Recommending New Features from Mobile App Descriptions

HE JIANG, Dalian University of Technology
JINGXUAN ZHANG, Dalian University of Technology
XIAOCHEN LI, Dalian University of Technology
ZHIHEI REN, Dalian University of Technology
DAVID LO, Singapore Management University
XINDONG WU, University of Vermont
ZHONGXUAN LUO, Dalian University of Technology

The rapidly evolving mobile applications (apps) have brought great demand for developers to identify new features by inspecting descriptions of similar apps and identify features that are missing from their apps. Unfortunately, due to the high number of apps, this manual process is time-consuming and unscalable. To help developers identify new features, we propose a new approach named Similar App based Feature Recommender (SAFER). SAFER needs to solve two research challenges: the identification of features from app descriptions (feature identification) and the identification of similar apps (similar app identification). In this study, to address the feature identification challenge, we first develop a new tool to automatically extract features from app descriptions. Then, we leverage a topic model applied to the extracted features and apps’ API invocations to identify similar apps and thus address the similar app identification challenge. Finally, the features of identified similar apps are aggregated and recommended. Experiments validate that SAFER can accurately recommend features to new apps from a collection of more than 8,000 apps; evaluated over a collection of 533 annotated features from 100 apps, SAFER achieves a Hit@15 score of up to 78.68% and outperforms a baseline approach by 17.23% on average.

CCS Concepts: • Software and its engineering → Requirements analysis; • Software and its engineering → Software libraries and repositories

Additional Key Words and Phrases: Mobile Applications, Feature Recommender System, Domain Analysis, Topic Model

ACM Reference Format:

1. INTRODUCTION

Recent years have witnessed the sharp growth of the number of mobile applications (apps). Up to Feb. 2016, two well-known app markets, namely Google Play and Apple
App Store, collect over 2 million apps respectively. In contrast to desktop software, apps update and evolve rapidly [Carreño et al. 2013]. When facing abundant similar apps, users tend to choose ones that provide their features of interest. Hence, it is important for developers to identify and implement new features; these new features can help developers in attracting and retaining users and promoting their apps [Lim et al. 2015]. Our survey with more than 100 app developers (see Section 2) shows that 86.1% of developers consider features that are offered by similar apps. Furthermore, an important way to identify these features is to read descriptions of similar apps in app markets. Unfortunately, due to the large number of apps available in app markets, it is time consuming and labor intensive for developers to manually identify features provided by similar apps [Sarro et al. 2015]. Therefore, it would be ideal if new features of apps can be automatically recommended.

In this study, we propose a new task of Feature Recommendation for Apps (FRA). When a developer implements an app, the new task takes in the initial features of this app as an input and aims to recommend new features of similar apps for the developer. In such a way, the developer can determine which new features should be implemented in the new app. The main challenges of the new task are as follows:

— **Feature Identification:** The descriptions detailing features of apps are usually written in free text containing a variety of information, e.g., brief introductions of apps, disclaimers, and contact addresses. Hence, a specific tool should be developed to identify sentences in the descriptions that describe app features.

— **Similar App Identification:** There exists no explicitly defined product type within app market. For example, over 2 million apps in Google Play are simply classified into 27 categories without sub-categories, except for Game category. Therefore, it is hard to detect closely-related similar apps.

A number of methods have been proposed to identify and recommend features to software systems. Some traditional methods, e.g., Feature Oriented Domain Analysis (FODA) [Kang et al. 1990] and Domain Analysis and Reuse Environment (DARE) [Frakes et al. 1998; Santo et al. 2009], propose methodologies to manually extract features from requirement documentation. Some other methods identify and rank requirements by analyzing a social network of stakeholders and letting stakeholders to introduce new features and rate features proposed by other stakeholders [Lim et al. 2011; Lim et al. 2010]. These methods cannot be used to automatically recommend features for apps. In contrast, a few automatic methods have been proposed to recommend features (e.g., [Alves et al. 2008; Chen et al. 2005; Rahimi et al. 2014;]). Some automatic methods employ data mining and natural language processing to recommend features from either a repository of requirement specifications [Alves et al. 2008; Chen et al. 2005] or forums [Rahimi et al. 2014] or user feedback [Carreño et al. 2013; Iacob et al. 2013]. These automatic methods mainly extract features from either the artifacts or the stakeholders of a software product itself, and cannot recommend features from other software products. These methods are not applicable for newly released apps, or apps under development since there could be few or no users of such apps yet.

Recently, an automatic feature recommendation approach KNN+ is proposed for Softpedia.com, a website collecting features for software products [Hariri et al. 2013]. However, Softpedia.com only covers thousands of apps, a small fraction of apps compared to existing apps in Google Play. Moreover, KNN+ cannot be directly used for common app markets (e.g., Google Play) due to several reasons. First, KNN+
cannot address the feature identification challenge, since features in Softpedia.com are explicitly provided in a bullet-point list format. In contrast, features of apps are implicitly provided in their descriptions mixed with other information, e.g., contact information, etc. Second, all the software products in Softpedia.com are manually organized hierarchically in three layers; more specifically, 9 categories, 292 subcategories, and 1096 product types [Hariri et al. 2013]. Under such a hierarchical structure, a product type contains similar software products. However, in Google play, apps are only coarsely partitioned into 27 categories. Hence, a non-trivial strategy is needed to identify closely-related apps to address the similar app identification challenge. In order for KNN+ to adapt to the new task of FRA, we modify KNN+ in the following ways. First, we use our developed tool (AFE) to extract feature-describing sentences from app descriptions and input them into KNN+. In such a way, KNN+ can recommend the identified features. Second, we introduce the apps in the same category with the new app to find similar ones.

In this study, we present a new approach to recommend new features for apps which can tackle the above-mentioned challenges. Our approach is named Similar App based FFeature Recommender (SAFER). SAFER automatically extracts and fully leverages domain-specific information of apps, including their likely features and API invocations, to identify similar apps belonging to the same product type and recommend features for an app. More specifically, we first develop a tool named App Feature Extractor (AFE) to tackle the feature identification challenge and effectively extract features (i.e., feature-describing sentences) from the descriptions of apps. We then learn a topic model from the extracted features, leveraging the popular Latent Dirichlet Allocation (LDA) approach, to identify similar apps that share similar topics, thus breaking through the similar app identification challenge. To better characterize apps, in this process, we also leverage API invocations of apps to complement the features – past studies have shown that specific features are often correlated to specific API invocations [Bavota et al. 2015; Thung et al. 2013]. Finally, all the features of identified similar apps are aggregated and ranked as the recommended features. In such a way, we hope that developers can receive some hints on what new features to implement to make their apps complete, competitive, and attractive.

To evaluate the effectiveness of SAFER, we collect a total of 8,359 apps coming from five categories of Google Play. Out of them, we have volunteers create an annotated dataset of 100 apps. The descriptions of these selected apps consist of 1,218 sentences, and 533 are identified as a gold set of features. Evaluated on the annotated dataset, the effectiveness of AFE is tested and it can achieve a Recall of 80.27% and Precision of 61.79%. Experimental results over the annotated dataset verify that SAFER can well recommend features to an app from descriptions of similar apps. In 68.29% of cases, SAFER can successfully identify new features when 15 features are recommended (i.e., Hit@15 = 68.29%). On average, SAFER improves the Hit@15 score of a baseline built by adapting KNN+ [Hariri et al. 2013] by 17.23%.

In summary, this study makes the following contributions:

— We collect 8,359 apps, and build a new manually annotated dataset containing features of 100 apps. A total of 533 features (i.e., feature-describing sentences) are identified. We have made this dataset publicly available for academic research¹.

¹ http://oscar-lab.org/FRA/
We build a tool AFE to extract features from the descriptions of apps. Experimental results illustrate that AFE could retain most of the feature-describing sentences, at the same time filter out a majority of non-feature-describing sentences.

We build a tool SAFER to recommend new features to an app. SAFER can effectively recommend new features by identifying similar apps and recommending missing features that are implemented by similar apps.

This paper is structured as follows. In Section 2, we present the usage scenario of SAFER and a survey with its results, which motivate us to proceed with this study. In Section 3, we present the process that we follow to download the 8,359 apps and to annotate the 100 randomly selected apps. In Sections 4 and 5, we elaborate the design of AFE and SAFER. In Sections 6 and 7, we present the experimental design and empirical results respectively. We discuss some interesting aspects of the work and threats to validity in Section 8. Then, we present related work in Section 9. Finally, we conclude this paper in Section 10.

2. MOTIVATION

In this section, we present the motivation of this study by describing the usage scenario of our proposed approach (Section 2.1) and a developer survey (Section 2.2).

2.1 Usage Scenario

When a developer plans to implement an app, typically he/she has already conceived a set of initial features of this app in his/her mind. As shown in Fig. 1, our proposed approach SAFER can take in these initial features to identify similar apps from a repository of apps. To identify similar apps, SAFER extracts feature-describing sentences (features, for short) from the descriptions of apps (see Section 4 for details). In addition, SAFER also extracts API invocations of these apps. With both features and API invocations, SAFER identifies similar apps and recommends a ranked list of new features to the new app from descriptions of these similar apps. Developers can check the recommended features from top to bottom to identify new features to implement. In addition, for every recommended feature, we also present three similar apps which possess this feature. The three apps are ranked by their user ratings in the corresponding app market, e.g., Google Play. In such a way, developers can evaluate the recommended new features by checking related apps.

2.2 Developer Survey

We hypothesize that developers often consider features provided by similar apps when they implement their own apps. To ascertain this hypothesis, we conduct a survey to app developers. The survey only contains two questions so that developers can complete it quickly. These questions investigate how developers identify new features to implement (see Table I). The first question investigates whether developers examine features of apps in the same product type when developing their own apps, and they can choose yes or no. The second question investigates the different ways developers identify new features. We provide some options for developers to choose and they can also provide their own answers.

To conduct the survey, we need to identify a population of app developers to contact. We check the top ranked apps in app markets, namely Apple App Store, Blackberry App World, and Google Play, and visit the webpages of these apps to obtain the detailed information. We totally collect 5,610 distinct contact addresses of...
Recommending New Features from Mobile App Descriptions

 apps, including 665 from Apple App Store, 1,860 from BlackBerry App World, and 3,085 from Google Play. Then, we send an email to each contact address with the survey containing the two questions.

Fig. 1. Application Scenario of SAFER

ACM Transactions on xxxxxxx, Vol. xx, No. xx, Article xx, Publication date: Month YYYY
Table I. Survey on new feature identification methods

<table>
<thead>
<tr>
<th>Question</th>
<th>Option 1</th>
<th>Option 2</th>
<th>Option 3</th>
<th>Option 4</th>
<th>Option 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1: When developing apps, will you examine features of other apps of the same product type?</td>
<td>A. Yes</td>
<td>B. No</td>
<td>C. Websites like Softpedia.com.</td>
<td>D. Similar app descriptions in app repositories, e.g., Google Play.</td>
<td>E. Customers’ feedback/reviews of my own app.</td>
</tr>
<tr>
<td>Q2: How do you find new features to implement?</td>
<td>A. Website like Softpedia.com.</td>
<td>B. Similar app descriptions in app repositories, e.g., Google Play.</td>
<td>C. Customers’ feedback/reviews of my own app.</td>
<td>D. Websites of similar apps.</td>
<td>E. Others: ____________________ [please provide more information].</td>
</tr>
</tbody>
</table>

*: exclusive choice; +: multiple choices

Table II. The results of the survey

<table>
<thead>
<tr>
<th>Question</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>Other answers for Q2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>86.1%</td>
<td>13.9%</td>
<td>-</td>
<td>-</td>
<td>1. Testing competitor apps</td>
</tr>
<tr>
<td>Q2</td>
<td>4.3%</td>
<td>58.3%</td>
<td>46.1%</td>
<td>20.9%</td>
<td>2. Brainstorm 3. Social media</td>
</tr>
</tbody>
</table>

We receive 115 responses and summarize the responses in Table II. There are three potential reasons for the response rate of ~2%. First, some developers may not be willing to share their methods of finding new features since they may view them as confidential. Second, around 15% of emails bounced back. Third, some app email addresses may be maintained by customer service personnel rather than developers. Nonetheless, 115 responses from industry practitioners are still a substantial number similar to many prior studies (e.g. [Zimmermann et al. 2010]).

Our survey results highlight the following findings: First, over 86% of App developers will examine features of apps in the same product type (Q1 in Table II). Hence, features from the same product type are important pieces of information to app developers when developing new apps. Second, the app descriptions are the main sources for developers to find new features. Customer’ reviews of their apps are other important sources for new features. However, since there are no customer reviews available for new apps with only some initial features, in this study, we do not take this source into consideration. Additionally, we find that developers also use other means to identify new features, such as testing similar apps, brainstorming, and reading materials posted on social media.

In conclusion, most developers examine features from apps of the same product type, and this is a common action that developers do to identify new features from similar app descriptions to implement. These findings motivate us to propose a new approach to recommend new features mined from descriptions of similar apps, given an initial set of features of an app that a developer wants to implement.

3. DATASETS

In this section, we first introduce a repository of apps that we used as input to our approach. Next, we present a set of apps, whose features have been manually identified, to test the results of AFE and SAFER.

3.1 Repository of Reference Apps

We create a repository of reference apps which we use as input to our approach. To create this repository, first we select five categories in Google Play, namely Business,
Table III. Characteristics of the Reference App Repository

<table>
<thead>
<tr>
<th>Category</th>
<th># of Apps</th>
<th>Sent. in Descriptions of Apps</th>
<th>Average # of API Invocations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Aver. #</td>
<td>Max. #</td>
</tr>
<tr>
<td>Business</td>
<td>2,118</td>
<td>5.24</td>
<td>124</td>
</tr>
<tr>
<td>Education</td>
<td>1,822</td>
<td>7.18</td>
<td>113</td>
</tr>
<tr>
<td>Health and Fitness</td>
<td>1,545</td>
<td>6.97</td>
<td>65</td>
</tr>
<tr>
<td>Finance</td>
<td>1,796</td>
<td>5.48</td>
<td>66</td>
</tr>
<tr>
<td>Music and Audio</td>
<td>1,078</td>
<td>6.85</td>
<td>182</td>
</tr>
</tbody>
</table>

Education, Health and Fitness, Finance, and Music and Audio. We pick these categories since there are a large number of apps with plentiful distinctive features under these categories. We download a collection of 8,359 apps from Google Play with a third party library tool, namely Google Play Unofficial Python API\(^2\). This tool can not only download APK file of each app, but also obtain its description, user rating, etc. To download apps and their information using the API, a category name should be specified and a random offset should be given to distinguish each run. This tool returns at most 100 apps in each run. We specify each of the five category names one at a time and run this tool 30 times for each category. After removing the duplicate apps downloaded by this tool, we eventually obtain a dataset with 8,359 apps in total belonging to the five different categories. This repository of apps is used as input to our approach to recommend new features to a new app.

Table III presents the characteristics of the apps in this repository, which we refer to as the Reference App Repository. In the table, we include the number of apps in each category, and the average and maximum number of sentences in the descriptions of the apps. In addition, we also present the average number of API invocations that an app makes for each category.

3.2 Annotated Feature Dataset

Since no dataset containing apps with annotated features is available, we have volunteers annotate a collection of apps to evaluate the performance of a feature recommendation system.

We recruit 6 graduate students from School of Software, Dalian University of Technology to annotate the apps. These volunteers all major in computer science and have experience with software development for at least 4 years. Thus, it is not difficult for them to identify features of apps from their descriptions. Before the annotation process, each volunteer is asked to read an annotation guide to explain the annotation procedure, criteria, and an example. After the volunteers get familiar with the whole process, they are requested to pick out the sentences detailing features. To obtain convincing results, we distribute every app description to three different volunteers. When two or three volunteers agree on a sentence detailing a feature, then this sentence is regarded as a golden feature. Finally, we collect the annotation results from the volunteers to form our Annotated Feature Dataset (AFD).

Due to the large number of apps in the downloaded Reference App Repository, we cannot annotate all the apps. Hence, we randomly select 20 apps from every category and employ volunteers to annotate their golden features. The descriptions of these apps contain 1,218 sentences in total, and for each category, more than 200 sentences

---

\(^2\) https://github.com/egirault/googleplay-api.
Table IV. Statistics of the Annotated Golden Features

<table>
<thead>
<tr>
<th>Categories</th>
<th>Golden Features</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ave. #</td>
<td>Max #</td>
<td>Std. Dev.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Business</td>
<td>5.85</td>
<td>10</td>
<td>2.10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Education</td>
<td>6.40</td>
<td>12</td>
<td>2.75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Health and Fitness</td>
<td>5.00</td>
<td>8</td>
<td>1.76</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finance</td>
<td>6.10</td>
<td>11</td>
<td>2.23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Music and Audio</td>
<td>3.30</td>
<td>8</td>
<td>2.00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

need to be annotated by three different volunteers. At the end of the annotation process, we have annotated 533 sentences detailing features of the 100 apps (golden features). As shown in Table IV, the number of golden features varies between apps. For example, the average and the maximum numbers of golden features are 5.85 and 10 respectively for apps in the category Business. In contrast, the corresponding values in the category Music and Audio are 3.3 and 8 respectively. We evaluate Fleiss' Kappa agreement on golden features voted by volunteers and we find that the Kappa coefficient is 0.45 showing moderate agreement. The 100 apps with the annotated golden features are removed from the Reference App Repository and only be used as new apps to test the effectiveness of AFE and SAFER. Our dataset is available for download from: http://oscar-lab.org/FRA/.

4. APP FEATURE EXTRACTOR (AFE)

Automatically extracting feature-describing sentences from app descriptions is not a trivial task. To address this feature identification challenge, we develop a tool named App Feature Extractor (AFE). AFE consists of three components, namely data cleaner, linguistic rule filter, and feature classifier (see Fig. 2). AFE first splits the descriptions of apps into sentences and noisy sentences are removed (data cleaner component). The remaining sentences are then filtered based on some linguistic rules (linguistic rule filter component). Finally, a classifier is employed to discriminate sentences that describe features from those that do not (feature classifier component). More details of these components are presented below:

Data Cleaner. Data cleaner first uses LingPipe\(^3\) to partition the description of an app into sentences. Then, sentences only containing non-letter symbols and punctuation marks (e.g., @, #, $, %, &) are filtered out. Moreover, interrogative sentences ending up with "?" are filtered out since they seldom describe features. Finally, sentences containing either email addresses or website URLs are removed since they also typically do not describe app features but rather contact information.

---

\(^3\) http://alias-i.com/lingpipe.
Table V. Linguistic Rules with Examples

<table>
<thead>
<tr>
<th>#</th>
<th>Linguistic Rules</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;VB&gt;&lt;NN&gt;</td>
<td>watch the video</td>
</tr>
<tr>
<td>2</td>
<td>&lt;NN&gt;&lt;VB&gt;</td>
<td>promotion offered</td>
</tr>
<tr>
<td>3</td>
<td>&lt;JJ&gt;&lt;NN&gt;</td>
<td>3D live wallpaper</td>
</tr>
<tr>
<td>4</td>
<td>&lt;NN&gt;&lt;JJ&gt;</td>
<td>user editable and changeable</td>
</tr>
<tr>
<td>5</td>
<td>&lt;JJ&gt;&lt;VB&gt;</td>
<td>lazy monitor</td>
</tr>
<tr>
<td>6</td>
<td>&lt;VB&gt;&lt;JJ&gt;</td>
<td>pause when asleep</td>
</tr>
<tr>
<td>7</td>
<td>&lt;VB&gt;&lt;VB&gt;</td>
<td>share and collect</td>
</tr>
<tr>
<td>8</td>
<td>&lt;NN&gt;&lt;NN&gt;</td>
<td>temperature measure</td>
</tr>
<tr>
<td>9</td>
<td>&lt;JJ&gt;&lt;JJ&gt;</td>
<td>safe and fast</td>
</tr>
</tbody>
</table>

**Linguistic Rule Filter.** Linguistic rule filter removes additional sentences based on the Part-Of-Speech (POS) tags of the constituent words in the sentences. Guzman et al. (2014) stated that verbs, adjectives, and nouns play an important role in defining features. For example, adjectives and nouns are often combined together to describe the characteristic of a software system. After a deep observation on plenty of descriptions of apps, especially the difference between POS of feature-describing sentences and POS of non-feature-describing sentences, we define nine linguistic rules (see Table V) to capture feature-describing sentences. The second column of Table V shows the linguistic rules. Each element in the rules corresponds to both the basic form of a POS and its variants. For example, <NN> means <NN>, <NNS>, <NNP>, and <NNPS> in Penn Treebank Tags. In the third column of Table V, a simple example is presented for each rule. We use the Stanford POS Tagger [Toutanova et al. 2003] to analyze each sentence and infer the POS tags of each word. We then apply the nine rules sequentially. If a sentence does not meet any of these linguistic rules, it is filtered out. Otherwise, it is retained.

**Feature Classifier.** After filtered by the above two components, the remaining sentences are predicted to be feature-describing or not by a classifier, namely Naï ve Bayes classifier in this study. To train the feature classifier, we manually build a training set containing both feature and non-feature sentences. For each of 27 categories of apps in Google Play, we select the top 10 ranked apps (based on their ratings) and download their descriptions. If the top ranked apps belong to the 100 randomly selected apps of AFD, we remove them from the training set. We process these descriptions using the data cleaner and linguistic rule filter components, and collect a total of 2,262 remaining sentences. Then, we manually label them and eventually have a set of 1,139 feature describing sentences (positive sentences) and 1,123 non feature-describing sentences (negative sentences). With such a training set, we use the Naï ve Bayes classification algorithm that is implemented in Weka [Hall et al. 2009] to learn a classifier from the sentences. The class labels of sentences retained by the linguistic rule filter are predicted by the trained classifier, and only the positive sentences are retained.

By following the three steps of AFE, we can obtain feature-describing sentences in the description of an app. AFE with the training set is also publicly open (available for download from: http://oscar-lab.org/FRA/), and we hope developers and researchers can benefit from it.

---

4 http://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html.
5. SIMILAR APP BASED FEATURE RECOMMENDER (SAFER)

In this section, we present the framework of SAFER with its main components (see Fig. 3). Given a new app, SAFER analyzes apps in the Reference App Repository. First, all the reference apps are processed by a Reference App Filter (see Section 5.1) to remove low quality ones. Next, for each retained app, SAFER extracts its features using AFE (described in Section 4) and extracts their API invocations using the API Extractor component (see Section 5.2). Then, SAFER builds a profile for each reference app based on its features and API invocations (see Section 5.3). LDA is then used to analyze these profiles and infer the topic distributions of each reference app (see Section 5.4). At the same time, a profile is also built for the new app from its initial features and this profile is input into the LDA model to identify similar apps (see Section 5.5). At last, features of the similar apps are recommended for this new app (see Section 5.6). We elaborate the various components of SAFER in the following subsections.

5.1 Reference App Filter

SAFER aims to mine features from similar competing apps, as a result, the qualities of the similar apps should be taken into consideration. To make a new app successful, developers would want to be inspired by high quality apps rather than low quality ones. Similar to past studies [Bavota et al. 2015; Guerrouj et al. 2015], we use an app’s user rating as an indicator of app’s quality and success. We set up two filtering criteria to remove low quality apps:

1. The average user rating score of a selected app should be more than 3. A user rating score ranges from 0 to 5 in Google Play, and we choose 3 as the threshold. The higher the score is, the more satisfied with the app users are.
2. The number of ratings that a selected app receives (rating count) should be more than 100. To make the average user rating score reliable and reduce bias, we restrict that the rating count should be more than 100. In this way, we indirectly set up the minimum number of times a selected app is downloaded.

For each app in the Reference App Repository, if it meets both of the two criteria, it is selected. Otherwise, it is filtered out.
5.2 API Extractor

To better characterize an app, we extract the API invocations that an app makes to complement features extracted from the description of the app. We consider API invocations since implementing a feature generally involves usage of some specific APIs [Bavota et al. 2015; Thung et al. 2013; Charrada et al. 2015; Heimdahl et al. 1997]. Mobile apps implementing a similar feature are likely to share a substantial proportion of API invocations [Gorla et al. 2014]. For example, just as the name suggests, the API “android.hardware.Camera.takePicture” is related to the feature “take pictures using a camera”. Although the names of identifiers may be obfuscated, API invocations will not change [Ruiz et al. 2012].

Since reference apps are saved in the form of Android PacKages (APKs), API Extractor extracts API invocations from these APK files by the following steps:

1. APK file decompression. We invoke the ‘tar’ command to decompress the APK files and obtain .dex files.
2. .dex file conversion. We leverage the dex2jar\(^5\) disassembler tool to convert the .dex files into .jar files. We filter out all APK files that cannot be converted.
3. .jar file decompression. In the same way as step (1), we decompress the .jar files into the .class files.
4. API invocation extraction. We leverage JClassInfo\(^6\) which is a toolkit for extracting API invocations from .class files.

Since there are plenty of API invocations in each app, we perform several additional heuristic steps to reduce the number of API invocations and uncover important and representative ones. First, we remove invocations of methods that appear in generic API packages which are likely to have little relevance with specific app features. For example, the aim of package “android.support” is to make an app compatible with old Android releases and the aim of package “android.database” is to allow an app to manipulate a database. The other generic API packages include “android.util” and “android.sax”. Second, inspired from the IDF term weighting scheme from Information Retrieval (IR), we filter out some non-discriminant API invocations that are commonly used across all the apps. We count the number of apps an API invocation appears, and rank all API invocations based on it. We treat the top 5% API invocations as non-discriminant API invocations and filter them out. Third, we only consider the class name and the method name of an API invocation rather than the whole API signature. We do this step to group related methods together.

At the end, for each app, API Extractor extracts a set of names of APIs that are called in the app.

5.3 App Profile Builder

App Profile Builder builds a profile for each app from two sources: (1) the app’s features extracted from its description using AFE and (2) the app’s API invocations. More specifically, App Profile Builder first creates an API vector and a feature vector for every app, and then combines both vectors together to create a profile for this app.

Given all the names of APIs invoked by an app, App Profile Builder performs a series of natural language processing steps, namely tokenization, camel case splitting,

---

stemming, and stop word removal [Butler et al. 2011; Annervaz et al. 2013]. We add “Java” and “Android” keywords to the list of stop words, since “Java” and “Android” keywords are the most common words that appear in API names. After that, every resulting term is mapped to a vector element whose value is its Term Frequency (TF) – the number of times the term appears. In such a way, an API vector can be created for each app.

For features extracted from the description of an app with AFE, we follow the same steps (except camel case splitting) to form a feature vector for the app.

Given the API vector \( V_{API} \) and the feature vector \( V_{feature} \) of an app, the profile of the app is created by merging the corresponding words from the API and feature vectors. The term weight of each word in the profile is defined as follows:

\[
W_{i \text{ in profile}} = 2 \times W_{i \text{ in feature}} + 1 \times W_{i \text{ in API}}
\]  

where \( W_{i \text{ in profile}} \) is the weight of the \( i \)th term in the profile, while \( W_{i \text{ in feature}} \) and \( W_{i \text{ in API}} \) are the weights of the \( i \)th term in the feature and API accordingly. Inspired by prior studies [Wang et al, 2008; Sun et al. 2010], we double the weights of the terms in the feature vector. In other words, we consider features extracted from descriptions to be more important than API invocations.

App Profile Builder can create profiles for both reference apps and a new app. For a new app with its initial features, its API vector \( V_{API} \) is empty.

5.4 Topic Model

Inspired by the work of Hindle et al. (2012), we leverage LDA to identify the topic distribution of each app profile, so that we can identify similar apps. Two apps with similar topic distributions are likely to belong to the same product type [Gorla et al. 2014]. We make use of the LDA implementation that comes with the TMT toolbox\(^7\). As suggested in [Gorla et al. 2014], we set the number of topics to 30, and the other parameters are set to their default values (e.g., topicSmoothing and termSmoothing are both set to 0.01). Better results may be achieved if we calibrate these parameters.

Table VI shows an example of 10 main topics with representative stemmed terms mined by LDA from descriptions of apps in our annotated dataset (see Section 3.2) that falls in the category Business. In the 3rd and 4th columns of Table VI, we list the topic distributions of two apps, namely Property Milestone and Gig Harbor Real Estate. Below Table VI, we also present the manually extracted golden features from the descriptions of the two apps. As shown in Table VI, the topic distributions could well characterize the features of the two apps. For example, one of the main features of Property Milestone is “property search”, and its probability for Topic 1, whose most representative terms include “search” and “find”, is 20.1%.

5.5 Similar App Identifier

Given a new app, Similar App Identifier identifies similar apps based on the topic distributions of the new app and apps in the Reference App Repository. More specifically, Similar App Identifier computes the cosine similarity between the new app and every reference app, and ranks all the reference apps in descending order of their cosine similarities. The top K similar reference apps are then returned for the next and final step of SAFER.

\( ^7 \) http://nlp.stanford.edu/software/tmt/tmt-0.4/.
Table VI. Topics and Topic Distribution of Some Apps

<table>
<thead>
<tr>
<th>Topic</th>
<th>Most Representative Stemmed Terms</th>
<th>Property Milestone</th>
<th>Gig Harbor Real Estate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>search, find, career, seek, company</td>
<td>20.1%</td>
<td>16.5%</td>
</tr>
<tr>
<td>2</td>
<td>file, document, pdf, text, office</td>
<td>3.3%</td>
<td>8.2%</td>
</tr>
<tr>
<td>3</td>
<td>expans, record, calcul, report, cost</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>4</td>
<td>map, locat, track, address, direct</td>
<td>4.9%</td>
<td>6.5%</td>
</tr>
<tr>
<td>5</td>
<td>send, email, photo, messag, attach</td>
<td>0%</td>
<td>5.9%</td>
</tr>
<tr>
<td>6</td>
<td>confer, session, attende, speaker, schedule</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>7</td>
<td>trade, market, bui, sell, exchang,</td>
<td>0.5%</td>
<td>0%</td>
</tr>
<tr>
<td>8</td>
<td>store, find, save, servic, offer</td>
<td>4.8%</td>
<td>4.6%</td>
</tr>
<tr>
<td>9</td>
<td>control, remot, record, secuir, fast</td>
<td>1.6%</td>
<td>0%</td>
</tr>
<tr>
<td>10</td>
<td>share, social, push, receive, media</td>
<td>0%</td>
<td>1.5%</td>
</tr>
</tbody>
</table>

Property Milestone:
- property searches: search properties to buy or rent and view full properties details.
- agents search: quickly search real estate agents nearby about listings.
- map: view properties on map in normal, satellite or traffic mode.
- manage account: you can register or sign in to save property and real estate agent listing for view later.
- saved listing: view your favourite property and real estate agents.

Gig Harbor Real Estate:
- search homes for sale.
- filter searches by price, beds, baths, and more.
- view full screen color photos.
- map and get the directions to each listing.
- access to agent/broker website, phone and email.

Given two apps $A_i$ and $A_j$, the cosine similarity of $A_i$ and $A_j$ can be calculated as follows:

$$\text{Simi} (A_i, A_j) = \frac{\sum_{t} P_{A_i \in t} \times P_{A_j \in t}}{\sqrt{\sum_{t} P_{A_i \in t} \times P_{A_i \in t}} \times \sqrt{\sum_{t} P_{A_j \in t} \times P_{A_j \in t}}}$$  \hspace{1cm} (2)

where $P_{A_i \in t}$ and $P_{A_j \in t}$ represent the probabilities of apps $A_i$ and $A_j$ belonging to a specific topic $t$.

5.6 Feature Recommender

After Similar App Identifier returns the set of top $K$ most similar apps, Feature Recommender processes the features of the top $K$ apps as well as the initial features of the new app, and recommends features for the new app.

The top $K$ apps may share similar features written in different forms. Hence, Feature Recommender aggregates features using the Affinity Propagation (AP) clustering algorithm [Frey et al. 2007], which is a density based clustering algorithm. Using AP, we do not need to specify the centers of clusters and the number of clusters. AP takes a matrix that defines the similarity between any two data points as input, and outputs the clusters with their exemplars. In this study, we define the matrix by calculating the cosine similarity between every two features. After clustering, the features belonging to a cluster are named by its exemplar. For example, apps Handy Lyrics and Atomic Kitten All Lyrics have features “lyrics categorized by album” and “browse lyrics by album” respectively. After clustering, the two features belong to the same cluster, and the exemplar “browse lyrics by album” is used as the
Table VII algorithm of feature recommendation

<table>
<thead>
<tr>
<th>Feature Recommendation Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> clusters and their connections</td>
</tr>
<tr>
<td><strong>Output:</strong> ranked recommended features</td>
</tr>
<tr>
<td>1. find all the occupied clusters;</td>
</tr>
<tr>
<td>2. locate all the neighboring clusters of occupied clusters;</td>
</tr>
<tr>
<td>3. rank exemplars of neighboring clusters based on their weights;</td>
</tr>
<tr>
<td>4. if (two exemplars have the same weight)</td>
</tr>
<tr>
<td>5. rank the two exemplars based on their sizes;</td>
</tr>
<tr>
<td>6. end if;</td>
</tr>
<tr>
<td>7. if (the number of recommended features is less than 15)</td>
</tr>
<tr>
<td>8. rank the rest exemplars of clusters based on their sizes;</td>
</tr>
<tr>
<td>9. end if;</td>
</tr>
</tbody>
</table>

---

representative of the two features.

After we cluster all the features, we connect all pairs of clusters by drawing lines between them. Each line between two clusters has a weight, which is the number of top K apps that contain the two features. The weight of a line between two clusters shows the degree of correlation between features. Feature Recommender ranks the feature clusters based on the weights of the lines and the sizes of the clusters.

The pseudocode of the feature recommendation algorithm is shown in Table VII. Feature Recommender first locates the clusters that the initial features of the new App belong to – we refer to these clusters as **occupied clusters**. It then finds all neighboring clusters of the occupied clusters and ranks these neighbors in descending order of the maximum weights of the lines linked with the occupied clusters. If two neighbors have the same weight, they are ranked based on their sizes. If the number of the neighbors of the occupied clusters is less than N (by default N is set to 15), Feature Recommender also includes and ranks non-neighboring clusters based on their sizes. In such a way, we can obtain the ranked recommended features.

Taking Fig. 4 as an example, each circle stands for a cluster and the size of the circle shows the size of the corresponding cluster (the number of features in this cluster). The numbers over the lines between the circles stand for the weights. In this example, the clusters 3 and 4 are occupied clusters, namely two initial features “turns radio automatically off, when you receive a call” and “where you are, the latest political talk news present with you” are in the two clusters. As we can see, the clusters 1, 2, and 5 are neighboring clusters. SAFER will first recommend the exemplar “control audio and view track titles from the lock screen” in cluster 2 to
developers, since the maximal weight of its lines is the largest, i.e., 15. Then, the exemplars of clusters 1 and 5 are recommended based on the weights, that is to say that “a player for listening to radio stations” and “browse popular tags or genres of the week” are recommended successively. At last, the exemplar “tracks interactive music chart of songs posted on twitter” in cluster 6 is recommended, since it has no connection with the occupied clusters.

6. EXPERIMENTAL SETUP

In this section, we describe our experiment settings, baseline, evaluation method, and evaluation metrics.

6.1 Experiment Settings

All the experiments are conducted on a Core i5 CPU PC with 8G memory running Windows 7. We implemented all the algorithms in Java compiled by MyEclipse 10 using Weka 3.6.5 library [Hall et al. 2009].

SAFEr takes in a parameter K, namely the number of similar apps to derive feature recommendations from. We set K = 60 as the default parameter value. In Section 7.1, we will show the impact of modifying K.

6.2 Baseline

As discussed in Section 1, no existing method in the literature can recommend new features for an app from descriptions of other apps in app markets. The closest work is by Hariri et al. (2013) who propose an approach named KNN+ to recommend new features from Softpedia.com, a website collecting features for software products [Hariri et al. 2013]. However, Softpedia.com contains a limited number of apps whose features have been manually curated and listed explicitly in bullet point form, and categorized into fine-grained product type. KNN+ thus does not address the feature identification challenge and in absence of fine-grained product types (Google Play does not have fine-grained app categories), it does not fully address the similar app identification challenge.

KNN+ works as follows. First, an incremental diffusive clustering is employed to aggregate and name features from a product type. Then, a product-by-feature matrix is created and several association rules are mined from a frequent item set graph generated from the matrix. Last, when some initial features of a new software product are provided, these initial features are extended by the association rules and new features can be recommended based on the standard K Nearest Neighbor clustering algorithm.

Since KNN+ cannot extract features from the descriptions of apps in app markets (e.g., Google Play), we employ AFE to extract features which are then used as input to KNN+ to address the feature identification challenge. We use the apps in the same category in Reference App Repository as the product type for KNN+ to address the similar app identification challenge.

6.3 Evaluation Method

Ideally, a feature recommender is said to successfully recommend a feature for an app developer, if this developer agrees that the recommended feature actually helps him/her in developing his/her app. However, this process is highly subjective and it is hard to invite a large number of app developers from industry. Hence, we evaluate the results following an elimination-recovery method proposed by Hariri et al. (2013)
which works as follows. For each golden feature \( f \) in the set \( F \) of golden features of app \( A \), we eliminate \( f \) from the set \( F \) and let a feature recommender takes in all the remaining features in \( F \setminus \{f\} \) as input. After the feature recommender recommends a ranked list of features, we employ three volunteers to check whether the eliminated feature \( f \) is hit in the ranked list. Here, a recommended feature \( f \) is said to be hit when two or three of the volunteers find \( f \) to be described by one of the features in the ranked list.

Following Hariri et al. (2013), we conduct a leave-one-out (LOO) test over each category of apps. More specifically, we choose every app \( A \) with golden features in FAD (see Section 3.2) as the test set, and have all the apps in the same category in the Reference App Repository as the training set. For each golden feature in app \( A \), we follow the elimination-recovery method to evaluate the performance of SAFER. After all the apps with golden features in FDA are used as test sets, we average the results over each category of apps.

6.4 Evaluation Metrics

In this study, Hit Ratio and Normalized Discounted Cumulative Gain (NDCG) are used as yardsticks to evaluate the performance of SAFER. In the experiments, we recommend 15 features for each App, and calculate Hit Ratio and NDCG at top 15 ranked features.

Hit Ratio is a widely used metric in feature recommendation [Hariri et al. 2013] to evaluate how many features can be successfully recommended. Hit Ratio can be calculated as follows:

\[
\text{Hit Ratio} = \frac{\text{\# of hit features}}{\text{\# of features}} \times 100\%
\]  

(3)

In addition to Hit Ratio, we compute NDCG to further evaluate the quality of the recommended list of features. NDCG is a well-known metric to evaluate a ranked list in information retrieval and recommendation systems [Järvelin et al. 2002]. NDCG can fully evaluate a recommended list of results from the top ranking result to the bottom ranking one with the gain of each result discounted by lower ranks. NDCG is defined as follows:

\[
\text{NDCG} = \frac{G}{G_{\text{ideal}}} = \frac{G}{\sum_{i=1}^{5} \frac{\text{score}_i - 1}{\log_2(i+1)}}
\]  

(4)

where \( \text{score}_i \) is equal to 1 when the \( i \)th recommended feature hits a golden feature, otherwise it is equal to 0. Here, ideal \( G \) [Mcmilan et al. 2013] is a special form of \( G \) with the best rank, namely all 1s rank higher than 0s.

In addition, we also introduce Precision, Recall, F-Measure, and Accuracy to evaluate the performance of AFE. These evaluation metrics are widely used in data mining and Information Retrieval (IR). The comparison results between true condition and predicted condition can be shown in a confusion matrix in Table VIII.

Based on the confusion matrix, Precision, Recall, F-Measure, and Accuracy can be calculated as follows:

\[
\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}
\]  

(5)

\[
\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}
\]  

(6)
Table VIII. confusion matrix

<table>
<thead>
<tr>
<th>Predicted Condition</th>
<th>True Condition</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive</td>
<td>True Positive (TP)</td>
<td>False Positive (FP)</td>
<td></td>
</tr>
<tr>
<td>Negative</td>
<td>False Negative (FN)</td>
<td>True Negative (TN)</td>
<td></td>
</tr>
</tbody>
</table>

\[ F\text{-Measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \]  
\[ \text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \]

7. EXPERIMENTAL RESULTS

In this section, we investigate five Research Questions (RQs) to evaluate the performance of AFE and SAFER.

7.1 RQ1: How will the parameter \( K \) influence the performance of SAFER?

**Motivation.** There is a parameter in SAFER, namely \( K \), which is the number of top similar apps used by SAFER to recommend new features. In this RQ, we try to find a good value of \( K \) and investigate its impact on SAFER’s performance.

**Approach.** We investigate a set of \( K \) values, namely \{20, 40, 60, and 80\}. Out of the five categories in AFD (see Section 3.2), we evaluate the behavior of SAFER over the first two categories, namely Business and Education. We run SAFER with each value of \( K \) on the two categories and evaluate the features recommended by SAFER using Hit Ratio and NDCG.

**Results.** In Fig. 5 and Fig. 6, we present the experimental results of SAFER over Business and Education category respectively, for different values of \( K \). As shown in Fig. 5 and Fig. 6, the performance of SAFER varies along with the changing of \( K \). However, the behavior of SAFER exhibits similar trends over both categories. For example, the value of Hit Ratio over the category Business improves from 60.20% to 73.99% when \( K \) grows from 20 to 60. When \( K \) grows to 80, the Hit Ratio drops to 69.48%. The NDCG curve of SAFER over the category Business also follows a similar trend. We can also note similar findings for the category Education in terms of both Hit Ratio and NDCG. For example, SAFER achieves the best Hit Ratio (78.68%) and NDCG (0.4955) when \( K \) is set to 60. The performance of SAFER declines when \( K \) is increased or reduced. Therefore, we set \( K = 60 \) in the remaining experiments.

The findings indicate that as we increase the value of \( K \), more and more closely related apps are detected, hence it becomes easier to recommend new features for a new app. However, as the value of \( K \) increases beyond a certain point, many unrelated apps may be introduced, which creates noise to the new feature identification process.

**Conclusion.** The parameter \( K \) influences the performance of SAFER. Based on the parameter tuning results over two categories, the best results are achieved when \( K \) is equal to 60. As a result, \( K \) is kept as 60 in the following RQs.
7.2 RQ2: How effective is AFE in extracting features from app descriptions?

**Motivation.** To extract features from the descriptions of apps, we construct a tool named AFE. In this RQ, we aim to investigate how effective is AFE in identifying feature-describing sentences from app descriptions.

**Approach.** We run AFE to extract feature-describing sentences from the descriptions of apps in the annotated feature dataset (AFD) (see Section 3.2). For each app in AFD, we use AFE to extract features from its description and compare these extracted features with the golden features annotated by volunteers. To measure the detailed performance of AFE, we count the average number of feature-describing (golden features) and non-feature-describing sentences (non-golden features) before and after each filtering step of AFE. In addition, we also show the other performance indicators for all the categories, such as Precision, Recall, F-Measure, and Accuracy.

**Results.** For every category of apps, we present in Fig. 7 and Fig. 8 the average number of feature-describing and non-feature-describing sentences remained before processing, after applying the data cleaner, after applying the linguistic rule filter, and after applying feature classifier. As seen from the figures, after applying the data cleaner, on average 5.70 golden features and 3.25 non-golden features are retained for the category Business. It indicates that the data cleaner component is effective since only 0.15 golden features are falsely filtered out, meanwhile 1.5 non-golden features are correctly filtered out. After processed by the linguistic rule filter, we can find that no golden features are removed, and 0.3 non-golden features are filtered out. After applying the feature classifier, 4.80 golden features are correctly predicted. Meanwhile only 1.50 non-golden features are falsely predicted as features and retained. We can find similar phenomenon for the other categories. A good tool for extracting features from the descriptions of apps should retain as many feature-describing sentences as possible, while retain as few non-feature-describing sentences as possible. By comparing Fig. 7 and Fig. 8, we can see that the areas of golden features are not reduced too much after each step. However, the areas of non-golden features decrease substantially. It reveals that AFE could correctly retain most of golden features and filter out most of non-golden features.
Fig. 7. Average number of feature-describing sentences after each step of AFE

Fig. 8. Average number of non-feature-describing sentences after each step of AFE

Table IX. Results of AFE for each category

<table>
<thead>
<tr>
<th>Category</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Measure</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business</td>
<td>76.19%</td>
<td>82.05%</td>
<td>79.01%</td>
<td>75.94%</td>
</tr>
<tr>
<td>Education</td>
<td>66.88%</td>
<td>82.03%</td>
<td>73.67%</td>
<td>73.40%</td>
</tr>
<tr>
<td>Health and Fitness</td>
<td>56.61%</td>
<td>77.00%</td>
<td>65.25%</td>
<td>67.46%</td>
</tr>
<tr>
<td>Finance</td>
<td>66.19%</td>
<td>75.41%</td>
<td>70.50%</td>
<td>69.92%</td>
</tr>
<tr>
<td>Music and Audio</td>
<td>43.08%</td>
<td>84.85%</td>
<td>57.14%</td>
<td>61.11%</td>
</tr>
<tr>
<td>Average</td>
<td>61.79%</td>
<td>80.27%</td>
<td>69.11%</td>
<td>69.57%</td>
</tr>
</tbody>
</table>

Table IX summaries the overall results of AFE in terms of Precision, Recall, F-Measure, and Accuracy for each category. For instance, when applying AFE on the category Business, it could achieve a Precision of 76.19%, and Recall of 82.05%. When considering F-Measure and Accuracy, AFE can achieve 79.01% and 75.94%
respectively. We can see from the table that, AFE could achieve an average \textit{F-Measure} of 69.11\% and average \textit{Accuracy} of 69.57\%. That’s to say that AFE is an effective tool to distinguish features from non-features in the descriptions of apps.

\textbf{Conclusion}. AFE is an effective tool to extract features from the descriptions of apps. It can retain most of the golden features and filter out a majority of the non-golden features.

7.3 \textbf{RQ3: Does the introduction of API invocations contribute positively to SAFER's effectiveness?}

\textbf{Motivation}. SAFER tries to combine features mined from app descriptions and API invocations to construct a profile for an app. In this RQ, we try to explore whether API invocations can complements features extracted from app descriptions.

\textbf{Approach}. We define and implement a variant of SAFER, namely SAFER-API which removes the API Extractor component and keeps the other components the same. We run SAFER and SAFER-API on the same annotated dataset. By comparing the results of SAFER against SAFER-API, we can evaluate the benefit of introducing API invocations.

\textbf{Results}. Table X shows the comparison results between SAFER and SAFER-API. In terms of \textit{Hit Ratio}, we can see that SAFER achieves better results than SAFER-API in most of the categories, except for the category \textit{Finance}, where the difference is little. For example, SAFER achieves a \textit{Hit Ratio} of 73.99\% and improves SAFER-API by 4.69\% for the category \textit{Business}. When considering the category \textit{Finance}, SAFER-API only outperforms SAFER by 0.64\%. On average, SAFER outperforms SAFER-API by 2.56\%. In terms of \textit{NDCG}, we can find that SAFER is superior to SAFER-API for all the categories. Overall, in terms of \textit{NDCG}, SAFER outperforms SAFER-API by 0.0186.

\textbf{Conclusion}. Taking API invocations into consideration can improve the results of SAFER. API invocations can be used as good complements to extracted features from app descriptions.

7.4 \textbf{RQ4: How SAFER performs with different number of initial features?}

\textbf{Motivation}. As the inputs to SAFER, the number of initial features may influence its performance. Through this RQ, we want to explore to what extent SAFER’s performance depends on the number of the initial features.
Approach. In real development process, developers may conceive different numbers of initial features. In this research question, we want to investigate the effectiveness of SAFER when we vary the number of initial features. We predefine these values by adjusting different percentages of the golden features, namely (25%, 50%, 75%, and leave-one-out (LOO)), as initial features. We then measure the Hit Ratio and NDCG for the different percentages.
Table XI. SAFER vs. KNN+

<table>
<thead>
<tr>
<th>Category</th>
<th>Hit Ratio</th>
<th></th>
<th>NDCG</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SAFER</td>
<td>KNN+</td>
<td>SAFER</td>
<td>KNN+</td>
</tr>
<tr>
<td>Business</td>
<td>73.99%</td>
<td>64.79%</td>
<td>0.4802</td>
<td>0.4258</td>
</tr>
<tr>
<td>Education</td>
<td>78.68%</td>
<td>56.10%</td>
<td>0.4955</td>
<td>0.4090</td>
</tr>
<tr>
<td>Health and Fitness</td>
<td>69.92%</td>
<td>54.70%</td>
<td>0.4536</td>
<td>0.3703</td>
</tr>
<tr>
<td>Finance</td>
<td>54.49%</td>
<td>38.01%</td>
<td>0.3714</td>
<td>0.3154</td>
</tr>
<tr>
<td>Music and Audio</td>
<td>64.38%</td>
<td>41.71%</td>
<td>0.4327</td>
<td>0.3476</td>
</tr>
<tr>
<td>Average</td>
<td>68.29%</td>
<td>51.06%</td>
<td>0.4467</td>
<td>0.3736</td>
</tr>
</tbody>
</table>

Results. Fig. 9 and Fig. 10 show the results of Hit Ratio and NDCG respectively. We can see from the figures that, as the percentage increases, both the values of Hit Ratio and NDCG show upward trends. For example, for category Business, when the inputs are 25% of all the golden features, the Hit Ratio is 63.86%. When the percentage is increased to 75%, the Hit Ratio is 71.58%. When using LOO, the Hit Ratio is 73.99%. We can also observe the same phenomenon for the other categories with some exceptions. We can find that there is no big difference between 75% and LOO. The reason is that the average number of features in AFD is close to 5. As a result, the input to SAFER using either 75% of the golden features or LOO is typically the same number of features (i.e., 4 features). The upward trends of the results may be due to the fact that the more the initial features are, the more likely they can better model the app profile. As a result, the recommended features have high probabilities to hit the target features.

Conclusion. As we increase the number of initial features, the Hit Ratio and NDCG scores increase.

7.5 RQ5: Can SAFER outperform the baseline approach?

Motivation. KNN+ can be adapted to solve the same problem as SAFER. In this RQ, we investigate whether SAFER can achieve better results than the adapted KNN+. We also try to explore whether the difference between the performance of the two approaches is statistically significant.

Approach. We run KNN+ on each app category in AFD. By conducting LOO and collecting the results, we can compare the results of SAFER against KNN+. Besides, we introduce paired Wilcoxon signed rank test, which is a non-parametric test, to explore the statistical significance of the difference between the performance of SAFER and KNN+ in recommending features for new apps. We formulate the two test hypotheses as follows:

H0: There is no significant difference between the performance of the two approaches.

H1: There is significant difference between the performance of the two approaches.

In this study, we set the significance level to be 5%, which means that significant difference between the performance of the two approaches could be detected if the p-value is below 0.05. In contrast, a p-value larger than 0.05 implies that the two approaches perform similarly considering the evaluation metric.
Results. The results of the two approaches are summarized in Table XI. As shown in Table XI, in terms of Hit Ratio, SAFER significantly outperforms KNN+ over all the categories of apps in AFD. For example, SAFER improves KNN+ by up to 22.67% over the category Music and Audio. On average, SAFER improves KNN+ by 17.23% in terms of Hit Ratio. This result implies that SAFER can successfully recommend far more golden features than KNN+. In terms of NDCG, SAFER also outperforms KNN+ for all the categories. On average, SAFER outperforms KNN+ by 0.0731 in terms of NDCG. The reason why SAFER performs well may be that, SAFER introduces API invocations as complements to features extracted from app descriptions, and builds app profiles based on both features and API invocations. In such a way, apps can be modeled accurately. At the same time, SAFER leverages topic model to better identify similar apps. As a result, SAFER performs better than KNN+ in terms of both Hit Ratio and NDCG.

When we consider Hit Ratio as the evaluation metric, the p-value obtained by the Wilcoxon test is 0.001, which means that $H_0$ is rejected, and there exists significant difference between the performance of SAFER and KNN+. Similar phenomenon could be observed when we consider the NDCG metric (p-value = 0.008). Considering that SAFER achieves better Hit Ratio and NDCG than KNN+ on average, we can conclude that SAFER is superior to KNN+.

Conclusion. SAFER outperforms KNN+ in terms of Hit Ratio and NDCG over all the five app categories, and the improvement achieved by SAFER is statistically significant.

8. DISCUSSION
We first discuss relationships among features and granularities of features which are directions that we plan to consider in a future work. Next, we discuss several threats to validity.

8.1 Relationship among Features
In this study, SAFER only considers two relationships among features. Two features are either the same (they are in the same cluster) or different (they are in two different clusters). In practice, relationships among features are likely to be more nuanced and varied. In this work, we ignore such relationships and plan to consider and use them to improve feature recommendation in a future work.

8.2 Granularities of Features
To avoid missing a feature, all the different granularities of features are taken into consideration, ranging from low-level configuration options to higher-level capabilities. In this way, we hope SAFER can help developers analyze all the available features of similar apps in the market and make the final decision. In the future, we plan to differentiate features of different granularities and recommend them separately.

8.3 Threats to Validity
Internal Validity. Since no dataset of features is available, we have volunteers annotate a dataset of apps and vote the resulting list of features. The backgrounds and personal opinions of volunteers may influence the annotation and evaluation of results. To reduce the bias of volunteers, we have three distinct volunteers annotate...
and vote for each feature and result. A sentence is treated as a golden feature and a recommended feature hits a golden feature, if it receives at least two out of the three votes. In such a way, we think that this threat is reduced.

Besides, since it is labor intensive and time consuming to directly evaluate whether SAFER can recommend new features for new apps, we evaluate SAFER following an elimination-recovery method, which was proposed earlier to evaluate another feature recommendation approach [Hariri et al. 2013].

External Validity. In this study, we use an annotated dataset with hundreds of features from 100 apps to evaluate the performance of SAFER. It is still uncertain how well SAFER performs over other apps. Still, we believe 100 apps and 533 features are large enough to illustrate the performance of SAFER.

9. RELATED WORK
A number of past studies propose approaches to identify and recommend features. These approaches can be roughly classified into three categories (see Table XII). Studies in the first category propose methodologies to either manually or semi-automatically extract features from requirement documentation. Studies under this category include Feature Oriented Domain Analysis (FODA) [Kang et al. 1990] and Domain Analysis and Reuse Environment (DARE) [Frakes et al. 1998; Santos et al. 2009]. The second category leverages a social network of stakeholders of different influence levels to identify and rank requirements by asking each stakeholder to write new requirements and rate requirements created by other stakeholders (e.g., [Lim et al. 2011; Lim et al. 2010]). Approaches in the above two categories cannot automatically recommend features to apps – a substantial manual step involving domain analysts or stakeholders is needed.

In contrast, the third category leverages data mining and natural language processing techniques to automatically recommend features. The methods in this category could be further divided into three sub-categories, based on their sources of features, namely repositories of requirement specifications [Alves et al. 2008; Chen et al. 2005], forums [Rahimi et al. 2014] or user feedback [Carreño et al. 2013; Iacob et al. 2013], and Softpedia.com [Hariri et al. 2013]. Some automatic methods mine and recommend features from repositories of requirement specifications [Alves et al. 2008; Chen et al. 2005]. However, a representative repository of requirement specifications does not exist for mobile apps. Rahimi and Cleland-Huang recommend features from

![Table XII. Existing Work on Feature Recommendation](image)

<table>
<thead>
<tr>
<th>#</th>
<th>Main Approach</th>
<th>Source of Features</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Social networks</td>
<td>Stakeholders</td>
<td>[Lim et al. 2011; Lim et al. 2010]</td>
</tr>
<tr>
<td>3</td>
<td>Data mining and natural language processing</td>
<td>Requirement specifications</td>
<td>[Rahimi et al. 2014; Carreño et al. 2013; Iacob et al. 2013]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Forum or user feedback</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Softpedia.com</td>
<td>[Hariri et al. 2013]</td>
</tr>
</tbody>
</table>
forums [Rahimi et al. 2014]. Carreño and Winbladh extract new requirements from user comments of apps [Carreño et al. 2013]. Iacob and Harrison define linguistic rules to retrieve feature requests from user comments of apps [Iacob et al. 2013]. However, these approaches are not applicable for newly released apps, or apps under development since there could be few or no users of such apps yet. The closest work to ours is the one that recommend features based on Softpedia.com [Hariri et al. 2013]. As highlighted in Section 1, this work cannot directly work on app descriptions in app markets since it cannot solve the feature identification and similar app identification challenges. We have adapted the approach proposed by Hariri et al. (named KNN+) so that it can work for our problem, and shown in our experiments that SAFER outperforms the adapted KNN+.

10. CONCLUSION AND FUTURE WORK

The features an app provides can greatly impact its success. In this study, we propose a new approach named SAFER to recommend features to a new app by analyzing descriptions of other similar apps in an app market. SAFER creates app profiles by identifying feature-describing sentences in an app's description and API invocations, and utilizes a topic model to process app profiles into latent topics to identify similar apps. Next, SAFER recommends features to a new app, by aggregating features of its similar apps and contrasting them to the initial features that the new app has. We have evaluated the effectiveness of SAFER on an annotated dataset containing 533 features extracted from 100 apps. Extensive experiments on the dataset show that, SAFER can recommend features well and is superior to a baseline approach adapted from the work of Hariri et al. (2013).

For future work, we intend to improve SAFER in several aspects. First, we plan to explore the interesting directions of research described in Sections 8.1 and 8.2. Second, we plan to better address the threats to validity by including more dataset and experiments. Third, we plan to investigate not only Android apps but also iOS apps.

ACKNOWLEDGMENTS

We would like to thank the volunteers who help to annotate the dataset of golden features (see Section 3.2). Also, we would like to thank developers participating in our survey (see Section 2.2).

REFERENCES


Kun Chen, Wei Zhang, Haiyan Zhao, and Hong Mei. 2005. An Approach to Constructing Feature Models


ACM Transactions on xxxxxxx, Vol. xx, No. x, Article x, Publication date: Month YYYY


