Recommending New Features from Mobile App Descriptions

HE JIANG, Dalian University of Technology
JINGXUAN ZHANG, Nanjing University of Aeronautics and Astronautics
XIAOCHEN LI and ZHILEI REN, Dalian University of Technology
DAVID LO, Singapore Management University
XINDONG WU, Hefei University of Technology, Mininglamp Technology
ZHONGXUAN LUO, Dalian University of Technology

The rapidly evolving mobile applications (apps) have brought great demand for developers to identify new features by inspecting the descriptions of similar apps and acquire missing features for their apps. Unfortunately, due to the huge number of apps, this manual process is time-consuming and unscalable. To help developers identify new features, we propose a new approach named SAFER. In this study, we first develop a tool to automatically extract features from app descriptions. Then, given an app, we leverage the topic model to identify its similar apps based on the extracted features and API names of apps. Finally, we design a feature recommendation algorithm to aggregate and recommend the features of identified similar apps to the specified app. Evaluated over a collection of 533 annotated features from 100 apps, SAFER achieves a Hit@15 score of up to 78.68% and outperforms the baseline approach KNN+ by 17.23% on average. In addition, we also compare SAFER against a typical technique of recommending features from user reviews, i.e., CLAP. Experimental results reveal that SAFER is superior to CLAP by 23.54% in terms of Hit@15.

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

Additional Key Words and Phrases: Mobile applications, feature recommender system, domain analysis, topic model

ACM Reference format:
He Jiang, Jingxuan Zhang, Xiaochen Li, Zhilei Ren, David Lo, Xindong Wu, and Zhongxuan Luo. 2019. Recommending New Features from Mobile App Descriptions. *ACM Trans. Softw. Eng. Methodol.* 28, 4, Article 22 (October 2019), 29 pages.
https://doi.org/10.1145/3344158

This work is partially supported by the National Key Research and Development Plan of China under Grants No. 2018YFB1003900 and the National Natural Science Foundation of China under Grants No. 61722202.
Authors' addresses: H. Jiang, X. Li, Z. Ren, and Z. Luo, Dalian University of Technology, School of Software and Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, Dalian, Liaoning, 116000, China; emails: jianghe@dlut.edu.cn, li1989@mail.dlut.edu.cn, {zren, zxluo}@dlut.edu.cn; J. Zhang, Nanjing University of Aeronautics and Astronautics, College of Computer Science and Technology, Nanjing, Jiangsu, 210016, China; email: jxzhang@nuaa.edu.cn; D. Lo, Singapore Management University, School of Information Systems, Singapore, Singapore; email: davidlo@smu.edu.sg; X. Wu, Hefei University of Technology, Key Laboratory of Knowledge Engineering with Big Data, Ministry of Education, Hefei, Anhui, 230009, China and Mininglamp Academy of Sciences, Mininglamp Technology, Beijing, China; email: wuxindong@mininglamp.com.

1049-331X/2019/10-ART22 $15.00
https://doi.org/10.1145/3344158

1 INTRODUCTION

Recent years have witnessed the sharp growth of the number of mobile applications (apps). Up to September 2018, 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; Narudin et al. 2016; Wu et al. 2019). When facing abundant similar apps (apps with similar functionalities or features), 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 the 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 inputs 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 the features of apps are usually written in free text containing a variety of information, e.g., brief introductions of apps, disclaimers, and contact addresses (Berardi et al. 2015; Fan et al. 2018; Martin et al. 2017). Hence, a specific tool should be developed to identify sentences that describe app features in the descriptions.
— *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 the *Game* category. Therefore, it is hard to detect closely-related similar apps.

In the literature, a number of methods have been proposed to identify and recommend features for 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 (NLP) techniques to recommend features from either a repository of requirement specifications (Alves et al. 2008; Chen et al. 2005) or forums (Rahimi et al. 2014). 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. In addition, 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 the features in Softpedia.com are explicitly provided in a bullet-point list format. In contrast, the features of apps are implicitly provided in their descriptions mixed with other information, e.g., contact information, and so on. Second, all the software products in Softpedia.com are organized hierarchically in three layers; more specifically, 9 categories, 292 sub-categories, and 1,096 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. 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, to obtain similar apps with the new app, we treat all the apps in the same category as potentials.

Another type of related studies to recommend features for apps leverages user reviews. Several typical studies have been proposed to analyze and extract new features from user reviews (Carreño et al. 2013; Nayebi et al. 2017; Chen et al. 2014; Panichella et al. 2015; Scalabrino et al. 2019). User reviews and app descriptions have different characteristics and both of them can be used as sources to acquire and recommend features. User reviews are usually short with huge amount, while app descriptions are relatively long containing mixed information. In addition, user reviews are informal and colloquial. In contrast, app descriptions are formal. These different properties motivate us to compare the approaches of recommending features from user reviews and app descriptions. Among these techniques, we select the most recent and effective approach, namely, CLAP (Scalabrino et al. 2019), for comparison.

In this study, we propose a novel approach named Similar App-based FEature Recommender (SAFER) to recommend features for new apps, which can tackle the above-mentioned challenges. SAFER fully leverages domain specific information of apps, including features and API names, to identify similar apps belonging to the same product type and recommend features for an app. More specifically, SAFER contains seven components, i.e., Reference App Filter, App Feature Extractor (AFE), API Extractor, App Profile Builder, Topic Model, Similar App Identifier, and Feature Recommender. It works as follows. First, a Reference App Repository containing a large scale of apps with different categories is constructed, and Reference App Filter is built to filter out low quality apps. To better characterize apps in the Reference App Repository, AFE is developed to extract features (i.e., feature-describing sentences) from the app descriptions, which tackles the *feature identification* challenge (Jiang et al. 2014). At the same time, API Extractor identifies and leverages API names to complement the features, since past studies have shown that specific features are often correlated to specific API names (Bavota et al. 2015; Nguyen et al. 2016). Then, App Profile Builder combines features in descriptions and API names together to generate profiles for apps. When a new app with initial features is taken into SAFER, App Profile Builder also generates a profile for the new app. Next, Topic Model is learned from the extracted app profiles to get topic distributions for both apps in Reference App Repository and the new app, and Similar App Identifier leverages the topic distributions to identify similar apps for the new app, thus breaking through the s*imilar app identification* challenge. Finally, Feature Recommender aggregates and ranks all the features of the identified similar apps, thus generating a feature list for the new app. Associated with each recommended feature, SAFER also presents its apps ranked by user ratings. 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 to form the Reference App Repository. Out of them, we have volunteers create an annotated dataset of 100 apps for testing. The descriptions of these selected apps consist

of 1,218 sentences, and 533 sentences are identified as a golden set of features. Evaluated on the annotated dataset, the effectiveness of AFE is tested and it can achieve a *Recall* value of 80.27% and a *Precision* value of 61.79%. Experimental results over the annotated dataset verify that SAFER can well recommend features to an app from the 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 baseline approach KNN+ by 17.23% in terms of Hit@15. Besides, by comparing with an advanced technique CLAP, which aims to recommend features from user reviews, we can find that SAFER significantly outperforms CLAP and improves Hit@15 by 23.54% on average.

In summary, this study makes the following contributions:

- —We collect 8,359 apps and build a new manually annotated dataset containing the features from 100 apps. A total of 533 features (i.e., feature-describing sentences) are identified. We have made this dataset publicly available for academic research.[1]
- —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, meanwhile filter out a majority of non-feature-describing sentences.
- —We propose a novel approach named SAFER to recommend new features for apps. SAFER can effectively recommend new features by identifying similar apps and recommending missing features that are implemented by similar apps.

This article 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 show 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 illustrate the experimental design and empirical results, respectively. We discuss the threats to validity in Section 8. Then, we mention related work in Section 9. Finally, we conclude this article in Section 10.

2 MOTIVATION

In this section, we present the motivation of this study by describing the usage scenario of our proposed approach SAFER (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 Figure 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 names of these apps. With both features and API names, SAFER identifies similar apps and recommends a ranked list of new features to the new app from the descriptions of these similar apps. Developers can check the recommended features from the top to the bottom to identify new features that should be implemented in the new app. In addition, for every recommended feature, we also present three similar apps that 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.

[1] http://oscar-lab.org/SAFER/.

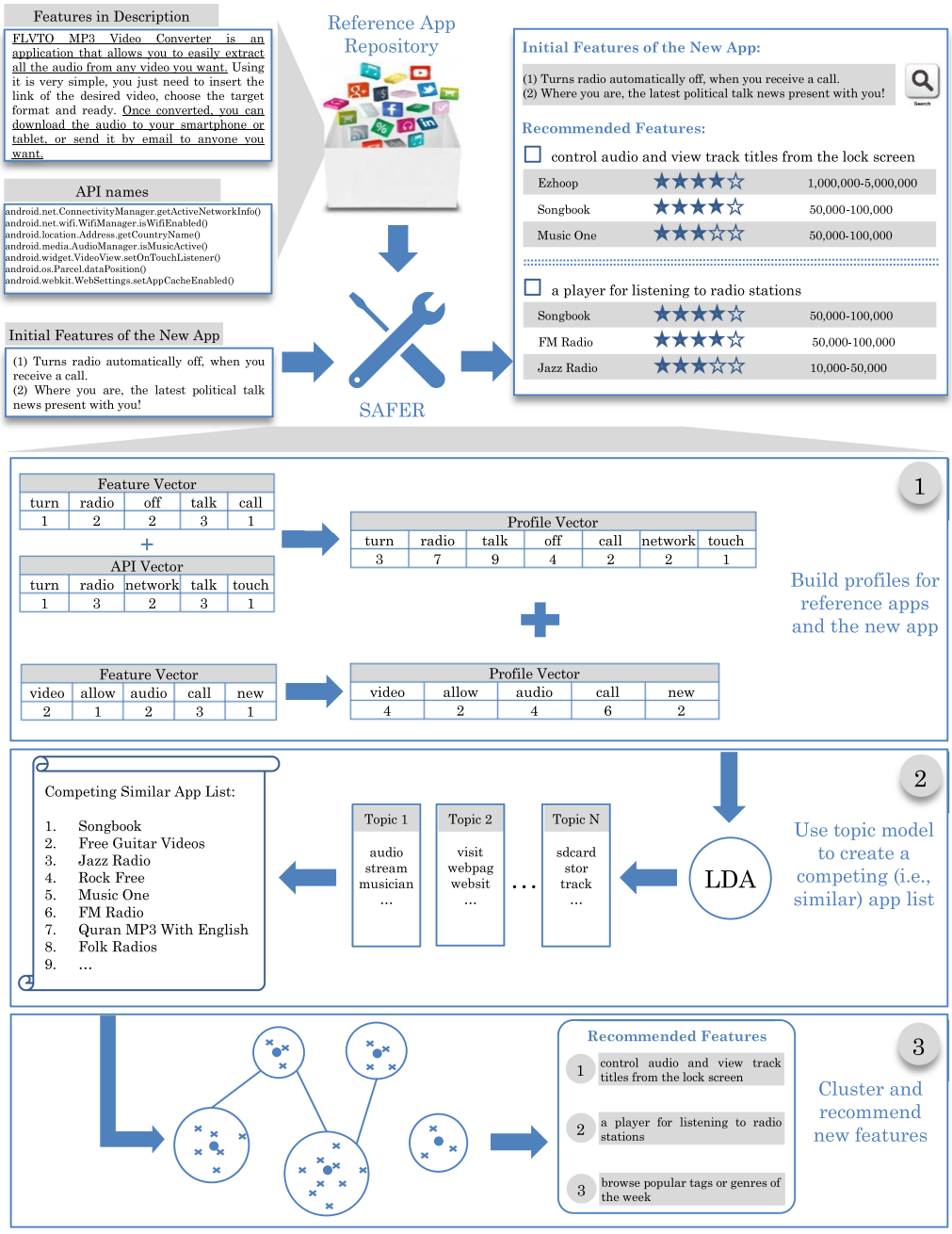


Fig. 1. The application scenario of SAFER.

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 1). The first question

Table 1. A Survey Investigating New Feature Identification Methods

*Q1: When developing apps, will you examine the features of other apps of the same product type?
A. Yes
B. No
+Q2: How do you find new features to implement?
A. Websites like Softpedia.com.
B. Similar app descriptions in app repositories, e.g., Google Play.
C. Customers' feedback/reviews of my own app.
D. Websites of similar apps.
E. Others: _____ [please provide more information].

*: exclusive choice; +: multiple choices

Table 2. The Results of the Survey

#	A	B	C	D	Other answers for Q2
Q1	86.1%	13.9%	-	-	1. Testing competitor apps
					2. Brainstorm
Q2	4.3%	58.3%	46.1%	20.9%	3. Social media

investigates whether developers examine the 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 in which developers identify new features. To reduce the response time and improve the response rate, we provide some predefined options with one blank option for developers to choose. If there is no satisfactory answer, then they can also provide their own answers based on their own methods of identifying new features. Other surveys also employ the same method to organize the contents of surveys, namely, predefined options combined with an undefined option (Zimmermann et al. 2010). These answer options are set up in a crowd-sourcing mechanism. We recruit 10 app developers to participate in it, and they are required to provide the sources to find new features to implement. By collecting and merging the results, the answer options are formed. In such a way, we think that the inducement to the participants is reduced as much as possible.

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. Eventually, we totally collect 5,610 distinct contact addresses of 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.

We receive 115 responses and summarize the responses in Table 2. There are three potential reasons for the response rate of ~2%. First, some developers may be unwilling to share their methods of acquiring new features, since they may view them as confidential data. Second, around 15% of emails are bounced back. Third, some app email addresses may be maintained by the 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 the features of apps in the same product type (Q1 in Table 2). Hence, features of apps from the same product type are important pieces of information to app developers when developing

new apps. Second, app descriptions are the main sources for developers to find new features. Customers' reviews of their apps are other important sources for new features (Gao et al. 2018). Hence, even though there is no customer review available for new apps with only some initial features, in this study, we also employ a baseline approach CLAP to compare (Scalabrino et al. 2019). 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.

After demonstrating the developer survey, we explain the difference between app descriptions and other informal documents (e.g., the requirement documents), which may make app description a good source to acquire and recommend new features. First, app descriptions are user-oriented, whereas informal documents are developer-oriented. In such a way, the features described in app descriptions are the most attractive functionalities, which are easy to understand. In contrast, the contents in informal documents involve too many technical details, which make them hard to understand. Second, app descriptions can be obtained easily, since they are publicly open to the users. Whereas, informal documents are usually not publicly open, due to the business privacies. The unique characteristics of app description make it a great source of acquiring and recommending new features.

In conclusion, most developers examine features from apps of the same product type, and this is a common action that developers identify new features from similar app descriptions to implement. These findings motivate us to propose a new approach to recommend new features mined from the 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 use as an input to our approach SAFER. Next, we present a set of apps, whose features have been manually identified, to test the results of AFE and SAFER.

3.1 Reference App Repository

We create a repository of reference apps that we use as an input to SAFER. To create this repository, we select five representative categories in Google Play, namely, *Business, Education, Health and Fitness, Finance*, and *Music and Audio*. We select these categories as case studies to validate the effectiveness of SAFER, since there are a large number of apps with plentiful distinctive features in these categories. We download a collection of 8,359 apps from Google Play with a tool, namely, Google Play Unofficial Python API.[2] This tool can not only download the APK file of each app but also obtain its description, user rating, and so on. To download apps and their information, a category name should be specified. 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. After removing the duplicate apps, we eventually obtain a dataset with 8,359 apps in total belonging to the five different categories. This repository of apps is used as an input of our approach to recommend new features to new apps.

Table 3 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, maximum, minimum, and standard deviation of number of sentences in the descriptions of the apps. In addition, we also present the average number of API names that an app makes for each category.

[2] https://github.com/egirault/googleplay-api.

Table 3. The Characteristics of the Reference App Repository

Category	# of Apps	Sent. in Descriptions of Apps				Avg. # of API names
		Avg. #	Max. #	Min. #	Std. Dev.	
Business	2,118	5.24	124	1	6.35	2,437
Education	1,822	7.18	113	1	8.16	1,964
Health and Fitness	1,545	6.97	65	1	6.93	2,468
Finance	1,796	5.48	66	1	6.07	2,336
Music and Audio	1,078	6.85	182	1	10.44	2,208

Table 4. The Statistics of the Annotated Golden Features

Categories	Golden Features		
	Ave. #	Max #	Std. Dev.
Business	5.85	10	2.10
Education	6.40	12	2.75
Health and Fitness	5.00	8	1.76
Finance	6.10	11	2.23
Music and Audio	3.30	8	2.00

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 different feature recommendation systems.

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 the features of apps from their descriptions. Before the annotation process, each volunteer is asked to read an annotation guideline to explain the annotation procedure, criteria, and an example. After these volunteers get familiar with the whole process, they are requested to pick out the sentences detailing features. First, the descriptions are segmented into sentences with an open tool named Lingpipe.[3] Then, each sentence is given to three different volunteers to obtain convincing results and reduce the impact of a single volunteer. These volunteers are required to pick out the sentences detailing the features in the descriptions. When two or three volunteers agree on a sentence detailing a feature, this sentence is regarded as a *golden feature*, which is the sentence describing a functionality or a service that an app can provide for users. Finally, we collect the annotation results from these volunteers to form our *Annotated Feature Dataset* (AFD).

Due to the large number of apps in the 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. Randomly selecting apps is a simple but effective method to reduce the sampling bias. The descriptions of these apps contain 1,218 sentences in total, and for each category, more than 200 sentences need to be annotated by three different volunteers. At the end of the annotation process, we have annotated 533 sentences detailing the features of the 100 apps (golden features). As shown in Table 4, the number of golden features varies among categories. For example, the average number and the maximum number of the golden features are 5.85 and

[3] http://alias-i.com/lingpipe/.

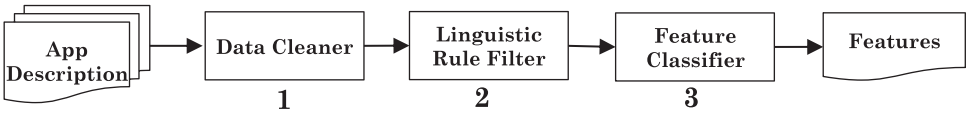


Fig. 2. The components of AFE.

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 calculate the Fleiss' Kappa agreement between volunteers on evaluating the golden features and 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 they are only used as new apps to test the effectiveness of AFE and SAFER.

4 APP FEATURE EXTRACTOR (AFE)

Automatically extracting feature-describing sentences from app descriptions is non-trivial. 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 Figure 2). AFE first splits the descriptions of apps into sentences and removes the noisy sentences (the data cleaner component). Then, AFE filters out the remaining sentences based on some linguistic rules (the linguistic rule filter component). Finally, AFE employs a classifier to discriminate sentences that describe features from those that do not (the feature classifier component). More details of these components are presented below:

Data Cleaner. Data cleaner first uses LingPipe 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 are usually used to ask questions or seek for help and seldom describe features based on our observation. Finally, sentences containing either email addresses or website URLs are removed, since they also typically do not describe app features but rather contact information.

Linguistic Rule Filter. Linguistic rule filter removes additional sentences based on the Part-Of-Speech (POS) tags of the constituent words in the sentences. Linguistic rule filter is proposed based on both literature review and data analysis. On the one hand, 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. On the other hand, 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 5) to capture feature-describing sentences. The second column of Table 5 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 the Penn Treebank Tags[4]. In the third column of Table 5, 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, then 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, Naïve Bayes classifier in this study. We also verify several commonly used classifiers, i.e., Decision Tree, Support Vector Machine (SVM), Random Forest, and Adaboost. Among them, the Naïve Bayes classifier achieves the

[4]http://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html.

Table 5. The Linguistic Rules with Examples

#	Linguistic Rules	Examples
1	<VB><NN>	watch the video
2	<NN><VB>	promotion offered
3	<JJ><NN>	3D live wallpaper
4	<NN><JJ>	user editable and changeable
5	<JJ><VB>	lazy monitor
6	<VB><JJ>	pause when asleep
7	<VB><VB>	share and collect
8	<NN><NN>	temperature measurement
9	<JJ><JJ>	safe and fast

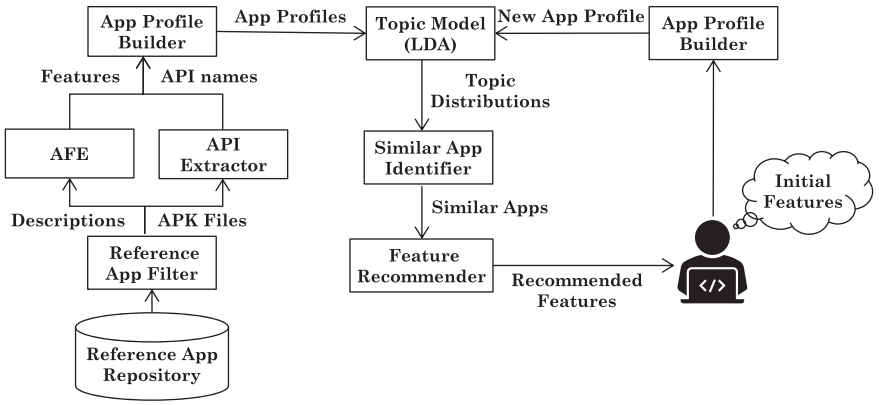


Fig. 3. The framework of SAFER.

best results. For example, leveraging the Naïve Bayes classifier, we can achieve a *F-Measure* value of 69.11% in detecting feature-describing sentences. In contrast, the *F-Measure* values of the other tested classifiers are all less than 69%. 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, then 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 2,152 remaining sentences. Then, we manually label them and eventually obtain a set of 1,073 feature describing sentences (positive sentences) and 1,079 non-feature-describing sentences (negative sentences). With such a training set, we use the Naïve Bayes classification algorithm that is implemented in Weka (Hall et al. 2009) to learn a classifier. 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.

5 SIMILAR APP-BASED FEATURE RECOMMENDER (SAFER)

In this section, we present the framework of SAFER with its main components (see Figure 3). Given a new app, SAFER recommends features for it by analyzing 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 API names using the API Extractor component (see Section 5.2). Then, SAFER builds a profile for each reference app based on its features and API names (see Section 5.3). Topic model 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 taken into the topic model to identify similar apps (see Section 5.5). At last, the features of the similar apps are recommended for this new app (see Section 5.6). We elaborate the 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 quality of the similar apps should be taken into consideration. To make a new app successful, developers 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; Tian et al. 2015), we employ the user rating of an app as the indicator for the quality and the success of the app. We set up two filtering criteria to remove low quality apps:

1. The average user rating of a selected app should be more than 3. A user rating ranges from 0 to 5 in Google Play, and we choose 3 as the threshold. The higher the user rating 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 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, then it is selected. Otherwise, it is filtered out.

5.2 API Extractor

To better characterize an app, we extract API names that an app makes to complement features extracted from the description of the app. We consider API names, since implementing a feature generally involves the usage of some specific APIs (Bavota et al. 2015; Nguyen et al. 2016; Charrada et al. 2015; Heimdahl et al. 1997). Mobile apps implementing a similar feature are likely to share a substantial proportion of API names (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 in Android PacKages (APKs), API names will not change (Ruiz et al. 2012).

API Extractor extracts API names 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 name extraction. We leverage the JClassInfo[6] toolkit to extract API names from .class files.

Since there are plenty of API names in each app, we perform several additional heuristic steps to reduce the number of API names and uncover important and representative ones (Shabtai et al.

[5] http://code.google.com/p/dex2jar.
[6] http://jclassinfo.sourceforge.net.

2010). First, we remove invocations of methods that appear in *generic* API packages that are likely to have little relevance with *specific* app features. We investigate the functions of each API package in the Android API reference and check whether they have a strong relationship with the features of apps. Finally, we pick out the *generic* API packages through an open discussion by the authors. For example, the aim of the package "android.support" is to make an app compatible with old Android releases, and the aim of the package "android.database" is to allow an app to manipulate a database. The other *generic* API packages include "android.utils" and "android.sax." Second, inspired from the IDF term weighting scheme from the Information Retrieval (IR) domain, we filter out some non-discriminant API names that are commonly used across all the apps. We count the number of apps an API name appears, and rank all API names based on it. We treat the top 5% API names as non-discriminant API names and filter them out. Third, we only consider the class name and the method name of an API name 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 API names 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 features of the app extracted from its description using AFE and (2) the API names of the app. More specifically, App Profile Builder first creates an API vector and a feature vector for every app separately, and then combines the two vectors together to create a profile for this app.

Given all the API names invoked by an app, App Profile Builder performs a series of Natural Language Processing (NLP) 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 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:

$$W_{i \text{ in profile}} = 2 \times W_{i \text{ in feature}} + 1 \times W_{i \text{ in API}}, \tag{1}$$

where $W_{i \text{ in profile}}$ is the weight of the ith term in the profile, while $W_{i \text{ in feature}}$ and $W_{i \text{ in API}}$ are the weights of the ith term in the feature and API vectors 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 the features extracted from descriptions to be more important than API names. Two reasons lead us to make such a choice. On the one hand, the features in app descriptions are expressed in natural language, which is convenient for developers to read and understand (Hierons et al. 2016). On the other hand, there are plentiful API names invocated by apps, whereas the features in descriptions are relatively infrequent. To reduce the influence of API names, we give API names a lower weight than the features in descriptions.

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 Latent Dirichlet Allocation (LDA) to identify the topic distribution of each app profile to identify similar apps. Two apps with similar

Table 6. The Topics and Topic Distributions of Some Apps

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

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 favorite 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.

topic distributions are likely to belong to the same product type (Gorla et al. 2014). We utilize 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 in TMT. Actually, we also test other values for the number of topics (e.g., 15 and 45) and find that the number of topics setting up to 30 achieves the best results. Better results may be achieved if we calibrate these parameters carefully.

Table 6 shows an example of 10 main topics with representative stemmed terms mined by LDA from the 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 6, we list the topic distributions of two apps, namely, *Property Milestone* and *Gig Harbor Real Estate*. Below Table 6, we also present the manually extracted golden features from the descriptions of the two apps. As shown in Table 6, 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 reference apps in the Reference App Repository. More specifically, Similar App Identifier computes the cosine similarity between the topic distributions of the new app and those of every reference app, and ranks all the reference apps in a descending order based on 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 7. The Algorithm of Feature Recommendation

Feature Recommendation Algorithm	
Input: clusters and their connections	
Output: the ranked recommended features	
1	find all the occupied clusters;
2	locate all the neighboring clusters of occupied clusters;
3	rank exemplars of neighboring clusters based on their weights;
4	**if** (two exemplars have the same weight)
5	rank the two exemplars based on their sizes;
6	**end if;**
7	**if** (the number of recommended features is less than N)
8	rank the rest exemplars of clusters based on their sizes;
9	**end if;**

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

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

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 the 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 similar 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 an 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, the app *Handy Lyrics* and the app *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 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 simultaneously. 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 7. 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 a descending order based on the maximum weights of the lines linked with the *occupied clusters*. If two neighbors have the same weight, then they are ranked based on their sizes. If the number of the neighbors of the *occupied clusters* is less than N, then Feature Recommender

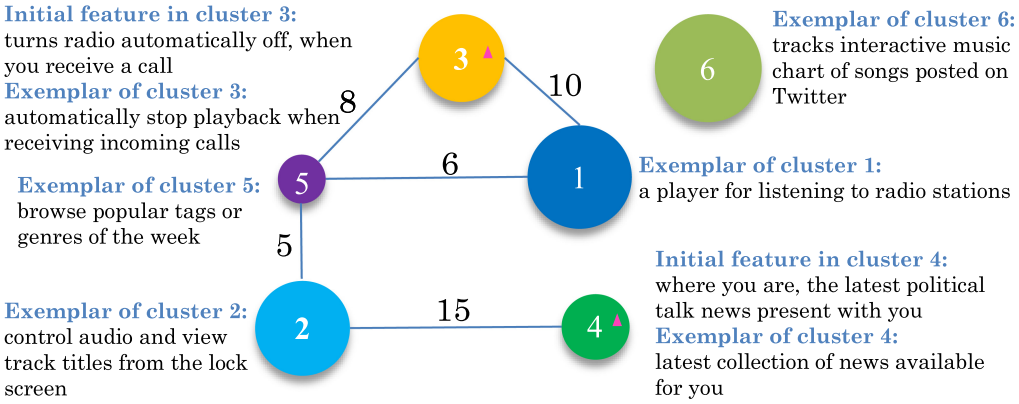


Fig. 4. An example of the resulted clusters.

also includes and ranks non-neighboring clusters based on their sizes. By default, N is set to 15, since recommending a small list is a common practice in software engineering (Wang et al. 2008). The recommendation list is not so large so that developers can check them from the top to the bottom sequentially without taking much time. In addition, SAFER associates each feature with its apps ranked by user ratings. In such a way, we can obtain the ranked recommended features.

Taking Figure 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, "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, baselines, 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 implement 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. We set K = 60 as the default parameter value. In Section 7.1, we will show the impact of modifying K.

6.2 Baselines

KNN+. As discussed in Section 1, no existing method in the literature can recommend new features for an app from the descriptions of similar apps in app markets. The most similar approach is KNN+, which aims 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 created and listed explicitly in bullet-point form and categorized into fine-grained product type. KNN+ thus does not address the *feature identification* challenge. Furthermore, in the absence of fine-grained product types (Google Play does not have fine-grained app categories), KNN+ 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 the input to KNN+ to address the *feature identification* challenge. We use the apps in the same category in the Reference App Repository as the product type for KNN+ to address the *similar app identification* challenge.

CLAP. Even though there is no user review for new apps with only initial features, we also try to compare SAFER against one of typical studies, which try to recommend features from user reviews, to show which source (description or user review) is better to acquire and recommend features. We employ a recent and similar study as another baseline approach, namely, CLAP (Scalabrino et al. 2019). CLAP aims to analyze user reviews for release planning. It consists of three main components to categorize, cluster, and prioritize user reviews. In the first component, CLAP uses the Random Forest classifier to automatically categorize user reviews into *new features, bug reports*, and *others*. Then, it clusters related reviews together using DBSCAN in the second component. Finally, CLAP prioritizes the user reviews to be implemented in the next release. In this study, we consider the top ranked user reviews (i.e., clusters with larger sizes) in the *new features* category as the recommended new features. The same as SAFER, we also evaluate the top 15 ranked new features for CLAP. If there are less than 15 recommended new features, then we evaluate all of them.

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. Inspired from the baseline approach KNN+ and for fair comparison, we employ the same *elimination-recovery* evaluation method to verify the two feature recommendation approaches SAFER and KNN+. Both SAFER and KNN+ are evaluated using the same evaluation method, so the better approach (SAFER or KNN+) can be shown. The *elimination-recovery* method works as follows. For each golden feature f in the golden feature set F of app A, we eliminate f from the set F and let a feature recommender takes in all the remaining features in $F\text{-}\{f\}$ 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 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.

We conduct the leave-one-out (LOO) test over each category of apps. More specifically, we choose every app A with golden features in AFD (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. This *elimination-recovery* method tested by LOO could reflect the real scenarios to some extent when developing new apps.

Table 8. The Confusion Matrix

		True Condition	
		Positive	Negative
Predicted Condition	Positive	True Positive (TP)	False Positive (FP)
	Negative	False Negative (FN)	True Negative (TN)

For the 20 randomly selected apps in each category in AFD, we also download their user reviews submitted before the release date from Google Play. By using CLAP to analyze and prioritize these downloaded user reviews, we can evaluate whether these user reviews can be used to acquire the new features. Different from SAFER and KNN+, CLAP tries to find features from user reviews. By comparing the results of the SAFER and CLAP, we can know which source (description or user review) is better to recommend features. In addition, by comparing the results of SAFER and KNN+, we can learn which approach is better to recommend features from descriptions.

6.4 Evaluation Metrics

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

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

$$\text{Hit Ratio} = \frac{\#\text{ of hit features}}{\#\text{ of features}} \times 100\% \tag{3}$$

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 (IR) and recommendation systems (Jiang et al. 2019). *NDCG* can fully evaluate a recommended list of results from the top ranked result to the bottom ranked one with the gain of each result discounted by lower ranks. *NDCG* is defined as follows:

$$\text{NDCG} = \frac{\text{G}}{\text{ideal G}}, \quad \text{G} = \sum_{\text{i}=1}^{15} \frac{2^{\text{score}_\text{i}} - 1}{\log_2(\text{i}+1)} \tag{4}$$

where 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 (Mcmillan 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. The comparison results between the true condition and the predicted condition can be shown in a confusion matrix in Table 8. Based on the confusion matrix, *Precision, Recall, F-Measure*, and *Accuracy* can be calculated as follows: >

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \tag{5}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \tag{6}$$

$$\text{F-Measure} = \frac{2 \times Precision \times Recall}{Precision + Recall}, \tag{7}$$

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}. \tag{8}$$

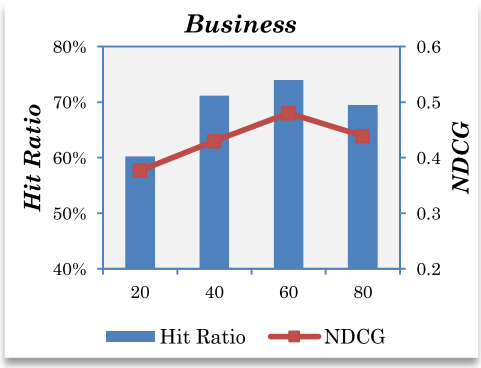


Fig. 5. The results of *Business*.

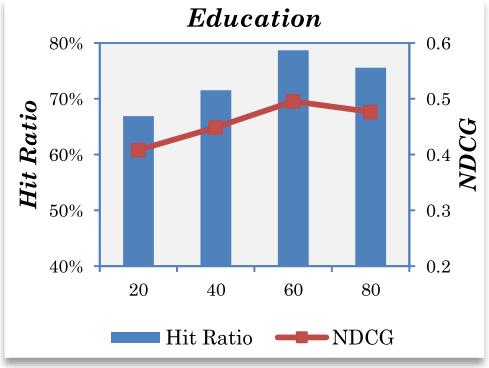


Fig. 6. The results of *Education*.

7 EXPERIMENTAL RESULTS

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

7.1 RQ1: How does the parameter K influence the performance of SAFER?

Motivation. There is a parameter in SAFER, namely, K, which is the number of the 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 the performance of SAFER.

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 Figures 5 and 6, we present the experimental results of SAFER over the *Business* and *Education* categories, respectively, for different values of K. As shown in Figures 5 and 6, the performance of SAFER varies along with the change 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* value 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) values 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 create 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. Hence, K is kept as 60 in the following RQs.

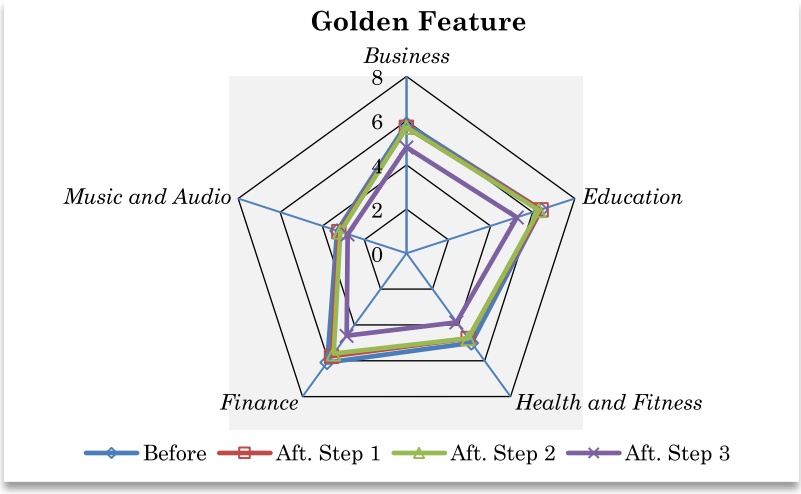


Fig. 7. The average number of feature-describing sentences after each step of AFE.

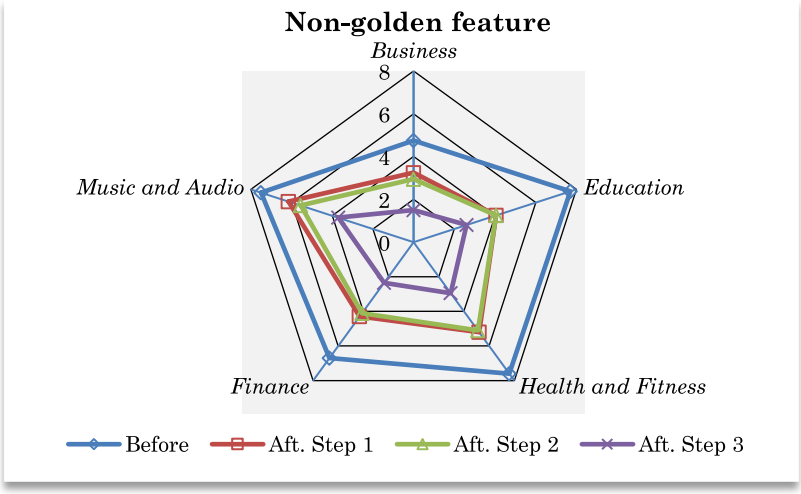


Fig. 8. The average number of non-feature-describing sentences after each step of AFE.

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 try 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 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

Table 9. The Results of AFE for Each Category

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

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, Figures 7 and 8 present the average number of golden features and non-golden features remained before processing, after applying the data cleaner, after applying the linguistic rule filter, and after applying the 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 golden features as possible, while retain as few non-golden features as possible. By comparing Figures 7 and 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.

Table 9 summarizes 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* value of 76.19% and a *Recall* value 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 *F-Measure* value of 69.11% and an average *Accuracy* value of 69.57%. That is to say that AFE is an effective tool to distinguish features from non-features in the descriptions of apps.

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 RQ3: Does the introduction of API names contribute positively to SAFER?

Motivation. SAFER tries to combine features mined from app descriptions and API names to construct a profile for an app. In this RQ, we try to explore whether API names can complement features extracted from app descriptions.

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

Results. Table 10 shows the comparison results between SAFER and SAFER$^{\text{-API}}$. In terms of *Hit Ratio*, we can see that SAFER achieves better results than SAFER$^{\text{-API}}$ in most of the categories, except

Table 10. The Comparison Results Between SAFER and SAFER^{-API}

Category	*Hit Ratio*		*NDCG*	
	SAFER	SAFER^{-API}	SAFER	SAFER^{-API}
Business	73.99%	69.30%	0.4802	0.4356
Education	78.68%	74.30%	0.4955	0.4697
Health and Fitness	69.92%	68.69%	0.4536	0.4431
Finance	54.49%	55.13%	0.3714	0.3696
Music and Audio	64.38%	61.25%	0.4327	0.4227
Average	68.29%	65.73%	0.4467	0.4281

for the category *Finance,* where the difference is trivial. For example, SAFER achieves a *Hit Ratio* value of 73.99% and improves SAFER^{-API} by 4.69% for the category *Business*. When considering the category *Finance*, SAFER^{-API} only outperforms SAFER by 0.64%. On average, SAFER outperforms SAFER^{-API} by 2.56%. In terms of *NDCG*, we can find that SAFER is superior to SAFER^{-API} in all the categories. Overall, SAFER outperforms SAFER^{-API} by 0.0186 in terms of *NDCG*.

Conclusion. Taking API names into consideration can improve the results of SAFER. API names can be used as good complements to the extracted features from app descriptions.

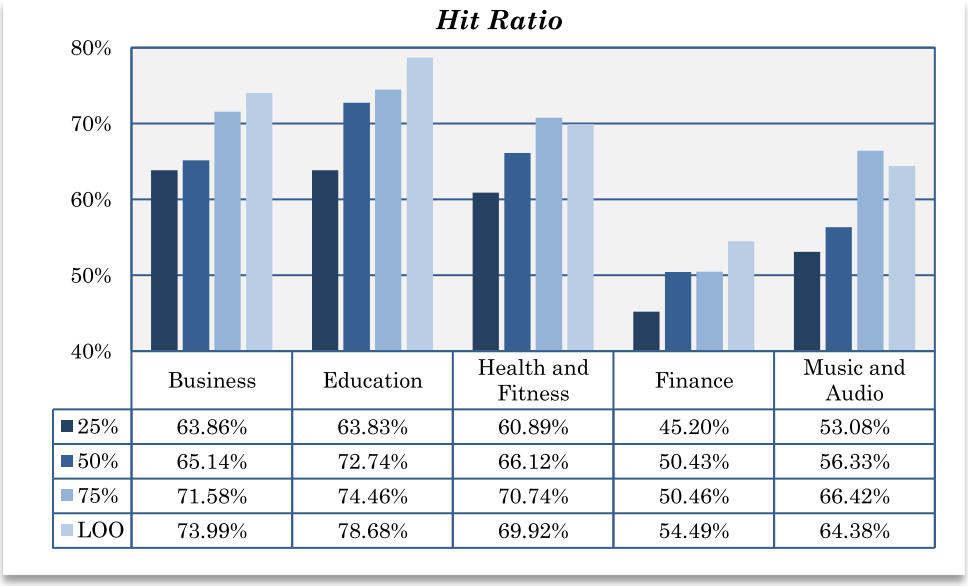
7.4 RQ4: To what extent can we obtain reasonable results when given a small set of initial features?

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 the performance of SAFER depends on a small set of initial features. Given a small set of initial features, we can test the performance of SAFER in extreme situations to show its robustness.

Approach. In the real development process, developers may conceive different number of initial features. In this RQ, we investigate the effectiveness of SAFER when we vary the number of initial features. Since we want to investigate the performance of SAFER starting with a small set of features, we predefine some values by adjusting different number of golden features, namely, {25%, 50%, 75%, and leave-one-out (LOO)}, as initial features. Since the average number of golden features is only 5.33 in the app descriptions in AFD, by selecting 25% and 50% features as initial features, we can test SAFER in extreme conditions (i.e., given less than 3 initial features).

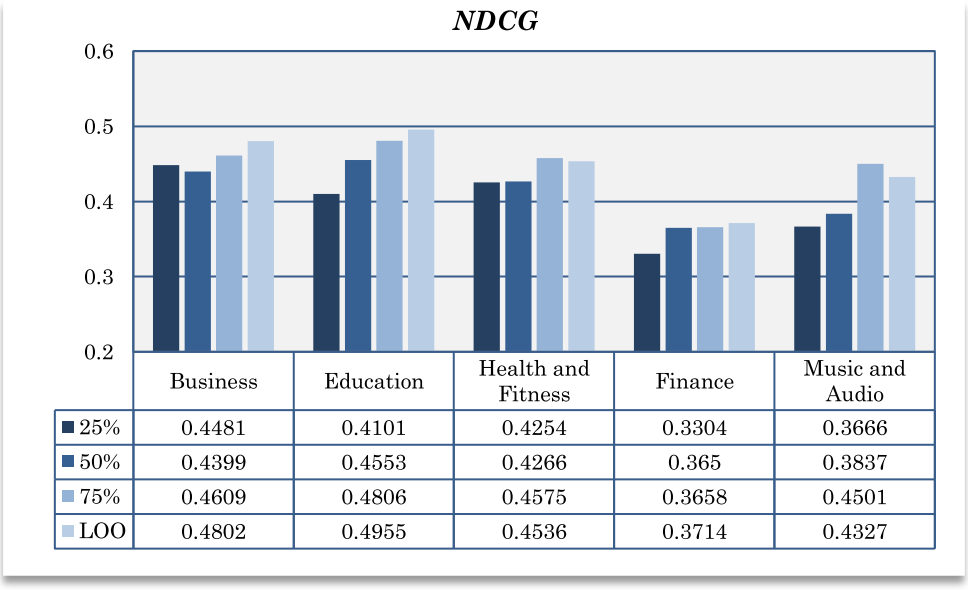
Results. Figures 9 and 10 show the results of *Hit Ratio* and *NDCG*, respectively. We can see from the figures that, as the percentage increases, both the values of *Hit Ratio* and *NDCG* show upward trends. For example, for the category *Business*, when the inputs are 25% of all the golden features, the *Hit Ratio* value is 63.86%. When the percentage is increased to 75%, the *Hit Ratio* value is 71.58%. When using LOO, the *Hit Ratio* value is 73.99%. We can also observe the same phenomenon for the other categories with some exceptions. 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 profile of the app. Hence, the recommended features have high probabilities to hit the target features. In addition, we can also 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 initial features).

When analyzing the reasons why SAFER performs well in some categories while not in the others, we find that the dispersion of the features in the reference apps could influence the results of SAFER. Intuitively, the more dispersive of the features the reference apps have for a category, the more difficult for SAFER to detect similar apps, so the less likely that SAFER can recommend



	Business	Education	Health and Fitness	Finance	Music and Audio
25%	63.86%	63.83%	60.89%	45.20%	53.08%
50%	65.14%	72.74%	66.12%	50.43%	56.33%
75%	71.58%	74.46%	70.74%	50.46%	66.42%
LOO	73.99%	78.68%	69.92%	54.49%	64.38%

Fig. 9. The *Hit Ratio* results for different percentages of golden features used as initial inputs.



	Business	Education	Health and Fitness	Finance	Music and Audio
25%	0.4481	0.4101	0.4254	0.3304	0.3666
50%	0.4399	0.4553	0.4266	0.365	0.3837
75%	0.4609	0.4806	0.4575	0.3658	0.4501
LOO	0.4802	0.4955	0.4536	0.3714	0.4327

Fig. 10. The *NDCG* results for different percentages of golden features used as initial inputs.

features for the test apps in the same category. To explore the dispersion of the features in the reference apps for a category, we first use AFE to identify features for all the apps in the Reference App Repository. Then, we calculate the average number of features with its standard deviation in the reference apps. We find that the average number of features is similar in all the categories. However, the *Finance* category and the *Music and Audio* category achieve the two largest standard

Table 11. The Comparison Results Between $SAFER_{v1}$ and KNN+

Category	*Hit Ratio*		*NDCG*	
	$SAFER_{v1}$	KNN+	$SAFER_{v1}$	KNN+
Business	73.99%	64.79%	0.4802	0.4258
Education	78.68%	56.10%	0.4955	0.4090
Health and Fitness	69.92%	54.70%	0.4536	0.3703
Finance	54.49%	38.01%	0.3714	0.3154
Music and Audio	64.38%	41.71%	0.4327	0.3476
Average	68.29%	51.06%	0.4467	0.3736

deviation values in all the categories in terms of the number of features in the reference apps, i.e., 16.97 and 5.83. It means that the number of features is spread out over a wide range of values in the two categories, making SAFER hard to detect similar apps and recommend features. Hence, SAFER does not perform well in the two categories.

Conclusion. Along with the growth of the number of initial features, the *Hit Ratio* and *NDCG* values increase. Even given a small set of initial features, SAFER still performs well and robustly.

7.5 RQ5: Can SAFER outperform the two baseline approaches?

Motivation. KNN+ can be adapted to recommend new features for apps as SAFER, and CLAP can resolve the same problem by acquiring features from user reviews. In this RQ, we investigate whether SAFER can achieve better results than the adapted KNN+ and CLAP. By investigating this RQ, we can learn which source is better to find features (description or user review) and which approach is better to recommend features from descriptions.

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+. At the same time, we collect the recommendation results from CLAP for the 100 apps in five categories in AFD. However, due to two reasons, CLAP cannot recommend features for all the 100 apps. First, there is no user review received before the release time of these apps. Different from our notion, some apps do not receive many user reviews, and only the popular and widely used apps can get plentiful user reviews. Second, since CLAP classifies user reviews into *new features, bug reports*, and *others*, no user review is classified into the *new features* category possibly due to that users tend to submit *bug reports* or *others*. Finally, CLAP can only recommend features for 36 apps in AFD. We collect the results from CLAP and compare it against SAFER. Hence, we first compare SAFER against KNN+ in the 100 apps in AFD. Then, we compare SAFER against CLAP in 36 apps in AFD.

In addition, we introduce the paired Wilcoxon signed rank test to explore the statistical significance of the difference between the performance of SAFER and KNN+, and between SAFER and CLAP in recommending features for new apps. We formulate the two test hypotheses as follows:

H_0: There is no significant difference between the performance of the two approaches.
H_1: 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 corresponding evaluation metric.

Results. Since the apps are different in KNN+ and CLAP for a category, we compare SAFER against KNN+ and CLAP separately, and give SAFER different version (i.e., $SAFER_{V1}$ in Table 11

Table 12. The Comparison Results Between $SAFER_{V2}$ and CLAP

Category	*Hit Ratio*		*NDCG*	
	$SAFER_{V2}$	CLAP	$SAFER_{V2}$	CLAP
Business	78.27%	60.00%	0.5074	0.4003
Education	81.44%	58.57%	0.5867	0.4657
Health and Fitness	77.50%	57.50%	0.5563	0.3874
Finance	71.54%	38.54%	0.4721	0.3239
Music and Audio	71.19%	47.62%	0.4936	0.2776
Average	75.99%	52.45%	0.5232	0.3710

and $SAFER_{V2}$ in Table 12). The results of SAFER and KNN+ are summarized in Table 11. As shown in Table 11, 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*.

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* values than KNN+ on average, we can conclude that SAFER is superior to KNN+.

The results of SAFER and CLAP are shown in Table 12. We can see that SAFER shows its advantages in all the categories. For example, SAFER achieves a *Hit Ratio* value of 78.27% in the *Business* Category. In contrast, CLAP only achieves 60.00% in the same situation. On average, SAFER achieves 75.99% and outperforms CLAP by 23.54% in terms of *Hit Ratio*. We can find similar results in terms of *NDCG*. For instance, SAFER is superior to CLAP by 0.1522 on average. We also conduct the Wilcoxon test for *Hit Ratio* and *NDCG*, and the p-values are 0.003 and 0.009, respectively. It means that there exits significant difference between SAFER and CLAP, and SAFER performs better than CLAP.

On average, the time to recommend features for each test app takes less than one minute. Along with the increase of the number of reference apps, the time required for SAFER to recommend features for new apps only has a linear growth. Hence, SAFER can recommend features within acceptable time, considering that there are thousands of apps in the Reference App Repository. After demonstrating the comparison results, we would like to explain why SAFER could perform better in recommending features from mobile app descriptions. First, SAFER is specially designed for recommending features for mobile apps. SAFER fully leverages the domain specific knowledge to retain high quality apps and detect similar apps, e.g., user rating, rating count, and API names, so it can recommend features accurately. Second, AFE is built to precisely extract pure feature lists from app descriptions, which can break through the *Feature Identification* challenge. Taking the pure features as input, SAFER does not suffer from the noise to achieve better results. Third, SAFER introduces API names as complements for features. Meanwhile, it utilizes topic model to achieve similar apps to break through the *Similar App Identification* challenge.

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. In addition, by

comparing the results of SAFER and CLAP, we find that app description is good source to acquire and recommend features.

8 THREATS TO VALIDITY

In this section, we introduce the threats to the validity, including the threats to internal validity and external validity.

Internal Validity. Since no dataset of golden features is available, we have volunteers annotate a golden feature dataset and vote the resulting list of features. The backgrounds and personal opinions of volunteers may influence the golden feature annotation and the evaluation of the results. To reduce the bias of volunteers, we have three distinct volunteers annotate each feature and vote for each result. A sentence is treated as a golden feature or a recommended feature hits a golden feature if it receives at least two out of three votes. In addition, we provide them an annotation guideline with annotation criterion, and they are required to read and understand them thoroughly before conducting the experiments. In such a way, we think that this threat is reduced as much as possible.

We construct a Reference App Repository to help SAFER recommends new features for apps. The construction of the Reference App Repository may be a threat. Employing different reference apps into the repository, SAFER may have different performance. If the Reference App Repository is large enough, then all the available features are included in it. In such a situation, SAFER may perform well. It is unknown how SAFER performs when the reference apps are changed in the repository. In the future, we will explore the performance of SAFER when providing different reference apps in the repository.

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

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). This evaluation method can reflect the real development process for new apps to some extent. In addition, both SAFER and KNN+ are evaluated by the same method so that they are evaluated equally. In such a way, we can learn which approach is better.

External Validity. In this study, we utilize an annotated dataset with hundreds of features of 100 apps from five app categories to evaluate the performance of SAFER. These 100 apps from the five categories are randomly selected, so the sampling bias can be reduced. It is still uncertain how well SAFER performs over other apps of other app categories. Still, we think that 100 apps and 533 features are large enough to illustrate the performance of SAFER. In the future, we plan to explore the performance of SAFER on more apps in different categories.

SAFER aims to recommend new features for apps by inspecting and combining features from similar apps. A winning app is expected to have some special features that are not possessed in other apps. In this situation, SAFER can only provide some special combinations of features that no apps have. In contrast, CLAP may provide some unique features that are suitable for new apps. To achieve some specific features that no apps have, combining both SAFER and CLAP could be a better solution. In the future, we plan to combine SAFER and CLAP together to help developers acquire special features for apps.

Table 13. Existing Work on Feature Recommendation

#	Main Approach	Source of Features	References
1	Manual/semi-manual effort	Requirement documentation	(Kang et al. 1990; Frakes et al. 1998; Santo et al. 2009)
2	Social networks	Stakeholders	(Lim et al. 2011; Lim et al. 2010)
3	Data mining and natural language processing	Requirement specifications	(Alves et al. 2008; Chen et al. 2005)
		Forum or user feedback	(Rahimi et al. 2014; Carreño et al. 2013; Nayebi et al. 2017; Chen et al. 2014; Panichella et al. 2015; Scalabrino et al. 2019)
		Softpedia.com	(Hariri et al. 2013)

9 RELATED WORK

Extracting software features from the textual software artifacts could help to resolve many software development tasks. A number of past studies propose approaches to identify and recommend features. These approaches can be roughly classified into three categories (see Table 13), i.e., the approaches using manual/semi-manual effort, the approaches using social networks, and the approaches using data mining and natural language processing.

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). Kang et al. propose FODA to help developers conduct domain analysis (Kang et al. 1990). FOAD consists of three basic activities, i.e., context analysis, domain modelling, and architecture modelling. Based on these activities, FODS supports software developers to understand and implement applications in the domain. Frakes et al. propose DARE, a case tool to support domain analysis (Frakes et al. 1998). DARE captures domain information from experts, documents, and code. In addition, DARE also helps to find reusable domain information by providing a search mechanism.

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)). Lim et al. propose StackNet to identify and prioritize stakeholders (Lim et al. 2010). StackNet consists of three steps, including identifying stakeholders, building a stakeholder social network, and prioritizing stakeholders with some social network metrics. Furthermore, Lim et al. also propose StackRare that leverages social networks to identify and prioritize requirements in a large-scale of software projects (Lim et al. 2011). StackRare asks stakeholders to recommend relevant requirement using collaborative filtering.

Approaches in the above two categories cannot *automatically* recommend features to apps, since a lot of manual steps involving domain analysts or stakeholders are needed. In contrast, the third category leverages data mining and Natural Language Processing (NLP) 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; Nayebi et al. 2017; Chen et al. 2014; Panichella et al. 2015; Scalabrino et al. 2019), 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). For example, Alves et al. conduct an exploratory study to investigate the performance of Information Retrieval (IR) techniques on identifying feature commonalities and variabilities in a large-scale of requirement specifications (Alves et al. 2008). However, a representative repository of requirement specifications does not exist for mobile apps. Hence, a lot of researchers shift their attention to online forums and user reviews to acquire features. Rahimi and Cleland-Huang recommend features from forums (Rahimi et al. 2014). Carreño and Winbladh extract new requirements from user comments of apps (Carreño et al. 2013). Nayebi and Abran systematically review the opinion mining studies from user reviews in mobile app stores (Nayebi et al. 2017). Chen et al. propose AR-Miner to extract informative user reviews (Chen et al. 2014). Panichella et al. combine Natural Language Processing (NLP), text analysis, and sentiment analysis techniques to classify user reviews into different categories (Panichella et al. 2015). We also introduce a recent and effective approach to compare, i.e., CLAP (Scalabrino et al. 2019). CLAP categorizes, clusters, and prioritizes user reviews that need to be implemented in the subsequent app releases. 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. (namely, KNN+) so that it can work for our problem and shown in our experiments that SAFER is superior to the adapted KNN+.

10 CONCLUSION AND FUTURE WORK

The features greatly impact the success of apps. To tackle the new task of recommending features for new mobile apps, we propose a novel approach named SAFER in this study. SAFER employs the descriptions of similar apps in app markets to help developers perform domain analysis. 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 better recommend features than the baseline approaches KNN+ and CLAP, and the improvement by SAFER is statistically significant. In this study, we do not distinguish different granularities of features and employ student volunteers rather than developers from industry to evaluate the performance of SAFER, which may be limitations of SAFER and could be improved in the future.

For future works, we intend to improve SAFER in several aspects. First, we plan to explore some interesting research directions, e.g., integrating the other types of relationships among features into SAFER. Second, we plan to better address the threats to validity of this study by including more datasets and experiments. Third, we plan to combine SAFER, which recommends features from the descriptions of apps and CLAP, which recommends features by mining user reviewers together. In such a way, developers can better perform the domain analysis and develop attractive apps.

ACKNOWLEDGMENTS

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

REFERENCES

Vander Alves, Christa Schwanninger, Luciano Barbosa, Awais Rashid, Peter Sawyer, Paul Rayson, Christoph Pohl, and Andreas Rummler. 2008. An exploratory study of information retrieval techniques in domain analysis. In *Proceedings of the 12th International Conference on Software Product Line (SPLC'08)*. 67–76.

K. M. Annervaz, Vikrant Kaulgud, Shubhashis Sengupta, and Milind Savagaonkar. 2013. Natural language requirements quality analysis based on business domain models. In *Proceedings of 28th International Conference on Automated Software Engineering (ASE'13)*. 676–681.

Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2015. The impact of API change- and fault-proneness on the user ratings of android apps. *IEEE Trans. Software Eng.* 41, 4 (2015), 384–407. DOI: 10.1109/TSE.2014.2367027.

Giacomo Berardi, Andrea Esuli, Tiziano Fagni, and Fabrizio Sebastiani. 2015. Multi-store metadata-based supervised mobile app classification. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC'15)*. 585–588.

Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2011. Improving the tokenisation of identifier names. *ECOOP Object-Orient. Program.* 6812 (2011), 130–154.

Laura V. Galvis Carreño and Kristina Winbladh. 2013. Analysis of user comments: An approach for software requirements evolution. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. 582–591.

Eya Ben Charrada, Anne Koziolek, and Martin Glinz. 2015. Supporting requirements update during software evolution. *J. Software: Evol. Process* 27, 3 (2015), 166–194.

Kun Chen, Wei Zhang, Haiyan Zhao, and Hong Mei. 2005. An approach to constructing feature models based on requirements clustering. In *Proceedings of the 13th International Conference on Requirements Engineering (RE'05)*. 31–40.

Ning Chen, Jialiu Lin, Steven C. H. Hoi, Xiaokui Xiao, and Boshen Zhang. 2014. AR-miner: Mining informative reviews for developers from mobile app marketplace. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. 767–778.

Lingling Fan, Ting Su, Sen Chen et al. 2018. Large-scale analysis of framework-specific exceptions in android apps. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. 408–419.

William Frakes, Ruben Prieto Diaz, and Christopher Fox. 1998. Dare: Domain analysis and reuse environment. *Ann. Software Eng.* 5, 1 (1998), 125–141.

Brendan J. Frey and Delbert Dueck. 2007. Clustering by passing messages between data points. *Science* 315, 16 (2007), 972–976.

Cuiyun Gao, Jichuan Zeng, David Lo, Chin-Yew Lin, Michael R. Lyu, and Irwin King. 2018. INFAR: Insight extraction from app reviews. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*. Lake Buena Vista, FL, USA, 904–907.

Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. 1025–1035.

Latifa Guerrouj, Shams Azad, and Peter C. Rigby. 2015. The influence of app churn on app success and stackoverflow discussions. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER'15)*. 321–330.

Emitza Guzman and Walid Maalej. 2014. How do users like this feature? A fine-grained sentiment analysis of app reviews. In *Proceedings of the 22nd International Conference on Requirements Engineering (RE'14)*. 153–162.

Mark Hall, Ebie Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The weka data mining software: An update. *ACM SIGKDD Explor. Newslett.* 11, 1 (2009), 10–18.

Negar Hariri, Carlos Castro-Herrera, Mehdi Mirakhorli, Jane Cleland-Huang, and Bamshad Mobasher. 2013. Supporting domain analysis through mining and recommending features from online product listings. *IEEE Trans. Software Eng.* 39, 12 (Dec. 2013), 1737–1751. DOI: 10.1109/TSE.2013.39.

Mats P. E. Heimdahl and David J. Keenan. 1997. Generating code from hierarchical state-based requirements. In *Proceedings of the 3rd International Conference on Requirements Engineering (RE'97)*. 210–219.

Robert M. Hierons, Miqing Li, Xiaohui Liu, Sergio Segura, and Wei Zhang. 2016. SIP: Optimal product selection from feature models using many-objective evolutionary optimization. *ACM Trans. Software Eng. Methodol.* 25, 2, (Apr. 2016). DOI: http://dx.doi.org/10.1145/2897760

Abram Hindle, Christian Bird, Thomas Zimmermann, and Nachiappan Nagappan. 2012. Relating requirements to implementation via topic analysis: Do topics extracted from requirements make sense to managers and developers? In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM'12)*. 243–252.

He Jiang, Liming Nie, Zeyi Sun et al. 2019. ROSF: Leveraging information retrieval and supervised learning for recommending code snippets. *IEEE Trans. Serv. Comput.* 12, 1 (2019), 34–46.

He Jiang, Hongjing Ma, Zhilei Ren, Jingxuan Zhang, and Xiaochen Li. What Makes a Good App Description? In *Proceedings of the 6th Asia-Pacific Symposium on Internetware (Internetware'14)*. 45–53.

Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. Feature-oriented domain analysis (FODA) feasibility study. Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA.

Soo Ling Lim, Peter J. Bentley, Natalie Kanakam, Fuyuki Ishikawa, and Shinichi Honide. 2015. Investigating country differences in mobile app user behavior and challenges for software engineering. *IEEE Trans. Software Eng.* 41, 1 (Jan. 2015), 40-64. DOI: 10.1109/TSE.2014.2360674

Soo Ling Lim and Anthony Finkelstein. 2011. StakeRare: Using social networks and collaborative filtering for large-scale requirements elicitation. *IEEE Trans. Software Eng.* 38, 3 (2011), 707–735. DOI: 10.1109/TSE.2011.36.

Soo Ling Lim, Daniele Quercia, and Anthony Finkelstein. 2010. StakeNet: Using social networks to analyse the stakeholders of large-scale software projects. In *Proceedings of 32nd International Conference on Software Engineering (ICSE'10).* 295–304.

William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. 2017. A survey of app store analysis for software engineering. *IEEE Trans. Software Eng.* 43, 9 (2017), 817–847.

Collin Mcmillan, Denys Poshyvanyk, Mark Grechanik, Qing Xie, and Chen Fu. 2013. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *ACM Trans. Software Eng. Methodol.* 37, 4 (Oct. 2013), 1–32. DOI: 10.1145/2522920.2522930

Fairuz Amalina Narudin, Ali Feizollah, Nor Badrul Anuar, and Abdullah Gani. 2016. Evaluation of machine learning classifiers for mobile malware detection. *Soft Comput.* 20, 1 (2016), 343–357.

Necmiye Genc-Nayebi and Alain Abran. 2017. A systematic literature review: Opinion mining studies from mobile app store user reviews. *J. Syst. Software* 125 (2017), 207–219.

Anh Tuan Nguyen, Michael Hilton, Mihai Codoban et al. 2016. API code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineer (FSE'16).* 511–522.

Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A. Visaggio, Gerardo Canfora, and Harald C. Gall. 2015. How can I improve my app? Classifying user reviews for software maintenance and evolution. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution.* 281–290.

Mona Rahimi and Jane Cleland-Huang. 2014. Personas in the middle: Automated support for creating personas as focal points in feature gathering forums. In *Proceedings of 29th International Conference on Automated Software Engineering (ASE'14).* 479–484.

Israel J. Mojica Ruiz, Meiyappan Nagappan, Bram Adams, and Ahmed E. Hassan. 2012. Understanding reuse in the android market. In *Proceedings of the 19th International Conference on Program Comprehension (ICPC'12).* 113–122.

Raimundo F. Dos Santos and William B. Frakes. 2009. DAREonline: A web-based domain engineering tool. *Formal Found. Reuse Domain Eng.* 5791 (2009), 246–257.

Federica Sarro, Afnan A. Al-Subaihin, Mark Harman, Yue Jia, William Martin, and Yuanyuan Zhang. 2015. Feature lifecycles as they spread, migrate, remain, and die in app stores. In *Proceedings of the 23rd International Requirements Engineering Conference (RE'15).* 76–85.

Simone Scalabrino, Gabriele Bavota, Barbara Russo, Massimiliano Di Penta, and Rocco Oliveto. 2019. Listening to the crowd for the release planning of mobile apps. *IEEE Trans. Software Eng.* 45, 1 (2019), 68–86.

Asaf Shabtai, Yuval Fledel, and Yuval Elovici. 2010. Automated static code analysis for classifying android applications using machine learning. In *Proceedings of the International Conference on Computational Intelligence and Security (CIS'10).* 329–333.

Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. 2010. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE'10).* 45–54.

Yuan Tian, Meiyappan Nagappan, David Lo, and Ahmed E. Hassan. 2015. What are the characteristics of high-rated apps? A case study on free android applications. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'15).* 301–310.

Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. 2003. Feature-Rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology (NAACL'03).* 173–180.

Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. 2008. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08).* 461–470.

Tianyong Wu, Xi Deng, Jun Yan, and Jian Zhang. 2019. Analyses for specific defects in android applications: A survey. *Front. Comput. Sci.* To appear.

Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. 2010. What makes a good bug report? *IEEE Trans. Software Eng.* 36, 5 (Sep. 2010), 618–643. DOI: 10.1109/TSE.2010.63

Received August 2016; revised April 2019; accepted July 2019