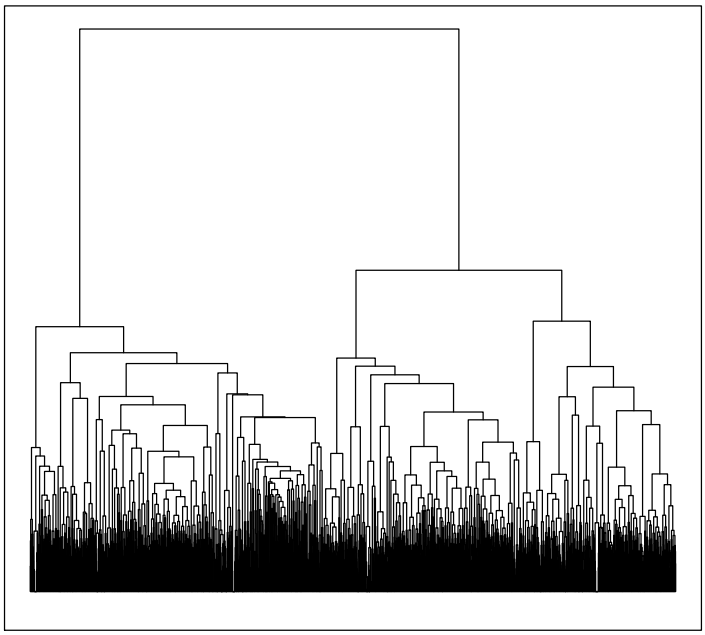
BlackBerry Apps Clustering Using Cosine Dissimilarity



dist_bb_cos

(a) Blackberry

Google Apps Clustering Using Cosine Dissimilarity



dist_google_cos

(b) Google

Figure 4.2: Dendrogram of the resulting agglomerative hierarchical clustering using cosine dissimilarity and Ward's criterion.

it contains; the more categories the more fine-grained is the granularity and the more detailed are the distinctions it makes between groups of apps.

Since we use agglomerative hierarchical clustering, a user of our clustering technique has the option of choosing particular granularity within the constraints of the clustering dendrogram that best suits their usage context. We assess the effectiveness of each choice using the silhouette width [216] (explained in Section 4.3.3), so that we aid in the selection process with a range of viable granularity options based on their silhouette scores. A higher silhouette score will tend to have more cohesive clusters of similar apps. RQ2 can be asked of both of the app store level and also within each of the given commercial categories of the app store. We therefore split RQ2 into two sub questions:

RQ2.1: What is the overall range of viable granularities for each app store studied?
We investigate the overall choice of granularity for each app store and compare it to the performance of the given commercial categorisation in terms of the silhouette width. The answer to this research question provides a baseline for comparison to future work with other clustering techniques. It may also be useful to app store providers, since it indicates a range of granularities at which it may be useful to re-categorise the app store into subcategories.

RQ2.2: What is the range of viable granularities for each given commercial category within each app store studied?
By answering RQ2.2 we seek to understand whether different categories within the commercial categorisation have different behaviours. We study how the silhouette width score performs as the granularity grows finer; and at what level of granularity does the silhouette score reaches its maximum value. Since the granularity refers to the number of distinct sets of feature claims that can be found to reside within the category, it would also indicate, loosely speaking, the 'amount' of functionality claimed within each.

RQ2.3: Which is the correlation between maximum cluster granularity and size?
We compare the maximum achieved cluster granularity of a certain category with the number of apps residing in that category. This gives us evidence as to whether this quantity is merely a product of the quantity of apps deployed within each category, or whether some categories have inherently more claimed functionality than others. We use the Spearman rank correlation test (explained in Section 4.3.3) to determine the degree of correlation between the ranking of commercial categories according to the best-performing granularity for our clustering, when compared to the number of apps in each category. A high correlation suggests that larger categories simply contain more features because they contain more apps; it will provide evidence that there is a consistent amount of feature claims per app category, and the categories in general, provide a similar quantity of claimed functionality. By contrast, a low correlation indicates that some commercial categories contain a larger number of feature claims per app than others.

RQ3. How does the clustering solution compare to a ground truth?
After analysing our clustering approach based on internal criteria (silhouette width score) in RQs 1-2 which show the general cohesion of clusters, we analyse it in a more qualitative manner based on external criteria (human judgement). To this end we use simple random sampling of app pairs (without replacement) and check if human raters concur with the cluster assignments at different levels of granularity. In particular, we investigate the Spearman rank correlation between the human similarity rating and the finest granularity that the app pair remain in the same cluster as we move

down the dendrogram. If there exists a positive correlation it shows that our technique is likely to cluster together apps that are deemed similar by humans.

RQ4. Efficiency: How efficient is the proposed framework?

In order to be usable, the set up cost and subsequent instantiation cost must be within reasonable bounds, to allow developers to use our approach to help understand the claimed-feature competitive space into which they deploy their apps. As explained in Section 4.2, our approach builds two matrices, i.e. FTM and AFM. Building the FTM is an expensive, upfront, once-only computation but it greatly reduces the dimensionality of the AFM.

4.3.2 Dataset

The data we used was collected in August 2014 from BlackBerry World [5] and Google Play stores [6]. The data was crawled from the web collecting the metadata of free and paid apps including the raw app description and category. A total of 14,258 apps from all 16 different categories was collected from the BlackBerry store, and 3,673 apps from all 23 high-level categories in the Google Play store. The list of categories for each app store and the sizes (number of apps) is shown in Table 4.4.

4.3.3 Evaluation Criteria

In this section, we explain the metrics and statistical analysis we perform to answer our research questions.

We use the **Spearman rank correlation** [226], to investigate the degree of correlation between the maximum feasible granularity for each category and the number of apps in the category (RQ1); and the degree of correlation between human-assigned similarity score of an app pair and the finest granularity the pair remains in the same cluster (RQ3). Spearman's correlation is based on the ranks, and therefore is more suitable to an ordinal scale metric [227], such as that provided by the silhouette method or a similarity score, than linear regression analysis or other parameterised correlation analyses. Spearman rank correlation between two orderings gives us the degree of correlation, expressed as a correlation coefficient, ρ, together with an indication of the significance of the correlation, computed as a p value. The value of ρ is constrained to lie between -1.0 and 1.0; the closer the value of ρ to 1.0, the stronger the correlation, while the closer to -1.0, the stronger the inverse correlation. Values of ρ close to 0 indicate an absence of any (strong) correlation, with a value of zero indicating exactly no correlation. The p value determines the probability of observing the given ρ value were there to be no correlation (that is, were the true ρ value to be zero).

In order to evaluate the clustering results (RQ2), we use the **silhouette width** [216] with the cosine distance. The silhouette width score derives from how similar each data point is to other data points in the same cluster in addition to how different it is from data points in other clusters. The silhouette value for each datum ranges from 1 to -1. 1 denotes a perfectly assigned cluster element. 0 denotes the border of two clusters, while -1 denotes a completely mis-assigned cluster element. By averaging silhouette scores for each member of each cluster, we obtain a measurement of how well assigned elements are to their clusters (and averaging over all clusters gives the corresponding silhouette width assessment for the clustering as a whole). The name of the technique, 'silhouette width', derives from the visualisation of the values for each element in each cluster. By plotting

[5] http://appworld.blackberry.com
[6] http://play.google.com

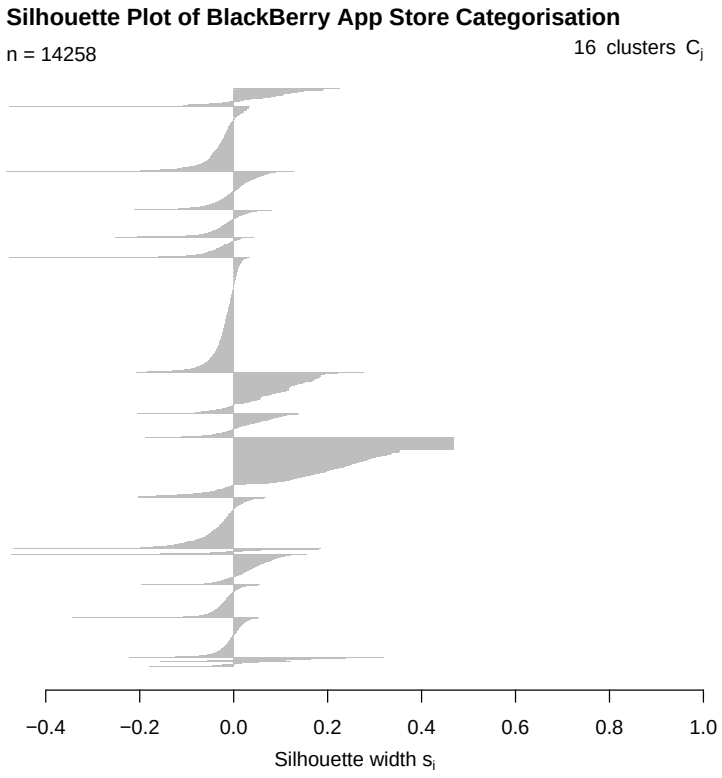
these values on a vertical axis, we obtain the 'shadow' (or silhouette) for each cluster, the upper bound of which is determined by the silhouette value of the best-placed element in the cluster. The elements of the cluster are plotted in ascending order of silhouette value, as we move up the vertical axis, giving a monotonically expanding 'shadow' for each cluster. If the shadow expands to the left (negative silhouette values), then the elements are (very) poorly placed, while expanding to the right (positive silhouette values) indicates elements that are better placed. This visualisation is extremely intuitive: More ink on the right-hand side of the vertical indicates a better clustering, while more ink on the left-hand side indicates worse clustering. Indeed, *any* ink on the left hand side of the vertical indicates elements that are probably in the wrong cluster. Furthermore, for two clusters that reside entirely on the right-hand side of the vertical, equal distribution of ink among the clusters (whether horizontal or vertical), indicates the relative quality of the two clusterings.

In RQ3, we assess the inter-rater agreement using the **Intraclass Correlation Coefficient** (ICC) [228]. There are many ways of measuring the degree of consistency of multiple raters depending on the number of participants and the type of scale used. Cohen's Kappa and Weighted Kappa [229] for example, are only used when there are two raters. Alternatively, Fleiss' Kappa [230] is used when there are more than two raters; however, it is only suited when the rating system is nominal or categorical. Since, we use a semantic differential scale (a Likert-like rating scheme), our rating scheme is ordered, thus we need a measurement that is sensitive to the degree of difference in the rating scale. For example, it should deem two ratings of 4 and 5 as more consistent than two ratings of 3 and 5. In such cases, Kendall's Coefficient of Concordance (W) and ICC are used. We select ICC to avoid the effect of rank ties that Kendall's W exhibits when the subjects of the ratings are not strictly ranked. ICC assigns a value of consistency among the raters that ranges between 0 and 1. Low values indicate high variations of scores given to each item by the raters; high values indicating more consensus. We use a two-way ICC model since both the rated app pairs and the raters are representative of a larger population.

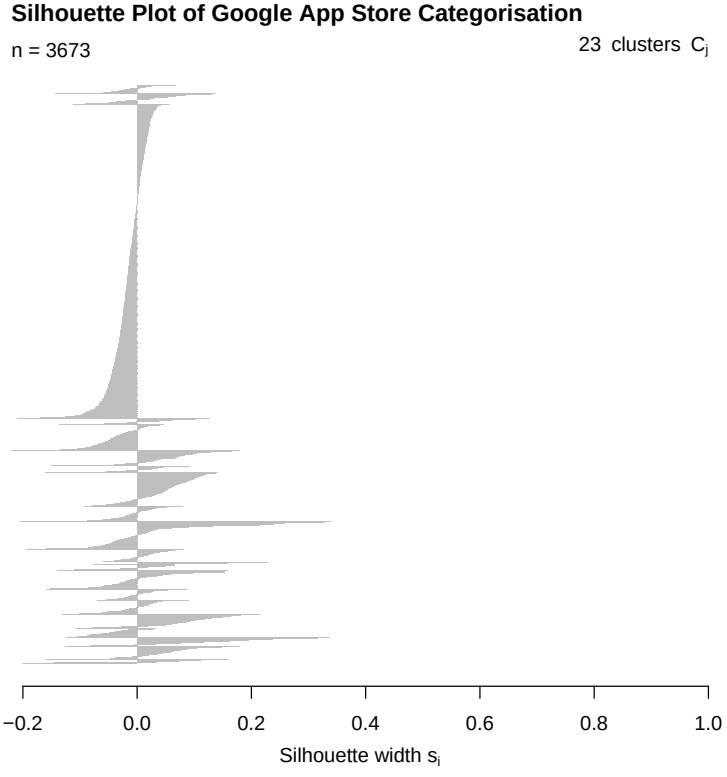
4.4 Results Analysis

RQ1. Sanity Check. We use the silhouette width to measure how well data points are grouped in the existing app store categorisation. A summary of silhouette scores found for each category of the two app stores is shown in Table 4.2. The slihouette width scores are depicted in Figure 4.3. In the BlackBerry store, the mean of categories' silhouette scores is 0.02, a mean of 0.09 when using our clustering technique for that particular granularity. In Google Play, the mean silhouette scores are 0.03 for category memberships and 0.08 for our clustering solution memberships. This shows that the existing categorisation does not excel in segmenting the apps according to their claimed features. This could be attributed to two factors: a) Our conjecture is correct in proposing that current categorisation is not based on the apps' claimed features; b) Current categorisation is of too coarse granularity. Using our clustering technique (though cut off at a non-optimal granularity) improves upon the silhouette score in most cases. For both options: using the existing categorisation as a preliminary cut-off point, or clustering all apps from scratch, the silhouette may improve upon exploiting finer granularity thereafter. This is investigated by the next research question (RQ2).

RQ2. Best performing granularity. Selecting an *optimal* clustering is an empirical task where the stakeholder balances the quality of member assignments to clusters and a feasible granularity



(a) Blackberry



(b) Google

Figure 4.3: RQ 1. The silhouette width for each category in the existing categorisation of the BlackBerry and Google datasets.

Table 4.2: RQ1. Summary measures (Min, Max, Mean and Median) of the silhouette widths of existing categories and those achieved when applying our clustering approach using as granularity the number of existing categories in the stores considered.

BlackBerry World (granularity = 16)				
	Min.	Max.	Mean	Median
Existing categorisation	**-0.04**	0.21	0.02	-0.01
Clustering solution	-0.09	**0.31**	**0.09**	**0.08**
Google Play (granularity = 23)				
	Min.	Max.	Mean	Median
Existing categorisation	**-0.05**	0.22	0.03	0.01
Clustering solution	-0.08	**0.95**	**0.08**	**0.03**

Table 4.3: RQ2.1. Comparing the individual element's silhouette widths of: The current quality of the existing categorisation, our clustering reusing existing coarse granularity, and our clustering using the maximum viable granularity.

(a) BlackBerry (commerical granularity: 16, ideal granularity: 7416)

	Min	1^{st} Q.	Med.	Mean	3^{rd} Q.	Max
Current quality	-0.48	-0.03	-0.01	0.02	0.03	0.47
Reusing coarse-grain	-0.45	-0.09	-0.01	0.02	0.08	0.55
With max. fine grain	-0.42	0.00	0.37	0.45	1.00	1.00

(b) Google (commerical granularity: 23, ideal granularity: 1769)

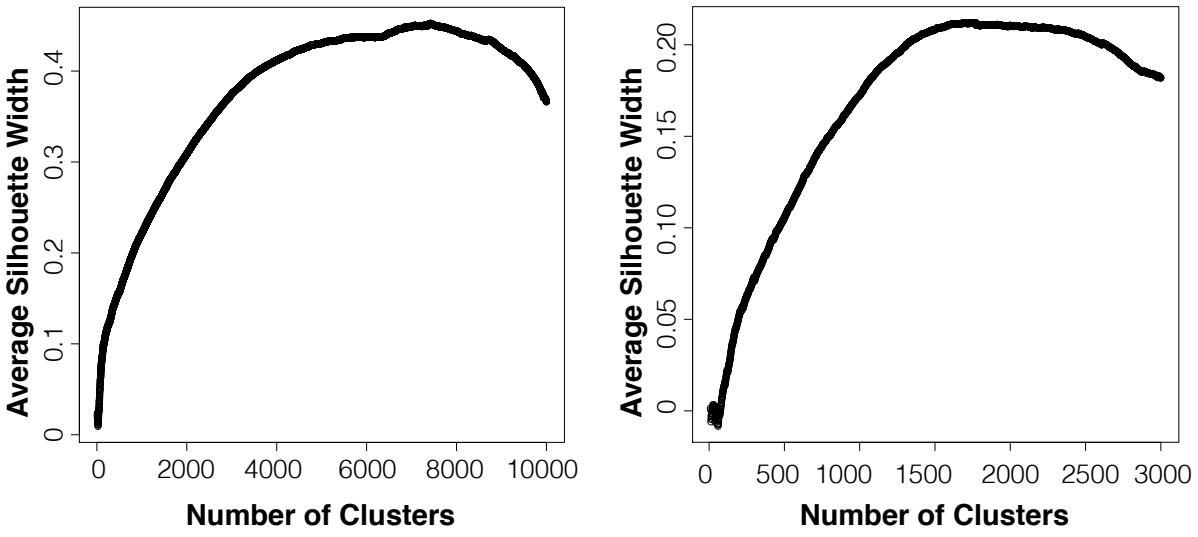
	Min	1^{st} Q.	Med.	Mean	3^{rd} Q.	Max
Current quality	-0.22	-0.03	-0.01	0.00	0.02	0.34
Reusing coarse-grain	-0.67	-0.06	-0.01	0.00	0.04	0.98
With max. fine grain	-0.32	0.00	0.10	0.21	0.33	1.00

depending on their usage context. It is then useful to study the behaviour of the silhouette scores as the granularity increases or decreases in the context of clustering apps. This also serves to provide an indication of the performance of the clustering technique at different granularity levels. To achieve this, we generate a solution for every possible granularity, then we measure the silhouette scores of each cluster in the generated solutions.

RQ2.1. Overall granularity. When observing the average silhouette scores of the clusters, we find that in both stores the scores steadily increase as the granularity increases (see Figure 4.4). The average silhouette score then peaks before dropping. The BlackBerry dataset yields a peak of 0.45 in mean silhouette when segmenting into 7,416 segments. Whereas when segmenting the Google dataset into 1,769 segments, the silhouette achieves its highest mean score of 0.21. This serves as an upper-bound granularity as it is the finest-granularity that can be achieved before sacrificing quality. We observed that at this level of granularity, clusters tend to have few (2-3) apps that are highly similar: They are the free (lite) and paid (full) versions of the app (e.g., 'App Task Manager Free' and 'App Task Manager Pro'), or a set of apps that belong in the same suite (e.g., 'MapMy:WALK', 'MapMy:FITNESS' and 'MapMy:HIKE'). This upper-bound granularity may be useful to stakeholders if their usage context requires a very fine distinction of apps (such as detecting app suites or free and paid versions of same apps); otherwise, a coarser granularity can be selected. We, hereinafter, call this level of granularity the *maximum feasible choice of granularity* for that particular app store. This clear stopping point shall aid in the analyses conducted in RQ.3 since there is no

Table 4.4: RQ2.2. Categories, their size, and granularity level that provides the highest silhouette width for each app store category when sub-clustered.

BlackBerry			
Category	Size	Granularity	Silhouette
Books	142	76	0.58
Business	813	397	0.33
Education & Reference	1260	706	0.46
Entertainment	1595	816	0.54
Finance	588	325	0.32
Health & Fitness	506	248	0.37
Music & Audio	1025	473	0.57
Navigation & Travel	953	480	0.34
News & Magazines	1474	662	0.62
Photo & Video	753	401	0.36
Productivity	974	460	0.26
Shopping	144	83	0.34
Social	668	379	0.31
Sports	439	179	0.49
Utilities	2832	1974	0.34
Weather	92	67	0.32
Total	14258	7726	Mean: 0.41
Google			
Category	Size	Granularity	Silhouette
Books & Reference	34	20	0.20
Business	23	17	0.35
Communication	65	26	0.17
Education	90	58	0.27
Entertainment	164	70	0.22
Family	79	46	0.19
Finance	20	11	0.20
Games	2002	964	0.21
Health & Fitness	84	46	0.23
Lifestyle	59	32	0.20
Media & Video	40	22	0.24
Music & Audio	98	57	0.20
News & Magazines	18	4	0.23
Personalization	121	53	0.32
Photography	89	53	0.19
Productivity	99	58	0.19
Shopping	42	14	0.17
Sports	213	120	0.19
Social	56	28	0.15
Tools	144	66	0.23
Transport	33	26	0.37
Travel & Local	69	37	0.20
Weather	31	24	0.24
Total	3673	1825	Mean: 0.23



(a) BlackBerry

(b) Google

Figure 4.4: RQ2.1 The average silhouette width for the different number of segments in BlackBerry and Google dataset.

need to go beyond that point for evaluation purposes.

RQ2.2. Refining Commercial Categories. To answer this question, we run our algorithm to analyse the average silhouette scores for each possible granularity of each commercial category. The process is further clarified in Figure 4.5. Note that the quality of the sub-clustering is influenced by the existing categorisation of the app stores. We show in Table 4.3 that clustering from scratch generates better silhouette scores on average. In analysing the behaviour of the silhouette scores when increasing the granularity, we find that they exhibit a 'plateau' effect where they reach a certain mean silhouette range, remain relatively stagnant, before dropping as the number of sub-clusters increases. This is insightful and may prove useful to stakeholders as it provides a wider range of granularity without a noticeable sacrifice in the clustering quality. In Figure 4.6 and 4.7 we show for each category the behaviour of the silhouette score as the number of clusters increases in the BlackBerry dataset. In general, categories seem to reveal higher tendency of good clustering towards the second quartile of the data size. We also notice that BlackBerry reveals better clustering tendency than Google. We conjecture that the size of the two datasets greatly influences the results, BlackBerry being the larger, more representative of the two. As in RQ 2.1, we also report the maximum point at which the mean silhouette peaks before dropping (Table 4.4) since it provides a clear cut-off point where further sub-clustering may not be feasible. In the Google Play store, the mean silhouette scores peaked at 0.35, whereas in the BlackBerry store, the highest score was achieved at 0.58, showing that the BlackBerry categories `Books, Entertainment, Music & Audio, and News & Magazines` may benefit from further meaningful sub-categorisation.

RQ2.3. Correlation between maximum cluster granularity and size. The results of RQ2.2 suggest that the granularity of different categories varies according to their sizes; this may give an insight on how homogeneous are the apps in these categories. For example, in the Google Play's `News & Magazines` category, 14 apps can be classified into 4 clusters without sacrificing the clustering quality; however, in the `Transport` category, the 33 apps are of highly diverse set of features that

26 sub-clusters are needed to make a proper distinction between those apps.

When investigating the degree of correlation between the maximum feasible granularity and the size of the category, we find high positive correlation ($\rho = 0.96, p - value < 0.001$ for Google Play and $\rho = 0.99, p - value < 0.001$ for BlackBerry). That is, the larger the size of the category, the larger the maximum granularity sub-clustering achieves for that category. This may indicate that larger categories contain more and larger variety of features.

RQ3. Comparison to a ground truth. To compare our clustering against human judgement, we manually label a gold set of app pairs to act as a baseline for comparison with the clustering approach. Due to the abundance of possible clustering solutions based on the selected granularity level (which can range from 2 clusters up to the maximum feasible granularity), we draw a simple random sample without replacement from the population after sratification. This resulted in a sample of 300 apps comprising 150 app pairs from 5 different levels distributed over the feasible granularity intervals [2,7416] and [2,1769] for BlackBerry and Google app stores, respectively. Simple random sampling is a basic sampling technique that randomly draws from the population with a uniform probability distribution (i.e. all individuals have the same chance of being selected). Stratified sampling is used to ensure all types of pairs are equally represented in the sample. This sampling results in 300 apps in total (150 app pairs). The selected strata lie at the 0%, 20%, 50%, 75% and 100% of the feasible interval for each app store. From each granularity level, we randomly select 15 app pairs that belong at that level but not beyond (they are separated at the following granularity level). The app pairs at level 0% are apps that are separated from granularity level = 2 to represent apps that are immediately deemed dissimilar by the algorithm. This is done to ensure that the sample is not biased towards a certain similarity level. Table 4.5 shows examples of app pairs together with the feature terms that they share.

To statistically analyse the behaviour of our sample compared to human evaluation of similarity, four of the researchers, who were not involved in selecting the sample and were not aware of the results of the clustering, rated the similarity of the selected random sample on a 5-level semantic differential scale [231][232] with 'unrelated' and 'similar' as the bipolar adjectives of the scale. To measure the inter-rater agreement we use Intraclass Correlation Coefficient (explained in Section 4.3.3). The achieved ICC is 0.7 ($p - value < 0.001$) thus rejecting the null hypothesis that the raters do not agree. We also compute the correlation between the mean of the similarity scores assigned by the 4 raters to each app pair with the finest level of granularity that the pair survives in the same cluster. 4.6 shows the found correlation coefficients for each rater and the combined mean similarity scors for all raters. We find mild positive correlation especially on the Google dataset

Table 4.5: RQ3. App pair examples and the feature terms that they share selected from the feature cluster prototype.

App Pair	Granularity	Common Feature Terms
Matalan Reward Cards Voucher Codes UK	1,796	Redeem, Save, Conduct, Trade, Manage, Selling.
Advanced Phone LED MissingLight - color LED	6,222	Color, Tint, Call, Ring, Direct.

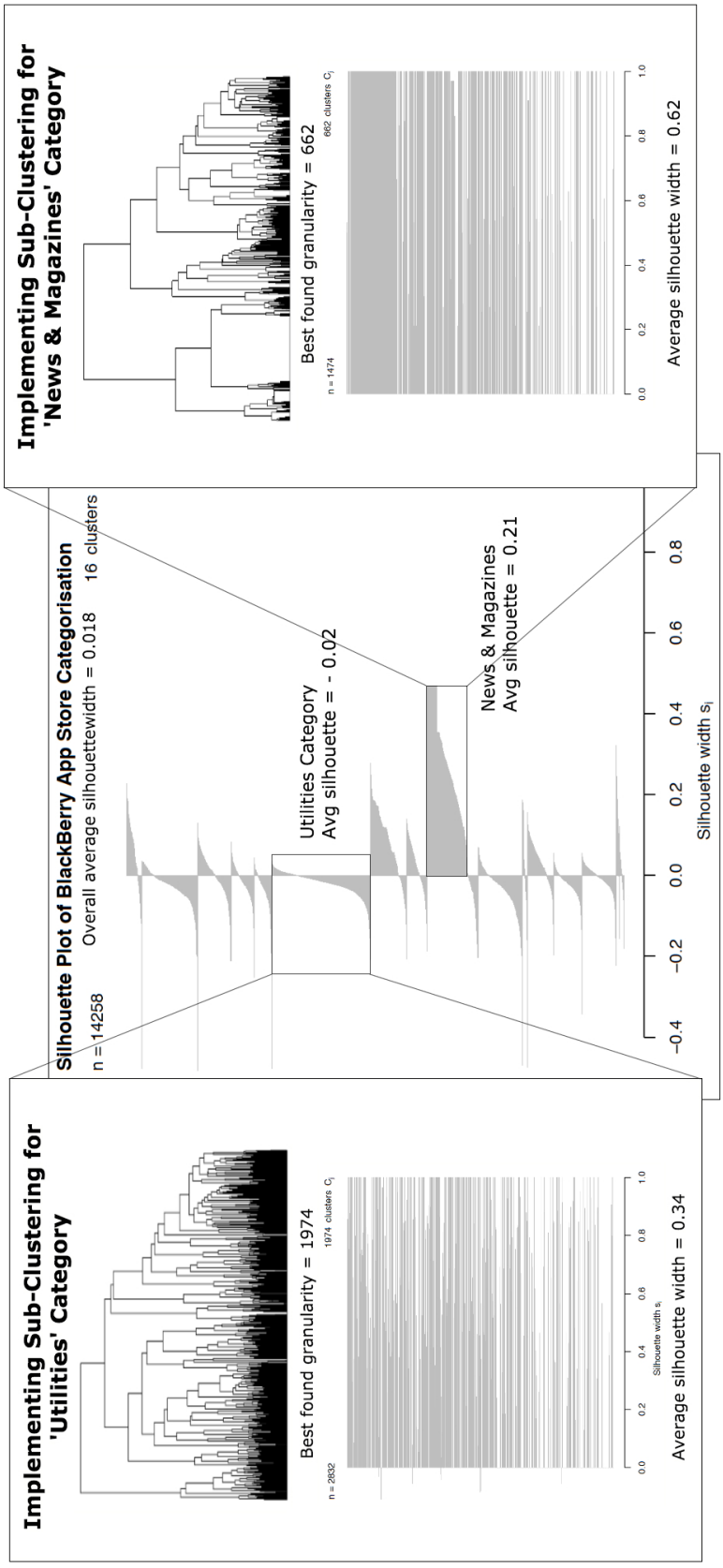


Figure 4.5: (Middle) The average silhouette score for each existing commercial category in the BlackBerry app store when measured using our app representation technique (RQ1). As an example, we zoom in the Utilities category (which exhibits a low current avg. silhouette) and the News & Magazines category (which has a relatively high current avg. silhouette). The left- and right-hand sub-figures illustrate how our approach can be used to uncover the clustering within each category (RQ2.2).

Table 4.6: RQ3. The Spearman rank correlation coefficient (ρ) between each of the raters similarity scores of pairs of apps and the finest granularity that these apps remain clustered together before they separate. All reported coefficients have $p-values < 0.001$.

	BlackBerry	Google
Rater 1	0.50	0.56
Rater 2	0.45	0.59
Rater 3	0.39	0.42
Rater 4	0.55	0.39
Mean Ratings	0.52	0.61

(Google: $\rho = 0.61, p-value < 0.001$, Blackberry: $\rho = 0.52, p-value < 0.001$). This shows that if our technique classifies apps together at deep levels, there is a likelihood that these apps are also deemed similar by human evaluation.

RQ4. Efficiency of the proposed solution. As explained in Section 4.2, our framework is based on two main phases: one is constructing the FTM from the mined features and the second is constructing the AFM. Constructing the FTM is a costly procedure. However, it is considered as once-only upfront cost as it is required less frequently. The reported time measures are susceptible to the size of the data, we provide the dimensions of our data for clarification purposes. Table 4.7 provides detailed run time of the major tasks of our framework. Building the FTM, Google Play dataset resulted in a total of 28,100 features and 8,747 unique terms (corresponding to the rows and columns of the matrix, respectively); whereas BlackBerry resulted in 23,337 features and 7082 unique terms. The construction was done in approximately five days on a standard machine (3.1 GHz Intel Core i7 Processor, 16 GB RAM). However, performing the clustering is substantially faster as it takes an hour on average. To expedite the clustering and silhouette evaluation steps, the dissimilarity matrix is first computed for the entirety of the data, this step costs approximately 43 minutes. The resulting clusters are then used as columns in the AFM. The Google dataset generated an AFM with 3,673 rows (number of apps) and 353 columns; the BlackBerry dataset generated 14,258 rows and 499 columns. Constructing the AFM is done in less than a minute and the hierarchical clustering in few seconds. Since the feature clusters are clustered using a prototype-based technique, the cluster prototypes are preserved, as new apps emerge, their extracted features

Table 4.7: RQ4. Computation time of tasks in our framework. The first two tasks are up-front costs.

Task	Time	Unit
Building FTM	5.00	days
FTM Dissimilarity Matrix (avg)	42.83	min.
Clustering FTM	43.38	min.
Building AFM	7.20	sec.
AFM Dissimilarity Matrices (avg)	0.37	sec.
Hierarchical Clustering (avg)	0.48	sec.
Calculating the Silhouette (avg)	12.00	sec.
Detecting Overall Granularity (avg)	6.00	hours
Detecting Category Specific Granularity (avg)	22.13	sec.

may be allocated to a cluster with the most similar prototype.

4.5 Threats to Validity

Internal Validity:

We carefully applied the statistical tests verifying all the required assumptions. As in every clustering solution, finding the optimal number of clusters remains ambiguous. To cluster the mined features, we use a popular method (Can's Metric) that has been used in similar problems with good results [21]. Another threat to internal validity could be due to the apps composing our datasets (a.k.a. App Sampling Problem [233]). Threats may also arise due to the procedure we used to build the gold set. However, the number of human raters is consistent with that in previous similar studies (e.g., [112]). Moreover, when selecting random app pairs, we prevent a bias towards a majority of a certain degree of similarity by using stratified sampling [234], thus ensuring that the sample contains apps with varying degrees of similarity.

Construct Validity: Previous studies have shown that it is possible to extract features from product descriptions available on-line [24][21][23][235][236]. However, these features are extracted from claims reported by app store developers and we cannot be sure that these necessarily correspond to features actually implemented in the code itself, since developers do not always deliver on their claims [111]. We mitigate this threat by extracting the features from a large and varied collection of app descriptions, and clarifying that it is clearly a constraint of our method (and of most NLP-based approaches [21]). Nevertheless, we believe that developers' technical claims about their apps are inherently interesting to requirement engineers and *however* we view them, they have interesting properties in real world app stores (see e.g., [63][66]).

External Validity: Though our features extraction method can be applied to any app stores, our empirical results are specific to the stores considered. More work would be needed to investigate whether the findings generalise to other time periods and app stores.

4.6 Conclusions

We proposed a new framework that segments the apps in an app store into groups of apps that claim similar features. An endeavour guided and motivated by the findings reported in Chapter 3. We show that current categorisations in Google Play and BlackBerry app stores do not exhibit a good classification quality in terms of this claimed feature space.

We then embark on devising the range of possible granularities for our discovered segmentation of the app store space. We also use our technique to find the possible sub-categorisation of the categorisation of the app store. We further report on the range of possible granularities for clustering, over the app store as a whole and within each existing commercial category in the app store, by computing the silhouette values for each possible choice of granularity. We also compare the performance of our approach to a sample of 300 apps manually labelled as score of similarity by four of the authors. The results reveal that there exists a positive correlation between the mean similarity score assigned by the raters and the finest granularity in which the rated apps remain together in our approach. Finally, we report on the computational cost of our approach, revealing that there is a large upfront cost in the semantic ontological analysis to determine features and their similarity, but the subsequent cost of constructing clusterings is cheap. We believe that the upfront

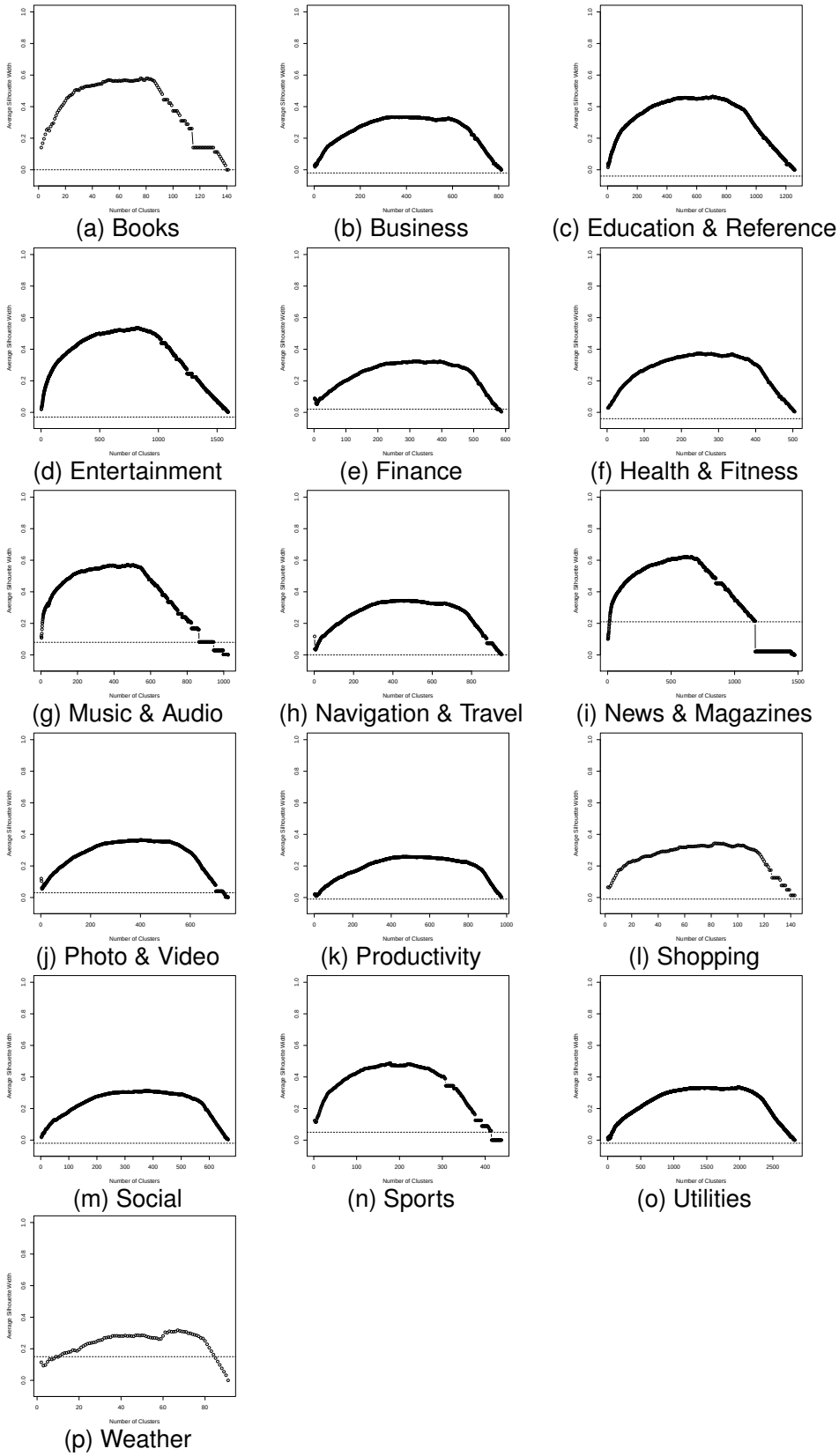


Figure 4.6: RQ2.2. Refining the existing BlackBerry app store categorisation by sub-clustering each category and observing how the average silhouette scores behaves as the granularity increases. The x axis is the granularity whereas the y axis is the achieved average silhouette for that granularity. For each category, the dotted line is the current average silhouette score for the category's members in relation of the entire app store.

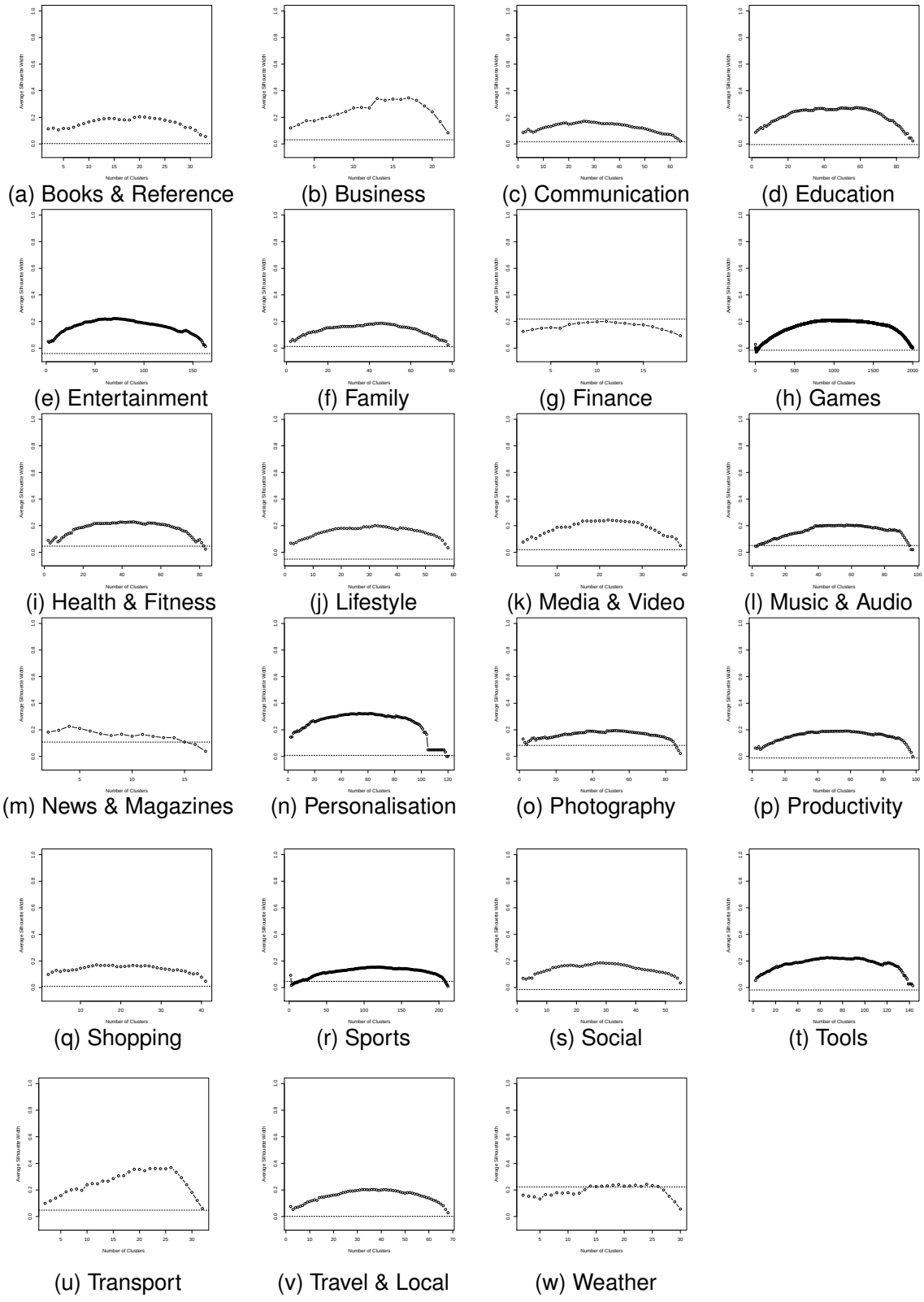


Figure 4.7: RQ2.2. Refining the existing Google app store categorisation by sub-clustering each category and observing how the average silhouette scores behaves as the granularity increases. The x axis is the granularity whereas the y axis is the achieved average silhouette for that granularity. For each category, the dotted line is the current average silhouette score for the category's members in relation of the entire app store.

cost is worthwhile, considering it need not be repeated as often as the clustering. Furthermore, this upfront cost took several days for us, but this computational cost was that required for when using a standard laptop. A commercial app store provider could exploit cloud resources to bring this cost down to very reasonable elapsed time, thereby allowing dynamic categorisation of app stores, to explore emerging trends on a weekly or even daily basis.

Chapter 5

Comparison of Feature Extraction Techniques for App Clustering

5.1 Introduction

Software classification has been used in many studies to help address several software engineering problems. In the case of mobile applications (apps), classification becomes a particularly viable option as apps are hosted in an app store ecosystem ripe with metadata [63][237]. Previous research investigated mobile application categorisation for various goals including detecting anomalies [70, 111, 238], finding software clones[239, 105], grouping of requirements into coherent sets [21, 103, 122], detecting semantically related code chunks [107, 92], finding actionable insights [66][108], and classifying systems into application domains [96, 106, 99].

In the exploratory study, presented in Chapter 3, we find that more than half of surveyed app developers look into similar app in the app store for features to include in their own during both design and perfective maintenance stages. We therefore proposed (in Chapter 4) a feature-based clustering algorithm that can segment the app store based on advertised features in a finer granularity.

Clustering (i.e. unsupervised categorisation) enables the detection of latent segmentation of the dataset without a ground truth known a priori. This is very useful when a taxonomy is not available and/or it is too costly to carry out the labelling process. It also enables grouping software according to different criteria depending on the feature extraction technique used. However, since clustering is an exploratory endeavour that helps uncover underlying, seemingly unknown, segmentation of data, the way of measuring similarity may affect both its efficiency and quality. On the other hand, natural language processing and information retrieval advances continue to greatly benefit several software engineering outstanding issues [240].

One of the major benefits of the technique presented in Chapter 4 is its reliance on natural language which is easily obtained from app stores. However, this method is not the only natural language feature extraction technique that is used in software engineering literature, and no study have been carried out to compare their performance for the task of categorising mobile applications based on their descriptions.

Therefore, in this chapter, we investigate a set of well-known text similarity techniques to determine how effective they are for this problem and provide an empirical comparison of their respective behaviour for 12,664 real-world apps extracted from the Google Play (Android) app store. Firstly, we use the evidence of feature claims present in developers' description of their apps, which we

extract using the text mining feature extraction framework proposed by Harman et al. [63] and used in Chapter 4. We then compare this technique with a clustering technique based on extracted topics using topic modelling (Latent Dirichlet Allocation), the regular Vector Space Model and finally, we use a variation of the feature extraction framework that relies on keyword extraction using sentence dependency parsing to build a similarity matrix. Finally, we employ agglomerative hierarchical clustering to detect natural groupings in the data based on all previous text representation techniques. A hierarchical technique gives flexibility with regards to the desired granularity of the final grouping.

In using textual descriptions to cluster mobile apps, we only consider those textual features that developers believe to be important to future users' decision to acquire the app. Whereas categorisation based on such features provides an interesting view of the app store relying on the developers' professed features, we do not claim that this is the only view of app store segmentation.

The primary contribution of this study lies in the comparison of a feature-based categorisation of app stores, using the vector space model as a baseline (with TF-IDF), topic modelling (LDA) and two keyword feature extraction approaches we dub the feature space model (FSM). The results reveal that, indeed, using LDA and/or FSM improves on the baseline with regards to three evaluation criteria: functional, domain and API similarities. We also observe that dependency-based keyword extraction performs best in terms of detecting application domain similarity.

5.2 Empirical Study Design

In this section we describe the design of the empirical study by presenting the similarity measurement techniques compare (Section 5.2.1), the research questions we aim to answer (Section 5.2.2), and the dataset (Section 5.2.3) and evaluation criteria (Section 5.2.4) used to this end.

5.2.1 Text Representation Techniques

At the heart of any clustering solution lie two important choices: the methodology used to represent the data points and the distance metric used to capture the difference among these data points. Common clustering techniques define a set of finite variables that describe the data which can be converted to numerals and appended to form a vector representing the data point. Then, given all the data points, they form a vector space representing the data, and cluster the data based on the vector space and a distance metric. Various geometrical distance metrics (e.g. Euclidean and cosine distances) can be used to quantify the similarity and lack thereof among the data points. More specifically, clustering methods usually only need, as input, a distance matrix of size $n \times n$ where n is the number of data points to be clustered, the columns and rows are the data points and each cell contains the distance between these two data points.

In this study we empirically observe how five textual feature extraction techniques perform when used as clustering input, applied on the case of mobile app descriptions. The textual description clustering baseline used in our study relies on the Vector Space Model representation of the data using Term Frequency-Inverse Document Frequency weighting, with and without latent semantic indexing. We compare this with three more advanced textual-based feature extraction techniques that have been used in software engineering research: topic modelling [241], collocation-based feature extraction [63] (presented in Chapter 4) and an enhancement on the latter we propose in this chapter relying on dependency parsing of sentences to extract software features. The following

subsections explain each of these techniques in further detail.

5.2.1.1 Vector Space Model

The Vector Space Model (VSM) is a textual representation baseline technique that relies on a bag-of-words (BOW) approach [242][243]. Bag of words approaches discard information regarding ordering of the terms in the document. The vector space is represented by a document-by-term matrix (DTM). Each data point (i.e. document) is a vector (i.e. row in the matrix). Each cell in the matrix convey whether the document being represented contains this term or not (or a weight that conveys the association between the document and the term).

Term Frequency/Inverse Document Frequency: TF-IDF [244] is a baseline weighting approach for VSM and has been shown to work well for ranking of documents in information retrieval literature [245]. TF-IDF is a weighting scheme that is used to link corpus terms with data points (i.e. documents) within a DTM. Term Frequency is the frequency of occurrence of a term in the document in question. This is then multiplied with the inverse document frequency: the logarithm of the total number of documents in the corpus divided by the number of documents that contain the term in question. Modifying the term weights with the IDF score increases the specificity of the terms, thus giving higher weight to less common terms. The assignment of the weights then comprises a document vector that is then used to solve various information retrieval problems. In this study, this vector is used to represent the applications in the dataset which is fed to the clustering algorithm.

Latent Semantic Analysis: LSA employs matrix dimensionality reduction in order to index the DTM in a way that approximates semantic closeness [246] thus revealing the higher order semantic structure of the text after eliminating noise introduced using synonyms or word-sense ambiguity. Latent semantic analysis is also referred to as latent semantic indexing and we do not make a distinction between the two terms in this chapter. At the heart of LSA, singular-value decomposition (SVD) is used to decompose the document-term matrix; this results in three matrices: a term vector matrix, a document vector matrix and the singular values matrix, such that the original matrix can be obtained from the product of the three matrices. The latent semantic space is then obtained by truncating the matrices to a certain number of dimensions k (i.e. only the k largest singular values are used to reconstruct the original matrix).

The number of selected dimensions can greatly affect LSA results. In our study, we operate a share-based dimensionality search routine in which k is the number of singular values (in a descending list) whose sum reaches a certain value. In our search, we set the value to 0.05% of the sum of all singular value. We also use several other techniques and different other share thresholds before settling on the aforementioned option as it produced the highest cluster quality in further steps according to the silhouette width score.

In our study, we include VSM-based representation as well as an LSA enhanced one in order to observe whether LSA can enhance the clustering results for the problem of classifying mobile applications based on their textual descriptions.

5.2.1.2 Latent Dirichlet Allocation

Topic modelling is the usage of statistical models to infer a set of topics in textual corpora.Latent Dirichlet Allocation (LDA) is a a generative probabilistic model. It operates by assuming that each document contains a latent mixture of topics thus uses a three-level hierarchical Bayesian model to

Table 5.1: Examples of extracted topics (represented by 4 terms) and the number of apps associated with each topic (occurrences).

Topic Terms	Occurrences
estate, home, property, real	860
gps, locate, time, map	664
account, bank, check, mobile	605
care, health, medical, patient	296
call, contact, phone, text	255
app , audio, listen, station	137
camera, image, photo, picture	85
book, item , library, search	48
airport , app, flight, hotel	29
country, currency, dollar, franc	17

discover the document's mixture of the corpus' underlying topics using a Dirichlet distribution. Topics are identified as being a distribution of terms [247].

Hence, the results of running LDA over a set of documents are the probabilities of relatedness of each document to each generated topic and each topic's distribution over the set of terms in the corpus. In this study, we use a variation of LDA that estimates LDA parameters using a sample of the dataset (since doing this over the entire dataset is typically not feasible). Gibbs sampling [248] is one commonly used sampling techniques used to solve this problem within LDA [249].

One drawback of LDA is the precondition that the number of latent topics in the dataset is already known and required to be set as a parameter. Since this is not the case in our study, we search for the number of topics that generates the lowest *perplexity* [250] (log likelihood on 10% held-out data). To this end, we generate LDA models over a large range of possible k values and select the k generating the lowest perplexity, as in previous work [115, 235, 251]. Chen et al. provide a survey of the usage of LDA in mining software repositories [252].

Table 5.1 shows examples of the extracted topics of this study's dataset (discussed in 5.2.3). Each topic is represented by the 4 top-most terms. The table also shows the number of occurrences of each topic (i.e the number of times an app is associated with that topic).

5.2.1.3 Feature Vector Space

The feature vector space model (FSM) in this study refers to a suite of techniques based on mobile app description feature extraction algorithm introduced by Harman et al. [63] which we have adapted to form a 'claimed feature space' used in app clustering in our previous work [67]. The term *feature* here refers to mobile applications' functional capabilities that are expressed in the natural language belonging to the user's domain of knowledge and can be provided by more than one app.

The feature vector space first clusters all raw extracted keywords (either collocation-based or dependency-based) to further abstract away the syntactical differences between features to capture their meaning. This is done using a modification of the vector space model and k-means clustering algorithm.

The vector space matrix is constructed such that features are rows and the columns represent all terms in the features' vocabulary. Each cell in the matrix is the multiplication of the inverse

document frequency of that term and the maximum of the semantic similarities of every term in the feature and that term. Semantic similarity are extracted from the WordNet English language ontology [253] since app store descriptions belong to the user's knowledge domain and not expected to be overly technical. The k-means clustering is conducted using cosine distance (spherical k-means). The number of clusters k is calculated using a variation of Can's metric [221] adapted by Dumitru et al. [21].

In the following we provide an overview of two variations of feature vector space model, in terms of how features are extracted from app descriptions. First we describe the technique proposed by Harman et al. relying on word collocations [63, 62]; second we describe a variation, we propose herein, that uses natural language dependency parsing. Both approaches are used in our empirical study, which compares their performance with other baseline techniques (namely, using the entire description with no keyword/feature extraction and LDA).

Collocation-based feature extraction: The original algorithm proposed by Harman et al. [63, 62] relied on extract mobile applications' claimed features from their app store descriptions. The algorithm identifies feature list patterns (if any) which are itemised lists preceded by a pre-compiled list of phrases that signifies the beginning of a feature list (e.g. 'includes' and 'latest features'), then it proceeds to extract word collocations in the form of bi-grams. Extracted similar collocations are then merged using a greedy clustering algorithm in which if a cluster of bi-gram terms shares over half of the words of another, a cluster consisting the union of the two clusters is formed, eliminating the two previous clusters. The resulting clusters are 'featurelets' of two or three terms representing one mobile application claimed feature. This algorithm has been shown to extract meaningful mobile application features (0.71 precision, 0.77 recall) [254]. It has been subsequently used to observe feature behaviour in app stores [66], their correlation with price, rating and rank [254] and their viability as features to discover latent categorisation of app stores [67] and to predict customer reactions to proposed feature sets [64]. A detailed description of this approach can be found elsewhere [62].

Dependency-based feature extraction: As the previous approach only extracts word collocations as the keywords representing a feature, this approach extracts software features from descriptions using dependency lexical parsing. Dependency parsers grammatically analyse the structure of a sentence annotating the word's role grammatically (beyond its part-of-speech). This is led by the intuition that apps features are described using common linguistic patterns. We perform the parsing using the Stanford dependency parser [255, 256].

To extract feature phrases using dependency parsing, first the app description is tokenised at the sentence level using common sentence termination punctuation in the English language, in addition to new lines. Then each sentence's dependency is parsed. Based on extensive analysis of dependency-parsed mobile app description, we devise a set of clauses that are likely to refer to the application's behaviour and offered features. These clauses are: [verb, direct object], [verb, indirect object], [noun, reduced non-finite verbal modifier], [verb, noun modifier] and [verb, nominal subject]. We also preserve part-of-speech tags and ensure that the first part of the pair is either a verb, noun or a phrase of each; and that the second part is either noun, adjective or adverb.

These pairs of words/phrases are then extracted and treated as 'featurelets' used in the clustering explained previously. Table 5.2 shows examples of featurelets extracted from the dataset described in 5.2.3 using both collocation- and dependency-based parsing.

Table 5.2: Examples of extracted featurelets representing each feature and the number of times these features appear in the dataset (number of apps that boast the feature) for collocation-based and dependency-based parsing.

Dataset	Featurelet terms	Occurrences
Collocation-Based	[push, notification]	54
	[news, search, ability]	19
	[font, change, size]	6
	[translate, dictionary, sentence]	2
	[photo, background, change]	1
Dependency-Based	[share, friends]	442
	[send, email]	192
	[choose, theme]	40
	[wake, alarm clock]	22
	[scan, business cards]	1

5.2.2 Research Questions

In order to assess the impact of using different similarity measurement techniques on mobile apps clustering we investigate five research questions.

The first question (RQ0) is a sanity check we carry out to assess the degree of difference in clustering solutions for different choices of similarity:

RQ0. Sanity Check: How much do clusterings based on different similarity measurement techniques differ from one another?

This is a sanity check in the sense that a low difference level would suggest there is little point in further study. To measure the similarity among different clustering solutions we use the Jaccard index as explained in Section 5.2.4.

The following three questions (RQs 1–3) are based on previous work [67] and aim to assess the effectiveness of the techniques we compared herein with respect to three main aspects as follows:

RQ1. How well do the similarity measurement techniques represent the commercially assigned app store categories?

In this research question, we investigate the degree to which each of the similarity measurement techniques deem apps in same app store category more similar than apps in different categories (i.e. which technique more closely represents the app similarities in current app store categorisation). This research does not regard app store categorisation as a ground truth of app similarity. We envisage a clustering algorithm that relies on capability-based feature extraction to result in a better and more fine-granularity segmentation of the dataset.

RQ2. How does the clustering granularity levels affect the clustering quality and what is the granularity level that results in the best clustering quality for each technique?

A granularity level of a clustering technique is the selected number of clusters, k. The clustering quality can be measured using the silhouette width score of the clustering solution. Different choices of k directly affect the clustering quality score. In answering this question, we verify that cluster quality does indeed change depending on the choice of k. We report the maximum scored silhouette width for each technique and at what granularity was achieved. Furthermore, in previous work [67]

we found that using hierarchical agglomerative clustering results in a range of viable granularities where the cluster quality (measured using silhouette width score) plateaus. This means that users of a clustering technique are able to select the granularity of the clustering based on desired result and the degree of distinction the clustering makes among the apps without large sacrifice in the cluster quality.

RQ3. What is the clustering solution quality for each technique based on human judgement? The silhouette width score is a method of internally measuring the cluster quality depending on each data point's assignment and overall cohesion. However, a more conclusive method of measuring the clustering solution's external quality is by relying on human judgement. Therefore, we analyse the resulting clustering hierarchies produced by the different techniques in a more qualitative manner. To this end we sample pairs of app from the dataset and proceed to build a gold set based on human-judgement. Due to the abundance of possible clustering solutions based on the selected granularity level, we draw a random sample of 300 apps comprising 150 app pairs from 5 different levels of the clustering dendrogram. The annotation is then carried out by eight human annotators to rate the similarity between apps in each pair based on their descriptions on a 5-item Likert scale. This enables us to investigate the correlation between the similarity score and the finest granularity at which these two apps remain in the same cluster for each technique.

The last question we answer aims at investigating the cost of using these approaches in practice:

RQ4. How efficient is each of the similarity measurement techniques? In order to be usable, the set up cost and subsequent instantiation cost of a clustering approach should be within reasonable bounds, to allow developers to use the approach to help understand the claimed-feature competitive space into which they deploy their apps. To this end, we compare the techniques with regards to their execution time.

5.2.3 Dataset

The dataset used in our empirical study has been built by sampling uniformly at random from a snapshot of the Google Play app store. This snapshot was collected by crawling the entirety of the app store in October, 2014 by Viennot et al. [105] amassing around 1.4 million Android apps[1]. To ensure the fairness of our comparison, we excluded apps that have very short descriptions (less than 100 characters) which may affect the performance of some techniques but not others. We also filtered out apps with non-English descriptions. Furthermore, we did not include the games category in our study as it has grown to the point of having a dedicate section in the store. Furthermore, recent research indicates that mobile apps might be presented/received differently from mobile games [257]. The final sample consists of 12,664 Android apps belonging to 24 categories[2].

5.2.4 Evaluation Criteria

This section explains the metrics and statistical analysis we use to evaluate the results of our study and answer our five research questions: Silhouette width score, Spearman rank correlation, and the

[1] A JSON file containing the metadata and URLs for all apps in the snapshot can be found here: `https://archive.org/download/playdrone-snapshots/2014-10-31.json`. The documentation of how to parse the JSON file is here: `https://archive.org/details/android_apps&tab=about`

[2] Our random sample can be found here: `https://afnan-s.github.io/appa/comparison.html`. This file contains Android app IDs as well as the index of the app in the aforementioned Playdrone JSON file.

intra-class correlation coefficient.

Perplexity [250] is the log likelihood of a model generating a held-out set. It is used in our study to find the most appropriate number of topics over our dataset when using latent dirichlet allocation (LDA). The concept of perplexity was first proposed to measure the complexity of speech recognition tasks. It is directly related to the entropy of a language model. Perplexity has since been used to evaluate the performance of a probability model by assigning it a score that shows how well the model predicts the probability distribution of a sample. The lower the perplexity, the better the model.

Jaccard Index, used in RQ0, is a measurement of agreement between two partitions by counting the number of element pairs that are classified together by the two partitions, divided by the number of pairs that are classified differently. Jaccard index value lies between 0 and 1 with 0 denoting complete dissimilarity and 1 is assigned when the two clustering solutions are identical.

The **silhouette width score**[216], used in RQ1 and RQ2, measures the similarity of data points in the same cluster to one another, and their dissimilarity with data points assigned to other clusters. Its value ranges from 1 (perfectly assigned) to -1 (completely mis-assigned). This score is assigned to each data point, hence an overall silhouette score of a clustering is typically the average of the silhouette scores of all data points.

Intraclass Correlation Coefficient (ICC) [228], used in RQ3, is a method of calculating inter-rater agreement. It is used in this study since there are more than two raters (thus, Cohen's Kappa and Weighted Kappa are unsuitable [229]). Whereas Fleiss' Kappa [230] is suitable for the case of more than two raters, it assumes the rating system to be nominal or categorical. In this study, we use a Likert scale represented by 5 Likert items, we need a rater-agreement system that deems two ratings of 4 and 5 as more consistent than two ratings of 3 and 5. To calculate ICC, we use a two-way model indicating that both rated statements and raters are representative of a larger sample. The ICC lies between 0 (extreme inconsistency among raters) and 1 (complete consensus).

Spearman rank correlation [226], used in RQ3, is a measurement of how well two paired series of values correlate with one another (i.e. change in one, leads to a change in another). Spearman correlation is based on the rank, therefore does not require a fixed rate of increase/decrease among correlating observations making it suitable for ordinal scale metric data (as used in this study). The value of the Spearman rank correlation coefficient (typically denoted ρ) lies between -1 and 1 and gives an indication of degree of correlation (1 means a strong positive correlation, -1 indicates a strong inverse one and 0 means a complete lack of correlation). Spearman rank correlation also produces a $p-value$ showing the probability of the given ρ when in fact there is not correlation (i.e. $\rho = 0$), hence, it is a proxy of the certainty of observing an accurate ρ.

5.3 Empirical Study Results

RQ0. Sanity Check: Degree of agreement among the different similarity measures.
To calculate the Jaccard index among all the different partitions that are based on the studied techniques, we need to select one k (number of clusters) for all techniques. We have opted to measure Jaccard index at $k = 24$ as the number of Google Play categories in our corpus is 24.

Table 5.3 shows the Jaccard index results: We can observe that the techniques produce clusterings that are different from one another. However, partitions produced by the collocations and

Table 5.3: RQ0. Jaccard similarity index between each of the clustering solutions of the studied techniques for 24 clusters (the number of app store categories).

	LDA	VSM	VSM+LSA	FSM+ Collocation	FSM+ Dependency
LDA	1	0.246	0.080	0.351	0.331
VSM	-	1	0.095	0.303	0.29
VSM+LSA	-	-	1	0.076	0.076
FSM+Collocations	-	-	-	1	0.617
FSM+Dependency	-	-	-	-	1

dependency parsing variations of the Feature Space Model are close. This is to be expected as both rely on keyword-based extraction from the corpus.

RQ1. How well do the similarity measurement techniques represent the commercially given app categories?

This shows whether the current app store categorisation represent a good clustering solution for each of the text representation and feature extraction techniques.

To answer this question, we build a distance matrix for the dataset that is calculated using each of the text representation techniques explained in section 5.2.1. Then, assigning a cluster to each data point that represents the app store category from which this app was mined. This forms a clustering solution of the dataset, albeit enforced by the state of the app store categorisation. Finally, we use the silhouette width score to measure the quality of this clustering solution. This measurement represents how well are app are grouped together based on the technique used to represent the app. This also helps ensuring that any further clustering stages that are app store category independent does indeed improve on the current categorisation of the app store.

Our results reveal that existing app store categories (24 categories) perform badly as a segmentation of mobile apps based on those representations (see Table 5.4). The results also reveal that performing hierarchical clustering yields an improvement of the silhouette scores of the partitions (cut-off at $k = 24$), albeit a slight one. Subsequent results for RQ2 will show that these cluster quality scores can be improved by increasing the number of clusters, thus supporting the observation that existing categorisation is of too coarse granularity to yield higher cluster quality scores based on the features studied.

RQ2. What is the clustering performance at different granularity levels for each technique?

Hierarchical clustering affords the user a range of possible k values. This can be selected depending on the desired granularity level and purpose of the clustering (broad sense of similarity vs. almost identical cluster members). However, the cluster memberships quality can suffer if an inadequate k is selected. To gauge the tendency of cluster quality compared to k we plot the silhouette score at each granularity level. Figure 5.1 shows the behaviour of the silhouette score as the granularity increases for each of the techniques.

The maximum silhouette scored by each technique can be an indication of the technique's performance compared to others. Table 5.5 lists the granularity level at which the silhouette score reaches its maximum value for each of the techniques. We observe that extracting features using topic modelling scores the largest silhouette with (0.48) whereas collocation-based feature space

Table 5.4: RQ1. Summary of silhouette width scores for each of the techniques when considering app store category as a cluster assignment (existing categorisation) and when selecting $k = 24$ (same number of categories in the app store).

		Min.	Max.	Mean	Median
LDA	Existing categorisation	-0.54	0.59	0.003	-0.01
	Clustering solution	-0.64	0.99	0.02	-0.01
VSM	Existing categorisation	-0.26	0.26	0.01	-0.001
	Clustering solution	-0.39	0.86	0.01	-0.02
VSM+LSA	Existing categorisation	-0.64	0.64	0.002	-0.02
	Clustering solution	-0.68	0.82	0.11	0.06
FSM+Col	Existing categorisation	-0.05	0.10	-0.0003	-0.002
	Clustering solution	-0.35	1	0.01	-0.01
FSM+Dep	Existing categorisation	-0.06	0.09	-0.0003	-0.003
	Clustering solution	-0.42	1	-0.004	-0.03

Table 5.5: RQ2.1. For each technique, the maximum viable granularity and the generated maximum silhouette score.

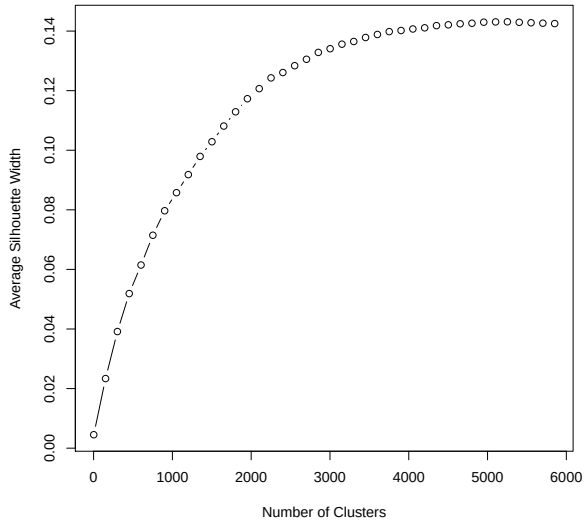
	Max Sil	Granularity
LDA	0.48	5702
VSM	0.14	5252
VSM+LSA	0.24	112
FSM+Collocations	0.12	6322
FSM+Dependency	0.13	6302

model scores the least. We also observe that using baseline VSM with LSA reduction can help achieve higher silhouette at an early stage (coarse granularity) of the dendrogram.

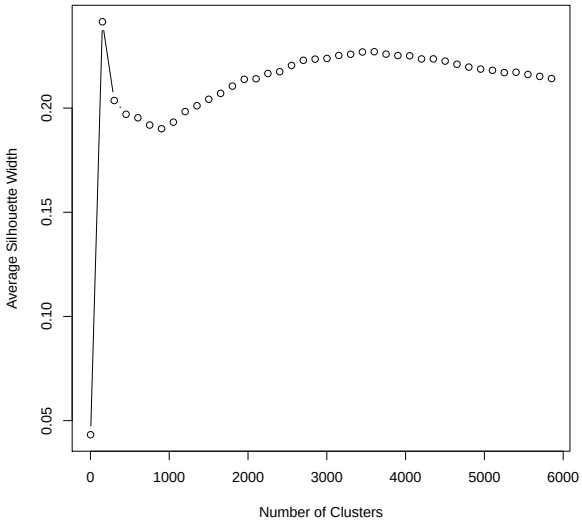
RQ3. How does the clustering solutions compare to the ground truth?

To answer this question, we employ human judgement in evaluating how well each of the techniques cluster apps based on their feature (functionality) similarity, application domain similarity, and underlying libraries/APIs similarity. Since the hierarchical clustering solutions provide a range of usable cut-off points (k clusters), we test the clustering at 5 different levels of the solution starting from $k = 2$ until the maximum viable k for each technique (i.e. maximum k before silhouette score starts dropping). The five sampling levels lie at 2, 25%, 50%, 75% and 100% of the maximum viable k for each technique where apps sampled from level 1 are apps that were separated immediately in the hierarchical dendrogram thus representing apps that are deemed completely different by the clustering technique, apps sampled at level 25% represent apps that survived together in the dendrogram but were separated at level 25% thus deemed somewhat different, and so on. From each of the levels, and for each technique, we randomly sample 6 app pairs (12 apps) representing the clustering technique's performance at that level, thus generating 30 app pairs for each technique representing all similarity levels. This results in a sample of 150 app pairs (300 apps in total).

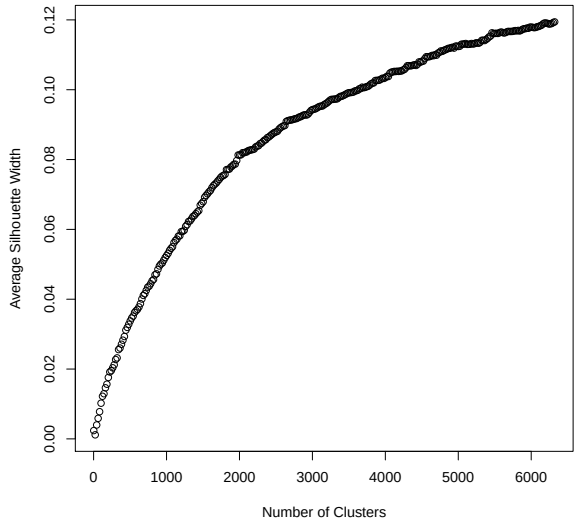
We asked eight annotators who are computer science students (2 undergraduate, 6 graduate students) with programming/coding experience to rate the similarity of each of the app pairs (after randomisation) on 5 similarity levels (5-item Likert scale) according to three criteria: feature similar-



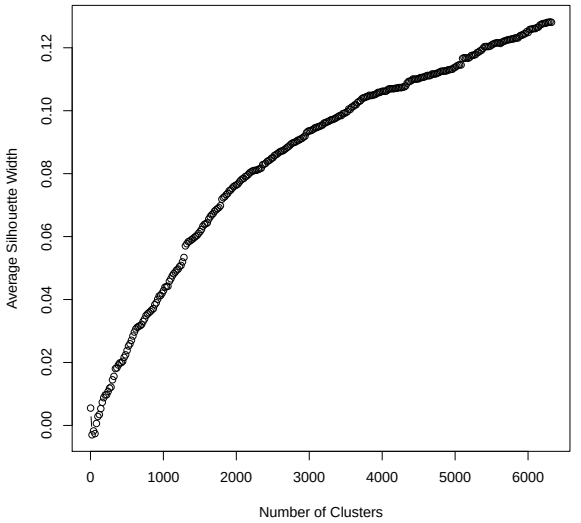
(a) VSM



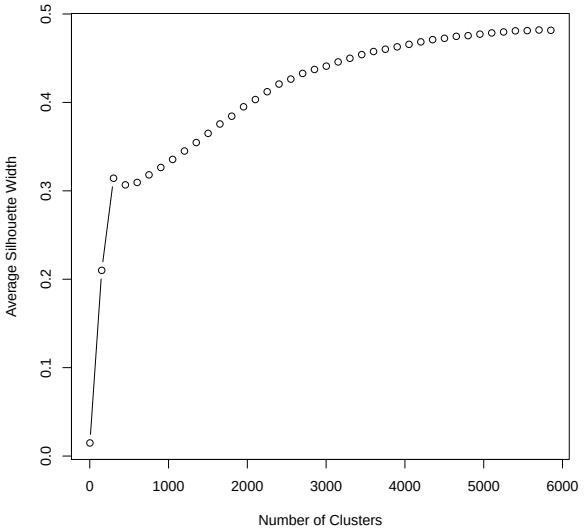
(b) VSM+LSA



(c) FSM+Collocations



(d) FSM+Dependency



(e) LDA

Figure 5.1: RQ2.1. Average silhouette score as the granularity (k) increases for each technique.

Table 5.6: RQ3. Inter-rater agreement of the obtained goldset (8 raters) on the three rating criteria using intra-class correlation.

	Feature Similarity	Application Domain Similarity	API Similarity
LDA	0.7	0.8	0.8
VSM	0.5	0.6	0.6
VSM+LSA	0.6	0.6	0.5
FSM+Collocations	0.6	0.6	0.7
FSM+Dependency	0.6	0.7	0.6
Overall	0.6	0.6	0.6

Table 5.7: RQ3. Spearman Rank correlation scores (p-value in brackets) between hierarchical sampling level (technique-assigned similarity) and human-assigned similarity scores. Scores are deemed statistically significant if p-value < 0.01.

	Feature Similarity	Application Domain Similarity	API Similarity
LDA	0.60 (0.0004)	0.60 (0.0003)	0.60 (0.0012)
VSM	0.40 (0.03)	0.60 (0.0008)	0.40 (0.03)
VSM+LSA	0.04 (0.85)	0.07 (0.7)	0.10 (0.6)
FSM+Collocations	0.60 (0.0016)	0.60 (0.0006)	0.60 (0.0011)
FSM+Dependency	0.60 (0.0012)	0.70 ($6.7e^{-05}$)	0.60 (0.0003)

ity (functionality/capability), application domain similarity (category) and underlying libraries (APIs) similarity based on the descriptions of the two apps in the pair. Table 5.6 shows the inter-rater agreement calculated using intra-class correlation confirming that indeed a correlation emerges.

In order to measure the performance of the techniques we analysed, we check whether exists a correlation between the mean of human-assigned similarity scores on the Likert scale and the level at which the app pair survives in the hierarchical clustering dendrogram before being separated into different clusters. Table 5.7 reports the Spearman rank correlation scores. The results reveal that there indeed exists a positive correlation between the goldset's similarity score (mean of the scores assigned by the annotators) and the level at which the clustering algorithm decides to separate the pair in the case of LDA and FSM-based techniques. The correlation is especially prominent in the dependency parsing based feature extractor when detecting similarity in the application domain.On the other hand, VSM-based techniques failed to generate strong correlations, or indeed statistically significant ones.

RQ4. How efficient is each of the similarity measurement techniques?

In table 5.8 we report the cost in terms of execution time of each of the techniques we compared. The times are broken down to four main phases required for each of the techniques. The first one is data preprocessing which, for LDA, included the amount of time it takes to select the number of topics that generates the lowest perplexity. For the feature-space model techniques data preprocessing also includes the feature extraction and clustering stages. The second phase is building the document-term matrix and any required subsequent reductions. This is followed by calculating the distance matrix (distance between each pair in the dataset) and, finally, the hierarchical clustering phase.

Table 5.8: RQ4. Efficiency of each of the studied techniques. The technique's run-time was measured on a standard laptop with an Intel Core i7 3.1 GHz and 16 GB RAM; d=days, h=hours, m= minutes, s=seconds.

	Data Preprocessing	DTM Reduction	Distance Matrix	Clustering
LDA	5.4 d	3.0 h	21.0 s	6.0 s
VSM	-	6.0 s	9.0 s	8.0 s
VSM+LSA	7.0 h	6.0 s	1.3 h	6.0 s
FSM+Collocations	1.5 m	8.7 d[1]	12.0 s	8.0 s
FSM+Dependency	5.0 d	17.9 d[1]	12.0 s	9.0 s

[1] The running time for FSM+Collocations and FSM+Dependency has been normalised to simulate sequential time in order to enable comparison with other phases/techniques, however the technique was actually run in parallel taking 1.17 days for FSM+Collocations (14 threads and an average of 15 hours per thread) and 5 days for FSM-Dependency (4 threads and an average of 4.5 days per thread).

As expected, feature-space model techniques require a large amount of time to conduct lexical feature extraction with the collocation-based one being considerably faster than the dependency-based algorithm. Topic modelling based technique is mostly hampered by the amount of time it takes to select the proper number of topics (by measuring the perplexity).

However, for all these techniques, except for VSM, the majority of the cost lies in early steps. Whereas LDA, and both variations of FSM require an upfront cost, folding-in (adding new unobserved data), is of significantly less cost for calculating the distance matrix and conducting the clustering. Using VSM with LSA promises larger folding-in cost as the distance matrix calculation step poses a bottleneck.

5.4 Discussion

The results of our empirical study show that mobile app similarity can be calculated by analysing their descriptions, with an acceptable degree of accuracy.

On one hand, using the feature extraction techniques discussed in this study to measure the current app store categorisation quality as a clustering solution (RQ1) resulted in very low silhouette measures. This, we believe, is a good indication that apps in different categories share more features than with their category members (i.e. more features tend to be ubiquitous than category-specific features). This supports the conjecture that app store categorisation may not provide an ideal feature-specific segmentation of the app store as a mobile software repository. Hence motivating the need to investigate better techniques to offer different view of the apps offered, especially when used by developers for re-use.

However, our study also showed that the task of detecting app store similarity cannot be carried absolutely conclusively by human raters (overall inter-rater agreement of 0.6) although the similarity criteria were broken down and clearly defined. This indicates that the task is somewhat difficult and has a certain degree of subjectivity, albeit low, thus the automation of such tasks should be handled with care especially with regards to the expectation of possible achievable accuracy.

In measuring the quality of a clustering solution, this study shows that internal cluster quality measures (i.e. silhouette score) are not a sufficient view of the resulting clustering and does not completely eliminate the need for a human rated ground truth. This is evident in the case of LDA-

based feature extraction which enabled hierarchical clustering to produce more cohesive clusters than other techniques (silhouette = 0.48), however, been shown to perform similarly to FSM-based techniques. In fact, dependency-parsing based clustering performs better that any other technique in terms of finding apps in similar application domain as measured by the rank correlation between human similarity rating and cluster agreement level in the dendrogram.

Finally, we deduce that baseline techniques (VSM), though the fastest and cheapest to carry out, do not seem to produce statistically significant results with regards to their similarity quality using human judgement. One interesting observation we find is that when using dimensionality reduction, namely latent semantic analysis, enabled the clustering to quickly converge to a high cluster quality earlier than other techniques (see Figure 5.1-b). This may indicate the usefulness of LSA if a clustering of coarser granularity is required.

5.5 Threats to Validity

Internal Validity: We carefully applied the statistical tests verifying all the required assumptions. As in every clustering solution, finding the optimal number of clusters remains ambiguous. To cluster the mined features, we use a popular method (Can's Metric) that has been used in previous work with good results [67, 21]. Another threat to internal validity could be due to the apps composing our datasets (a.k.a. App Sampling Problem [233]) as collecting all existing apps is not currently allowed for existing app stores, including the Google Play one. Threats may also arise due to the procedure we used to build the gold set. However, the number of human raters is consistent with that in previous similar studies (e.g., [67, 112]) and their agreement. Moreover, when selecting random app pairs, we prevent a bias towards a majority of a certain degree of similarity by using purposive sampling [258], thus ensuring that the sample contains apps with varying degrees of similarity, as done in previous study [67].

Construct Validity: Previous studies have shown that it is possible to extract features from product descriptions available on-line [24, 21, 23, 235, 236]. However, these features are extracted from claims reported by app store developers and we cannot be sure that these necessarily correspond to features actually implemented in the code itself, since developers do not always deliver on their claims [111]. We mitigate this threat by extracting the features from a large and varied collection of app descriptions, and clarifying that it is clearly a constraint of most NLP-based approaches [21]. Nevertheless, previous work has shown that developers' technical claims about their apps are inherently interesting and *however* we view them, they have interesting properties in real world app stores (see e.g., [62, 66, 64]).

External Validity: The features extraction methods analysed in this study can be applied to any kind of software and software repository, however our empirical results are specific to mobile application and to the store considered. More work would be needed to investigate whether the findings generalise to other time periods, app stores, and software type.

5.6 Conclusions

This study empirically analyses how different text representation techniques perform when used for mobile app clustering. As Arnaoudova et al. [240] estimate that NLP and text retrieval can address more than 20 software engineering issues, such techniques are particularly useful in the

case of mobile applications as the app store provides a rich repository of software in which textual description is readily available while source code might not necessarily be.

To this end we have used a textual description clustering baseline, which relies on the Vector Space Model representation of the data using Term Frequency-Inverse Document Frequency weighting, along with latent semantic indexing. We compare this baseline with three more advanced textual-based feature extraction techniques that have been used in software engineering research: topic modelling [241], collocation-based feature extraction [63] (Chapter 4) and an enhancement on the latter we propose in this study relying on dependency parsing of sentences to extract software features. We have performed this comparison on a randomly sampled dataset of 12,664 mobile app descriptions extracted from the Google Play (Android) app store.

The results of this study revealed that quantitative cluster quality (measured in silhouette score) tends to favour clustering solution produced by using topic modelling (silhouette = 0.48). However, qualitative evaluation (human judgement) shows a good clustering quality for all techniques, barring the baseline.

This study also confirms that current app store categorisation performs badly as a segmentation of the dataset based on all of these representation techniques, thus motivating the need for a better segmentation of applications in mobile app stores.

Chapter 6

Feature Migration in the Samsung App Store

6.1 Introduction

Features play a central role in the App Store ecosystem. A feature poses as unit of interaction between users and developers. It is observed that about 40% of user feedback is provided on the features level: complaining about certain features, liking others, and requesting the addition of some [116]. Therefore, previous research endeavoured to employ user feedback to enhance requirement elicitation and maintenance tasks [77][82].

However, for newly deployed apps, user reviews might be scarce. Requirement elicitation from sources other than user reviews can be very useful. This might include investigating features provided in other competing apps, features that are trending and have high adoption rate, or features popular in other categories that might have transferable value.

The study presented in Chapter 3 reported that 56% of surveyed developers browse similar apps for features to include in their own. This gives the intuition that features can exhibit certain movement among the app store.

Chpaters 4 and 5 showed that mobile applications' features can be adequately extracted from mobile app description on the app store. This relied on collocation-based extraction of keywords from natural language descriptions.

Therefore, this chapter investigates applying this feature extraction technique to observe whether features exhibit any movement tendencies beyond the category boundaries of the app stores; furthermore, it observes characteristics of these features with regards to the average price, rating and rank of apps that adopt them. This is carried out by empirically analysing data from the Samsung mobile app store at two different time points to observe feature spreading behaviour over categories during that time span.

The goal of this work is to investigate whether feature adoption from other categories exists, and if so, whether features exhibiting different spread behaviour, show certain trends in terms of their price, rating and popularity in order to guide developers decisions when adopting features from other categories. The work in this chapter has augmented an initial research carried out over only the BlackBerry dataset done by Sarro et al. [259]. The results obtained with both datasets have been published in the the 23rd IEEE International Requirements Engineering Conference [66]. My main contribution to this study paper was the empirical analysis of the Samsung app store data set, which is reported in this chapter. The complete study using the two datasets is presented in the full paper [66].

The organization of the next sections will be as follows: A theoretic characterization of app behaviour in the app store is proposed in section 6.2. Afterwards, the design is presented in section 6.3. Then feature behaviour is analysed over a period of time for the Samsung app store to inspect if different behaviour carries different tendencies in the three main metrics: price, rating and rank of downloads with results presented in section 6.4. Threats of validity are presented in section 6.5. Finally, the chapter concludes in section 6.6.

6.2 Overview of Feature Migratory Behaviour

This work pays special attention to the movement of features, guided by an intuition that developers elicit requirements from the app store as found in Chapter 3. It proposes a complete taxonomy of feature migration behaviour among categories in the app store. Each feature belongs to at least one category if there exists an app belonging to that category that exhibits this feature. These same categories are then investigated at a subsequent point in time. The existence of old features and new features then enable us to classify feature into one of the classes of migratory behaviours. These classes are:

- **Weak Migration (WM)**: Feature appears in at least one new category.
- **Strong Migration (SM)**: Feature appears in at least one new category, while remaining in original category(ies).
- **Weak Exodus (WE)**: Feature appears in at least one new category, while leaving at least one original category.
- **Strong Exodus (SE)**: Feature appears in at least one new category, while leaving all original category(ies).
- **Intransitive (I)**: Feature remains in all original category(ies), and do not appear in any new category.
- **Weak Extinction(WX)**: Feature disappears from at least one original category and do not appear and any new category.
- **Strong Extinction (SX)**: Feature disappears from all original category(ies) and do not appear and any new category.
- **Non Migratory features (NM)**: is the set of Intransitive and weakly extinct features (including strongly extinct features). This set was created to serve the purpose of grouping non migratory features in the same way that encompasses all migratory features.
- **Birth**: Feature only appears in last time point. These features are not observed throughout this study.

Due to the previous definitions, it must be noted that these sets are not mutually exclusive but carry the following relationships:

$$SM \subseteq WM$$

$$SE \subseteq WE \subseteq WM$$

$$SX \subseteq WE \subseteq NM$$

$$I \subseteq NM$$

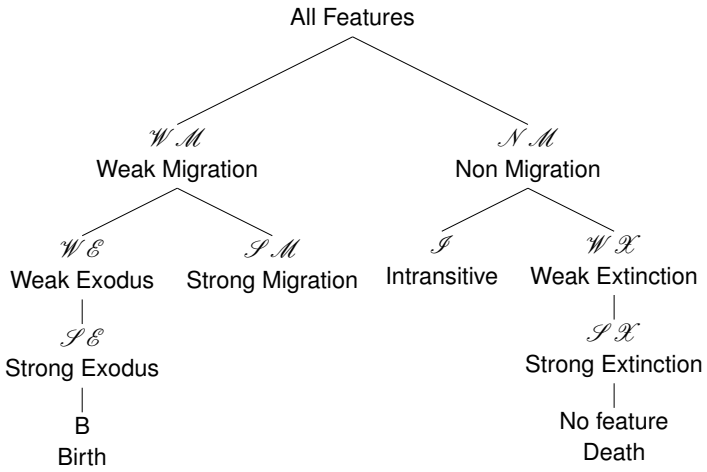


Figure 6.1: The Feature Migration Subsumption Hierarchy.

Table 6.1: Summary Data for the Samsung Apps Studied Between Two Time Intervals.

	2011-week05										2011-week36									
Category	Apps	Features	Mean Price	Median Price	Mean Rating	Median Rating	Mean Rank of Downloads	Median Rank of Downloads	Min Rank of Downloads	Max Rank of Downloads	Apps	Features	Mean Price	Median Price	Mean Rating	Median Rating	Mean Rank of Downloads	Median Rank of Downloads	Min Rank of Downloads	Max Rank of Downloads
E-Book/Education	34	3	12.51	3.00	1.24	0.00	3924	4497	320	5204	72	67	6.49	1.00	1.03	0.00	8502	9116	543	12353
Entertainment	186	54	2.23	1.25	2.71	3.25	3200	3435	0	5185	407	95	1.83	1.00	1.48	0.00	7677	7951	89	12313
Games	715	98	2.08	1.50	2.78	3.50	2719	2677	5	5197	1082	75	1.70	1.25	2.01	2.00	8090	9696	3	12355
Handmark	25	0	5.62	3.00	2.12	0.00	4083	4717	1659	5162	26	11	5.63	4.00	0.38	0.00	11970	12000	9238	12319
Health/Life	189	58	1.32	1.00	3.02	4.00	3210	3093	68	5198	254	52	1.28	1.00	1.88	1.00	9529	11139	23	12368
Music/Video	35	19	1.48	1.25	1.64	1.00	2165	1988	528	4300	74	35	1.39	1.25	1.92	2.00	7121	8109	138	11785
Navigation	57	59	5.01	1.25	2.88	3.50	2614	2339	45	5012	130	80	10.69	3.00	1.68	1.00	8121	9045	139	12044
News/Magazine	9	13	1.67	1.00	2.50	3.00	3149	3314	1633	4250	12	7	1.50	1.00	2.25	3.00	9121	9564	1000	12093
Productivity	76	114	4.50	1.50	2.83	3.50	3046	2924	544	5189	147	87	2.91	1.25	2.16	2.50	7891	8847	48	12357
Reference	259	59	12.65	12.00	1.45	0.00	4313	4686	141	5207	352	53	10.72	6.00	0.73	0.00	9812	10679	558	12371
Social	15	14	4.03	1.25	2.83	3.00	2302	2322	481	4784	34	19	2.46	1.25	1.90	2.00	6965	9476	97	12255
Theme	533	0	1.16	1.25	1.85	0.00	3256	3469	47	5110	4041	15	1.07	1.00	1.25	0.00	6995	7031	0	11326
Utilities	215	96	2.67	1.00	2.82	3.50	2924	3058	33	5187	468	93	1.88	1.00	1.76	0.00	8138	8519	4	12328
Mean	168	42	4.07	2.16	2.36	2.17	3147	3271	423	4999	507	53	3.81	1.85	1.57	1.04	8456	9321	914	12174
Median	67	37	2.45	1.25	2.71	3.00	3149	3093	141	5185	139	53	1.88	1.25	1.76	1.00	8121	9116	97	12319

$$NM = WE + I$$

These relationships resemble a subsumption hierarchy as better depicted in figure 6.1.

6.3 Empirical Study Design

This section provides an overview of the investigated dataset, research questions and their motivation.

6.3.1 Dataset

To carry out the experiment, we mined non-free mobile application descriptions and complete profiles of their metadata from the Samsung app store. The data was collected during 2011 at two time points (week 5 and week 36) by crawling and parsing the html files of the applications' pages in the app store. The choice of time points is arbitrary and was specifically selected to imitate BlackBerry App Store data used in [259]. Table 6.1 depicts a basic summary of the apps in the dataset for the first time point and the second time point. We excluded from the Samsung Apps study the Brand category since it contained only eight free apps and the Handmark category that contained different kind of apps (e.g., games, advertisement) developed by the same software company (i.e., Handmark), so it does not represent a category of apps offering similar functionalities.

6.3.2 Research Questions

The research questions investigated in this study are as follows:

RQ0. Feature Evolution: Is there any change in features between the start and end time points? We investigate the results of this research question as a baseline sanity check: whether features in the subsequent snapshot differ than in the original one. The assertion that features have moved between the two points in time, facilitates subsequent questions. We answer RQ0 simply by measuring the number of features in each category at the start and end of the time period.

RQ1. Feature Migration: How do the features distribute over the different migratory behaviours in the subsumption hierarchy? This question investigates whether any of the behavioural classes defines in section 6.2 exist. If indeed features exhibit any of them, than how do they distribute over the taxonomy.

RQ2. Are there any significant differences in the price, rating, popularity of features that exhibit different migratory behaviours? classification of features over the different migratory behaviours would prove essential for requirement elicitation and app design if indeed we find a relationship between certain migratory behaviours and app success metrics.

RQ3. Are there differences in the correlations between price, rating, and popularity within each form of migratory behaviour? This question investigates whether there exists a correlation between each of the rating, price and rank for each of the migratory behaviours in the dataset. The study inspects whether each of the migratory behaviours exhibits different correlation trends for each pair of the attributes investigated (price, rating and rank).

6.3.3 Methodology

In order to mine pertinent data to this study, we have employed the framework proposed by Harman et al.[63] explained in Chapter 4 and [66].

The framework parses the raw data extracted from the Samsung Apps market and identifies lists of features recognised using specific common patterns; such lists are extracted for each app in addition to the app's price, rating and rank of downloads.

The next phase extracts feature keywords from the application's extracted feature lists. In order to extract features, a collocation-based keyword extraction has been employed (presented in detail in Chapters 4 and 5). This methodology has been first proposed by Harman et al. [63] and has been shown to adequately extract app features from natural language descriptions found in app stores. The algorithm is a four-step NLP feature extractor that uses the Python Natural Language Tool Kit (NLTK). For each raw features list, the algorithm first removes non-English and stop words (e.g. the, and, to, etc.). The words are then stemmed (returned to their original lemma form) using the WordNetLemmatizer. Afterwards, the algorithm extracts commonly co-located words for each raw feature (using NLTK's N-gramCollocationFinder). Theses word collocations are then clustered using a simple greedy clustering algorithm to group similar collocating word to represent the same feature.

Throughout this process, the framework maintains the link among the extracted features and the apps the exhibit these features. Afterwards, each feature is assigned a price, rating and rank of downloads that is the mean (average) price, rating and rank of each of the apps that exhibit this feature.

This process has been carried out over the two snapshots of the dataset. The existence and

frequency of each of the features from the first snapshot in the second one has been investigated and each of these features has been assigned one of the feature migratory behaviours discussed in Section 6.2. We extracted 623 and 689 unique features from the first and second Samsung snapshots, respectively.

In investigating whether features in the subsequent snapshot differs from the first one, we answer **RQ0**. In order to answer **RQ1**, we count the number of features that exhibit each of the migratory behaviours to observe how the features distribute over the taxonomy in Section 6.2. The third research question (**RQ2**) inspects whether any of the apps exhibiting features belonging to migratory behaviours differ in price, rating and rank. To uncover this, We use a 2-tailed, unpaired Wilcoxon test [260] to compare the median values of the price, rating, and popularity of each of the migratory behaviours which are the median of those of the apps that exhibit these features. If indeed a difference was detected between price, rating and/or popularity among the different behaviour classes, we also use the Vargha-Delaney $\hat{A}_{12}$ metric for effect size [261] to detect the degree of the effect size between the groups. Finally, in answering question **RQ3**, the study uses both Pearson [262] and Spearman statistical correlation tests [226].

6.4 Results Analysis and Discussion

6.4.1 RQ0. Feature Evolution

Table 6.2 reports the number of features contained in the Samsung app store at two different periods of time (i.e., weeks 5 and 36 of the year 2011, denoted T_0 and T_1 respectively) and the Jaccard Similarity (JS) of each category over the time (i.e., we measure how the features contained in the same category change over the time). Through investigating this data, we find, as expected, that most categories exhibit certain changes in terms of the features they contain. The categories Education, Entertainment, Music/Video, Navigation, Social and Theme have seen growth in the number of features their apps boast; whereas the remaining categories lost some features.

6.4.2 RQ1. Feature Migration

Figure 6.2 shows the subsumption hierarchy of the migratory behaviours with the number of features found in each category for the Samsung store. We find that of the 623 found Samsung apps features, only 3 exhibit weak migration. When investigating the migrated features, we find that none of these features moved because of the original app containing them has re-categorised (i.e. the features appeared in a different app in a different category). These features have moved to other similar categories. There are also a significant number of features (62%) that die out (i.e. observed in first snapshot, but not in the second). One the other hand, about one third (38%) are intransitive and remain stagnant.

6.4.3 RQ2. Differences in Migratory Behaviours

Figure 6.3 and 6.4 show the boxplots of the Mean and Median Price, Rating and Rank of Downloads values of the features that have the same migratory behaviours, respectively. The first four boxplots of each figure are non-migratory, while the second four are migratory. A higher Rank of Downloads indicates lower popularity. We observe that the migratory features are cheaper and lower rated (based on the mean price/rating of apps that exhibit them) than the non-migratory ones, however, differently from BlackBerry, they are more popular. These results reveal some interesting differences,

Table 6.2: RQ0. Number of features contained in a given category in the Samsung app store; and Jaccard Similarity (JS) of the initial and final categories over the time period.

Category	T_0	T_1	JS
Education/Ebook	3	67	0.03
Entertainment	54	95	0.25
Games	98	75	0.27
Health/ Life	58	52	0.37
Music/Video	19	35	0.32
Navigation	59	80	0.17
News/Magazine	13	7	0.17
Productivity	114	87	0.11
Reference	59	53	0.13
Social	14	19	0.62
Theme	0	15	0
Utilities	96	93	0.28
Total	587	689	-

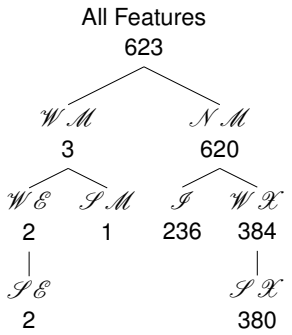


Figure 6.2: RQ1. Observed Number of Features for each Migratory Behaviour for the Samsung App Store.

particularly with regard to the price of intransitive features relative to that of others. This analysis reveals that intransitive features appear to have a lower monetary value than those which die out. When investigating the degree of difference, we found a statistically significant difference in the median price values between $\mathscr{I}$ and $\mathscr{W}\mathscr{X}$: the features that die out are higher priced than features that remain intransitive ($p = 0.048, \hat{A}_{12} = 0.55$). This shows that intransitive features in the Samsung app store mirror the behaviour of their counterparts in the BlackBerry store. This difference carries value that may affect decision making done by app developers in the requirement elicitation, design and maintenance processes.

Tables 6.3, 6.4, 6.5 and Tables 6.6, 6.7, 6.8 report the results of the Wilcoxon test obtained by comparing the Mean and Median Price, Rating and Rank of Dowloads of the considered migratory behaviours, respectively. Each table reports the p-value, the corrected p-value and the corresponding A^{12} effect size.

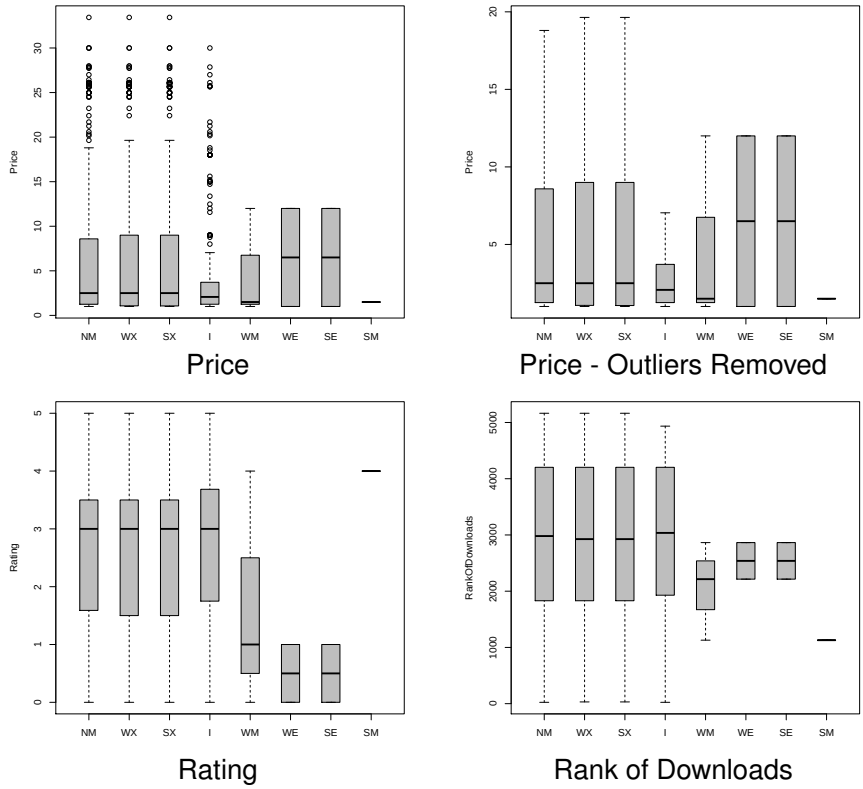


Figure 6.3: RQ2. Boxplots of Mean Price, Rating and Popularity (Rank of Downloads) for each of the non-migratory behaviours.

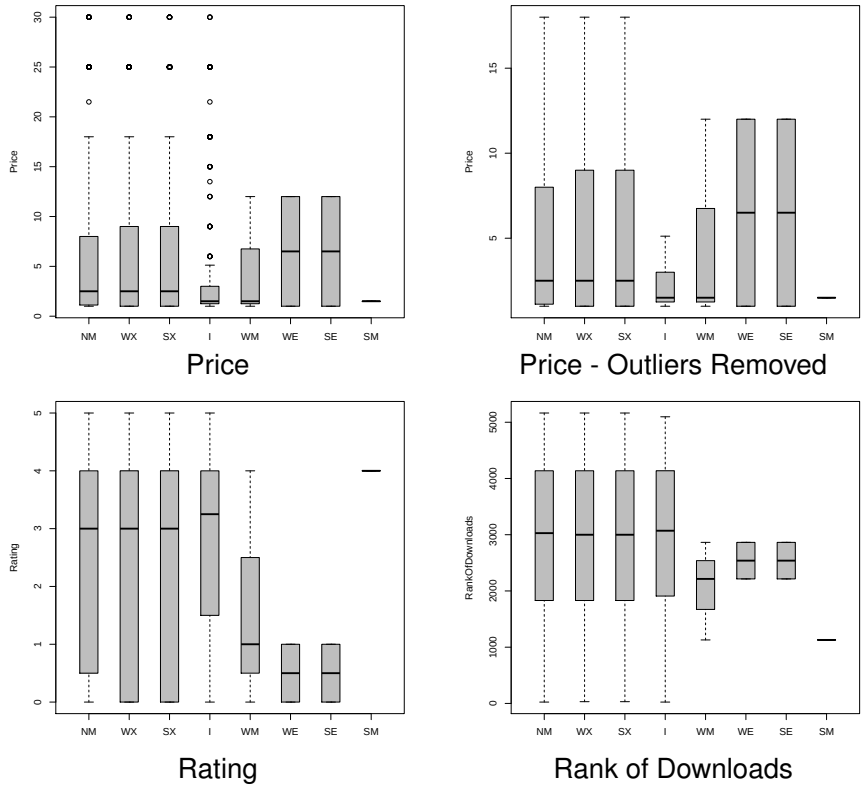


Figure 6.4: RQ2. Boxplots of Median Price, Rating and Popularity (Rank of Downloads) for each of the non-migratory behaviours.

Table 6.3: Wilcoxon Test Results: mean price.

	SX		WX		I		SM		WE	
	p	A^{12}	p	A^{12}	p	A^{12}	p	A^{12}	p	A^{12}
SX		-								
WX	0.897	0.503		-						
I	0.154	0.534	0.119	0.537		-				
SM	0.693	0.614	0.683	0.618	0.639	0.638		-		
WE	0.846	0.459	0.838	0.458	0.901	0.474	1	0.5		-
SE	0.846	0.459	0.838	0.458	0.901	0.474	1	0.5	1	0.5

6.4.4 RQ3. Correlations among Price, Popularity and Rating

Figures 6.5, 6.6, 6.7 show the scatter plots of each pair of $\{$Price, Popularity, Rating$\}$ values for each feature. These scatter plots give an insight into the Pearson and Spearman correlation analyses results presented in table 6.9.

We only report the correlation coefficient (rho value) where the p value indicates that the correlation coefficient is reliable (i.e., we have evidence that it is significantly different to zero). Where the $p > 0.05$ we leave the entry blank, since there are insufficiently many data points to allow us to draw reliable conclusions about correlations. We observe a strong positive correlation (Pearson $\rho = 0.70$ and $\rho = 0.71$) between price and rank of download (mean and median values) in features that face weak ($\mathscr{WX}$) and strong ($\mathscr{SX}$) extinction. This reveals that the higher price the higher rank of download (i.e., the less popular) for features that go extinct. We also observe a mild negative correlation between median price and rating for $\mathscr{WX}$ (Pearson $\rho = -0.60$) and $\mathscr{SX}$ (Pearson $\rho = -0.61$) features, respectively, i.e., the higher the prices the lower the rating (and vice versa).

Since prices are charged at price points, we can also compute the median rating (respectively rank of downloads) for all features that share a given price point (see Figure 6.8). Figure 6.9 show the scatter plots of each pair of $\{$Median Price, Popularity, Rating$\}$ values for non migratory behaviour and Table 6.10 reports the Spearman and Person correlation values. The results confirm the positive correlations between price and rank of download for $\mathscr{WX}$ and $\mathscr{SX}$ features (Pearson $\rho = 0.75$, Spearman $\rho = 0.77$). We find high correlation between Rating and download at median price points.

This difference in behaviour confirms those found in the BlackBerry dataset analysis. Furthermore, it confirms that the classification of features based on their migratory behaviour promises interesting insights and can aid developers in prioritising possible features.

6.5 Threats to Validity

Threats to External Validity: The feature behaviour taxonomy reported by Sarro et al. [259] is general. However, the empirical results presented in this chapter are specific to the Samsung app store. The agreement of most findings to those reported by Sarro et al. aids in their generalizability over the time period studied.

Internal Validity Threat Risk Reduction: The inferential statistical values and correlations, and all the derived metrics reported in this study were cross-checked as two researchers independently calculated the metrics arriving at the same results.

Table 6.4: Wilcoxon Test Results: mean rating

	SX		WX		I		SM		WE	
	p	A^{12}	p	A^{12}	p	A^{12}	p	A^{12}	p	A^{12}
SX		-								
WX	0.946	0.501		-						
I	0.353	0.522	0.323	0.524		-				
SM	0.28	0.813	0.277	0.815	0.289	0.809		-		
WE	0.474	0.906	0.049	0.904	0.028	0.951	0.54	1		-
SE	0.474	0.906	0.049	0.904	0.282	0.951	0.54	0.1	1	0.5

Table 6.5: Wilcoxon Test Results: mean rank of downloads comparison among the migratory behaviours.

	SX		WX		I		SM		WE	
	p	A^{12}	p	A^{12}	p	A^{12}	p	A^{12}	p	A^{12}
SX		-								
WX	0.986	0.5		-						
I	0.912	0.497	0.895	0.497		-				
SM	0.148	0.919	0.147	0.921	0.127	0.945		-		
WE	0.665	0.589	0.661	0.59	0.574	0.617	0.54	1		-
SE	0.665	0.589	0.661	0.59	0.574	0.383	0.54	1	1	0.5

Table 6.6: Wilcoxon Test Results: Median Price.

	SX		WX		I		SM		WE	
	p	A^{12}	p	A^{12}	p	A^{12}	p	A^{12}	p	A^{12}
SX		-								
WX	0.898	0.5		-						
I	0.065	0.544	0.048	0.547		-				
SM	0.748	0.593	0.736	0.598	0.929	0.528		-		
WE	0.876	0.468	0.869	0.466	0.929	0.481	1	0.5		-
SE	0.876	0.468	0.869	0.466	0.929	0.481	1	0.5	1	0.5

Table 6.7: Wilcoxon Test Results: median rating.

	SX		WX		I		SM		WE	
	p	A^{12}	p	A^{12}	p	A^{12}	p	A^{12}	p	A^{12}
SX		-								
WX	0.976	0.5		-						
I	0.386	0.521	0.369	0.521		-				
SM	0.428	0.727	0.427	0.728	0.384	0.752		-		
WE	0.157	0.787	0.159	0.785	0.097	0.838	0.54	1		-
SE	0.157	0.787	0.159	0.785	0.097	0.839	0.54	1	1	0.5

Table 6.8: Wilcoxon Test Results: median rank of downloads.

	SX		WX		I		SM		WE	
	p	A^{12}	p	A^{12}	p	A^{12}	p	A^{12}	p	A^{12}
SX		-								
WX	0.984	0.5		-						
I	0.958	0.499	0.935	0.498		-				
SM	0.153	0.914	0.152	0.915	0.127	0.945		-		
WE	0.66	0.591	0.656	0.592	0.595	0.61	0.54	1		-
SE	0.66	0.591	0.656	0.592	0.595	0.389	0.54	1	1	0.5

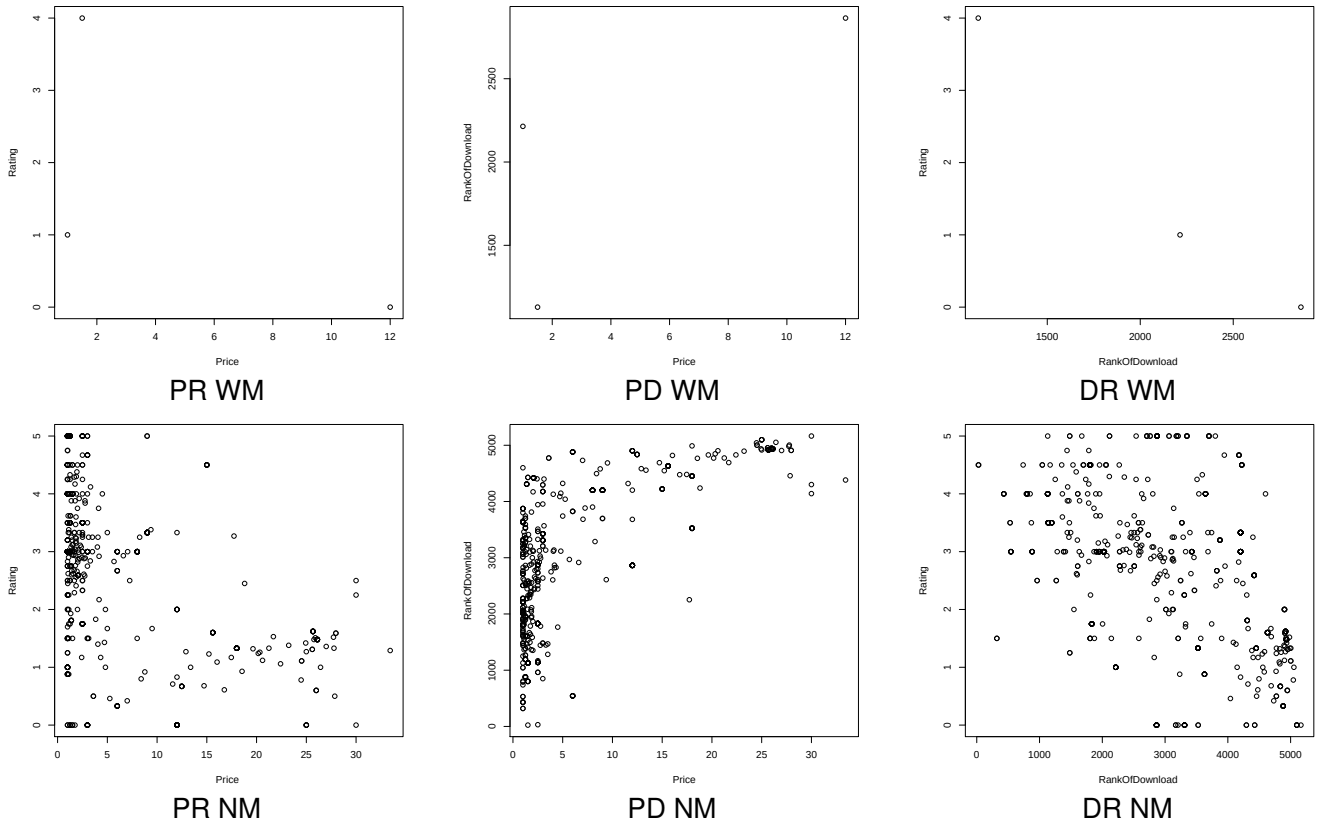


Figure 6.5: RQ3. Scatterplot of Mean Price (P), Rank of Downloads (D) and Rating (R) for the migratory behaviours (W)eak (M)igration and (N)o (M)igration

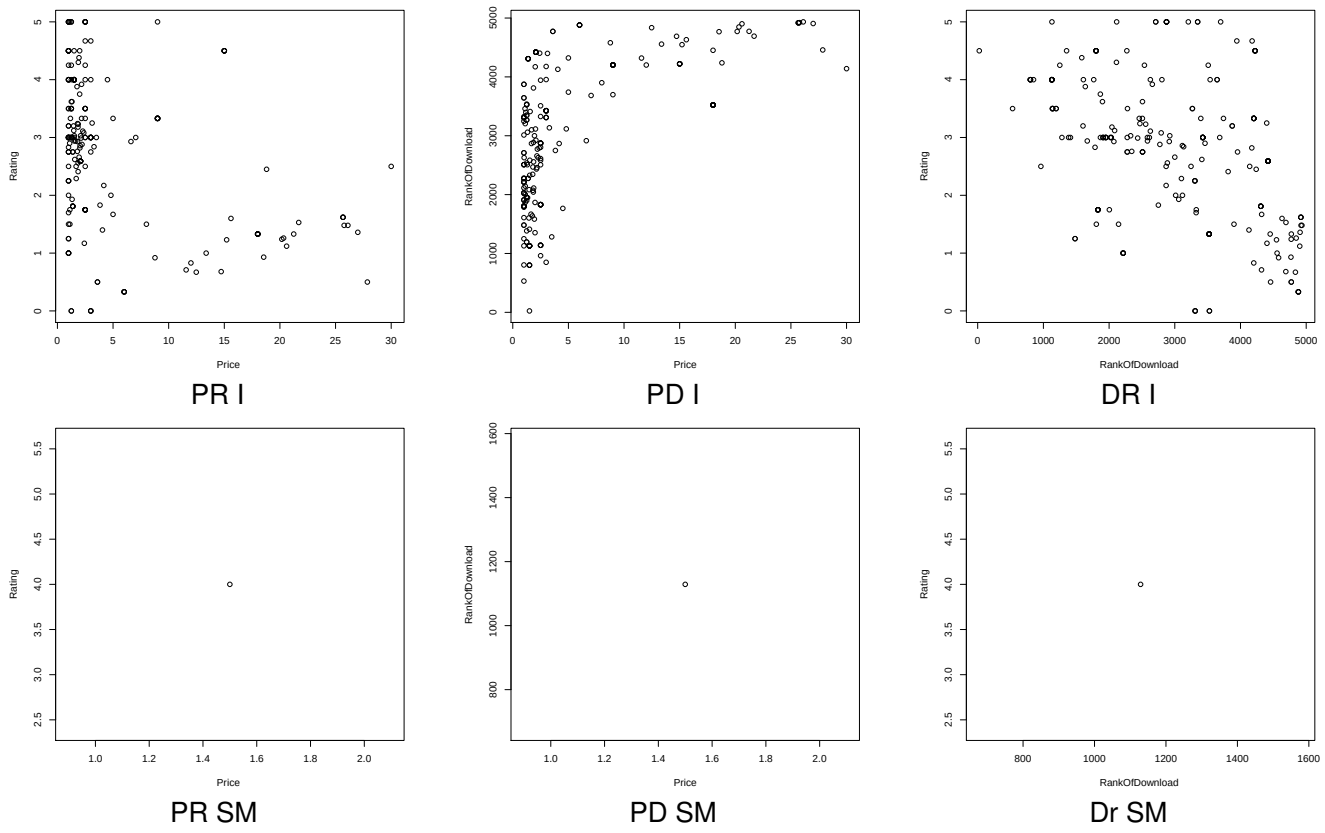


Figure 6.6: RQ3. Scatterplot of Mean Price (P), Rank of Downloads (D) and Rating (R) for the migratory behaviours (I)ntransitive and (S)trong (M)igration.

Table 6.9: RQ3. Raw Value Correlations.

Pearson and Spearman Correlation values for (P)rice, (R)ating and Rank of (D)ownloads. For completeness, all migratory behaviours are listed in the rows of the table. However, only significant correlation values ($p \leq 0.05$) are reported.

	Pearson						Spearman					
Migratory Behaviour	Mean PR	Median PR	Mean PD	Median PD	Mean RD	Median RD	Mean PR	Median PR	Mean PD	Median PD	Mean RD	Median RD
$\mathscr{NM}$	-0.47	-0.57	0.65	0.65	-0.44	-0.45	-0.50	-0.39	0.66	0.61	-0.46	-0.37
$\mathscr{WX}$	-0.51	-0.60	0.70	0.71	-0.44	-0.46	-0.51	-0.46	0.66	0.66	-0.46	-0.37
$\mathscr{SX}$	-0.51	-0.61	0.70	0.71	-0.44	-0.46	-0.50	-0.46	0.66	0.66	-0.46	-0.37
$\mathscr{I}$	-0.37	-0.48	0.56	0.56	-0.45	-0.44	-0.34	-0.27	0.58	0.50	-0.48	-0.37
$\mathscr{WM}$	-	-	-	-	-	-	-	-	-	-	-	-
$\mathscr{SM}$	-	-	-	-	-	-	-	-	-	-	-	-
$\mathscr{WE}$	-	-	-	-	-	-	-	-	-	-	-	-
$\mathscr{SE}$	-	-	-	-	-	-	-	-	-	-	-	-

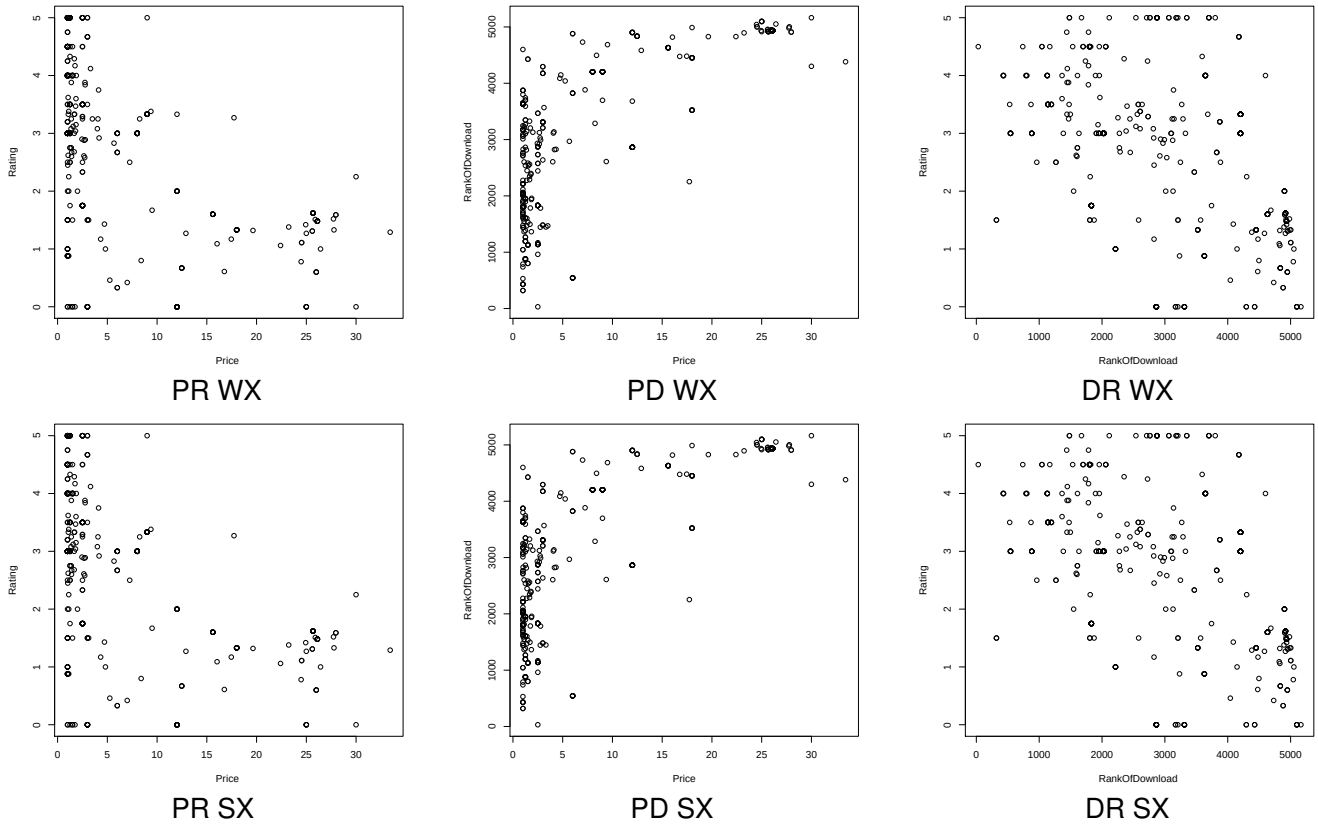


Figure 6.7: RQ3. Scatterplot of Mean Price (P), Rank of Downloads (D) and Rating (R) for the migratory behaviours (W)eak e(X)tinction and (S)trong e(X)tinction.

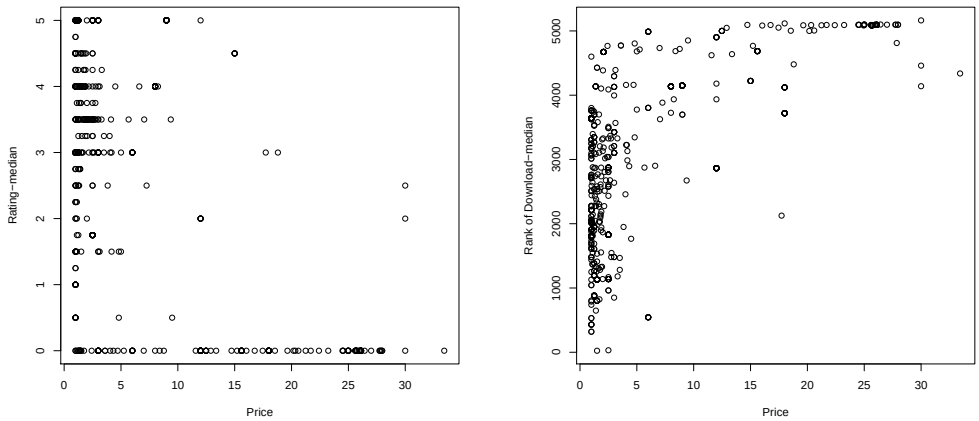


Figure 6.8: RQ3. Scatterplot of Median Price (P), Rank of Downloads (D) and Rating (R) for all the features. Please, note that we grouped the points based on their median values.

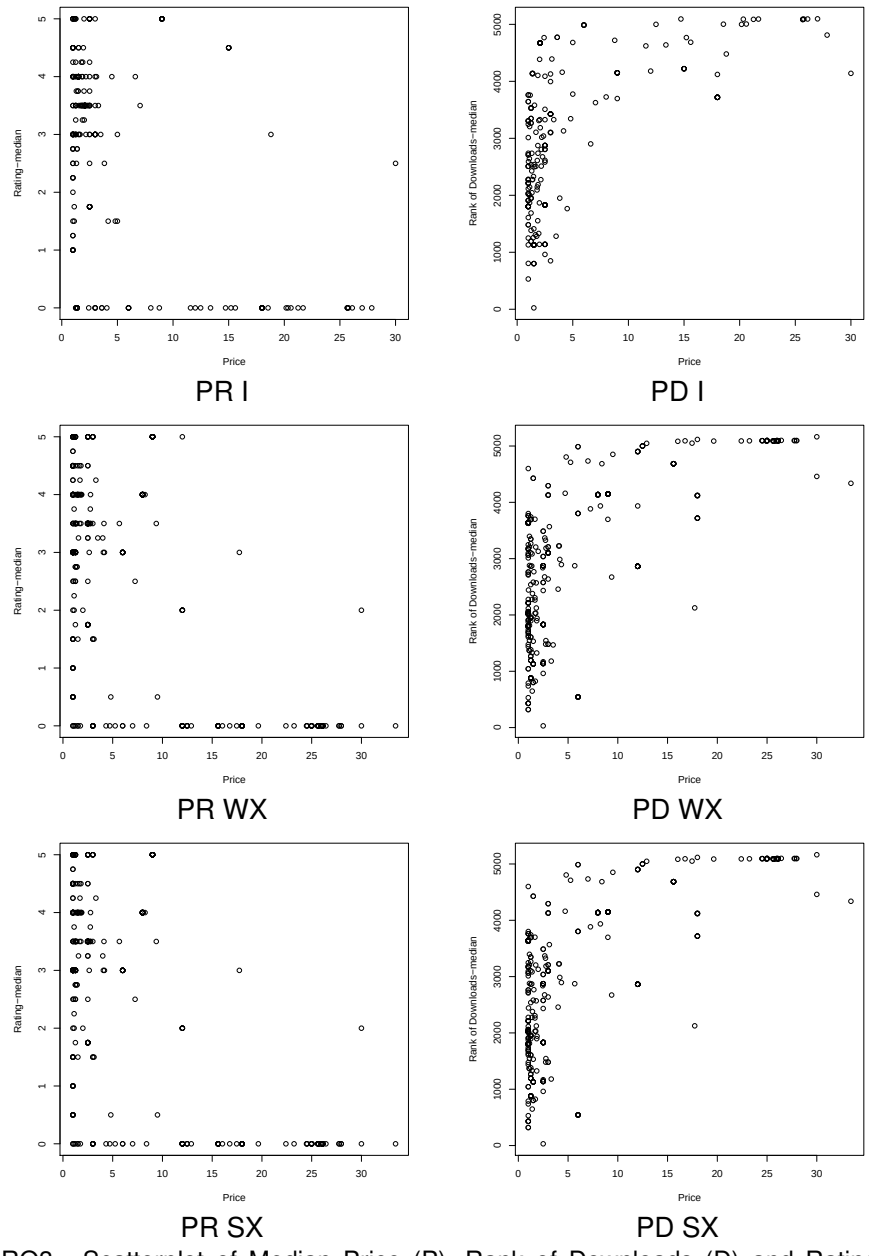


Figure 6.9: RQ3. Scatterplot of Median Price (P), Rank of Downloads (D) and Rating (R) for the non-migratory behaviours I, WX, SX. Please, note that we grouped the points based on their median values.

Table 6.10: RQ3. Median Price Point Correlations.

Pearson and Spearman correlation values for median (R)ating and Rank of (D)ownloads for each median price point. For completeness, all migratory behaviours are listed in the rows of the table. However, only significant correlation values ($p \leq 0.05$) are reported.

Migratory	Pearson		Spearman	
Behaviour	PR	PD	PR	PD
$\mathscr{NM}$	-0.53	0.70	-	0.78
$\mathscr{WX}$	-0.59	0.75	-0.40	0.77
$\mathscr{SX}$	-0.59	0.75	-0.40	0.77
$\mathscr{I}$	-0.44	0.64	-	0.74
$\mathscr{WM}$	-	-	-	-
$\mathscr{SM}$	-	-	-	-
$\mathscr{WE}$	-	-	-	-
$\mathscr{SE}$	-	-	-	-

Table 6.11: RQ3. Mean Price Point Correlations.

Pearson and Spearman correlation values for mean (R)ating and Rank of (D)ownloads for each mean price point. For completeness, all migratory behaviours are listed in the rows of the table. However, only significant correlation values ($p \leq 0.05$) are reported.

Migratory	Pearson		Spearman	
Behaviour	PR	PD	PR	PD
$\mathscr{NM}$	-0.65	0.76	-0.62	0.80
$\mathscr{WX}$	-0.64	0.80	-0.63	0.83
$\mathscr{SX}$	-0.64	0.80	-0.63	0.84
$\mathscr{I}$	-0.58	0.70	-0.59	0.73
$\mathscr{WM}$	-	-	-	-
$\mathscr{SM}$	-	-	-	-
$\mathscr{WE}$	-	-	-	-
$\mathscr{SE}$	-	-	-	-

Threats to Construct Validity: The features extracted from the app descriptions are what the developer deems interesting to include. They are the features claimed by the developer and we have not method of insuring they are indeed included in the app or actual requirements. However, we believe that the feature claims still hold as a valuable unit of analysis that holds value to other developers.

6.6 Conclusions

This research proposes a new taxonomy of classifying application feature behaviour among the different categories of the app store and whether different behaviours hold different values and promises towards the app success[259].

The results showed that, indeed, features in the app store exhibit these kinds of migration behaviour, some of which shows differences in the price, rating and popularity they achieve compared to other migratory behaviours. We found, for example, that migratory features, ones that have transferable value, tend to be cheaper and lower rated than non-migratory ones, however, they tend to be

more downloaded. We also found that some correlates with either price, rating or popularity of the feature. We found that features that eventually die out usually have prices that correlate positively with their popularity. These analyses uncover interesting findings regarding the relationship between a feature's intransitive nature and how it is perceived by users and developers. This type of analysis can greatly aid decision making processes for mobile app developers when eliciting requirements, prioritising features for implementation and in designing the end product.

Chapter 7

Conclusion and Future Work

The concept of features in software engineering is crucial to a number of its activities, whereas app stores are newly emerging form of software repositories, ripe for mining. This thesis leverages this valuable source of information to uncover interesting and actionable information regarding mobile application features to facilitate their development and evolution tasks.

Chapter 2 presents a review of the literature pertaining to the study of features (forming, locating and extracting), automatic software categorisation and, finally, app store analysis.

Chapter 3 reports a comprehensive exploratory empirical study of mobile app developers' interactions with app stores. It investigates app stores' involvement throughout the classical software engineering processes; in addition to skill sets and metrics that mobile developers deem important which have been introduced by the nature of app stores. The study finds that mobile app stores are essential during requirements elicitation and maintenance phases. Furthermore, it provides evidence that app stores play a major role in developer-user interaction; increase market transparency and alter release management decisions. The study highlighted developers' reliance on feature elicitation from other similar applications on the app store, motivating the subsequent chapters in this thesis.

Chapter 4 employs a feature extraction algorithm from natural language descriptions of apps, incorporating semantic coding, in order to represent apps in the app store for classification. This representation enhances over the baseline (as shown in Chapter 5). Representing apps using their advertised features enables carrying out clustering techniques to help cluster applications based on their prevalent functionality. This clustering is of finer granularity than app store categorisation and can help developers find similar application with common co-locating features. Chapter 5 extends feature extraction by using sentence dependency parsing for a keyword based representation of apps. It then compares several feature extraction techniques from natural language for the task of app clustering.

Chapter 6 further applies the aforementioned feature extraction algorithm from natural language descriptions of mobile apps to study the migratory behaviour of features in the app store [259]. The study reveals that features of different migratory behaviours tend to have differences in terms of their price, rating and popularity. In the BlackBerry app store, features that are intransitive and tend to remain in the same category carry higher monetary value than others; however, in the Samsung app store, features with higher price tend to die out. The study of migratory behaviour of feature promises to guide mobile developers in eliciting requirements from similar and competing apps. In

facilitating the task of exploring such applications, the remaining two chapters investigate clustering techniques of the mobile app store.

Feature extraction using the algorithms presented in this thesis have been applied and used in industry during a 3-month internship at a London branch of a global investment bank. Natural language feature extraction, when used to represent applications, can be used to quantify the similarity among applications and their artefacts. This algorithm was found helpful in augmenting the firm's documentation linking applications with their formal capabilities, classifying the applications into application domains and extracting the roles of different entities from the applications' descriptions. Such tasks, performed manually prior to the introduction of our code, helped in populating the reference architecture which is crucial to ensure the firm's adherence to financial regulators. This has shown the transferability of the developed techniques throughout this PhD to solve another software engineering task in practice.

The method of feature extraction from unstructured artefacts can be further utilised in future research. When coupled with source code analysis, for example, co-occurring features and chunks of codes can be linked to certain degree of accuracy which may help in the feature location problem, bug location problem, and code understandability.

The clustering of application devised in Chapter 4 and demonstrated in Chapter 5 can be further extended to provide several views of the app store repository depending on other criteria of interest. The clustering can cater to code-reuse (therefore can be carried out using source code or using APIs extracted features), it can cater to finding applications that share similar user interfaces, bug complaints, usage scenarios, and so on.

App stores, furthermore, provide large amounts of historical data of applications' evolution. Along with the evolution of provided features, the success of the app can be observed. This can be utilised for data-driven prediction tasks that can support developers in their decision making process when considering new features to include. App store *predictive analytics*, we believe, can provide several advantages to the mobile development team members throughout the lifecycle of the mobile app.

Appendix A

Questionnaire - App Store Ecosystems Effects

3120

App Store Effects on Software Lifecycle

Introduction

How are app stores changing traditional software engineering life cycle processes?
Help research design and build the proper tools for you.

How to fill this survey

Think about your practices when interacting with the app store. The survey is arranged to take you through the journey of building an app starting from conception and ending with performance metrics. Finally, we'll ask you about demographics before concluding the survey.

More About This Study
This is an exploratory study conducted by a team of researchers in the Systems Software Engineering research group at University College London. The goal of this study is to measure how much, and in what way, do app stores change how software engineers design, develop, test and maintain mobile apps. We believe that an exploratory study will aid in understanding the current practices of developers which will lead to a better informed research and the design of better tools.

UCL Ethics project number: 6917/001

UCL Data Protection Registration reference No Z6364106/2015/04/16, section 19, research: social research.

All data will be collected and stored in accordance with the Data Protection Act 1998.

App Store Effects on Software Lifecycle

We'd like to know more about you..

1. What is your age?

- ◯ 18 to 24
- ◯ 25 to 34
- ◯ 35 to 44
- ◯ 45 to 54
- ◯ 55 to 64
- ◯ 65 to 74
- ◯ 75 or older

2. In which country are you based?

App Store Effects on Software Lifecycle

Idea Conception and Requirements Gathering

3. I survey the app store to validate the viability/feasibility of my app idea (main functionality)

Strongly disagree	Disagree	Neutral	Agree	Strongly Agree	*Not Applicable*
○	○	○	○	○	○

4. I explore other apps in the app store for GUI design ideas and trends.

Very rarely	Rarely	Occasionally	Frequently	Very frequently	*Not Applicable*
○	○	○	○	○	○

Other sources you consider:

5. When I already settle on a main app idea, I gather what other features to include in my app from these sources:

	Very rarely	Rarely	Occasionally	Frequently	Very frequently	*Not Applicable*
Similar apps on the app store	○	○	○	○	○	○
Similar apps in general (web/desktop)	○	○	○	○	○	○
User surveys and focus groups	○	○	○	○	○	○

Other (please specify)

6. If I use the app store to gather features for my app by looking at similar apps, I would pay attention to these elements:

(Rate how interesting these things are for you)

	Not at all interested	Not very interested	Neutral	Interested	Very interested	*Not Applicable*
Icon and name	○	○	○	○	○	○
Developer's Name	○	○	○	○	○	○
Screenshots	○	○	○	○	○	○
Description	○	○	○	○	○	○
User feedback and reviews	○	○	○	○	○	○
Rating	○	○	○	○	○	○
Version number	○	○	○	○	○	○

Other (please specify)

7. Other comments regarding gathering requirements?

App Store Effects on Software Lifecycle

Call-To-Action [Part 1]

Call-to-action: a pop-up within your app asking the user to rate/review the app in the app store.

* 8. Do you have a call-to-action within your app for users to rate it?

- ◯ Yes
- ◯ No

App Store Effects on Software Lifecycle

Call-To-Action

Because you answered 'Yes'

9. I only trigger the rating call-to-action when I'm confident the user is enjoying the app (e.g. after several uses)

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	*Not Applicable*
◯	◯	◯	◯	◯	◯

10. My app asks the user for their rating and only directs them to insert their rating in the app store if it is high enough.

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	*Not Applicable*
◯	◯	◯	◯	◯	◯

App Store Effects on Software Lifecycle

Alpha/Beta Testing [Part 1]

* 11. Do you release Alpha and/or Beta versions of your app

- ◯ Yes
- ◯ No

App Store Effects on Software Lifecycle

Alpha/Beta Testing

Because you answered 'Yes'

12. When releasing an Alpha or Beta version of my app in the app store, I'm interested in:

	Not at all interested	Not very interested	Neutral	Interested	Very interested	*Not Applicable*
Performance issues (non-functional requirements)	◯	◯	◯	◯	◯	◯
Bugs	◯	◯	◯	◯	◯	◯
Feedback on missing features (functional requirements)	◯	◯	◯	◯	◯	◯
Feedback on unwanted features	◯	◯	◯	◯	◯	◯
Generic reception by users (rating/reviews/recommendations/social hype)	◯	◯	◯	◯	◯	◯

13. Other comments regarding Alpha and/or Beta testing?

App Store Effects on Software Lifecycle

A/B Testing [Part 1]

A/B Testing: having two concurrent versions of your app in order to compare which versions performs better.

* 14. Do you perform A/B Testing for your app?

- Yes
- No

App Store Effects on Software Lifecycle

A/B Testing

Because you answered 'yes' to the previous question.

15. I am interested in differentiating the user rating and reviews that my app gets in app stores for both A and B versions.

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	*Not Applicable*
○	○	○	○	○	○

App Store Effects on Software Lifecycle

Maintenance Tasks (Bug fixes and enhancements)

3125

16. How often do you receive bug reports from the following sources:

	Very Rarely	Rarely	Occasionally	Frequently	Very frequently	*Not Applicable*
Automatic in-app crash reporting	○	○	○	○	○	○
User-initiated bug reporting functionality inside the app	○	○	○	○	○	○
Private messages from users (emails and direct messages in social media)	○	○	○	○	○	○
User public complaints on social media	○	○	○	○	○	○
User reviews on the Google Play app store	○	○	○	○	○	○
User reviews on the Apple app store	○	○	○	○	○	○
User reviews on other app stores	○	○	○	○	○	○

Other (please specify)

17. Of these sources, rate how often you actually fix these bugs based on their source:

	Very Rarely	Rarely	Occasionally	Frequently	Very frequently	*Not Applicable*
Automatic in-app crash reporting	○	○	○	○	○	○
User-initiated bug reporting functionality inside the app	○	○	○	○	○	○
User reviews on the app store	○	○	○	○	○	○
User public complaints on social media	○	○	○	○	○	○
Private messages from users (emails and direct messages in social media)	○	○	○	○	○	○

Can you explain why? and add any other missing sources if any.

18. I find it easy to extract bug reports from user reviews in the app store.

Strongly disagree	Disagree	Neutral	Agree	Strongly agree	*Not Applicable*
○	○	○	○	○	○

If answered 'Disagree' or 'Strongly Disagree' please tell us why?

19. Rate how important are these types of app reviews for app maintenance and enhancement:

	Not Important	Slightly Important	Mild Importance	Important	Very Important	*Not Applicable*
Generic praise	◯	◯	◯	◯	◯	◯
Usage scenario	◯	◯	◯	◯	◯	◯
Features users like	◯	◯	◯	◯	◯	◯
Bug reports	◯	◯	◯	◯	◯	◯
Features users hate	◯	◯	◯	◯	◯	◯
Features users request	◯	◯	◯	◯	◯	◯

Other (please specify)

20. Other comments regarding bug fixes, user reviews, and maintenance tasks in general?

App Store Effects on Software Lifecycle

Release Management [Part 1]

* 21. Have you released an update (or more) of your app?

(I.e does your app have more than one release)

- ◯ Yes
- ◯ No

App Store Effects on Software Lifecycle

Release Management

Because you have answered 'Yes' to the previous question.
Questions regarding your decision making after the initial release.

22. When you are planning on enhancing your app by including new features, how often do you use these sources to find new features to include?

	Very Rarely	Rarely	Occasionally	Frequently	Very frequently	*Not Applicable*
Initial app strategy and vision	○	○	○	○	○	○
User surveys and focus groups	○	○	○	○	○	○
Private messages from users (emails and direct messages through social media)	○	○	○	○	○	○
User reviews of your app in the app store	○	○	○	○	○	○
Similar apps in the app store	○	○	○	○	○	○
User reviews of similar apps in the app store	○	○	○	○	○	○

Other (please specify)

23. How frequently is a new release triggered by these events (i.e. main cause of new release)

	Very rarely	Rarely	Occasionally	Frequently	Very frequently	*Not Applicable*
Bug fixes	○	○	○	○	○	○
Performance improvement	○	○	○	○	○	○
Adding new feature / removing unwanted feature	○	○	○	○	○	○
Packaging changes (to change name, icon, screenshots and/or description)	○	○	○	○	○	○

Other (please specify)

24. I have changed how I plan releases because of the app store reviewing and approval period.

Strongly disagree	Disagree	Neutral	Agree	Strongly agree	*Not Applicable*
○	○	○	○	○	○

How did you change your plan?

25. Other comments regarding releasing new updates of your app in the app store?

App Store Effects on Software Lifecycle

Emerging New Skill sets

26. How do you define 'success' in the app store?

27. I find it clear how to reach that success in the app store.

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	*Not Applicable*
◯	◯	◯	◯	◯	◯

28. How do you measure the success of your app?

Based on your defined success goal

29. Rate how important are these factors to build a successful app:

	Not Important	Slightly Important	Neutral	Important	Very Important
App's Novelty	◯	◯	◯	◯	◯
The quality of the UX (including app performance)	◯	◯	◯	◯	◯
The quality of the code (well coded and well documented)	◯	◯	◯	◯	◯
Having a good brand (attractive page on the app store) and marketing strategy (including user engagement).	◯	◯	◯	◯	◯
App visibility (easy to discover by users).	◯	◯	◯	◯	◯
Luck.	◯	◯	◯	◯	◯

Other (please specify)

30. It is important to have someone in the team responsible for marketing and business intelligence.

Strongly disagree	Disagree	Neutral	Agree	Strongly agree
◯	◯	◯	◯	◯

31. Think of the person in the team who is responsible for any of app marketing tasks (could be you).

(App marketing tasks: writing description, screenshots, video promo, user engagement, app analytics, ad campaigns, monetisation, etc.)

This person..

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	*Do Not Know*
..is dedicated to these tasks (i.e. have no other roles)	◯	◯	◯	◯	◯	◯
..has formal training in marketing and/or business intelligence.	◯	◯	◯	◯	◯	◯
..is self-taught and relies on experience.	◯	◯	◯	◯	◯	◯
..imitates the strategies of successful apps.	◯	◯	◯	◯	◯	◯
..mainly relies on tuition and common sense in some of these tasks.	◯	◯	◯	◯	◯	◯

Other (please specify)

32. Any other comments regarding new skill nowadays required of development teams to succeed in an app store environment?

App Store Effects on Software Lifecycle

Demographics

Tell us more about yourself.

33. How many years of experience do you have..

in developing (web/desktop) applications?

in developing mobile apps?

in dealing with app stores?

34. What is your formal education?

- [ ] Technical / Engineering
- [ ] Business
- [ ] Other (please specify)

35. What app stores have you developed/managed apps for?

You may pick more than one.

- [ ] Apple app store (iOS)
- [ ] Google Play Store (Android)
- [ ] Windows Phone Store
- [ ] Other (please specify)

App Store Effects on Software Lifecycle

About your App

Tell us more about your app. If you have more than one app, think of your primary one, could be the most successful or the current one.

3130

36. My main app is deployed in (or planning to be deployed in..)

You may pick more than one.

- [ ] iOS app store
- [ ] Google Play store (Android)
- [ ] Other (please specify)

37. How many active users are using your app?

38. What is the size of the team working on the app?

Number of people working full time:

Number of people working part time:

39. What is your role in the team?

List your responsibilities.

40. What kind of tasks are you outsourcing at the moment?

App Store Effects on Software Lifecycle

App Analytics

* 41. We would like to retrieve basic statistics regarding your app. Would you be willing to share the App's page on the app store? Alternatively you can manually enter the statistics.

The name of your app will not be included in the study and we'll ensure your anonymity.

- ( ) I don't mind providing the link to my app
- ( ) I would rather type in the numbers manually

App Store Effects on Software Lifecycle

App Analytics

Because you don't mind sharing the link to your app.

* 42. Link to your app(s):

If more than one URL, please separate with a semi-colon.

App Store Effects on Software Lifecycle

Your App's Metrics

Provide approximate numbers to the metrics below.
If you have more than one app, think of the main one you'd like us to consider.

43. Number of downloads:

44. Number of versions:

45. Number of reviews:

46. Average rating:

47. Revenue model:

- ◯ Free (and no ads)
- ◯ Free with Ads
- ◯ Paid
- ◯ Freemium (in-app purchases)
- ◯ Subscription
- ◯ Other (please specify)

App Store Effects on Software Lifecycle

48. Price:

App Store Effects on Software Lifecycle

Thank you!

49. Thank you for completing the survey!
If you would like us to send you the results of this study, type your email below. Otherwise, click 'Done'.
Your email will not be associated with your answers.

Bibliography

[1] Bosch and Jan, “From software product lines to software ecosystems,” 2009.

[2] J. Bosch and P. Bosch-Sijtsema, “From integration to composition: On the impact of software product lines, global development and ecosystems,” *Journal of Systems and Software*, vol. 83, pp. 67–76, jan 2010.

[3] K. Manikas and K. M. Hansen, “Software ecosystems âĂŞ A systematic literature review,” *Journal of Systems and Software*, vol. 86, pp. 1294–1306, may 2013.

[4] S. L. Lim, P. J. Bentley, N. Kanakam, F. Ishikawa, and S. Honiden, “Investigating Country Differences in Mobile App User Behavior and Challenges for Software Engineering,” *IEEE Transactions on Software Engineering*, vol. 41, pp. 40–64, jan 2015.

[5] C. R. Turner, A. L. Wolf, A. Fuggetta, and L. Lavazza, “Feature Engineering,” *Proceedings of the 9th international workshop on Software specification and design (IWSSD98)*, p. 162, apr 1998.

[6] “Ieee standard for software and system test documentation,” July 2008.

[7] A. Abran, P. Bourque, R. Dupuis, and J. W. Moore, eds., *Guide to the Software Engineering Body of Knowledge - SWEBOK*. Piscataway, NJ, USA: IEEE Press, 2001.

[8] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, “Feature location in source code: a taxonomy and survey,” *Journal of Software: Evolution and Process*, vol. 25, pp. 53–95, Jan. 2013.

[9] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, “Feature-Oriented Domain Analysis (FODA) Feasibility Study,” tech. rep., Software Engineering Institute, Carnegie Melon, nov 1990.

[10] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, “FORM: A feature-oriented reuse method with domain-specific reference architectures,” *Annals of Software Engineering*, vol. 5, no. 1, pp. 143–168, 1998.

[11] Jing Sun, Hongyu Zhang, Yuan Fang, and Hai Wang, “Formal Semantics and Verification for Feature Modeling,” in *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS’05)*, pp. 303–312, IEEE, 2005.

[12] K. C. Kang, P. Donohoe, and J. Lee, “Feature-oriented product line engineering,” *IEEE Software*, vol. 19, pp. 58–65, July 2002.

[13] S. Apel and C. Kästner, "An Overview of Feature-Oriented Software Development," *Journal of Object Technology*, vol. 8, no. 5, pp. 49–84, 2009.

[14] C. Reid Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf, "A conceptual basis for feature engineering," *Journal of Systems and Software*, vol. 49, pp. 3–15, Dec. 1999.

[15] A. Classen, P. Heymans, and P.-Y. Schobbens, "What's in a feature: a requirements engineering perspective," *Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering (FASE'08/ETAPS'08)*, pp. 16–30, mar 2008.

[16] Kun Chen, Wei Zhang, Haiyan Zhao, and Hong Mei, "An approach to constructing feature models based on requirements clustering," in *13th IEEE International Conference on Requirements Engineering (RE'05)*, pp. 31–40, IEEE, 2005.

[17] N. Niu and S. Easterbrook, "On-Demand Cluster Analysis for Product Line Functional Requirements," in *2008 12th International Software Product Line Conference*, pp. 87–96, IEEE, Sept. 2008.

[18] N. Weston, R. Chitchyan, and A. Rashid, "A framework for constructing semantically composable feature models from natural language requirements," *Proceedings of the 13th International Software Product Line Conference*, pp. 211–220, aug 2009.

[19] J. S. Thakur and A. Gupta, "Identifying domain elements from textual specifications," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, (New York, New York, USA), pp. 566–577, ACM Press, 2016.

[20] B. Wang, W. Zhang, H. Zhao, Z. Jin, and H. Mei, "A Use Case Based Approach to Feature Models' Construction," in *2009 17th IEEE International Requirements Engineering Conference*, pp. 121–130, IEEE, Aug. 2009.

[21] H. Dumitru, M. Gibiec, N. Hariri, J. Cleland-Huang, B. Mobasher, C. Castro-Herrera, and M. Mirakhorli, "On-demand feature recommendations derived from mining public product descriptions," in *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, (New York, New York, USA), p. 181, ACM Press, May 2011.

[22] M. Acher, A. Cleve, G. Perrouin, P. Heymans, C. Vanbeneden, P. Collet, and P. Lahire, "On extracting feature models from product descriptions," in *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems - VaMoS '12*, (New York, New York, USA), pp. 45–54, ACM Press, Jan. 2012.

[23] N. Hariri, C. Castro-Herrera, M. Mirakhorli, J. Cleland-Huang, and B. Mobasher, "Supporting Domain Analysis through Mining and Recommending Features from Online Product Listings," *IEEE Transactions on Software Engineering*, vol. 39, pp. 1736–1752, Dec. 2013.

[24] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, and P. Heymans, "Feature model extraction from large collections of informal product descriptions," in *Proceedings of*

the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013, (New York, New York, USA), p. 290, ACM Press, Aug. 2013.

[25] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *Proceedings of 1993 15th International Conference on Software Engineering*, pp. 482–498, IEEE Comput. Soc. Press, 1993.

[26] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *Proceedings 10th International Workshop on Program Comprehension*, pp. 271–278, IEEE Comput. Soc, 2002.

[27] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: a taxonomy," *IEEE Software*, vol. 7, pp. 13–17, Jan. 1990.

[28] S. Sim, C. Clarke, and R. Holt, "Archetypal source code searches: a survey of software developers and maintainers," in *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*, pp. 180–187, IEEE Comput. Soc, 1998.

[29] N. Wilde, J. A. Gomez, T. Gust, and D. Strasburg, "Locating user functionality in old code," in *Proceedings Conference on Software Maintenance 1992*, pp. 200–205, IEEE Comput. Soc. Press, 1992.

[30] N. Wilde and M. C. Scully, "Software reconnaissance: Mapping program features to code," *Journal of Software Maintenance: Research and Practice*, vol. 7, pp. 49–62, Jan. 1995.

[31] A. D. Eisenberg and K. De Volder, "Dynamic feature traces: finding features in unfamiliar code," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 337–346, IEEE, 2005.

[32] E. Wong, S. S. Gokhale, J. R. Horgan, and K. S. Trivedi, "Locating program features using execution slices," in *Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99 (Cat. No.PR00122)*, pp. 194–203, IEEE Comput. Soc, 1999.

[33] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Transactions on Software Engineering*, vol. 29, pp. 210–224, Mar. 2003.

[34] K. Chen and V. Rajlich, "Case study of feature location using dependence graph," in *Proceedings of the 8th International Workshop on Program Comprehension*, IWPC '00, (Washington, DC, USA), pp. 241–, IEEE Computer Society, 2000.

[35] M. P. Robillard, "Topology analysis of software dependencies," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, pp. 18:1–18:36, Aug. 2008.

[36] M. P. Robillard and G. C. Murphy, "Concern graphs: finding and describing concerns using structural program dependencies," in *Proceedings of the 24th international conference on Software engineering*, pp. 406–416, ACM, 2002.

[37] Z. M. Saul, V. Filkov, P. Devanbu, and C. Bird, "Recommending random walks," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 15–24, ACM, 2007.

[38] M. Trifu, "Using dataflow information for concern identification in object-oriented software systems," in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pp. 193–202, IEEE, 2008.

[39] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic, "An information retrieval approach to concept location in source code," in *11th Working Conference on Reverse Engineering*, pp. 214–223, IEEE Comput. Soc, 2004.

[40] D. Poshyvanyk and A. Marcus, "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code," in *15th IEEE International Conference on Program Comprehension (ICPC '07)*, pp. 37–48, IEEE, June 2007.

[41] A. Kuhn, S. Ducasse, and T. Girba, "Enriching Reverse Engineering with Semantic Clustering," in *12th Working Conference on Reverse Engineering (WCRE'05)*, pp. 133–142, IEEE, 2005.

[42] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, pp. 230–243, mar 2007.

[43] B. Cleary, C. Exton, J. Buckley, and M. English, "An empirical analysis of information retrieval based concept location techniques in software comprehension," *Empirical Software Engineering*, vol. 14, no. 1, pp. 93–130, 2009.

[44] D. Shepherd, L. Pollock, and K. Vijay-Shanker, "Towards supporting on-demand virtual remodularization using program graphs," in *Proceedings of the 5th international conference on Aspect-oriented software development*, pp. 3–14, ACM, 2006.

[45] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 232–242, IEEE Computer Society, 2009.

[46] S. L. Abebe and P. Tonella, "Natural language parsing of program element names for concept extraction," in *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pp. 156–159, IEEE, 2010.

[47] M. Würsch, G. Ghezzi, G. Reif, and H. C. Gall, "Supporting developers with natural language queries," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 165–174, ACM, 2010.

[48] Y. S. Maarek, D. M. Berry, and G. E. Kaiser, "An information retrieval approach for automatically constructing software libraries," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 800–813, 1991.

[49] I. van Willegen, L. Rothkrantz, and P. Wiggers, "Lexical affinity measure between words," in *Text, Speech and Dialogue: 12th International Conference, TSD 2009, Pilsen, Czech Republic, September 13-17, 2009. Proceedings*, vol. 5729, p. 234, Springer, 2009.

[50] N. Niu and S. Easterbrook, "Extracting and Modeling Product Line Functional Requirements," in *2008 16th IEEE International Requirements Engineering Conference*, pp. 155–164, IEEE, sep 2008.

[51] J. Cleland-Huang, H. Dumitru, C. Duan, and C. Castro-Herrera, "Automated support for managing feature requests in open forums," *Communications of the ACM*, vol. 52, p. 68, oct 2009.

[52] M. Rahimi and J. Cleland-Huang, "Personas in the middle: automated support for creating personas as focal points in feature gathering forums," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*, (New York, New York, USA), pp. 479–484, ACM Press, 2014.

[53] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of Duplicate Defect Reports Using Natural Language Processing," in *29th International Conference on Software Engineering (ICSE'07)*, pp. 499–510, IEEE, may 2007.

[54] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 13th international conference on Software engineering - ICSE '08*, (New York, New York, USA), p. 461, ACM Press, 2008.

[55] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pp. 253–262, IEEE, nov 2011.

[56] A. Alipour, A. Hindle, and E. Stroulia, "A contextual approach towards more accurate duplicate bug report detection," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, pp. 183–192, IEEE, may 2013.

[57] A. Hindle, A. Alipour, and E. Stroulia, "A contextual approach towards more accurate duplicate bug report detection and ranking," *Empirical Software Engineering*, vol. 21, pp. 368–410, apr 2016.

[58] M. S. Rakha, W. Shang, and A. E. Hassan, "Studying the needed effort for identifying duplicate issues," *Empirical Software Engineering*, vol. 21, pp. 1960–1989, oct 2016.

[59] M. S. Rakha, C.-P. Bezemer, and A. E. Hassan, "Revisiting the Performance Evaluation of Automated Approaches for the Retrieval of Duplicate Issue Reports," *IEEE Transactions on Software Engineering*, pp. 1–1, 2017.

[60] T. Merten, M. Falis, P. Hubner, T. Quirchmayr, S. Bursner, and B. Paech, "Software Feature Request Detection in Issue Tracking Systems," in *2016 IEEE 24th International Requirements Engineering Conference (RE)*, pp. 166–175, IEEE, sep 2016.

[61] N. H. Bakar, Z. M. Kasirun, and N. Salleh, "Feature extraction approaches from natural language requirements for reuse in software product lines: A systematic literature review," *Journal of Systems and Software*, vol. 106, pp. 132–149, aug 2015.

[62] A. Finkelstein, M. Harman, Y. Jia, W. Martin, F. Sarro, and Y. Zhang, "Investigating the relationship between price, rating, and popularity in the blackberry world app store," *Information & Software Technology*, vol. 87, pp. 119–139, 2017.

[63] M. Harman, Y. Jia, and Y. Zhang, "App store mining and analysis: MSR for app stores," in *IEEE International Working Conference on Mining Software Repositories*, (Zurich, Switzerland), pp. 108–111, 2012.

[64] F. Sarro, M. Harman, Y. Jia, and Y. Zhang, "Customer rating reactions can be predicted purely using app features," in *Proceedings of the 26th IEEE International Requirements Engineering Conference*, RE'18, 2018.

[65] M. Nayebi and G. Ruhe, "Optimized Functionality for Super Mobile Apps," in *2017 IEEE 25th International Requirements Engineering Conference (RE)*, pp. 388–393, IEEE, sep 2017.

[66] F. Sarro, A. A. Al-Subaihin, M. Harman, Y. Jia, W. Martin, and Y. Zhang, "Feature lifecycles as they spread, migrate, remain, and die in app stores," in *23rd IEEE International Requirements Engineering Conference, RE 2015*, pp. 76–85, 2015.

[67] A. A. Al-Subaihin, F. Sarro, S. Black, L. Capra, M. Harman, Y. Jia, and Y. Zhang, "Clustering mobile apps based on mined textual features," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2016, Ciudad Real, Spain, September 8-9, 2016*, pp. 38:1–38:10, 2016.

[68] W. Martin, F. Sarro, and M. Harman, "Causal impact analysis applied to app releases in google play and windows phone store," tech. rep., University College London, 2015.

[69] W. Martin, F. Sarro, and M. Harman, "Causal impact analysis for app releases in google play," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, (New York, NY, USA), pp. 435–446, ACM, 2016.

[70] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, (New York, New York, USA), pp. 1025–1035, ACM Press, May 2014.

[71] K. Kuznetsov, V. Avdiienko, A. Gorla, and A. Zeller, "Checking app user interfaces against app descriptions," in *Proceedings of the International Workshop on App Market Analytics - WAMA 2016*, (New York, New York, USA), pp. 1–7, ACM Press, 2016.

[72] W. Maalej and H. Nabil, "Bug report, feature request, or simply praise? On automatically classifying app reviews," in *2015 IEEE 23rd International Requirements Engineering Conference (RE)*, pp. 116–125, IEEE, aug 2015.

[73] A. Di Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. 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 - FSE 2016*, (New York, New York, USA), pp. 499–510, ACM Press, 2016.

[74] A. Di Sorbo, S. Panichella, C. V. Alexandru, C. A. Visaggio, and G. Canfora, "SURF: Summarizer of User Reviews Feedback," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 55–58, IEEE, may 2017.

[75] H. Khalid, E. Shihab, M. Nagappan, and A. Hassan, "What Do Mobile App Users Complain About? A Study on Free iOS Apps," *IEEE Software*, vol. PP, no. 99, pp. 1–1, 2014.

[76] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall, "How can i improve my app? Classifying user reviews for software maintenance and evolution," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 281–290, IEEE, sep 2015.

[77] E. Guzman and W. Maalej, "How Do Users Like This Feature? A Fine Grained Sentiment Analysis of App Reviews," in *2014 IEEE 22nd International Requirements Engineering Conference (RE)*, pp. 153–162, IEEE, Aug. 2014.

[78] E. Bakiu and E. Guzman, "Which Feature is Unusable? Detecting Usability and User Experience Issues from User Reviews," in *2017 IEEE 25th International Requirements Engineering Conference Workshops (REW)*, pp. 182–187, IEEE, sep 2017.

[79] J. Gui, M. Nagappan, and W. G. J. Halfond, "What Aspects of Mobile Ads Do Users Care About? An Empirical Study of Mobile In-app Ad Reviews," tech. rep., University of Southern California, 2017.

[80] E. C. Groen, S. Kopczynska, M. P. Hauer, T. D. Krafft, and J. Doerr, "Users âĂŤ The Hidden Software Product Quality Experts?: A Study on How App Users Report Quality Aspects in Online Reviews," in *2017 IEEE 25th International Requirements Engineering Conference (RE)*, pp. 80–89, IEEE, sep 2017.

[81] T. Johann, C. Stanik, A. M. A. B., and W. Maalej, "SAFE: A Simple Approach for Feature Extraction from App Descriptions and App Reviews," in *2017 IEEE 25th International Requirements Engineering Conference (RE)*, pp. 21–30, IEEE, Sep 2017.

[82] C. Iacob and R. Harrison, "Retrieving and analyzing mobile apps feature requests from online reviews," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, pp. 41–44, IEEE, May 2013.

[83] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall, "ARdoc: app reviews development oriented classifier," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, (New York, New York, USA), pp. 1023–1027, ACM Press, 2016.

[84] X. Gu and S. Kim, ""What Parts of Your Apps are Loved by Users?" (T)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 760–770, IEEE, nov 2015.

[85] S. Scalabrino, G. Bavota, B. Russo, R. Oliveto, and M. Di Penta, "Listening to the Crowd for the Release Planning of Mobile Apps," *IEEE Transactions on Software Engineering*, pp. 1–1, 2017.

[86] S. Ugurel, R. Krovetz, and C. L. Giles, "What's the code?: automatic classification of source code archives," in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '02*, (New York, New York, USA), p. 632, ACM Press, jul 2002.

[87] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "MUDABlue: An automatic categorization system for Open Source repositories," *Journal of Systems and Software*, vol. 79, pp. 939–953, July 2006.

[88] S. Kawaguchi, P. Garg, M. Matsushita, K. Inoue, and Z. Source, "Automatic categorization algorithm for evolvable software archive," in *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings.*, pp. 195–200, IEEE, 2003.

[89] K. Tian, M. Revelle, and D. Poshyvanyk, "Using Latent Dirichlet Allocation for automatic categorization of software," in *2009 6th IEEE International Working Conference on Mining Software Repositories*, pp. 163–166, IEEE, May 2009.

[90] J. Härtel, H. Aksu, and R. Lämmel, "Classification of APIs by hierarchical clustering," in *Proceedings of the 26th Conference on Program Comprehension - ICPC '18*, (New York, New York, USA), pp. 233–243, ACM Press, 2018.

[91] D. Altarawy, H. Shahin, A. Mohammed, and N. Meng, "Lascad : Language-agnostic software categorization and similar application detection," *Journal of Systems and Software*, vol. 142, pp. 21–34, aug 2018.

[92] C. McMillan, M. Linares-Vasquez, D. Poshyvanyk, and M. Grechanik, "Categorizing software applications for maintenance," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pp. 343–352, IEEE, sep 2011.

[93] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 364–374, IEEE, jun 2012.

[94] M. Linares-Vásquez, C. McMillan, D. Poshyvanyk, and M. Grechanik, "On using machine learning to automatically classify software applications into domain categories," *Empirical Software Engineering*, vol. 19, pp. 582–618, Oct. 2012.

[95] T. Wang, H. Wang, G. Yin, C. X. Ling, X. Li, and P. Zou, "Mining Software Profile across Multiple Repositories for Hierarchical Categorization," in *2013 IEEE International Conference on Software Maintenance*, pp. 240–249, IEEE, Sept. 2013.

[96] J. Escobar-Avila, M. Linares-Vásquez, and S. Haiduc, "Unsupervised software categorization using bytecode," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pp. 229–239, IEEE Press, may 2015.

[97] G. Berardi, A. Esuli, T. Fagni, and F. Sebastiani, "Multi-store metadata-based supervised mobile app classification," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing - SAC '15*, (New York, New York, USA), pp. 585–588, ACM Press, apr 2015.

[98] N. Chen, S. C. Hoi, S. Li, and X. Xiao, "SimApp: A Framework for Detecting Similar Mobile Applications by Online Kernel Learning," in *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining - WSDM '15*, (New York, New York, USA), pp. 305–314, ACM Press, feb 2015.

[99] S. Vakulenko, O. Müller, and J. Brocke, "Enriching iTunes App Store Categories via Topic Modeling," in *Proceedings of the Thirty Fifth International Conference on Information Systems (ICIS)*, (Auckland, New Zealand), 2014.

[100] D. Surian, S. Seneviratne, A. Seneviratne, and S. Chawla, "App Miscategorization Detection: A Case Study on Google Play," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, pp. 1591–1604, aug 2017.

[101] J. Kim, Y. Park, C. Kim, and H. Lee, "Mobile application service networks: Apple's App Store," *Service Business*, vol. 8, pp. 1–27, feb 2013.

[102] H. Zhu, E. Chen, H. Xiong, H. Cao, and J. Tian, "Mobile App Classification with Enriched Contextual Information," *IEEE Transactions on Mobile Computing*, vol. 13, pp. 1550–1563, jul 2014.

[103] D. Lavid Ben Lulu and T. Kuflik, "Functionality-based clustering using short textual description," in *Proceedings of the 2013 international conference on Intelligent user interfaces - IUI '13*, (New York, New York, USA), p. 297, ACM Press, 2013.

[104] S. Mokarizadeh, M. T. Rahman, and M. Matskin, "Mining and Analysis of Apps in Google Play," in *9th International Conference onWeb Information Systems and Technologies, WEBIST '13*, 2013.

[105] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, pp. 221–233, jun 2014.

[106] M. Linares-Vásquez, A. Holtzhauer, and D. Poshyvanyk, "On Automatically Detecting Similar Android Apps," in *24th IEEE International Conference on Program Comprehension*, IEEE Comput. Soc, 2016.

[107] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher, "Recommending source code for use in rapid software prototypes," *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, pp. 848–858, jun 2012.

[108] M. Nayebi, H. Farrahi, A. Lee, H. Cho, and G. Ruhe, "More insight from being more focused: analysis of clustered market apps," in *Proceedings of the International Workshop on App Market Analytics - WAMA 2016*, (New York, New York, USA), pp. 30–36, ACM Press, 2016.

[109] A. Shabtai, Y. Fledel, and Y. Elovici, "Automated Static Code Analysis for Classifying Android Applications Using Machine Learning," in *2010 International Conference on Computational Intelligence and Security*, pp. 329–333, IEEE, Dec. 2010.

[110] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, and P. G. Bringas, "On the automatic categorisation of android applications," in *2012 IEEE Consumer Communications and Networking Conference (CCNC)*, pp. 149–153, IEEE, Jan. 2012.

[111] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "Whyper: Towards automating risk assessment of mobile applications," in *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, (Berkeley, CA, USA), pp. 527–542, USENIX Association, 2013.

[112] S. Seneviratne, A. Seneviratne, M. A. Kaafar, A. Mahanti, and P. Mohapatra, "Early detection of spam mobile apps," in *Proceedings of the 24th International Conference on World Wide Web (WWW15)*, WWW '15, pp. 949–959, 2015.

[113] D. E. Krutz, N. Munaiah, A. Meneely, and S. A. Malachowsky, "Examining the relationship between security metrics and user ratings of mobile apps: a case study," in *Proceedings of the International Workshop on App Market Analytics - WAMA 2016*, (New York, New York, USA), pp. 8–14, ACM Press, 2016.

[114] P. Teufl, M. Ferk, A. Fitzek, D. Hein, S. Kraxberger, and C. Orthacker, "Malware detection by applying knowledge discovery processes to application metadata on the Android Market (Google Play)," *Security and Communication Networks*, vol. 9, pp. 389–419, mar 2016.

[115] N. Chen, J. Lin, S. C. H. Hoi, X. Xiao, and B. Zhang, "AR-miner: mining informative reviews for developers from mobile app marketplace," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, (New York, New York, USA), pp. 767–778, ACM Press, May 2014.

[116] D. Pagano and W. Maalej, "User feedback in the appstore: An empirical study," in *2013 21st IEEE International Requirements Engineering Conference (RE)*, pp. 125–134, IEEE, July 2013.

[117] H. Khalid, "On identifying user complaints of ios apps," in *Software Engineering (ICSE), 2013 35th International Conference on*, pp. 1474–1476, May 2013.

[118] S. McIlroy, N. Ali, H. Khalid, and A. E. Hassan, "Analyzing and automatically labelling the types of user issues that are raised in mobile app reviews," *Empirical Software Engineering*, pp. 1–40, 2015.

[119] F. Palomba, M. Linares-Vasquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "User reviews matter! Tracking crowdsourced reviews to support evolution of

successful apps," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 291–300, IEEE, sep 2015.

[120] F. A. Shah, Y. Sabanin, and D. Pfahl, "Feature-based evaluation of competing apps," in *Proceedings of the International Workshop on App Market Analytics - WAMA 2016*, (New York, New York, USA), pp. 15–21, ACM Press, 2016.

[121] L. Villarroel, G. Bavota, B. Russo, R. Oliveto, and M. Di Penta, "Release planning of mobile apps based on user reviews," in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*, (New York, New York, USA), pp. 14–24, ACM Press, 2016.

[122] M. Lu and P. Liang, "Automatic Classification of Non-Functional Requirements from Augmented App User Reviews," in *Proceedings of the 21st conference on Evaluation and Assessment in Software Engineering, EASE'17*, (Karlskrona, Sweden.), 2017.

[123] C. Gao, J. Zeng, M. R. Lyu, and I. King, "Online app review analysis for identifying emerging issues," *Proceedings of the 40th International Conference on Software Engineering*, pp. 48–58, 2018.

[124] H. Khalid, M. Nagappan, E. Shihab, and A. E. Hassan, "Prioritizing the devices to test your app on: A case study of android game apps," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, (New York, NY, USA), pp. 610–620, ACM, 2014.

[125] P. M. Vu, T. T. Nguyen, H. V. Pham, and T. T. Nguyen, "Mining User Opinions in Mobile App Reviews: A Keyword-Based Approach (T)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 749–759, IEEE, Nov 2015.

[126] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. 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*, ESEC/FSE 2013, (New York, NY, USA), pp. 477–487, ACM, 2013.

[127] H. Khalid, M. Nagappan, and A. Hassan, "Examining the Relationship between FindBugs Warnings and End User Ratings: A Case Study On 10,000 Android Apps," *IEEE Software*, vol. PP, no. 99, pp. 1–1, 2015.

[128] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, "A Survey of App Store Analysis for Software Engineering," *IEEE Transactions on Software Engineering*, pp. 1–1, 2016.

[129] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia, "Understanding Android Fragmentation with Topic Analysis of Vendor-Specific Bugs," in *2012 19th Working Conference on Reverse Engineering*, pp. 83–92, IEEE, oct 2012.

[130] G. L. Scoccia, S. Ruberto, I. Malavolta, M. Autili, and P. Inverardi, "An investigation into Android run-time permissions from the end users' perspective," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems - MOBILESoft '18*, (New York, New York, USA), pp. 45–55, ACM Press, 2018.

[131] L. Pascarella, F.-X. Geiger, F. Palomba, D. Di Nucci, I. Malavolta, and A. Bacchelli, "Self-reported activities of Android developers," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems - MOBILESoft '18*, (New York, New York, USA), pp. 144–155, ACM Press, 2018.

[132] G. G. Parker and M. W. Van Alstyne, "Two-Sided Network Effects: A Theory of Information Product Design," *Management Science*, vol. 51, pp. 1494–1504, Oct 2005.

[133] A. Holzer and J. Ondrus, "Mobile application market: A developerâĂŹs perspective," *Telematics and Informatics*, vol. 28, pp. 22–31, Feb. 2011.

[134] "Engineers are becoming a lot like marketers too." `http://chiefmartec.com/2012/05/engineers-are-becoming-a-lot-like-marketers-too/`. Accessed: 2017-06-09.

[135] M. E. Joorabchi, A. Mesbah, and P. Kruchten, "Real Challenges in Mobile App Development," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 15–24, IEEE, Oct. 2013.

[136] R. Francese, C. Gravino, M. Risi, G. Scanniello, and G. Tortora, "Mobile app development and management: results from a qualitative investigation," *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, pp. 133–143, 2017.

[137] M. Nayebi, B. Adams, and G. Ruhe, "Release Practices for Mobile Apps – What do Users and Developers Think?," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 552–562, IEEE, mar 2016.

[138] C. Rosen and E. Shihab, "What are mobile developers asking about? A large scale study using stack overflow," *Empirical Software Engineering*, vol. 21, pp. 1192–1223, jun 2016.

[139] R. E. Boyatzis, *Transforming qualitative information : thematic analysis and code development*. Sage Publications, 1998.

[140] K. Roulston, "Data analysis and 'theorizing as ideology'," *Qualitative Research*, vol. 1, pp. 279–302, dec 2001.

[141] B. E. Whitley, M. E. Kite, and H. L. Adams, *Principles of research in behavioral science*. Routledge, 3 ed., 2012.

[142] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative research in psychology*, vol. 3, no. 2, pp. 77–101, 2006.

[143] M. D. Syer, M. Nagappan, A. E. Hassan, and B. Adams, "Revisiting prior empirical findings for mobile apps: an empirical case study on the 15 most popular open-source Android apps," 2013.

[144] A. Miniukovich and A. De Angeli, "Computation of Interface Aesthetics," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems - CHI '15*, (New York, New York, USA), pp. 1163–1172, ACM Press, 2015.

[145] T.-H. Chang, T. Yeh, and R. Miller, "Associating the visual representation of user interfaces with their internal structures and metadata," in *Proceedings of the 24th annual ACM symposium on User interface software and technology - UIST '11*, (New York, New York, USA), p. 245, ACM Press, 2011.

[146] M. Harman, A. Al-Subaihin, Y. Jia, W. Martin, F. Sarro, and Y. Zhang, "Mobile app and app store analysis, testing and optimisation," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, (New York, NY, USA), pp. 243–244, ACM, 2016.

[147] "Number of Smartphone Users Worldwide 2014-2020." `https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide`. Accessed: 2017-06-09.

[148] "Annual Number of Mobile App Downloads Worldwide 2021." `https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads`. Accessed: 2017-06-09.

[149] A. I. Wasserman, "Software engineering issues for mobile application development," in *Proceedings of the FSE/SDP workshop on Future of software engineering research - FoSER '10*, (New York, New York, USA), p. 397, ACM Press, nov 2010.

[150] R. Minelli and M. Lanza, "Software Analytics for Mobile Applications–Insights & Lessons Learned," in *2013 17th European Conference on Software Maintenance and Reengineering*, pp. 144–153, IEEE, mar 2013.

[151] M. D. Syer, B. Adams, Y. Zou, and A. E. Hassan, "Exploring the Development of Micro-apps: A Case Study on the BlackBerry and Android Platforms," in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, pp. 55–64, IEEE, sep 2011.

[152] I. J. M. Ruiz, M. Nagappan, B. Adams, and A. E. Hassan, "Understanding reuse in the Android Market," in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pp. 113–122, IEEE, jun 2012.

[153] I. J. M. Ruiz, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A. E. Hassan, "A Large-Scale Empirical Study on Software Reuse in Mobile Apps," *IEEE Software*, vol. 31, pp. 78–86, mar 2014.

[154] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the Test Automation Culture of App Developers," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 1–10, IEEE, apr 2015.

[155] M. Linares-Vasquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk, "How do Developers Test Android Applications?," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 613–622, IEEE, sep 2017.

[156] G. Sethumadhavan, "Sizing Android mobile applications," in *6th IFPUG International Software Measurement and Analysis Conference (ISMA)*, 2011.

[157] T. Preuss, "Mobile Applications, Functional Analysis, and the Customer Experience," in *The IFPUG Guide to IT and Software Measurement* (IFPUG, ed.), pp. 408–433, Auerbach Publications, 2012.

[158] H. van Heeringen and E. Van Gorp, "Measure the Functional Size of a Mobile App: Using the COSMIC Functional Size Measurement Method," in *Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-MENSURA), 2014 Joint Conference of the International Workshop on*, pp. 11–16, IEEE, 2014.

[159] F. Ferrucci, C. Gravino, P. Salza, and F. Sarro, "Investigating functional and code size measures for mobile applications: A replicated study," in *Proceedings of the 16th International Conference on Product-Focused Software Process Improvement, PROFES 2015*, pp. 271–287, 2015.

[160] F. Ferrucci, C. Gravino, P. Salza, and F. Sarro, "Investigating functional and code size measures for mobile applications," in *41st Euromicro Conference on Software Engineering and Advanced Applications, EUROMICRO-SEAA 2015, Madeira, Portugal, August 26-28, 2015*, pp. 365–368, 2015.

[161] G. Catolino, P. Salza, C. Gravino, and F. Ferrucci, "A set of metrics for the effort estimation of mobile apps," *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, pp. 194–198, 2017.

[162] H. K. Flora, X. Wang, and S. V.Chande, "An Investigation into Mobile Application Development Processes: Challenges and Best Practices," *International Journal of Modern Education and Computer Science*, vol. 6, pp. 1–9, jun 2014.

[163] M. Linares-Vasquez, C. Vendome, Q. Luo, and D. Poshyvanyk, "How developers detect and fix performance bottlenecks in Android apps," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 352–361, IEEE, sep 2015.

[164] P. Salza, F. Palomba, D. D. Nucci, C. D'uva, A. De Lucia, and F. Ferrucci, "Do Developers Update Third-Party Libraries in Mobile Apps," in *Proceedings of 26th International Conference on Program Comprehension (ICPC 2018)*, (New York, New York, USA), ACM, 2018.

[165] M. Nayebi, H. Farahi, and G. Ruhe, "Which Version Should Be Released to App Store?," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 324–333, IEEE, nov 2017.

[166] I. Benbasat, D. K. Goldstein, and M. Mead, "The Case Research Strategy in Studies of Information Systems," *MIS Quarterly*, vol. 11, p. 369, sep 1987.

[167] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, pp. 131–164, Dec. 2008.

[168] C. Robson and K. McCartan, *Real world research : a resource for users of social research methods in applied settings*. John Wiley & Sons, 4 ed., 2015.

[169] B. A. Kitchenham and S. L. Pfleeger, "Principles of survey research part 2," *ACM SIGSOFT Software Engineering Notes*, vol. 27, pp. 18–20, jan 2002.

[170] B. Glaser and A. L. Strauss, *Discovery of Grounded Theory Strategies for Qualitative Research*. Taylor and Francis, 1967.

[171] C. Wohlin and A. Aurum, "Towards a decision-making structure for selecting a research design in empirical software engineering," *Empirical Software Engineering*, vol. 20, pp. 1427–1455, dec 2015.

[172] D. S. Cruzes and T. Dyba, "Recommended Steps for Thematic Synthesis in Software Engineering," in *2011 International Symposium on Empirical Software Engineering and Measurement*, pp. 275–284, IEEE, sep 2011.

[173] I. Bohnet, H. Harmgart, S. H. (Ucl), and J.-R. Tyran, "Learning Trust," *Journal of the European Economic Association*, vol. 3, pp. 322–329, may 2005.

[174] A. A. Al-Subaihin, A. Finkelstein, M. Harman, Y. Jia, W. Martin, F. Sarro, and Y. Zhang, "App store mining and analysis," in *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile, DeMobile 2015*, pp. 1–2, 2015.

[175] A. Finkelstein, M. Harman, Y. Jia, W. Martin, F. Sarro, and Y. Zhang, "Investigating the relationship between price, rating, and popularity in the blackberry world app store," *Information and Software Technology*, vol. 87, no. Supplement C, pp. 119 – 139, 2017.

[176] A. Finkelstein, M. Harman, Y. Jia, W. Martin, F. Sarro, and Y. Zhang, "App store analysis: Mining app stores for relationships between customer, business and technical characteristics," *UCL - Research Note RN/14/10*, September 2014.

[177] A. A. Al-Subaihin, F. Sarro, S. Black, L. Capra, M. Harman, Y. Jia, and Y. Zhang, "Clustering mobile apps based on mined textual features," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '16, (New York, NY, USA), pp. 38:1–38:10, ACM, 2016.

[178] Y. Liu, L. Liu, H. Liu, X. Wang, and H. Yang, "Mining domain knowledge from app descriptions," *Journal of Systems and Software*, vol. 133, pp. 126–144, nov 2017.

[179] F. Sarro, M. Harman, Y. Jia, and Y. Zhang, "Customer rating reactions can be predicted purely using app features," in *Proceedings of the 26th IEEE International Requirements Engineering Conference*, RE'18, p. to appear, IEEE, 2018.

[180] G. Grano, A. Ciurumelea, S. Panichella, F. Palomba, and H. C. Gall, "Exploring the integration of user feedback in automated testing of Android applications," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 72–83, IEEE, mar 2018.

[181] F. Sarro, "Predictive analytics for software testing," in *Proceedings of the 11th International Workshop on Search-Based Software Testing - SBST '18*, (New York, New York, USA), pp. 1–1, ACM Press, 2018.

[182] F. Palomba, P. Salza, A. Ciurumelea, S. Panichella, H. Gall, F. Ferrucci, and A. De Lucia, "Recommending and Localizing Change Requests for Mobile Apps Based on User Reviews," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 106–117, IEEE, May 2017.

[183] N. Jha and A. Mahmoud, "Using frame semantics for classifying and summarizing application store reviews," *Empirical Software Engineering*, pp. 1–34, mar 2018.

[184] S. Shen, X. Lu, Z. Hu, and X. Liu, "Towards Release Strategy Optimization for Apps in Google Play," in *Proceedings of the 9th Asia-Pacific Symposium on Internetware - Internetware'17*, (New York, New York, USA), pp. 1–10, ACM Press, 2017.

[185] H. Khalid, M. Nagappan, and A. Hassan, "Examining the Relationship between FindBugs Warnings and End User Ratings: A Case Study On 10,000 Android Apps," *IEEE Software*, vol. PP, no. 99, pp. 1–1, 2015.

[186] W. Martin, F. Sarro, and M. Harman, "Causal Impact Analysis Applied to App Releases in Google Play and Windows Phone Store," tech. rep., University College London, Research Note, RN/15/07, 2015.

[187] M. Nayebi, H. Cho, H. Farrahi, and G. Ruhe, "App store mining is not enough," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 152–154, May 2017.

[188] E. Shaw, A. Shaw, and D. Umphress, "Mining Android Apps to Predict Market Ratings," in *Proceedings of the 6th International Conference on Mobile Computing, Applications and Services*, ICST, 2014.

[189] L. Guerrouj, S. Azad, and P. C. Rigby, "The influence of App churn on App success and StackOverflow discussions," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 321–330, IEEE, mar 2015.

[190] G. Bavota, M. Linares-Vasquez, C. E. Bernal-Cardenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "The Impact of API Change- and Fault-Proneness on the User Ratings of Android Apps," *IEEE Transactions on Software Engineering*, vol. 41, pp. 384–407, apr 2015.

[191] H. Hu, C.-P. Bezemer, and A. E. Hassan, "Studying the consistency of star ratings and the complaints in 1 and 2-star user reviews for top free cross-platform Android and iOS apps," *Empirical Software Engineering*, pp. 1–34, mar 2018.

[192] H. Hu, S. Wang, C.-P. Bezemer, and A. E. Hassan, "Studying the consistency of star ratings and reviews of popular free hybrid Android and iOS apps," *Empirical Software Engineering*, pp. 1–26, apr 2018.

[193] E. Noei, M. D. Syer, Y. Zou, A. E. Hassan, and I. Keivanloo, "A study of the relation of mobile device attributes with the user-perceived quality of Android apps," *Empirical Software Engineering*, vol. 22, pp. 3088–3116, dec 2017.

[194] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan, "What are the characteristics of high-rated apps? A case study on free Android Applications," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 301–310, IEEE, sep 2015.

[195] F. J. Gravetter and L.-A. B. Forzano, *Research methods for the behavioral sciences*. Wadsworth Cengage Learning, 2012.

[196] E. Murphy-Hill, T. Zimmermann, and N. Nagappan, "Cowboys, ankle sprains, and keepers of quality: how is video game development different from software development?," in *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, (New York, New York, USA), pp. 1–11, ACM Press, 2014.

[197] I. Manotas, C. Bird, L. Pollock, and J. Clause, "An empirical study of practitioners' perspectives on green software engineering," Tech. Rep. 2014/003, University of Delaware, 2014.

[198] B. A. Kitchenham and S. L. Pfleeger, "Personal Opinion Surveys," in *Guide to Advanced Empirical Software Engineering*, pp. 63–92, London: Springer London, 2008.

[199] M. Nayebi, K. Kuznetsov, P. Chen, A. Zeller, and G. Ruhe, "Anatomy of Functionality Deletion," in *Proceedings of the Conference on Mining Software Repositories (MSR'18)*, (Gothenburg, Sweden), 2018.

[200] E. Guzman, M. El-Haliby, and B. Bruegge, "Ensemble Methods for App Review Classification: An Approach for Software Evolution (N)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 771–776, IEEE, Nov 2015.

[201] S. Mcilroy, W. Shang, N. Ali, and A. E. Hassan, "User reviews of top mobile apps in Apple and Google app stores," *Communications of the ACM*, vol. 60, pp. 62–67, Oct 2017.

[202] S. McIlroy, W. Shang, N. Ali, and A. Hassan, "Is It Worth Responding to Reviews? A Case Study of the Top Free Apps in the Google Play Store," *IEEE Software*, vol. PP, no. 99, pp. 1–1, 2015.

[203] G. Lee and T. S. Raghu, "Determinants of mobile apps' success: Evidence from the app store market," *Journal of Management Information Systems*, vol. 31, no. 2, pp. 133–170, 2014.

[204] I. J. Mojica Ruiz, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. E. Hassan, "Examining the Rating System Used in Mobile-App Stores," *IEEE Software*, vol. 33, pp. 86–92, nov 2016.

[205] B. Fu, J. Lin, L. Li, C. Faloutsos, J. Hong, and N. Sadeh, "Why People Hate Your App âĂŤ Making Sense of User Feedback in a Mobile App Store," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '13*, (New York, New York, USA), p. 1276, ACM Press, 2013.

[206] E. Guzman, O. Aly, and B. Bruegge, "Retrieving Diverse Opinions from App Reviews," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–10, IEEE, Oct 2015.

[207] G. Grano, A. Di Sorbo, F. Mercaldo, C. A. Visaggio, G. Canfora, and S. Panichella, "Android apps and user feedback: a dataset for software evolution and quality improvement," in *Proceedings of the 2nd ACM SIGSOFT International Workshop on App Market Analytics - WAMA 2017*, (New York, New York, USA), pp. 8–11, ACM Press, 2017.

[208] N. Genc-Nayebi and A. Abran, "A systematic literature review: Opinion mining studies from mobile app store user reviews," *Journal of Systems and Software*, vol. 125, pp. 207–219, Mar 2017.

[209] Z. Xie and S. Zhu, "AppWatcher : unveiling the underground market of trading mobile app reviews," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks - WiSec '15*, (New York, New York, USA), pp. 1–11, ACM Press, 2015.

[210] S. Li, J. Caverlee, W. Niu, and P. Kaghazgaran, "Crowdsourced App Review Manipulation," in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval - SIGIR '17*, (New York, New York, USA), pp. 1137–1140, ACM Press, 2017.

[211] X. Chen, Q. Zou, B. Fan, Z. Zheng, and X. Luo, "Recommending software features for mobile applications based on user interface comparison," *Requirements Engineering*, pp. 1–15, jul 2018.

[212] B. Adams and S. McIntosh, "Modern Release Engineering in a Nutshell – Why Researchers Should Care," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 78–90, IEEE, Mar 2016.

[213] E. Ries, *The lean startup : how today's entrepreneurs use continuous innovation to create radically successful businesses*. Crown Publishing Group, 2011.

[214] D. Rowinski, "Another Reason Why App Discovery Is Completely Broken." `http://arc.applause.com`.

[215] K. Sangaralingam, N. Pervin, N. Ramasubbu, A. Datta, and K. Dutta, "Takeoff and Sustained Success of Apps in Hypercompetitive Mobile Platform Ecosystems: An Empirical Analysis," in *ICIS'12*, pp. 1850–1867, 2012.

[216] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *Journal of Computational and Applied Mathematics*, vol. 20, pp. 53–65, nov 1987.

[217] A. Finkelstein, M. Harman, Y. Jia, F. Sarro, and Y. Zhang, "Mining app stores: Extracting technical, business and customer rating information for analysis and prediction," *RN*, vol. 13, p. 21, 2013.

[218] G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Commun. ACM*, vol. 18, pp. 613–620, Nov. 1975.

[219] G. A. Miller, "Wordnet: A lexical database for english," *Commun. ACM*, vol. 38, pp. 39–41, Nov. 1995.

[220] N. H. Timm, *Applied Multivariate Analysis*. Springer Science & Business Media, 2007.

[221] F. Can and E. A. Ozkarahan, "Concepts and effectiveness of the cover-coefficient-based clustering methodology for text databases," *ACM Trans. Database Syst.*, vol. 15, pp. 483–517, Dec. 1990.

[222] I. S. Dhillon and D. S. Modha, "Concept Decompositions for Large Sparse Text Data Using Clustering," *Machine Learning*, vol. 42, no. 1-2, pp. 143–175, 2001.

[223] S. C. Johnson, "Hierarchical clustering schemes," *Psychometrika*, vol. 32, no. 3, pp. 241–254, 1967.

[224] F. Murtagh and P. Legendre, "Ward's Hierarchical Agglomerative Clustering Method: Which Algorithms Implement Ward's Criterion?," *Journal of Classification*, vol. 31, pp. 274–295, Oct 2014.

[225] J. H. Ward, "Hierarchical Grouping to Optimize an Objective Function," *Journal of the American Statistical Association*, vol. 58, pp. 236–244, Mar 1963.

[226] C. E. Spearman, "The proof and measurement of association between two things," *The American Journal of Psychology*, vol. 15, pp. 72–101, January 1904.

[227] M. J. Shepperd, *Foundations of software measurement*. Prentice Hall, 1995.

[228] J. J. Bartko, "The intraclass correlation coefficient as a measure of reliability.," *Psychological reports*, vol. 19, no. 1, pp. 3–11, 1966.

[229] J. Cohen, "Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit.," *Psychological Bulletin*, vol. 70, no. 4, pp. 213–220, 1968.

[230] J. L. Fleiss, "Measuring nominal scale agreement among many raters.," *Psychological Bulletin*, vol. 76, no. 5, pp. 378–382, 1971.

[231] W. Albert and T. Tullis, *Measuring the User Experience: Collecting, Analyzing, and Presenting Usability Metrics*. Newnes, 2013.

[232] C. E. Osgood, "The nature and measurement of meaning.," *Psychological bulletin*, vol. 49, pp. 197–237, May 1952.

[233] W. Martin, M. Harman, Y. Jia, F. Sarro, and Y. Zhang, "The app sampling problem for app store mining," in *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR)*, pp. 123–133, 2013.

[234] J. E. Trost, "Statistically nonrepresentative stratified sampling: A sampling technique for qualitative studies," *Qualitative Sociology*, vol. 9, no. 1, pp. 54–57, 1986.

[235] A. Massey, J. Eisenstein, A. Anton, and P. Swire, "Automated text mining for requirements analysis of policy documents," in *IEEE International Requirements Engineering Conference*, pp. 4–13, 2013.

[236] A. Sutcliffe and P. Sawyer, "Requirements elicitation: Towards the unknown unknowns," in *IEEE International Requirements Engineering Conference*, pp. 92–104, 2013.

[237] M. Nagappan and E. Shihab, "Future Trends in Software Engineering Research for Mobile Apps," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 21–32, IEEE, mar 2016.

[238] V. Avdiienko, K. Kuznetsov, I. Rommelfanger, A. Rau, A. Gorla, and A. Zeller, "Detecting Behavior Anomalies in Graphical User Interfaces," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 201–203, IEEE, may 2017.

[239] J. Crussell, C. Gibler, and H. Chen, "AnDarwin: Scalable Detection of Semantically Similar Android Applications," in *18th European Symposium on Research in Computer Security* (J. Crampton, S. Jajodia, and K. Mayes, eds.), (Egham, UK), pp. 182—-199, Springer Berlin Heidelberg, 2013.

[240] V. Arnaoudova, S. Haiduc, A. Marcus, and G. Antoniol, "The use of text retrieval and natural language processing in software engineering," 2015.

[241] H. M. Wallach and H. M., "Topic modeling: beyond bag-of-words," in *Proceedings of the 23rd international conference on Machine learning - ICML '06*, (New York, New York, USA), pp. 977–984, ACM Press, 2006.

[242] H. P. Luhn, "A Statistical Approach to Mechanized Encoding and Searching of Literary Information," *IBM Journal of Research and Development*, vol. 1, pp. 309–317, oct 1957.

[243] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information Processing & Management*, vol. 24, pp. 513–523, jan 1988.

[244] K. S. Jones, "A STATISTICAL INTERPRETATION OF TERM SPECIFICITY AND ITS APPLICATION IN RETRIEVAL," *Journal of Documentation*, vol. 28, no. 1, pp. 11–21, 1972.

[245] S. Robertson, "Understanding inverse document frequency: on theoretical arguments for IDF," *Journal of Documentation*, vol. 60, pp. 503–520, oct 2004.

[246] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, pp. 391–407, sep 1990.

[247] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet Allocation," *Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

[248] T. L. Griffiths and M. Steyvers, "Finding scientific topics.," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 101 Suppl 1, pp. 5228–35, apr 2004.

[249] X.-H. Phan, L.-M. Nguyen, and S. Horiguchi, "Learning to classify short and sparse text & web with hidden topics from large-scale data collections," in *Proceeding of the 17th international conference on World Wide Web - WWW '08*, (New York, New York, USA), p. 91, ACM Press, 2008.

[250] F. Jelinek, R. L. Mercer, L. R. Bahl, and J. K. Baker, "PerplexityâĂŤa measure of the difficulty of speech recognition tasks," *The Journal of the Acoustical Society of America*, vol. 62, pp. S63–S63, dec 1977.

[251] M. Nayebi, H. Cho, and G. Ruhe, "App store mining is not enough for app improvement," *Empirical Software Engineering*, pp. 1–31, feb 2018.

[252] T.-H. Chen, S. W. Thomas, and A. E. Hassan, "A survey on the use of topic models when mining software repositories," *Empirical Software Engineering*, vol. 21, pp. 1843–1919, oct 2016.

[253] "About WordNet." `http://wordnet.princeton.edu/`. Accessed: 2016-01-29.

[254] A. Finkelstein, M. Harman, Y. Jia, W. Martin, F. Sarro, and Y. Zhang, "App store analysis: Mining app stores for relationships between customer, business and technical characteristics," *UCL - Research Note RN/14/10*, September 2014.

[255] M.-C. de Marneffe and C. D. Manning, "The Stanford typed dependencies representation," in *Coling 2008: Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation*, (Manchester, UK), pp. 1–8, Association for Computational Linguistics, 2008.

[256] M.-C. De Marneffe and C. D. Manning, "Stanford Dependencies."

[257] D. Lin, C.-P. Bezemer, Y. Zou, and A. E. Hassan, "An empirical study of game reviews on the Steam platform," *Empirical Software Engineering*, pp. 1–38, jun 2018.

[258] E. R. Babbie, *The practice of social research*, vol. 112. Wadsworth publishing company Belmont, CA, 1998.

[259] F. Sarro, Y. Jia, M. Harman, W. Martin, and Y. Zhang, "Life and death in the app store: Theory and analysis of feature migration," tech. rep., University College London, 2014.

[260] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[261] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proc. of the 33rd International Conference on Software Engineering (ICSE'11)*, pp. 1–10, 2011.

[262] K. Pearson, “Notes on regression and inheritance in the case of two parents,” *Proc. of the Royal Society of London*, vol. 58, pp. 240–242, June 1895.