Information Sciences 580 (2021) 129–151

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Supporting features updating of apps by analyzing similar products in App stores

Huaxiao Liu [a,b], Yihui Wang [a,b], Yuzhou Liu [a,c,*], Shanquan Gao [a,b]

[a] College of Computer Science and Technology, Jilin University, Changchun, China
[b] Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, China
[c] College of Electronic Science and Engineering, Jilin University, Changchun, China

ARTICLE INFO

Article history:
Received 18 September 2020
Received in revised form 10 August 2021
Accepted 13 August 2021
Available online 17 August 2021

Keywords:
App store mining
Feature extraction
Reviews analysis
App evolution

ABSTRACT

Facing the increasingly fierce competition, app developers have to update features of their products continually. In this process, developers need to not only consider users' demands but also pay attention to what other similar apps do so that they can stay one step ahead in the competition. App stores provide large-scale useable data for achieving this goal while how to use them efficiently becomes a new challenge for developers. In this paper, we aim at helping developers make feature updating strategies of their apps by analyzing data of similar products in App stores. Firstly, we identify similar apps by using texts in app descriptions and UI. Then, we gain and integrate the information of updated features in these apps from their release texts. Furthermore, we match reviews with the related updated features, which helps developers predict the payback if they adopt a similar updating strategy. To validate the proposed approach, we conducted a series of experiments based on Google Play. The results show that our approach can analyze the data reasonably and provide useful information for developers making feature updating strategy in the evolution of their own products.

1. Introduction

With the popularity of smartphones, the mobile is no longer just a traditional communication tool but becomes a kind of powerful computation terminal [1,2]. Mobile applications (apps) are the key factors for supporting people to fully use smartphones, and they have become one of the most important software products in our daily life. To satisfy users' different kinds of demands, the market of apps has developed rapidly in recent years [3–6]. According to a recent survey, the users in global have downloaded apps with 218 billion times and spent $143 billion in App stores in 2020, and these values have increased 12% and 42% respectively compared with 2018. The prosperity of the app market not only brings huge economic benefit, but also generates increasingly fierce competition [7–9]: since 2010, there are more than 4 million apps have released Apple Store, but only less than 2 million of them retained until 2018, this means one out of every two apps has eliminated.

To survive in the competition, app developers have to update features of their products continually to make themselves outstanding in the market. In this process, developers need to not only consider users' demands, but also pay attention to what other similar apps do so that they can stay one step ahead in the competition [10]. By analyzing similar products, developers can get a deeper insight into the market before giving or updating products, and gain actionable insights as well as

* Corresponding author at: College of Computer Science and Technology, Jilin University, Changchun, China.
E-mail address: liuyuzhou@jlu.edu.cn (Y. Liu).

https://doi.org/10.1016/j.ins.2021.08.050
0020-0255/

inspiration to improve their apps. Specifically, by understanding which features have been updated by these products in the recent versions, the developers can not only consider making similar changes to their own app, but also predict the future trend and achieve it first to occupy the market; meanwhile, by gaining the opinions proposed by the users of these products, developers could get the preferences of the potential users and make corresponding feature updating strategy to satisfy their demands.

Apps have concentrative online sale platforms, such as Google Play, Apple Store, and they provide usable data resource for analyzing the updated features (including improved features and added features) of similar products [11,12]: for one thing, apps use the descriptions as well as release texts to introduce the main features and their changes in each version, these texts are important data for us to track the evolution process of their features; for another, App stores allow users to give reviews on their downloaded apps, and this offers materials for analyzing the users' feedback on the existing feature updating actions. However, as more and more similar products have been launched, it becomes difficult for developers to analyze the increasing and large-scale data resources within an acceptable time.

To better illustrate the problem, let us consider the following scenario. John is a developer of *HERE WeGo*, a navigation app on App store. Recently, John gets a survey of the market conditions[1] on his product, which shows that the ranking of *HERE WeGo* has slipped out of 50 in 2019. This makes John anxious and he wants to know what happened to other products, so that he could make a better feature updated strategy of *HERE WeGo*. In this process, he encounters at least three obstacles:

1) John needs to firstly identify the products that need to be analyzed in App store, and collect data from them. This seems easy as the App stores have categorized the products, but John wants to select the ones more similar to *HERE WeGo* from thousands of (even more) products.
2) Secondly, John reads the release texts of these similar products to gain information of updated features. This is time-consuming because an app may have a large number of similar products, and each product often has more than one version even in recent time.
3) Eventually, John obtains the updated information of the features, but it is hard for him to integrate them together to gain the overall understanding about these products because the information is unsystematic.

Until now, John just gained the information of updated features, if he wants to further understand users' opinions on these updated features, there are still thousands of reviews that have not been analyzed yet. Facing such a predicament, John might turn to existing automatic methods of App store mining. For instance, he can exploit MaRK [13] to analyze descriptions or release texts of apps and CLAP [14] to mine reviews. Although these methods can help John analyze part of the data resource, none of them can solve the above problems properly. Thus, John could not (i) understand how these similar products update their features directly; (ii) get users' feedback on these updated features; (iii) and further analyze the evolution trend of these features.

In this paper, we propose an approach to analyze the data resources (app descriptions, texts in the UI, release texts, and reviews) in App stores for helping John make the feature updating strategy of his product. The main ideas are as follow:

- By using texts in app descriptions and the UI as the corpus, we leverage Latent Dirichlet Allocation (LDA) to build the topic distributions of apps to further identify similar apps for John.
- We gain sentences containing the information about updated features from release texts by training a classifier, and extract such information from sentences by defining keyword-based linguistic rules and semantic rules.
- We integrate the information of updated features and analyze the results from two dimensions. For one thing, we analyze the features frequently updated by the products, and these features may be also considered by John; for another, we analyze reviews to gain the users' opinions on the updated features, so that John can know users' attentions and preferences.

To evaluate the performance of our method, we conducted a series of experiments with the data of 13335 apps on Google Play.

Firstly, we evaluate each step of our approach quantitatively. The results show that our approach can identify similar products for apps effectively, the accuracy can be up to 64.98% at top 10, 70.16% at top 50, and 71.89% at top 100. Meanwhile, we can mine sentences containing updated features from release texts and extract valuable information from the gained sentences efficiently, their F-measure values are 88.03% and 87.80% respectively. In addition, our approach can integrate reviews into their related features with precision 83.05%. These results indicate that our approach can gain the feature updating actions of similar apps from the resource effectively.

Secondly, we conduct a survey to evaluate the usefulness of the information provided by our approach. The results of the survey show that our approach can help developers understand the features updated by other similar products and further analyze the users' opinions on these updated features. It indicates that our approach can reduce the time for analyzing large-scale data and give developers useful information for making feature updating strategy.

The paper is organized as follows. Section 2 presents the work related to our approach. Section 3 gives an overview of our approach. Sections 4 and 5 give the methods for identifying similar apps and mining the updated related information from

[1] www.qimai.cn/app/rank/appid/955837609/country/us

their data. In Section 6, we introduce the process of integrating the gained information and giving the visualization of the results. Finally, the experiments and the conclusion are shown in Sections 7 and 8 respectively.

2. Related work

Our method mines the release texts of similar products to understand the feature updating activities for further helping developers make feature updating strategies. Thus, the related work is discussed from the following two directions.

2.1. Studies on information mining for supporting app updating

In order to support developers to update app products, many researchers provide methods to mine valuable information from various types of data, and there are two main types of information mining methods based on the analyzed data resources.

For one thing, some methods gain valuable information to help developers by analyzing the data resources of their own products. For example: Shi et al. [15] propose a novel approach to detect feature-request dialogues from the online issue tracking system, and thereby benefiting the work of requirements gathering during app evolution; Gao et al. [16–18] give methods to process review texts to identify the issues (e.g., features and bugs) that users concern, so as to provide informative evidence for app developers in updating their apps. Haering et al. [19] introduce DeepMatcher, an automatic approach using a deep learning method to match problem reports in app reviews to bug reports in issue trackers, this approach can be used to provide support when developers update their products, including: detecting bugs earlier, identify duplicated bug reports and enhancing bug reports with user feedback; Palomba et al. [4] devise an approach, named CRISTAL, for tracing informative crowd reviews onto source code changes, and thereby monitoring the extent to which developers accommodate crowd requests and follow-up user reactions as reflected in their ratings; Scalabrino et al. [8] presented CLAP, a tool supporting the release planning activity of mobile apps by mining information from user reviews: after classifying and clustering user reviews, CLAP recommend which review cluster developers should satisfy in the next release.

For another, many studies mine the information useful for app updating by performing market-wide analysis. For example: Jiang et al. [20] propose an approach named SAFER to employ the descriptions of similar apps in app markets to help developers perform domain analysis, SAFER can effectively recommend new features released in the next version by identifying similar apps and recommending missing features that are implemented by similar apps; Chen et al. [21] propose STORYDROID, a system to return visualized storyboard of Android apps by extracting relatively complete ATG, rendering the UI pages statically, and inferring semantic names for obfuscated activities, such a storyboard can assist app development teams including PMs, designers, and developers to quickly have a clear overview of other similar apps.

The above work has demonstrated that whether the data from the products themselves or wide markets, it can provide valuable information for developers to evolve their products from different angles. Our work pays attention to the release texts of products in the market and gains their feature updating activities, this can help developers understand the market trend and make the feature updating strategies of their own products.

2.2. Studies on text mining in the research of apps

The purpose of text mining in the research of apps is to extract valuable information from texts, and there are two important topics: classifying texts according to contents and extracting the feature information in texts [5].

Existing methods of classifying texts are mainly used to process user reviews. For example: Ciurumelea et al. [22] built URR (User Request Reference), a prototype that is able to group reviews according to the high and low level categories of taxonomy, and a developer will be able to analyze either reviews belonging to a very specific category, or all reviews grouped per single or multiple categories; Tao et al. [23] propose a novel review summarization approach (SRR-Miner) to automatically summarizing security issues, and their work leverages a keyword-based approach to identify security-related review sentences; Grano et al. [24] explore reviews from a maintenance and testing perspective with the Gradient Boosted Classifier algorithm to further integrate into the testing process. Different from them, we aim at dividing the sentences in release texts into two types: sentences containing the information of feature updating and others. Considering different data resources and classification purposes, we determined the more appropriate classification factors and algorithm to build an effective classifier.

Feature information in texts (e.g., app descriptions and user reviews) is the valuable data resource, and many researchers have given methods to extract it from texts. For example: Johann et al. [25] propose SAFE to extract and match features from the app description and the reviews, they identify a set of general textual patterns that are frequently used to describe features based on a deep manual analysis, and apply these patterns to extract features from reviews and description; Harman et al. [26] use data mining to extract feature information from reviews, which is used to combine with more readily available information to analyze apps' technical, customer, and business aspects; Wu et al. [27] give a novel approach, KEFE, to identify key features that have significant relationships with app ratings, KEFE exploits grammatical relations of multi-lingual sentences and utilizes a more powerful deep machine learning classifier to address the feature extraction challenge; Malik et al. [28] propose a methodology to facilitate both app developers and customers to automatically extract and compare the "hot

features" among mobile apps, from the given set of mobile app reviews, and gauge sentiments towards them. Different from existing methods in terms of the analyzed data resources, we try to mine the information from the sentences related to feature updating in release texts. For this purpose, we establish a set of extraction rules based on keywords and common sentence patterns in this type of sentences to gain the feature information accurately. In addition, the information we extract contains not only features but also more detailed updated information, and such information can help developers understand how the features are updated by other products and give them inspiration when updating their own apps.

3. Overview of the proposed approach

In our approach, developers can interact with the system to acquire information on updated features of similar products, and this can help them make the feature updating strategy for their app. This process mainly contains three steps as shown in Fig. 1.

1) **Constructing background knowledge.** The developer provides the ID of the product, and similar apps of the given product are identified by building a topic distribution for each product in the same category. Then, we extract features from description texts and string files of these similar products and integrate them to construct a high-level feature framework. The framework can give information of these features as the background knowledge to the developer, and we also use it as the basis for subsequent analysis of updated information.
2) **Mining updated information.** We classify sentences in release texts of similar apps to obtain those containing the information of updated features. Then, such information is extracted from these sentences by defining two kinds of rules: keyword-based linguistic rules and semantic rules. Furthermore, we mine reviews related to the updated features by comparing words in them to analyze users' opinions.
3) **Updated Information analysis.** In this step, we integrate the information of updated features into the high-level feature framework. Then, we calculate the updating frequency of features in different periods, so as to trace the concerns of developers of similar apps. In addition, we use the number of reviews to reflect users' attention, and mine sentiments in reviews to show the users' preference on features. To display analysis results intuitively, we design the visualization for this information, so that the developers can gain the information according to their demands.

4. Constructing background knowledge

There are a large number of apps in one category of App stores, and many products may contain absolutely different features with the given app even in the same category. Therefore, we select products with similar features to the given app rather than those from the entire domain. In addition, we mine description texts and string files of these similar apps to gain app features, and integrate them as the background knowledge for subsequent analysis of feature updating.

4.1. Similar apps identification

Topic modeling algorithm can generate latent topics for texts and it can be used for similar text categorization by representing each text as a probability distribution of topics [29]. Inspired by this idea, many researches leverage this algorithm to generate topic distributions of app features from app descriptions, which are the texts to introduce the products' features to users, and use the results to measure the similarity between products [30–32].

In our work, we also consider adopting LDA to gain feature topic distributions of app products and further identifying similar products for apps based on the distributions. However, the description texts of products usually give their main features (some descriptions even contain only a few features), so it is not enough to only consider descriptions when building topic distributions for app products. User Interface (UI) of an app is everything that users can see and interact with, and developers would like to show the functionalities of their products in the UI clearly for helping users use them easily. In addition, most of features in the UI are described by giving texts, and these texts are stored in the string files of apps. Thus, we can get more comprehensive feature information of apps by mining their string files. Refer to the example in Fig. 2, the features (e.g., avoid toll roads) in the left screenshot are not mentioned in the description of app *Waze*. However, by mining texts in the string file of this app, we can get these features.

Based on the above analysis, we use string files of apps as the supplementary data of descriptions to build topic distribution profiles for apps. For each app, we first parse its apk with apktool [33] to get the string file, then texts extracted from the string file and description text are used to establish a document for the app. After that, we leverage LDA to identify the topic distributions of app documents. Finally, we evaluate the similarity between apps by calculating the cosine similarity values between their topic distributions, and top K (K can be adjusted according to developers' needs) products similar with the given app are used as the reference apps of subsequent analysis. Suppose that there is an app A, the similarity between it and the given app G is calculated with the formula as below:

$$Similarity(A,G)=\frac{\sum_{\forall t}P_{A\in t}\times P_{G\in t}}{\sqrt{\sum_{\forall t}P_{A\in t}\times P_{A\in t}}\times\sqrt{\sum_{\forall t}P_{G\in t}\times P_{G\in t}}} \tag{1}$$

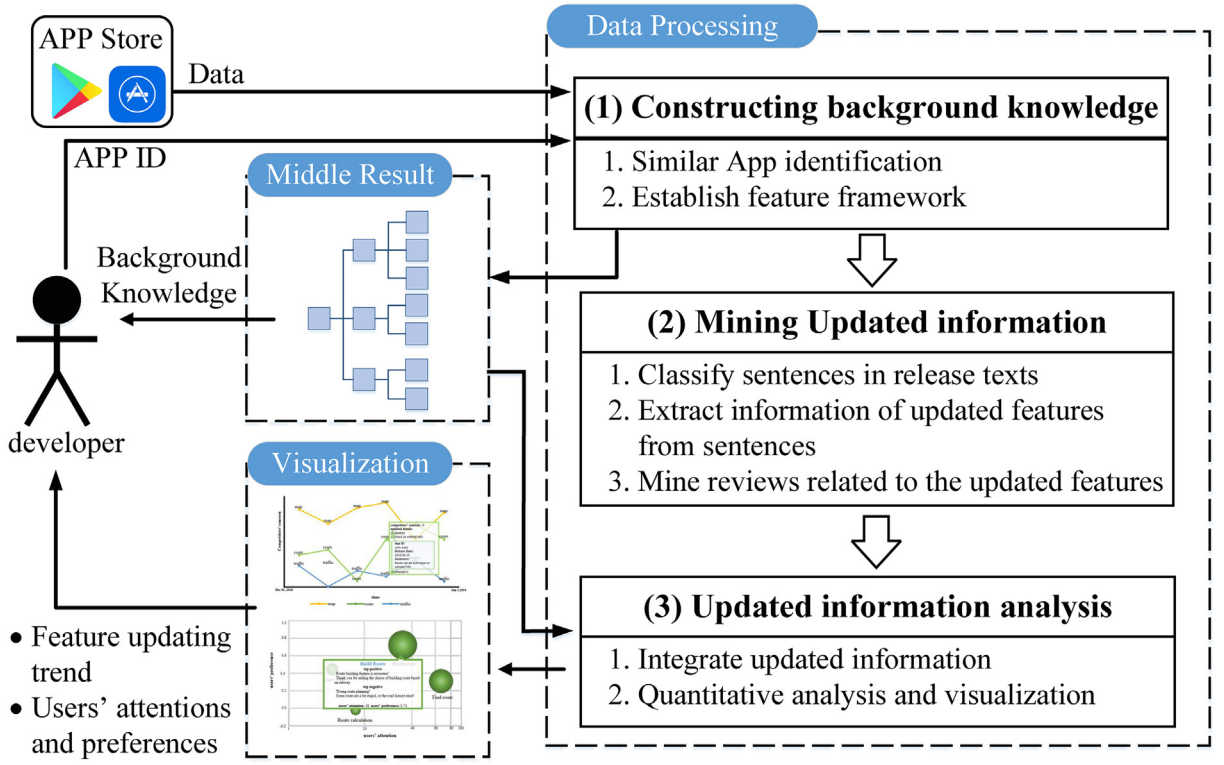


Fig. 1. Overview of approach.

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

4.2. Establishing feature framework

After identifying similar apps, we construct the feature framework as the background knowledge when analyzing the updated information. The construction process consists of two steps: extracting the features from descriptions as well as string files of similar apps, and integrating the gained feature information into a framework.

4.2.1. Feature extraction from similar apps

In our previous work [34], we have defined a set of feature extraction rules by analyzing and summarizing the relationships between structures of sentences and features, and used them to extract features from app descriptions effectively. The feature extraction rules are defined by giving operations of the parsing tree of a sentence, and they are classified into two kinds: one kind is tree transform rules, they aim at overall analyzing and transforming the parsing tree by replacing pronouns with their anaphors and eliminating useless grammar structures; the other kind is information extraction rules, they define operations to gain features from the transformed parsing tree. For each sentence in the app description, we build its parsing tree by analyzing its syntax, and then gain the information of features by using these pre-defined rules. In this way, we analyze the descriptions of all the similar apps to extract their features.

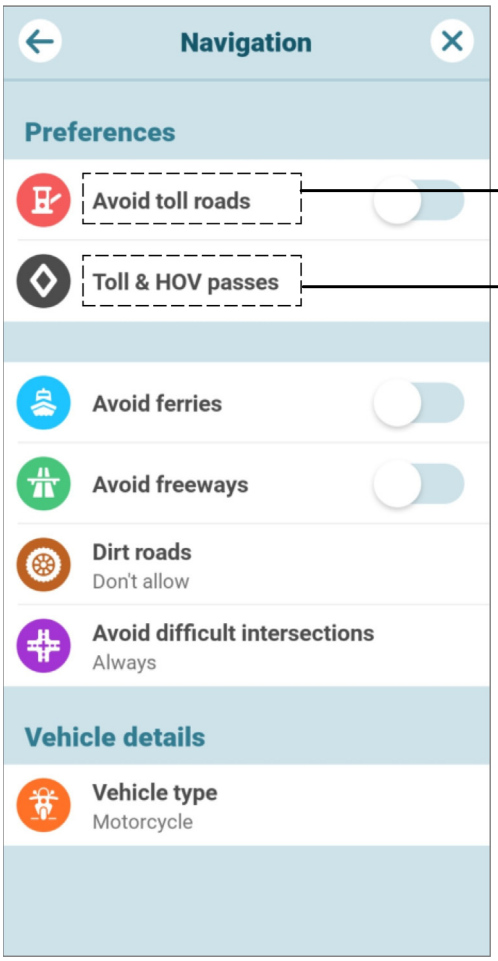
As discussed in Section 4.1, description texts of apps usually cannot reflect their comprehensive feature information, so we also consider the texts in string files when extracting features from similar apps. However, since some texts in the UI are related to other contents such as advertisements and emails, rather than feature information, we do not treat all texts gained from string files as the ones related to features. Previous work [25,34] has summarized expression forms of feature-related texts (e.g., *verb + noun* and *verb*), and only texts that satisfy these expression forms are used as app features.

All features gained from description texts and string files of similar apps are used to build a feature framework.

4.2.2. Feature integration

Our feature framework is organized into a tree structure as shown in Fig. 3, and it contains three levels. The content of each level and the construction method is as follows:

The first level contains a root node.




```
// string.xml
<?xml version="1.0" encoding="utf -8"?>
 <resources>
   <string name="avoidToll">
   Avoid toll roads
   </string>
   <string name="tollPasses">
   Toll & HOV passes
   </string>
   ...
```

// description of waze

Always know what's happening on the road with Waze. Even if you know the way, Waze tells you about traffic, construction, police, crashes, and more in real-time. If traffic is bad on your route, Waze will change it to save you time.

- See what's happening - Alerts about traffic, police, hazards
- Get there faster - Instant routing changes to avoid traffic
- Drive with Waze Carpool - Save time & money
- Play music & more - listen to your favorite apps for music
- Know when you'll arrive - your ETA is based on traffic data
- Pay less for gas - find the cheapest gas along your route
- Drive with Android Auto - use Waze on your car's display
- Always find the way - choose from a variety of voices

Be prepared, drive with Waze!

Fig. 2. Screenshot, string file and description of the app *Waze*.

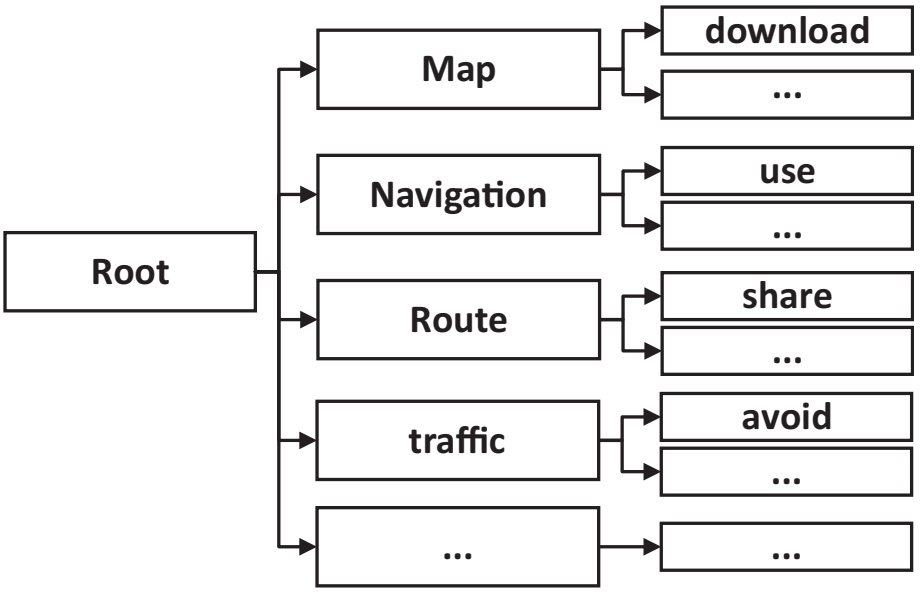


Fig. 3. Example of feature framework.

The second level is the aspect level. The aspect level is the core of the feature framework, and it summarizes the main aspects of features. Each aspect node is a noun, and these nodes are gained by summarizing the noun part of features of similar apps. The specific process is shown as follows.

We first sort nouns of features gained from similar apps according to the frequency and select top-k nouns to form a keyword set. The similarity between words in the keyword set should be smaller than the set threshold. Then, we use the features to expand the keyword set. For a feature, we calculate the relevancy of the noun part of feature (denoted by f^{noun}) with each keyword: if all relevancy values are smaller than the threshold, the noun part of feature is added into keyword set; otherwise, the feature is associated with the most related keyword. The relevancy between f^{noun} and the keyword is identified by calculating the relevancy between their words:

$$relevancy(keyword, f^{noun}) = relate^{words}(keyword, f^{noun}) \tag{2}$$

The formula to calculate the relevancy between two *wordSets* is as following:

$$relate^{words}(wordSet_i, wordSet_j) = \max_{\substack{word_i \in wordSet_i \\ word_j \in wordSet_j}} Similarity(word_i, word_j) \tag{3}$$

where $Similarity(word_i, word_j)$ denotes the similarity between $word_i$ and $word_j$ calculated by word2vec [35]. After processing features in this way, we can obtain a set of keywords as nodes of the aspect level.

The third level is the verb level. Each node in this level is expressed by a verb, which can present a feature when combined with parent noun node of it. When constructing the aspect level of feature framework, each node in the aspect level is associated with a set of features. The verb part of these features form a verb set, and we remove the identical and similar words in this set. The left verbs are taken as the child nodes of the aspect node.

5. Mining updated information

Each time a new version of an app is released, developers would give a release text to describe changes of products, and this provides us an opportunity to get the information about updated features of similar products. In addition, we can understand the feedback of updated features by mining related user reviews. In this section, we describe the process of mining the valuable information related to updated features in detail.

5.1. Classifying sentences in release texts

Some sentences in the release texts are unrelated to features but give the contents about other aspects, such as bugs fix and communication with users. Thus, we firstly train a classifier to divide sentences into feature related sentences and others to support the subsequent work.

To classify sentences in release texts, we manually selected 3000 sentences containing features and 3000 sentences unrelated to features, and summarized some important classification factors by observing them:

- **Feature related keywords.** We summarized the frequent words in sentences containing features. Once a sentence contains these words, it has a high probability of being a sentence containing features. Table 1 shows these 23 keywords.
- **Feature unrelated keywords.** By observing sentences that do not contain features, we also summarized some keywords in them as shown in Table 1. Note that *function (functionality, feature)* is the feature related keyword, but the sentences containing its plural form are usually about the summarization of the product updating and do not contain features. For example, the sentence *"crop photo feature added"* and *"add some features in this version"*.
- **Sentence pattern.** There are two common sentence patterns in sentences containing products' features, while the ones without feature information rarely belong to these two sentences patterns. Two patterns are *sentences starting with verbs* and *sentences not containing verbs*. For instance, the sentences *"Rotate image with pinch gesture"* and *"Voice Alert about traffic light"* meet two sentence patterns respectively, and both of them are related to features.
- **Comparative adjectives (more).** By observing release texts, we found that the sentences containing comparative adjectives (or the word *more*) are usually about the improvements of features, such as the sentence *"More natural photographic processing"*.
- **Special punctuation.** Through observing the data, we found that features in many sentences are surrounded by the punctuation "". For example, the sentence *Added "Asset History Map" feature* fits this situation.

Based on these classification factors, each sentence in release texts is transformed into a 64-dimensional vector $(r_1, r_2, \ldots r_{64})$, where:

- r_1 to r_{60} is assigned with 0 or 1 according to whether the sentence includes feature related keywords or feature unrelated keywords, each keyword corresponds to an element in the vector: if the sentence includes a keyword, the corresponding element in the vector is assigned with 1; otherwise, the corresponding element of this keyword is assigned with 0.

Table 1
Feature related keywords and feature unrelated keywords.

Feature related keywords		
Add feature: add, new, function (functionality, feature), available, ability, able, can, include, now, allow, provide, enable, possibility, way, introduce, such as, implement **Improve feature:** update, improve, improvement, adjust, enhance, optimize		
Feature unrelated keywords		
Solve problem: fix, bug, resolve, issue, error, address, correct, crash, minor (small), modify, problem, defect **Other improvement:** UI, performance, code, service, SDK, API, usability, stability, experience, upgrade, change, redesign **Summarization:** functions (functionalities, features), improvements, release, version		
User communication: thank, contact, please, email	**Device compatibility:** device, support, android	**Remove feature:** reduce, remove

- r_{61} and r_{62} is assigned with 0 or 1 according to the structure of the sentence: if the sentence starts with a verb, v_{61} is assigned with 1, otherwise 0 is assigned to it; v_{62} is assigned with 1 if the sentence does not include any verb, otherwise 0.
- r_{63} is assigned with 0 or 1 according to whether the sentence has comparative adjectives (or *more*): 1 is assigned to this element if the sentence has comparative adjective (or *more*); otherwise 0.
- r_{64} is assigned with 0 or 1 according to whether the sentence includes the special symbol: 1 is assigned to this element if the sentence has the symbol “”, otherwise, 0.

The keywords are considered separately rather than overall when converting sentences to vectors for two reasons: for one thing, sentences that contain different keywords have distinctly different probabilities of being sentences containing features; for another, the relationship between keywords will be ignored if we only simply consider whether sentences contain keywords. In fact, different combinations of keywords can affect the probability that sentences contain features, for example, the probability of sentences with *can* and *now* is much higher than sentences with one of them.

After vectorizing sentences of release texts, we manually labeled a certain number of sentences, and the label is 0 or 1: 1 represents that the sentence contains features, and 0 represents that the sentence does not contain any feature. Using these sentences as the training data, we train a machine learning classifier to get sentences containing features from release texts.

5.2. Extracting updated features and their details from sentences

After obtaining the sentences containing features from release texts, we extract updated features and their updated details from these sentences. Updated detail is the more detailed information about a feature, for example, we can extract feature *use map* from the sentence *“you can use map with vivid color”* and its updated detail is *with vivid color*. Features and their updated details are extracted from sentences through two kinds of rules[2]: keyword-based linguistic rules and semantic rules.

***RuleSet*$_1$: Keyword-based linguistic rules**

We have given some feature related keywords when training the classifier of sentences in release texts, and these keywords are also useful for extracting features and their updated details from sentences.

We observed more than 3000 sentences containing feature related keywords based on the POS of words in sentences to analyze position relationships between features and keywords, and summarized 381 keyword-based linguistic rules. Table 2 shows 6 linguistic rules and examples of using them to extract features as well as their updated details from sentences. Letters marked in red represent the POS tag of words, where V$\in$ {VB, VBD, VBN, VBG, VBP, VBZ}, N$\in$ {NN, NNS, NNP, NNPS}. The explanation of these POS tags is shown in Table 3. The letters with subscript $< f >$ represent features in sentences, and letters with $< d >$ represent updated details.

***RuleSet*$_2$: Semantic rules**

By observing dependency-based parsing trees of large number of sentences, we found that features and their updated details have some fixed patterns in these trees. We summarized these patterns and established 24 semantic rules for extracting features and their updated details from sentences.

Table 4 lists some typical semantic rules we use. The letters at the beginning (e.g., V and N) represent the POS tag of the root. The children of the word are listed in the right square brackets. For example, the rule in the first row means a root node

[2] https://github.com/Hylance1224/data/tree/master/rules

Table 2
Keyword-based linguistic rules to extract features and updated details.

Linguistic Rule	Sentence	Feature	Updated details
added $N_{<f>}$ function	Added soft border eraser function.	soft border eraser	{}
$N_{<f>}$ feature available	GIF feature available.	gif	{}
you can $RB_{<d>} + V_{<f>} + N_{<f>} + IN(any\,content)_{<d>}$	Now you can easily create videos with your own photos.	create videos	{easily, with your own photos}
$N_{<f>}$ option provided	Gallery folder view option provided.	gallery folder view	{}
added possibility to $V_{<f>} + JJ_{<d>} + N_{<f>}$	added possibility to record clear video.	record video	{clear}
allow user to $V_{<f>} + N_{<f>} + IN(any\,content)_{<d>}$	Allowing user to find routes via maps.	find route	{via maps}

Table 3
The explaination of POS tags.

POS tag	Explanation	POS tag	Explanation
VB	Verb, base form	NN	Noun, singular or mass
VBD	Verb, past tense	NNS	Noun, plural
VBN	Verb, past participle	NNP	Proper noun, singular
VBG	Verb, gerund or present participle	NNPS	Proper noun, plural
VBP	Verb, non-3rd person singular present	VBZ	Verb, 3rd person singular present

Table 4
Sementic rules to extract features and updated details from release texts.

Linguistic Rule	Sentence	Feature	Updated details
V[dobj-N, advmod-RB]	Users can exchange destination quickly.	V + N	RB
V[dobj-N[amod-JJ]]	Plan a fastest route.	V + N	JJ
V[nsubjpass-N, prep-IN]	Map can be used with offline mode.	V + N	subtree rooted at IN
N[amod-JJ]	Natural wallpaper.	N	JJ
V[dobj-N]	Delete photo.	V + N	–

with POS tag of V and two children: a direct object (dobj) with a POS tag of N and an adverb modifier (advmod) with a POS tag of RB.

For a sentence, spacy [36] is applied to generate a dependency-based parsing tree for it. Then, we travel from the root to all other nodes, checking nodes, edges and corresponding children to determine the feature and its updated details according to rules. For example, given the parsing tree in Fig. 4, we first check the POS tag of the root. Since it is a verb (V) that matches more than one rule, we further check its child nodes. It can be seen that the verb has two children: a passive nominal subject (nsubjpass) with a POS tag of noun (N), a prepositional modifier (prep) with a POS tag of preposition (IN). The third template is matched and we can extract the verb *use* with its child *map* as the feature, and the subtree rooted at IN *with offline mode* as updated details.

Based on the above rules, we process the sentences containing features as follows:

- If the sentence conforms a linguistic rule in $RuleSet_1$, we extract features and their updated details by using the corresponding rule.
- If the sentence does not conform to any rule in $RuleSet_1$, we use semantic rules in $RuleSet_2$ to extract features and their updated details.

After analyzing sentences containing features in release texts by using the method described above, we can obtain a set of updated features and their updated details. These features are used to establish a set *FeatureSet*.

5.3. Mining reviews related to updated features

After extracting information from release texts, we mine related reviews for each updated feature to support the subsequent quantitative analysis work.

A review usually consists of more than one sentence and different sentences may evaluate different features of an app, for example, the review *"Offline map is amazing. However, the route planning is not reasonable.*" contains two sentences, the first is about the map with positive emotion but the second is about route planning with negative emotion. Thus, we firstly split corresponding reviews into sentences. Then, we calculate the relevancy between features and review sentences to determine

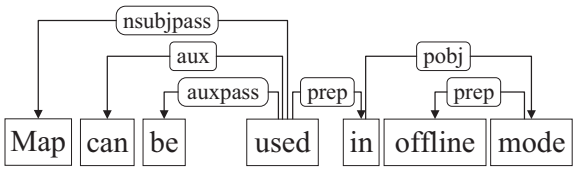


Fig. 4. Example of dependency-based parsing tree.

whether review sentences comment on the features. The relevancy between a feature f and a review sentence RS is calculated by comparing the nouns and verbs between them, and the specific formula is as follows:

$$relevancy(RS,f) = w_n relate^{words}(f^{noun}, RS^{noun}) + w_v relate^{words}(f^{verb}, RS^{verb}) \tag{4}$$

where w_n and w_v are weights of nouns and verbs respectively, and $w_n + w_v = 1$; $relate^{words}$ is calculated according to Eq. (3). If the $relevancy(RS,f)$ is bigger than the set threshold, the review sentence RS mentions the feature f, and RS is associated with this feature.

6. Updated information analysis

To help developers understand the updated features of similar products systematically, we integrate the updated information into the feature framework. Then, we further analyze the information and give a method to visualize results.

6.1. Integrating updated information

In Section 4.2, we have built a feature framework as the background knowledge for analyzing similar apps. Now, we associate features in *FeatureSet* with the nodes of feature framework by comparing features with nodes in the framework to complete the task of information integration.

The updated features we extract can be divided into two categories. The first kind of features contain both the verb part and the noun part, and we associate them with the nodes of the verb level. The second kind of features contain only the noun part, and we associate them with nodes on the aspect level. If the updated feature cannot be associated with any existing node of the feature framework, this information is about a new feature. We record them by adding new nodes into the feature framework. In visualization, we give them special tags to help developers understand the new direction of the market more clearly.

Algorithm 1 describes the process of integrating updated features in *FeatureSet* based on the feature framework. For each updated feature in *FeatureSet*, we first compare the feature with aspects in the aspect level. We calculate the relevancy between the noun part of the feature and each aspect in aspect level, and find out the most related aspect (denoted as $aspect^{max}$) (3–5). If $related^{word}(aspect^{max}, noun\_part)$ (calculated by using Eq. (3)) is bigger than the threshold, we need to further identify the associated nodes (6–22): if the feature does not contain the verb part, it is associated to the most relate aspect directly (20); otherwise, we need to compare it with the verbs, which are child nodes of $aspect^{max}$, to find out the associated node (7–18). If the feature could not be associated with any existing node in the feature framework, it is a new feature and we add a new aspect node to record this information (23–33).

According to Algorithm 1, the updated information is integrated into the feature framework. Fig. 5 shows a part of the feature framework after associating updated information.

6.2. Information analysis and visualization

After integrating updated information by associating them with the feature framework, we analyze updated information based on the hierarchy structure of the feature framework and visualize the result.

6.2.1. Quantification of updated information

In order to facilitate the quantification of updated information, we firstly give the structure of updated information in a node as shown in Fig. 6. For a $node_i$ in the feature framework, there is a set of updated information which is associated with $node_i$. We divide information according to their updated time, and establish the set $InfoSet_{node_i}(t_j, t_{j+1}) = \{Info_1, \ldots, Info_n\}$ where $t_j < Info.time \leqslant t_{j+1}$. $InfoSet_{node_i}(t_j, t_{j+1})$ describes the updated information related to the $node_i$ between time t_j and t_{j+1}, and the element in it is the $Info$ defined as a 4-tuple. We use $InfoSet_{node_i}(t_j, t_{j+1})$ as the basic unit to analyze the information.

1) To obtain aspects concerned by developers, we define a parameter, $Concern^{Developer}_{aspect_i}(t_j, t_{j+1})$. This parameter is the updated frequency of features related to the $aspect_i$. For an $aspect_i$ in the feature framework, its related features consist

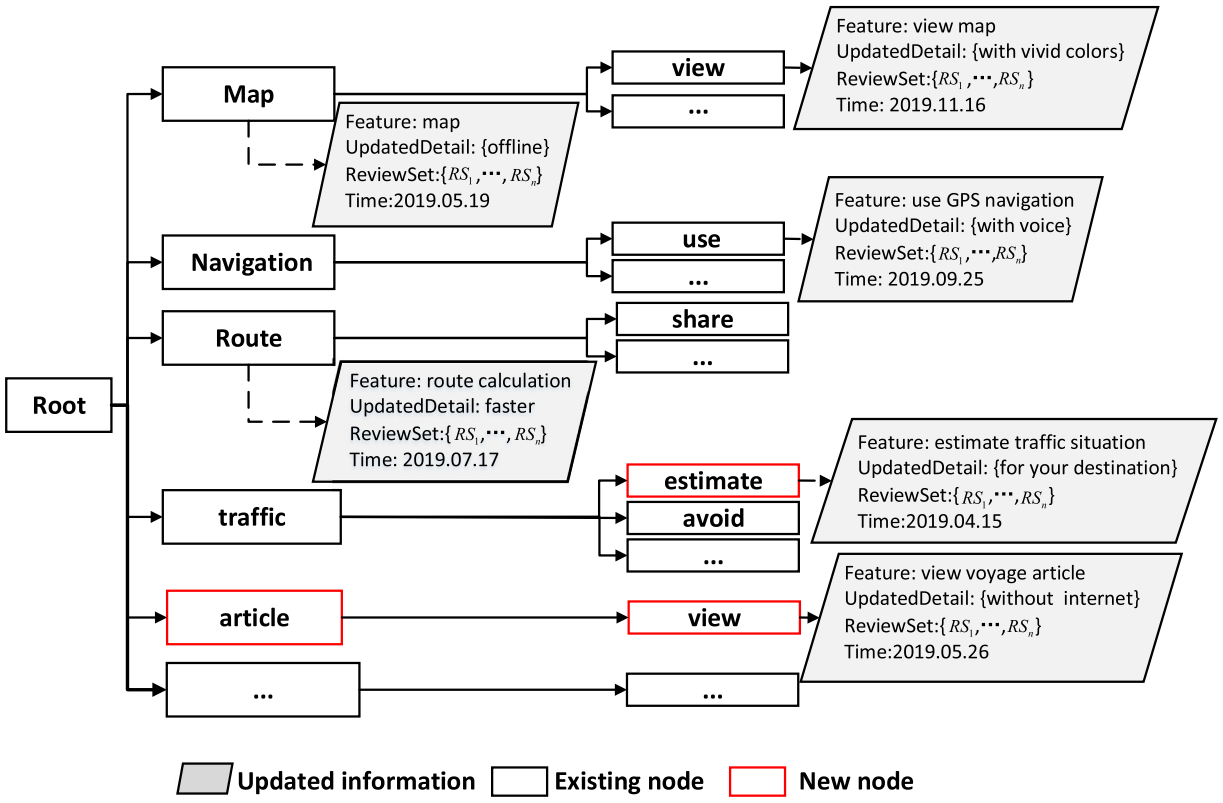


Fig. 5. Example of feature framework with updated information.

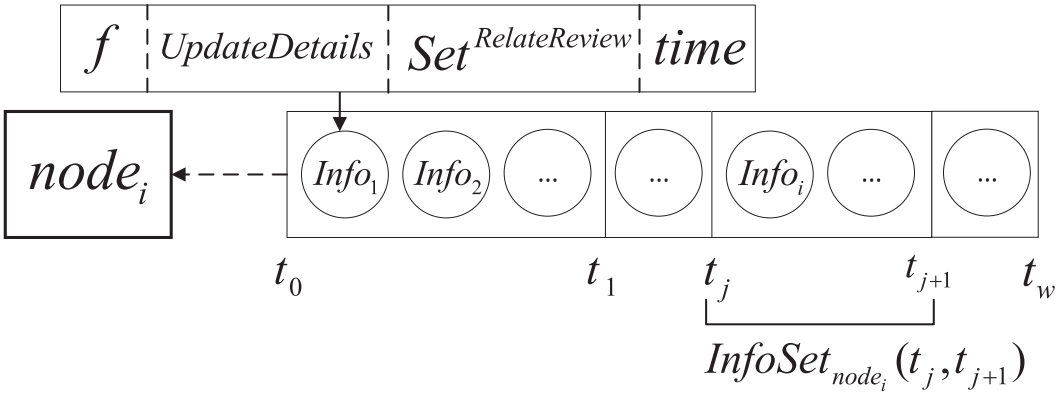


Fig. 6. Structure of updated information in a node.

of two parts: one part is the features described by the aspect node directly; and the other part is the features described by the children nodes of the aspect. Thus, we summarize the updated information related to the two parts of features together to calculate $Concern^{Developer}_{aspect_i}(t_j, t_{j+1})$:

$$Concern^{Developer}_{aspect_i}(t_j, t_{j+1}) = \left|InfoSet_{aspect_i}(t_j, t_{j+1})\right| + \sum_{k \in aspect_i.Set^{verb}} \left|InfoSet_k(t_j, t_{j+1})\right| \tag{5}$$

where $aspect_i.Set^{verb}$ is the set of child nodes of $aspect_i$.

Input: $FeatureSet, FeatureFramework$
Output: $FeatureFramework$ $with$ $updated$ $information$

```
root ← FeatureFramework.root;
foreach f ∈ FeatureSet do
    nodes^aspect ← getChildNodes(root);
    noun_part ← getNounPart(f);
    aspect^max ← getMostRelateNode(noun_part, nodes^aspect);
    if related^words(aspect^max, noun_part) ≥ μ then
        if have_verb(f) then
            nodes^verb ← getChildNodes(aspect^max);
            verb_part ← getVerbPart(f);
            verb^max ← getMostRelateNode(verb_part, nodes^verb);
            if relate^words(verb^max, verb_part) ≥ θ then
                associate(verb^max, f);
            end
            else
                new_verb ← aspect^max.addVerbNode(verb_part);
                associate(new_verb, f);
            end
        end
        else
            associate(aspect^max, f);
        end
    end
    else
        new_aspect ← root.addAspectNode(noun_part);
        if have_verb(f) then
            verb_part ← getVerbPart(f);
            new_verb ← new_aspect.addVerbNode(verb_part);
            associate(new_verb, f);
        end
        else
            associate(new_aspect, f);
        end
    end
end
```

2) To obtain features attracting more attention of users, we define a parameter, $Attention^{User}_{node_i}(t_j, t_{j+1})$, and it is calculated by:

$$Attention^{User}_{node_i}(t_j, t_{j+1}) = \frac{\sum_{Info_w \in InfoSet_{node_i}(t_j, t_{j+1})} \left| Info_w.Set^{RelateReview} \right|}{ReviewNum^{sum}(t_j, t_{j+1})} \tag{6}$$

where $ReviewNum^{sum}(t_j, t_{j+1})$ is the review number of all similar apps between time t_i and t_{j+1}.

3) To obtain features preferred by users, we define a parameter $Pref^{User}_{node_i}(t_j, t_{j+1})$. This parameter is the average sentiment of review sentences about a feature described by the $node_i$. The sentiment of a review sentence (denoted as *Senti*) is calculated base on the three important values: $rating, subjective$ and *sentiment*. The formula to calculate *Senti* is as follow:

$$Senti = (1 + w \times subjective)^{-INT(sentiment)} \left(\frac{rating}{10} + \frac{sentiment + 1}{4}\right) \tag{7}$$

where: *rating* represents the star rating of reviews given by users, and $rating \in [1, 5]$ from the most negative to the most positive; *subjectivity* reflects the emotion of users, and we use it to evaluate whether the review is tendentious and help to alleviate the affection of such emotion, $subjectivity \in [0, 1]$, the smaller value indicates the review can reflect the truth better; *sentiment* reflects the sentiment of users, and $sentiment \in [-1, 1]$ from the most negative to the most positive. The *sentiment* and *subjective* are calculated based on Pattern.en in natural language processing (NLP) given by python. $w \in (0, 1]$ is the weight of subjective and can be set according to characteristics of reviews in different domains. The users' preference on the feature described by $node_i$ is calculated by the formula $Pref^{User}_{node_i}(t_j, t_{j+1})$:

$$Pref^{User}_{node_i}(t_j, t_{j+1}) = \frac{\sum_{RS \in Set^{Review}_{node_i}(t_j, t_{j+1})} Senti(RS)}{\left| Set^{Review}_{node_i}(t_j, t_{j+1}) \right|} \tag{8}$$

where

$$Set^{Review}_{node_i}(t_j, t_{j+1}) = \bigcup_{Info_w \in InfoSet_{node_i}(t_j, t_{j+1})} Info_w.Set^{RelateReview} \tag{9}$$

6.2.2. Visualization

We designed two interactive diagrams, namely *Feature updating Trend of Similar Apps (FTSA)* figure and *User Feedback on the Feature updating Actions (UFFA)* figure, to visualize the results of the quantitative analysis of updated information. The design of these diagrams refers to [37].

Fig. 7 shows examples of the figure *FTSA* and *UFFA* that gained by analyzing similar apps of *HERE WeGo*, and we use it to illustrate the information provided by our method briefly.

Condition 1: When developers want to view updating trends of some aspects, they could select several aspects in the feature framework. Fig. 7(a) is the result after selecting three aspects: *map*, *route* and *traffic*. Each line in the figure indicates the updating trend of features in one aspect. This figure can help developers understand which aspects have been frequently updated in the recent period so that they can obtain the concerns of other developers. For example, the line of aspect *route* shows an upward trend, and this reflects a growing concern of developers on features route, so the developer could also consider improving the features about *route* in his product. In addition, the developers can view the updated details of features, so that they can understand how this feature is updated and consider making similar changes for the feature of their app.

Condition 2: When developers want to get user feedback about updated features of one aspect, they can click the node in the figure *FTSA*. Fig. 7 (b) is the result gained after clicking the *route* aspect in April. Each circle in the figure represents one feature. The horizontal axis represents the user's attention, and the vertical axis represents the users' preference. Therefore, circles in the top right represent the most popular and preferred features. The size of the circle shows the result combining users' attention and preference, and its calculation formula is $size = \log_2(w_a \times Attention^{User}_{node_i}(t_j, t_{j+1}) + w_p \times (Pref^{User}_{node_i}(t_j, t_{j+1}) + 1)$, where w_a and w_p are weights of attention and preference respectively, $w_a + w_p = 1$, and they can be adjusted according to the importance that developers attach to the attention and preference. Developers could click each feature (circle) to view related reviews. The reviews with the top positive and negative sentiment could make developers understand why users like or dislike the updated features, so that developers can take a more comprehensive consideration before updating the identical feature of their products. For example, it can be seen that the feature *plan route* in the figure has low user sentiment. By checking the relevant reviews, we can find the main reason is that the fast route planned by the app sometimes does not consider the road condition. Therefore, if developers also intend to update the feature *plan route*, they should consider not only the time required to reach the destination, but also the road conditions.

The demo of the *Feature updating Trend of Similar Apps (FTSA)* figure and *User Feedback on the Feature updating Actions (UFFA)* figure are available on our project website.[3]

7. Evaluation and results

Considering that effectiveness and usefulness are two angles commonly used for evaluating the performance of methods in software engineering [23,27,37,38], we designed the research questions in our experiments also from these two angles as follow:

- ***RQ*$_1$ (effectiveness):** Whether our approach can mine the information of updated features from the data of similar apps reasonably.
- ***RQ*$_2$ (usefulness):** Whether the information proposed by our approach is useful for making feature updating strategy?

For measuring the effectiveness of our method, we analyzed the performance of each step in the method, including: identification of similar products, text classification, information extraction, and information association. Meanwhile, we conducted a survey on developers to evaluate the usefulness of our method in practice.

7.1. Dataset and participants

We chose 4 categories of apps on Google Play as the objects of our experiments, covering the popular mainstream domain to the relatively professional one, including: Education, Navigation, Photograph and Social. There are two main reasons for choosing these categories: for one thing, the features provided by these categories are quite different from each other, so we believe using them as experimental subjects can validate the generalization of our method; for another, there are a large

[3] https://hylance1224.github.io/Display/

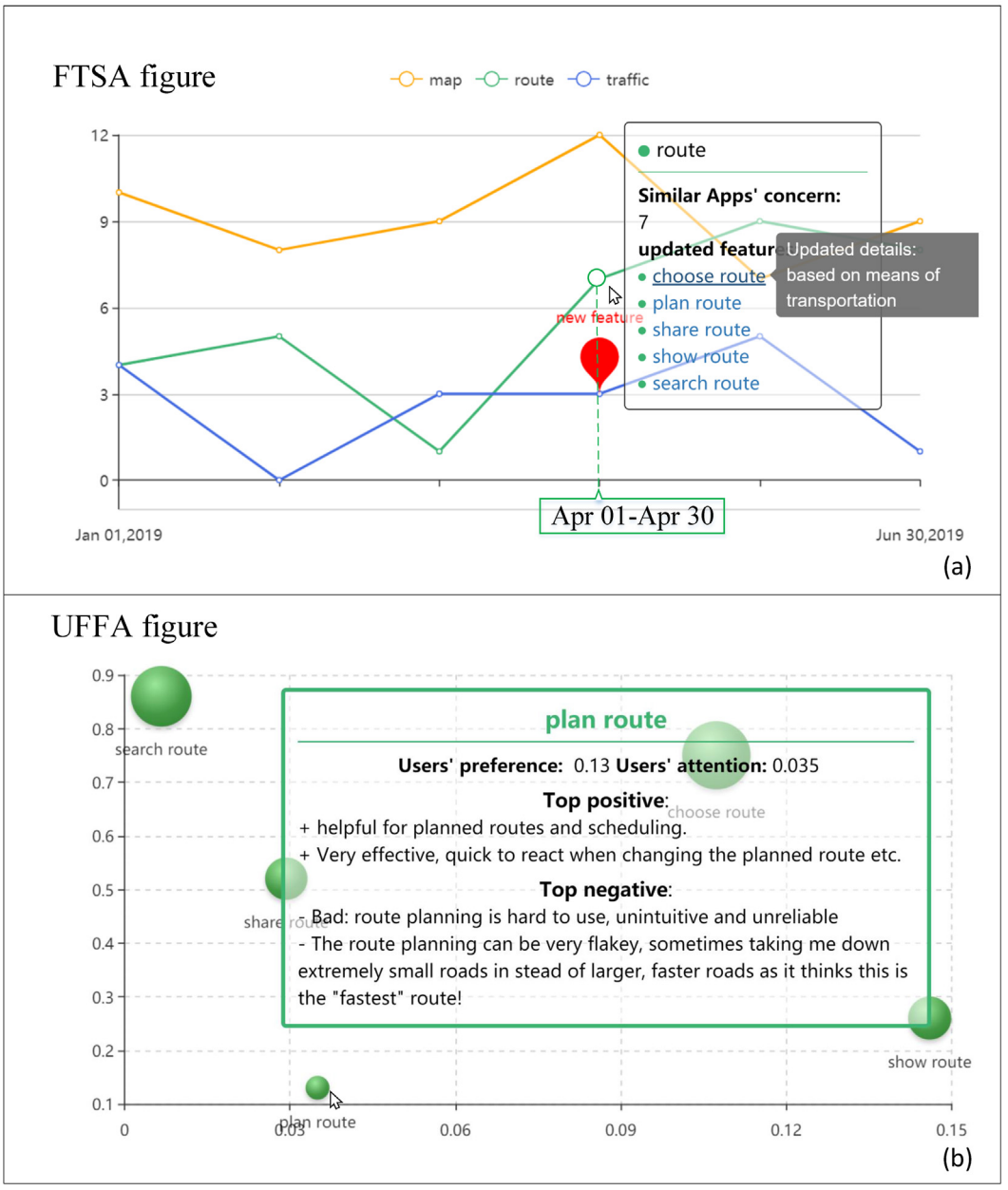


Fig. 7. Example of Feature updating Trend in Domain (FTSA) figure and User Feedback on the Feature updating Actions (UFFA) figure.

number of products in these categories, which provide enough samples to cover kinds of practice situations. For each app, we collected the description, release texts, user reviews, and apk file to support our experiments. In this process, we wrote a crawler[4] to gain app reviews and descriptions from Google play directly. The apk files are downloaded from Google Play with a tool, namely, Google Play Unofficial Python API.[5] As the apps in Google Play only show release texts of the latest version, we need to crawl appannie,[6] a website that collects and manages the data in Google Play and Apple Store, to obtain release texts of early versions. The information of our dataset is shown in Table 5.

[4] https://github.com/Hylance1224/codes/tree/master/Google-crawler
[5] https://github.com/egirault/googleplay-api
[6] https://www.appannie.com/en/

Table 5
Dataset of the experiments.

Category	Number of Apps	Number of Versions	Number of Reviews
Education	3520	12459	1097458
Navigation	2984	10840	795632
Photograph	3672	21366	1445896
Social	3159	16983	1145963
Total	13335	61648	4484949

Table 6
Participants of the experiments.

Participant	Number	Major	Development skills
PhD students	3	Requirements engineering, Data mining.	2 of them have 1–2 years industrial experience
Developers	10	App development.	More than 3 years

We invited 3 PhD students and 10 app developers to participate in our experiments, and their information is given in Table 6:

1) 3 PhD students, majoring in software engineering, especially requirements engineering, data mining. 2 of them had 1 to 2 years of industrial experience in app development. They were responsible for creating the truth set used in the experiments and analyzing the results.
2) 10 developers with more than 3 years of app development experience. They were from 3 different companies, and we invited them to evaluate the information gained by our approach and discussed it with them to get suggestions.

7.2. The experiments for RQ_1

RQ_1 focuses on checking whether the proposed approach can mine the information of updated features from the data of similar apps reasonably. We answered this question by evaluating the performance of each step of our approach, and divided RQ_1 into four sub-questions:

- Sub-question (1) whether our approach can identify similar products for apps effectively?
- Sub-question (2) whether our approach can classify sentences in release texts effectively?
- Sub-question (3) whether our approach can extract information of updated features from the data resource effectively?
- Sub-question (4) whether our approach can associate reviews and features effectively?

We conducted the experiments for RQ_1 based on the rules in [39] to eliminate bias as much as possible: for one thing, we randomly selected experimental data used to evaluate each step of our method from different categories to guarantee the fairness of experiments and ensure the generalization of the method; for another, we invited multiple participants to co-create ground truth for data to ensure the reliability of experiments.

The data and results in our experiments were provided online.[7]

7.2.1. The experiment for sub-question (1)

Identifying similar products for apps is the first step in our method, which is the basis of mining the activities about feature updating of products in the market. Thus, we conducted an experiment to evaluate the performance of our method used in this step, the experimental design and result are presented in this section.

Experimental design for sub-question (1).

Evaluating the performance of our method used to identify similar products for apps is not easy because it is very subjective to judge whether app products are similar [40]. Thus, we designed an experiment to evaluate the method indirectly.

To help users find suitable app products, Google Play divides apps into different categories, such as the ones shown in Table 5. We assume that an app has the higher similarity with products in the same category than the ones in other categories. Based on this assumption, we evaluated our method of identifying similar products. The evaluation process consists of three steps:

Firstly, we randomly selected 2000 apps from our dataset for each category as data basis for answering sub-question (1). Note that, we selected 2000 apps from each category as the experimental data rather than all products in our dataset because we want to balance the number of apps participating in this experiment in each category.

[7] https://github.com/Hylance1224/data

Secondly, we used description text and the text gained from string file to establish a document for each app, and leveraged LDA to process all the documents obtained to generate topic distributions for app products. In this process, we adopted perplexity [41] to identify the suitable number of topics.

Thirdly, we identified top N similar products for each app by comparing its topic distribution and the ones of other apps. As discussed above, we believe that apps have the higher similarity with products in the same category than the ones in other categories, so we evaluated the accuracy (*accuracy@N*) of identifying similar products for each app by calculating the percentage of apps in the same category among top N similar products. Based on the accuracy values of all apps in a category, we can further gain the accuracy for a category.

Experimental result for sub-question (1).

Fig. 8 shows the experimental result of sub-question (1), we can see that our method can identify similar products for apps effectively: the accuracy of our method can be up to 64.98% at top 10, 70.16% at top 50, and 71.89% at top 100. However, of top N similar products that we gained for apps by using our method, many are still not in the same category as them. By observing the practice data, we summarized a main reason for this situation: even app products in different categories may share some features, which causes LDA to generate similar topic distributions for these apps.

Compared with existing methods, the characteristic of our method is to introduce texts in the UI (texts in the string files) into the establishment of app documents, so we constructed an experiment to evaluate whether the introduction of these texts contributes positively to our method. We only used description texts to establish documents for apps selected in this experiment, and leveraged LDA to process these documents to generate topic distributions for further identifying similar products for each app. By using the same evaluation method, we analyzed the performance of this method (denoted by $Method^{-UItext}$). Fig. 9 gives the experimental result of $Method^{-UItext}$, it can be seen from the figure that three metrics (*accuracy*@10, *accuracy*@50, and *accuracy*@100) of this method are all lower than our method. As mentioned in Section 4.1, the UI of apps contains more comprehensive feature information than description texts, so combining texts in string files with description texts can better characterize apps and this is conducive to the work of similar app identification.

The methods proposed in studies [20,42] use API names as a data resource of establishing app documents so as to improve the performance of similar product identification, but our method does not consider API names in this process. We also carried out an experiment to explore whether introducing this data resource into our method can gain better results. For each app product, we utilized androguard [43] to parse apk file for collecting its API names, and established a document with considering three types of data: description text, text gained from the string file, and API names. We measured the similarity between each app and other products based on new documents, and used the same method to evaluate the performance of this method (denoted by $Method^{+APIname}$). Fig. 9 also gives the experimental result of $Method^{+APIname}$, we can see that its performance is worse than our method: although *accuracy*@50 of $Method^{+APIname}$ is slightly higher than our method (70.49% versus 70.16%), its values of *accuracy*@10 and *accuracy*@100 are significantly lower than ours (62.31% versus 64.98%, and 67.42% versus 71.89%). Obviously, the introduction of API names does not contribute positively to our method. By observing and analyzing the practice data, we summarized two reasons for this result: for one thing, the feature information gained from the UI covers almost all valuable information contained in API names; for another, since a lot of APIs (e.g., *android.widge.EditText; getText*) can be used to support the implementation of various different features, the names of these APIs will interfere with the generation of topic distributions about app features instead of helping our works.

According to the above analysis, we believe that our method can identify similar products for apps effectively.

7.2.2. The experiment for sub-question (2)

To gain the updated features from similar apps, we mine the sentences containing features from release texts by training a classifier. We constructed an experiment to evaluate the performance of our classifier, the experimental design and results are presented in this section.

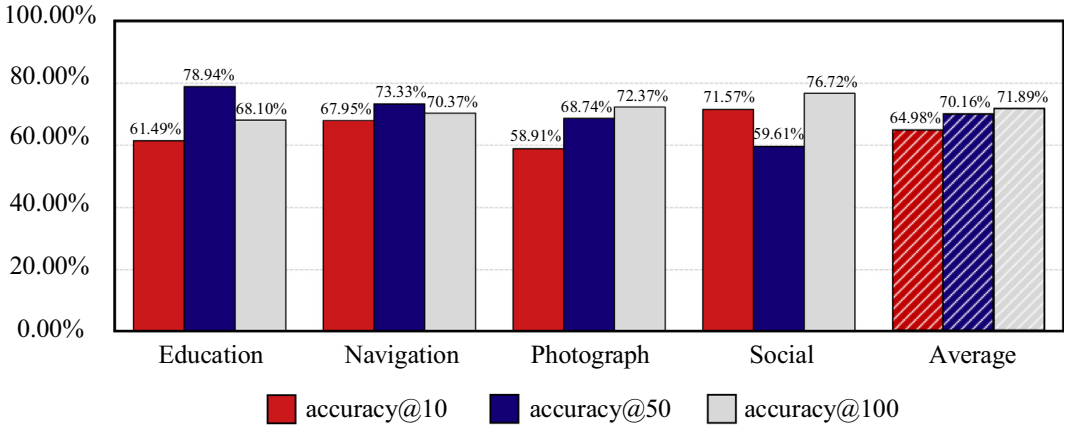


Fig. 8. The experimental result for identifying similar apps.

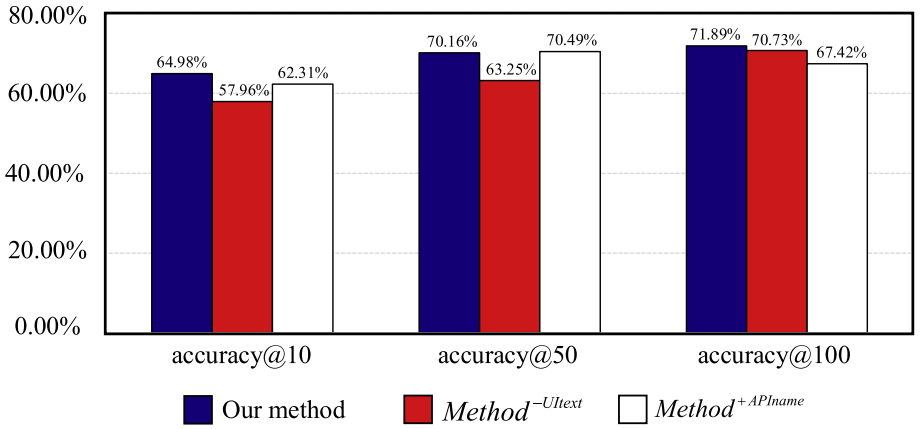


Fig. 9. The results of comparison experiment for identifying similar apps.

Experimental design for sub-question (2).

For answering the sub-question (2), we first trained the classifier for sentences in release texts. We randomly selected about 900 sentences that were split from release texts to establish the training set, in which the data should cover all the classification factors we defined so that the classifier could be trained comprehensively. The training process consists of 3 steps:

Firstly, each sentence in the training set is converted to a 64-dimensional vector according to the method described in Section 5.1.

Secondly, 3 PhD students labelled sentences in the training set separately: sentences containing features are labelled as 1, and others are labelled as 0. The consistent labels were used as labels of sentences in the training set directly. For disagreements, the 3 students discussed together and then voted, only sentences which reach an agreement by all the raters were added into the training set.

Finally, we trained the SVM classifier based on the training set, and here are the important parameters set in our experiment: kernel = 'poly', degree = 1, C = 1.0, gamma = 'auto_deprecated', Coef0 = 0.0.

After training the classifier of sentences, we establish truth sets. For each category, we randomly selected 400 sentences of release texts as samples to establish the truth set in the experiment. Note that sentences in the training set could not be added into truth sets. Same as the training set, 3 PhD students labelled the samples in the truth sets separately, and the labels are consistent with the ones in the training set. The consensus labels were used as labels of sentences in the truth sets, and the participants discussed together for disagreements.

We compared the classification results gained by our approach and labels of samples in the truth set to evaluate the performance of our classifier. Here we give 3 evaluation parameters: TP, FP and FN.

TP is the number of sentences which are correctly classified as 1 in the truth set; FP is the number of instances which were classified as 1 but labelled as 0 in the truth set; FN, the number of sentences which were classified as 0 but labelled as 1 in the truth set. Based on these 3 parameters, we used F-measure to evaluate the performance of our classifier, the definition of F-measure is shown as follow:

$$Precision = \frac{TP}{TP + FP} \tag{10}$$

$$Recall = \frac{TP}{TP + FN} \tag{11}$$

$$F - measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{12}$$

Experimental result for sub-question (2).

Table 7 gives the performance of our approach for classifying sentences in release texts, it can be seen that the precision and recall are 84.15% and 92.30% on average respectively; the F-measure is 88.03% and its variation is stable (the lowest is 86.53% and the highest is 89.03%). This indicates that our approach achieves good performance and is suitable for the release texts in different categories. The average recall 92.30% shows that our classifier can find most of the sentences containing features in release texts, and this provides the plentiful data basis for our subsequent analysis. In addition, We can find that the average precision is lower than recall (84.15% versus 92.30%). By observing and analyzing the data, we summarized two main reasons. For one thing, some sentences contain feature related keywords, but they give contents about the overall change of products rather than describing specific features. For example, the sentence "*Additional Functionality Added*" is clas-

Table 7
Experimental result of classifying sentences in released texts.

Category	Number of sentences	Precision	Recall	F-measure
Education	400	84.38%	93.56%	88.73%
Navigation	400	86.48%	91.74%	89.03%
Photograph	400	81.10%	95.14%	87.56%
Social	400	84.34%	88.83%	86.53%
Average	400	84.15%	92.30%	88.03%

sified as the sentence containing features because it contains the keyword *functionality* and *add*, but it is about the overall change of the product. For another, many sentences are about the communication of developers to users, but their sentence patterns meet the ones given by us (one is *sentences starting with verbs*, and the other is *sentences not containing verbs*), and this makes our classifier incorrectly process them. For example, the sentence "*Welcome to the Noisy May Emojis keyboard*" conforms to the first pattern, and it is labelled as the sentence containing feature information by our classifier.

In order to further analyze the performance of our method, we compared SVM with some classical algorithms, including Random Forest, Logistic Regression and Naive Bayes in our classification problem. We utilized these compared algorithms to train classifiers for classifying sentences in the truth set and then used the same evaluation metrics to analyze their performances. Table 8 shows the comparison results. It can be seen that SVM outperforms random forest consistently in all three metrics. In addition, although the precision and recall of SVM are slightly lower than logistic regression and naive Bayes respectively, the F-measure of SVM is much higher than them. Thus, we conclude that SVM is better at solving the problems in our work.

Based on the above analysis, we can draw a conclusion: our approach can classify sentences of release texts effectively in different categories of apps.

7.2.3. The experiment for sub-question (3)

Our approach extracts the information of app features from two types of texts: app descriptions and release texts. The method for analyzing app descriptions is proposed in our previous work and its effectiveness had been evaluated [34]. Thus, we only need to evaluate the performance of extracting information from release texts.

Experimental design for sub-question (3).

We firstly established truth sets in order to answer sub-question (3). For each category, we randomly selected 100 sentences, which are labeled as 1 by both PhD students and the classifier, from the truth set of sub-question (2). Three participants labeled each sample, and the label is the information of feature and its updated details. For example, for the sentence - *Sharing your current position.*, its label is *(share position, current)*. Similar as sub-question (2), sentences with consistent labels are used as experimental samples directly, and for the disagreements, the three participants had a discussion to determine the final results.

Then, we processed sentences in the truth sets by our method to get features and their updated details, and the results were compared with the labels of truth sets. Three evaluation parameters were gained: TP is the number of sentences which the extracted information was not empty and consistent with the label in the truth set; FP is the number of sentences which the extracted information was not empty but not consistent with the labels in the truth set; FN is the number of sentences which the extracted information were empty according to our approach but not empty in the truth set. Then we evaluate the performance of our feature extraction method by calculating: Precision, Recall, and F-measure.

Experimental result for sub-question (3).

Table 9 shows the experimental result of the sub-question (3). It can be seen that our approach is capable to complete the task of extracting the information of updated features from sentences in release texts: the precision and recall are 82.59% and 93.71% respectively, and the average F-measure is 87.80%. There are about 17.41% of sentences that are extracted incorrect information by our method. We analyzed them and found a main reason: some words in sentences are marked with wrong POS, and this makes our method cannot extract information from these sentences accurately. For example, for the sentence "*– Interact with weather station directly from the moving map*", the word interact is labelled as NN (noun) rather than VB

Table 8
Experimental results of other algorithms.

Category	Random Forest			Logistic Regression			Naive Bayes		
	Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
Education	82.20%	77.72%	79.90%	83.67%	81.19%	82.41%	74.91%	99.00%	85.29%
Navigation	88.48%	73.48%	80.29%	86.14%	75.65%	80.56%	78.32%	97.39%	86.82%
Photograph	80.77%	79.46%	80.11%	84.78%	84.32%	84.55%	67.66%	98.38%	80.18%
Social	82.98%	82.98%	82.98%	84.21%	85.11%	84.66%	72.05%	97.34%	82.81%
Average	83.64%	78.13%	80.80%	84.72%	81.24%	82.94%	73.33%	98.01%	83.89%

Table 9
Experimental results of extracting updated information of features from released texts.

Category	Number of sentences	Precision	Recall	F-measure
Education	100	79.38%	96.25%	87.01%
Navigation	100	77.89%	93.67%	85.06%
Photograph	100	87.50%	95.45%	91.30%
Social	100	85.71%	89.66%	87.64%
Average	100	82.59%	93.71%	87.80%

(verb), so we extracted incorrect feature *interact* rather than *interact with station* by using the rule *N[prep-IN]*. In addition, the average recall 93.71% shows that we can extract the most of valuable information from release texts, but there are still some sentences that cannot gain the feature information by using our method. We further analyzed the data and found that most parsing trees generated for the false-negative sentences are not accurate, it makes that our rules cannot apply to these sentences.

Feature extraction is an important step of our approach. We further compared our approach with some existing feature extraction approaches: the approach provided by Johann et al. [25] and the approach provided by Harman et al. [26]. We quoted the performance of methods from their research paper directly as baselines for comparing feature extraction approaches; the results are shown in Table 10. It can be seen that our method achieves better performance in both precision (82.59% versus 55.90% and 28.70%) and recall (93.71% versus 43.40% and 29.20%). The results of our method are far better than the comparison methods, we analyzed this situation and summarized two main reasons: for one thing, we classified the sentences before extracting features from them, so we can avoid the interference caused by useless sentences in release texts; for another, our method is used to process sentences in release texts while the comparison methods extract the features from description texts, and release texts have the more fixed writing pattern than descriptions, so we can get a series of effective extraction rules by summarizing these patterns.

Note that we did not evaluate the performance of compared methods on sentences used in our experiment to further compare two methods on a common data set, this is because the research purpose of compared methods is not completely consistent with our method. Compared methods are used to process description texts, and their effect on analyzing release texts is not ideal, so it is unfair for compared methods to do the comparison experiment on our data set. Also due to different research purposes, the experiment could not prove that our method is better, and it only reflects that our method can reach the level of practical application as compared methods.

7.2.4. The experiment for sub-question (4)

In our approach, we establish relationships between features and reviews by comparing keywords in them. We conducted an experiment to evaluate the performance of our association method, the experimental design and results are presented in this section.

Experimental design for sub-question (4).

We only considered the precision in this evaluation process for two reasons: for one thing, app may receive thousands of user reviews each day, so we believe that the precision is more important than recall for the problem of associating reviews with features; for another, we could not establish an appropriate truth set to calculate the recall value of out method. Firstly, PhD students selected some release texts together to identify the features in them, and only when a feature was identified by all of them can the feature be the sample in our experiment. We randomly selected 10 different features from the samples for each category. Then, we associated reviews with these features by using our method, and randomly selected 50 reviews from the associated ones for each feature as the data basis to evaluate the performance of our method. Finally, raters labelled reviews and the label of each review is 1 or 0: 1 represents that the review is indeed related to the associated feature, otherwise 0. Based on the gained labels, we calculated the precision for each category.

Experimental result for sub-question (4).

Fig. 10 shows the results of sub-question (4). It can be seen that our method can complete the work of associating the reviews with the features for different categories of apps: the average precision is 83.05% and its variation is stable (the lowest is 80.40% and the highest is 86.00%). The reason for gaining the good result is that our method associates reviews with the features by matching keywords in them, and this method is very accurate. There are 16.95% of associated reviews are incorrect, we analyze these reviews and summarize two main reasons: for one thing, some reviews are very shot, and the POS of

Table 10
The results of comparison experiment for feature extraction.

Method	Precision	Recall	F-measure
Our approach	82.59%	93.71%	87.80%
Johann et al. [25]	55.90%	43.40%	45.80%
Harman et al. [26]	27.80%	29.20%	27.00%

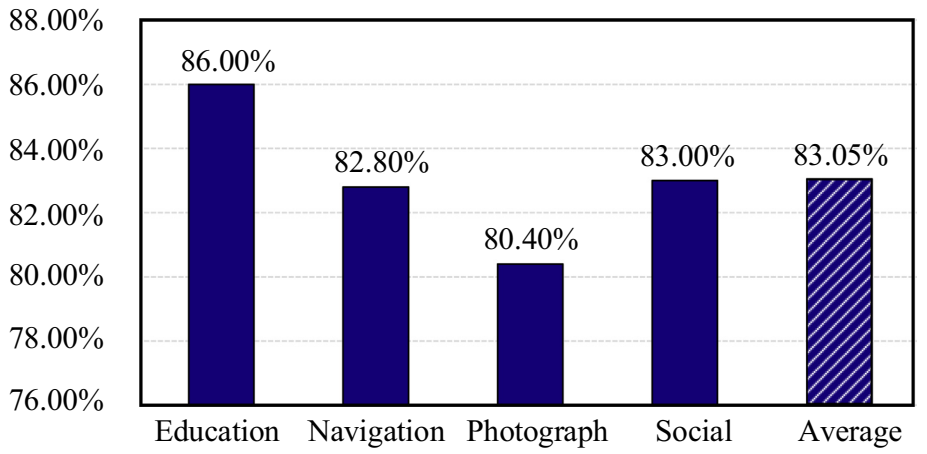


Fig. 10. Experimental result of establishing relationships between reviews and features.

words in short texts tagged by NLTK are not very accurate; for another, the vectors of some words obtained by word2vec are unreasonable, which leads to the errors of similarity calculation between words.

In general, our approach can associate reviews to their related features with the precision about 83.05%, so we believe that the association method of comparing keywords is effective.

7.3. The experiments for RQ_2

RQ_2 aims at evaluating the usefulness of the information gained by our approach for making feature updating strategy. This is a nontrivial task because usefulness is very subjective, we cannot establish a truth set and gain quantitative evaluation parameters. In such condition, conducting surveys in practice is a good way to measure the usefulness of the method and it has been used by many excellent researches in software engineering to evaluate their approaches (e.g., [23,27,44]). In this experiment, we conducted the survey on app developers guided by the activities suggested in [39] to analyze whether the information provided by our method can give help for them in the evolution of the products.

7.3.1. Design of the survey

For each category, we randomly selected one app in the dataset, then we identified its similar apps from the dataset and applied our method to analyze the updated information of these similar apps, the results were visualized in the form introduced in Section 6.2.2. Then, we designed a questionnaire for evaluating the information. As our approach analyzes updated information of similar apps from two dimensions (the concerns of similar apps and the sentiments of users), the questionnaire also includes two parts. Table 11 shows the main questions in each part of the questionnaire.

The first part evaluates whether our *Feature updating Trend of Similar Apps (FTSA)* figure can help developers understand the changing trend of concerns of these products. This part includes three sub-questions: firstly, we want to know whether the information in *FTSA* figure is easy to understand for developers; secondly, the *FTSA* gives the detailed information that shows how the similar apps update their features, so we check whether the participants confirm the usefulness of such information; thirdly, the *FTSA* gives the updated frequency of app features to reflect the changing of concerns in the similar products, and we ask the participants whether such information can help them understand the feature updating trend of the similar apps.

The second part evaluates whether our *User Feedback on the Feature updating Actions (UFFA)* figure can help developers gain users' opinions on these feature updating actions of similar apps. This parts also includes three sub-questions: the first one evaluates the understandability of the *UFFA* figure; the second one checks whether the developers can get users' attention on the feature updating actions, that is which updated features are concerned by users; the third one analyzes whether developers can gain users' preference on the feature updating actions, that is the users feedback positive or negative sentiments on the updated features.

We provided five options for each question (1 strongly disagree, 2 disagree, 3 neither, 4 agree and 5 strongly agree). Furthermore, a square frame named "Any other feedback?" was given at the last of each section in the questionnaire, so that the participants could give any comments or ideas freely. In the survey, we introduced the tasks to the 10 developers in detail, and gave the questionnaire as well as the visualized information gained by our approach to them. The participants could respond to the questionnaire according to their schedules.

Table 11
Main questions in each part of the questionnaire.

Part	Main Questions
P1	1. Is the information in *Feature updating Trend of Similar Apps* figure understandable? 2. Does the *FTSA* reflects how the similar Apps updating their features? 3. Does the *FTSA* reflects the updating trend of the features in the similar Apps?
P2	1. Is the information in *User Feedback on the Feature updating Actions* figure understandable? 2. Can you gain the users' attentions on the feature updating actions from *UFFA* figure? 3. Can you gain the users' sentiments on the feature updating actions from *UFFA* figure?

7.3.2. Results and analysis

Table 12 summaries the feedback of the participants. After the survey, we further had an open discussion with all the developers to get a deeper understanding on their opinions. Next, we analyze the feedback of each part in the questionnaire separately.

Feedback on Part 1.

Part 1 of the questionnaire evaluates the *FTSA* figure generated in our approach.

With respect to the understandability of *FTSA* figure, 9 of the 10 participants agreed that the figure is easy to understand, and none of the developers gave negative feedback on question 1. It indicates the information in *FTSA* figure is represented clearly and its understandability is acceptable.

For question 2, most developers showed great interest in the information, 9 of them indicated that they can know the feature updating actions of similar apps from the *FTSA* figure, and 4 in them strongly agreed with the usefulness of the information. Only one developer disagreed with question 2, and he pointed out that there is some useless detailed information which cannot reflect how the similar apps update their products. As our approach extracts updated information from release texts automatically, it is reasonable that there exists some redundancy or useless information in the *FTSA* figure.

For question 3, the response from developers is a little lower than question 2, but most of them still gave positive feedback. 7 of 10 developers agreed on the usefulness of the information on analyzing the updating trend of the features in similar apps, while one held conservative opinions and two gave negative feedback. By discussing with the participants, we found that a main reason the two developers disagree the question 3 is that: they think only the updated frequency is not enough for supporting the analysis process, and more parameters should be considered. This suggestion is reasonable and we will consider it for future improvements in our approach.

Feedback on Part 2.

Part 2 of the questionnaire evaluates the *UFFA* figure generated in our approach, and the results are quite encouraging.

With respect to the understandability of *UFFA* figure, all of the 10 participants agreed that the figure is comprehensible. This is because the information in *UFFA* figure gives how users comment on the updated features, and it is represented by review sentences.

For question 2, 8 developers agreed that the *UFFA* figure can give users' attentions to the feature updating actions clearly, while the other 2 developers held conservative opinions. As *UFFA* figure shows the number of users paying attention to one updated feature by the size of the nodes directly, the developers can get such information intuitively.

For question 3, 6 developers believed that the *UFFA* figure reflects the users' sentiments on the feature updating actions, but 2 developers disagreed with this conclusion. By discussing with the 2 developers, we found that they have a similar concern: they think it is inappropiate to aggregate reviews of different products when analyzing user sentiment, because each product may have its own advantage or disadvantage. However, our approach aims at providing reference information for making feature updating strategy, so we need to integrate information and help developers overall understand the feature updating actions of similar apps.

In summary, most developers accepted the usefulness of the information we provided for helping make feature updating strategy, and they confirmed that our approach can reduce their time on analyzing large-scale data.

Table 12
Results of the survey on developers.

Questionaire		Strongly Disagree	Disagree	Neither	Agree	StronglyAgree	Total
Part	Questions						
Part 1	1	0	0	1	3	6	10
	2	0	1	0	5	4	10
	3	0	2	1	6	1	10
Part 2	1	0	0	0	3	7	10
	2	0	0	2	2	6	10
	3	0	2	2	3	3	10

7.4. Threats to the validity of experiments

Although we gained good results in the experiments and the survey, there are threats to the validity of our study. We give a discussion on them from the following two aspects.

The dataset used in our approach is limited. The data used in our experiments is mainly collected from four categories of apps on Google Play, so it is unclear whether our approach can also achieve similar results on other categories or App stores. Unfortunately, due to the complexity of our experiments, we could not evaluate our approach with additional datasets in a short time. However, the data in different App stores share similar characteristics, and the developers and users would not change their expression habits in different platforms, so we believe that the generalizability of our approach can be acceptable.

The participants in the survey are not the developers of the objected apps. They might not have a deep understanding on the objected apps and the information we provided. To alleviate this threat, we adopted three actions. Firstly, the four objected apps in our experiment are popular and from the categories of apps that are used by people in their daily life. Secondly, we did not limit the time for the participants to answer the questionnaire so that they could download the apps and use them during the survey. Finally, the questions in the questionnaire are relevant simple, and developers could answer them according to their industry experience without understanding the apps deeply. In the future, we wish to cooperate with these developers to analyze their own apps to further evaluate the usefulness of our approach on supporting feature updating strategy.

8. Conclusions and future work

In this paper, we propose an approach to provide information for helping developers make feature updating strategy by analyzing the data in App stores. To achieve this goal, we firstly help developers identify similar apps, and mine the descriptions and string files of these products to establish the feature framework to overall describe their features. Then, we gain the updated information from the release texts and reviews, and further integrate the information to represent the feature updating actions of these products; meanwhile, we establish relationships between reviews and the updated features, so that the developers can get the users' attention and sentiments on these actions. The experiments based on the data on Google Play show that the performances of the steps in our approach are acceptable, which means our approach can mine and integrate the information of updated features from the data in App stores efficiently. In addition, the survey indicates that the information provided by our approach is useful for making the feature updating strategy. Specifically, it helps developers to understand the feature updating behavior of similar products and users' opinions on these updating actions.

Our future researches mainly focus on two aspects. Firstly, we want to develop a well-established tool to support our approach in practice. Secondly, we hope to evaluate our approach in more cases and help developers to analyze their own apps.

CRediT authorship contribution statement

Huaxiao Liu: Conceptualization, Methodology, Validation, Software, Writing - original draft. **Yihui Wang:** Methodology, Software, Validation, Data curation. **Yuzhou Liu:** Conceptualization, Validation, Writing - review & editing. **Shanquan Gao:** Conceptualization, Investigation, Validation.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The work is funded by the Fundamental Research Funds for the Central Universities, National Key Research and Development Program of China 2017YFB1003103, Natural Science Research Foundation of Jilin Province of China under Grant Nos. 20190201193JC, Graduate Innovation Fund of Jilin University 101832020CX181, and Interdisciplinary Research Funding Program for Doctoral Students of Jilin University 101832020DJX064.

References

[1] Q.A. Chen, H. Luo, S. Rosen, Z.M. Mao, K. Iyer, J. Hui, K. Sontineni, K. Lau, Qoe doctor: Diagnosing mobile app qoe with automated ui control and cross-layer analysis, in: Proceedings of the 2014 Conference on Internet Measurement Conference (IMC), 2014, pp. 151–164.

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

[3] K. Moran, C. Watson, J. Hoskins, G. Purnell, D. Poshyvanyk, Detecting and summarizing gui changes in evolving mobile apps, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE), 2018, pp. 543–553.

[4] F. Palomba, M. Linares-Vásquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, A. De Lucia, Crowdsourcing user reviews to support the evolution of mobile apps, Journal of Systems and Software 137 (2018) 143–162.

[5] W. Martin, F. Sarro, Y. Jia, Y. Zhang, M. Harman, A survey of app store analysis for software engineering, IEEE Transactions on Software Engineering 43 (9) (2016) 817–847.
[6] S. Hassan, C.-P. Bezemer, A.E. Hassan, Studying bad updates of top free-to-download apps in the google play store, IEEE Transactions on Software Engineering 46 (7) (2020) 773–793.
[7] X. Chen, Q. Zou, B. Fan, Z. Zheng, X. Luo, Recommending software features for mobile applications based on user interface comparison, Requirements Engineering 24 (2018) 545–559.
[8] S. Scalabrino, G. Bavota, B. Russo, M.D. Penta, R. Oliveto, Listening to the crowd for the release planning of mobile apps, IEEE Transactions on Software Engineering 45 (2019) 68–86.
[9] D. Cao, L. Nie, X. He, X. Wei, J. Shen, S. Wu, T.-S. Chua, Version-sensitive mobile app recommendation, Information Sciences 381 (2017) 161–175.
[10] A. Al-Subaihin, F. Sarro, S. Black, L. Capra, M. Harman, App store effects on software engineering practices, IEEE Transactions on Software Engineering 47 (2021) 300–319.
[11] S. Panichella, A. Sorbo, E. Guzman, C.A. Visaggio, G. Canfora, H. Gall, Ardoc: app reviews development oriented classifier, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2016, pp. 1023–1027.
[12] B. Fu, J. Lin, L. Li, C. Faloutsos, J. Hong, N. Sadeh, Why people hate your app: making sense of user feedback in a mobile app store, in: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2013, pp. 1276–1284.
[13] X. Lian, M. Rahimi, J. Cleland-Huang, L. Zhang, R. Ferrai, M. Smith, Mining requirements knowledge from collections of domain documents, in: 2016 IEEE 24th International Requirements Engineering Conference (RE), 2016, pp. 156–165.
[14] L. Villarroel, G. Bavota, B. Russo, R. Oliveto, M. Di Penta, Release planning of mobile apps based on user reviews, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016, pp. 14–24.
[15] L. Shi, M. Xing, M. Li, Y. Wang, S. Li, b. I. n. I.C. o. S.E.I. Wang, Qing, Detection of hidden feature requests from massive chat messages via deep siamese network, 2020, pp. 641–653. .
[16] C. Gao, J. Zeng, M.R. Lyu, I. King, Online app review analysis for identifying emerging issues, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), 2018, pp. 48–58.
[17] C. Gao, W. Zheng, Y. Deng, D. Lo, J. Zeng, M.R. Lyu, I. King, Emerging app issue identification from user feedback: Experience on wechat, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2019, pp. 279–288.
[18] C. Gao, J. Zeng, Z. Wen, D. Lo, X. Xia, I. King, M.R. Lyu, Emerging app issue identification via online joint sentiment-topic tracing, IEEE Transactions on Software Engineering (2021), 1–1.
[19] M. Häring, C. Stanik, W. Maalej, Automatically matching bug reports with related app reviews, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021.
[20] H. Jiang, J. Zhang, X. Li, Z. Ren, D. Lo, X. Wu, Z. Luo, Recommending new features from mobile app descriptions, ACM Transactions on Software Engineering and Methodology (TOSEM) 28 (2019) 1–29.
[21] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, L. Xu, Storydroid: Automated generation of storyboard for android apps, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 596–607.
[22] A. Ciurumelea, A. Schaufelbühl, S. Panichella, H. Gall, Analyzing reviews and code of mobile apps for better release planning, in: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017, pp. 91–102.
[23] C. Tao, H. Guo, Z. Huang, Identifying security issues for mobile applications based on user review summarization, Information and Software Technology 122 (2020) 106290.
[24] G. Grano, A. Ciurumelea, S. Panichella, F. Palomba, H. Gall, Exploring the integration of user feedback in automated testing of android applications, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 72–83.
[25] T. Johann, C. Stanik, B. AlirezaM, W. Maalej Alizadeh, Safe: A simple approach for feature extraction from app descriptions and app reviews, in: 2017 IEEE 25th International Requirements Engineering Conference (RE), 2017, pp. 21–30.
[26] M. Harman, Y. Jia, Y. Zhang, App store mining and analysis: Msr for app stores, in: 2012 9th IEEE Working Conference on Mining Software Repositories (MSR), 2012, pp. 108–111.
[27] H. Wu, W. Deng, X. Niu, C. Nie, Identifying key features from app user reviews, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021.
[28] H. Malik, E. Shakshuki, W.-S. Yoo, Comparing mobile apps by identifying 'hot' features, Future Generation Computer Systems 107 (2020) 659–669.
[29] A. Hindle, C. Bird, T. Zimmermann, N. Nagappan, Relating requirements to implementation via topic analysis: Do topics extracted from requirements make sense to managers and developers?, in: 2012 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp 243–252.
[30] A. Gorla, I. Tavecchia, F. Gross, A. Zeller, Checking app behavior against app descriptions, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 1025–1035.
[31] C. Zhang, H. Wang, R. Wang, Y. Guo, G. Xu, Re-checking app behavior against app description in the context of third-party libraries., in: SEKE, 2018, pp. 665–664. .
[32] L. Yu, X. Luo, C. Qian, S. Wang, H.K. Leung, Enhancing the description-to-behavior fidelity in android apps with privacy policy, IEEE Transactions on Software Engineering 44 (9) (2017) 834–854.
[33] Y.L. Arnatovich, L. Wang, N.M. Ngo, C. Soh, A comparison of android reverse engineering tools via program behaviors validation based on intermediate languages transformation, IEEE Access 6 (2018) 12382–12394.
[34] Y. Liu, L. Liu, H. Liu, X. Wang, H. Yang, Mining domain knowledge from app descriptions, Journal of Systems and Software 133 (2017) 126–144.
[35] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, NIPS (2013).
[36] J.D. Choi, J. Tetreault, A. Stent, It depends: Dependency parser comparison using a web-based evaluation tool, in: Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), 2015, pp. 387–396.
[37] X. Gu, S. Kim, What parts of your apps are loved by users?, in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) , 2015, pp 760–770.
[38] F. Palomba, P. Salza, A. Ciurumelea, S. Panichella, H. Gall, F. Ferrucci, A.D. Lucia, Recommending and localizing change requests for mobile apps based on user reviews, in: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), 2017, pp. 106–117.
[39] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer Science & Business Media, 2012.
[40] A. Al-Subaihin, F. Sarro, S. Black, L. Capra, Empirical comparison of text-based mobile apps similarity measurement techniques, Empirical Software Engineering 24 (2019) 3290–3315.
[41] L. Huang, J. Ma, C. Chen, Topic detection from microblogs using t-lda and perplexity, in: 2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW), 2017, pp. 71–77.
[42] H. Yu, Y. Lian, S. Yang, L. Tian, X. Zhao, Recommending features of mobile applications for developer, in: International Conference on Advanced Data Mining and Applications, 2016, pp. 361–373.
[43] F. Mercaldo, C.A. Visaggio, G. Canfora, A. Cimitile, Mobile malware detection in the real world, in: 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), 2016, pp. 744–746.
[44] M. Li, Y. Yang, L. Shi, Q. Wang, J. Hu, X. Peng, W. Liao, G. Pi, Automated extraction of requirement entities by leveraging lstm-crf and transfer learning, IEEE International Conference on Software Maintenance and Evolution (ICSME) 2020 (2020) 208–219.