Science of Computer Programming 202 (2021) 102556

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

API recommendation for the development of Android App features based on the knowledge mined from App stores

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

[a] College of Computer Science and Technology, Jilin University, Changchun 130012, 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 20 May 2020
Received in revised form 30 September 2020
Accepted 30 September 2020
Available online 15 October 2020

Keywords:
API recommendation
App store mining
UI analysis
Reusable knowledge
Feature extraction

ABSTRACT

To improve the efficiency, developers tend to use APIs to avoid reinventing wheels in the development of Apps. However, there are thousands of APIs for various purposes, so it is difficult for developers to identify suitable APIs according to the functionalities to be realized. App stores manage millions of products, which embody the experience and wisdom of developers, and they provide valuable data resource for solving this problem. By summarizing the API usage for the same or similar functionalities in Apps, reusable knowledge can be mined for the API recommendation. In this paper, we utilize the data resource in App stores and provide an API recommendation method for the development of Android Apps. Firstly, by using UI elements as the bridge, we establish mapping relationships between functionalities and APIs. Secondly, we build a framework to describe functionalities of Apps in the same category, and utilize relationships between functionalities and APIs to construct the API knowledge for each node in the framework. Thirdly, we identify nodes according to queried features and show the API knowledge to developers by giving recommendation lists. We conducted experiments based on Google Play, and the result shows that our method has a good recommendation performance.

1. Introduction

In current society, mobile Apps have penetrated into every aspect of people's lives [1,2]. No matter for simple communication and entertainment, or sensitive activities such as bank transfers and e-business, people tend to do everything at their fingertips. To stay one ahead in the market, App developers need to release their products as soon as possible [3]. APIs (Application Programming Interface) are predefined functions that can be used in the development process, and they prevent developers from reinventing the wheel to improve the development efficiency [4]. However, with the rapid growth of the number of APIs, it is impossible for developers to systematically master the usage of all APIs to get the ones they need when they develop the features/functionalities of Apps.

In such condition, developers tend to describe their needs in natural language as the queries to find the proper API information from relevant documents (such as Android official documents) or community websites (such as StackOverflow). To reduce developers' effort in this process, many researches paid attention to mining these data resources for API recommendation [4,5]. Generally, these methods take the problems in the stage of programming as main objects to gain related

* Corresponding author at: College of Computer Science and Technology, Jilin University, Changchun 130012, China.
E-mail address: liuyuzhou@jlu.edu.cn (Y. Liu).

https://doi.org/10.1016/j.scico.2020.102556
0167-6423/

Fig. 1. Two screenshots from the App *Pixlr*.

API knowledge, and they are more adapted to answer the queries described at code level (e.g., *matrix rotation* and *clear StringBuilder*). However, the development of Apps focuses on features [6,7], so developers often find usable APIs starting from the feature level, especially at the beginning of development and for the novices, they usually want to firstly identify the APIs could be used for the App features to be developed. Facing the queries given from features, existing methods often fail to return valuable recommendations for suffering two important shortcomings:

- On one hand, limited by the information in the data resource being analyzed, a lot of queries cannot be recommended with proper APIs. For example, the documents *Android Tutorial* and *Android SDK Reference* are used as the data resource by [4] to mine the recommendation information of APIs, but this method cannot provide recommendations for many App features (even simple ones, such as *crop photo* and *add text*), because there is no information of realizing these features in the two kinds of documents.
- On the other hand, according to the information demands and expression habits of developers, features in the queries may be described differently, not only the words used by them (such as *photo* vs. *picture*) but also their abstract levels (such as *message* vs. *send message*). However, existing methods do not consider both of the expression characters, which means they could not match the queries with the API knowledge correctly, so they are not able to provide the proper information of APIs to satisfy developers' needs.

Different from existing methods, we try to tackle the problem of API recommendation from a new perspective: using App features as the skeleton to organize the information of APIs. Specifically, we choose App stores as the data resource to mine API knowledge for the recommendation based on two important reasons. Firstly, the huge App market contains almost all App feature information, so for the feature to be developed, there always exists Apps having same or similar functionality in App stores, and we can summarize the APIs used for realizing this functionality in these products as the basis to answer the queries related to the feature. Secondly, products in App stores provide diverse expressions on similar features and they also give their features in different granularities. By mining features and their relationships from large scale of Apps, we define a hierarchical feature framework and use it as the index of corresponding API knowledge. In this way, we can better meet the requirements of developers for identifying APIs that can be used to realize the features of their products.

To achieve the above goal, a key question need to be solved is how to establish the relationships between App features and APIs used for realizing them. Here, we use UI elements as the bridge to mine such relationships from App stores. UI (User Interface) is the place where users interact with Apps and the elements in it can reflect almost all the features of products [8]. Meanwhile, the UI elements call the corresponding code to realize their functionalities, and this allows us to identify the APIs employed by them in the products. In addition, the features in UI are often described clearly so that users can understand them easily, and such expressions are given by developers and likely to be also used in the queries for searching related APIs. This means that taking these expressions of features as the index is benefit for matching the API knowledge with the developers' queries more easily and directly.

In this paper, we propose an approach to summarize the API usage around App features for the API recommendation. Our approach faces to developers of Android Apps to help them realize products' features, and it consists of three parts. Firstly, we give a method to get features served by UI elements according to their attributes, such as *text label* and *idname*, and

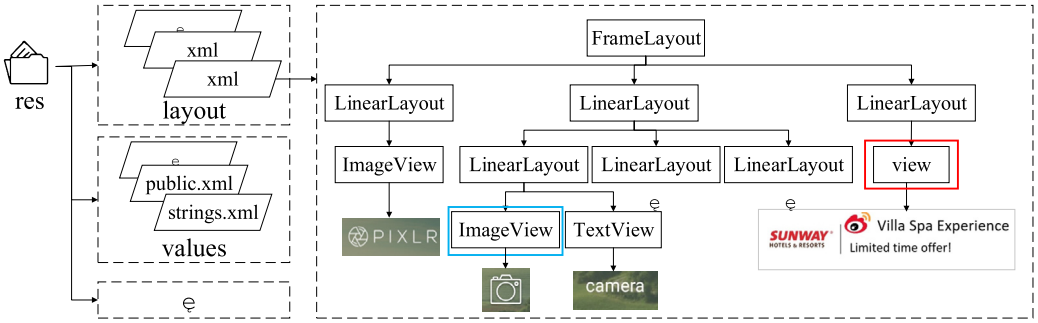

Fig. 2. The structure diagram of a *res folder*. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

gain APIs used by UI elements to establish mapping relationships between features and used APIs. Secondly, we organize features gained from the UI into a framework with a hierarchical structure, which nodes describe features of App products, and construct the API knowledge for each node based on relationships between features and APIs. Thirdly, we identify nodes in the feature framework based on keywords and expression forms in queries, and show their API knowledge to developers by giving recommendation lists, so that developers can use the recommendation information effectively. Overall, our major contributions include:

1) Give the method to get features from UI and trace APIs used by them from the code, thereby establishing the relationships between features and APIs. According to our knowledge, we firstly introduce such relationships to the work of API recommendation.
2) Define a feature framework to summarize various expressions of features and give features with different abstract levels, so that we can alleviate the impact of expression gap between functionality descriptions of API knowledge and queries of developers.
3) Evaluate the performance of our recommendation method by comparing it with an excellent method [4] based on the information collected from StackOverflow. The results show that our method can effectively recommend API information for the queries related to features: both the *Precision@N* and *Mean Average Precision@N* of our method are better than the compared method.

The paper is organized as follows. Section 2 gives the method for establishing relationships between features and APIs used by them. Section 3 shows the process of constructing the API knowledge system, and section 4 gives the method of recommending APIs for developers based on this knowledge system. In section 5, we introduce the experiment for evaluating our method. Finally, the related work and the conclusion are shown in section 6 and 7 respectively.

2. Establishing mappings between features and APIs

Users interact with Apps through UI, so developers would like to show the functionalities of their products in the UI elements clearly so that the users can use them easily. Moreover, developers can call relevant APIs to support the functionality realization of UI elements. For example, the screenshot on the left in Fig. 1 shows that a UI element of the App *Pixlr* serves the feature *camera*, and many APIs are used to realize the functionality of this UI element, such as *Camera.open*. This provides the possibility for us to establish the mappings between features and used APIs. We can extract UI elements from the UI of Apps in the category, and then identify features reflected by UI elements as well as APIs used by them to establish mappings between features and APIs.

2.1. Collecting UI elements

UI design of the Android App is controlled by xml files of the *layout folder* in *res folder*. The *res folder* is stored in the APK file and manages many important folders besides *layout folder*, such as *values folder*.

For getting the information related to UI elements, we parse the APK file of the App by using *Apktool*[1] to obtain the *res folder* and extract xml files in its *layout folder*, all xml files are used to established a set Set^{xml}. Each xml file manages a UI page of the App or part of the UI page, it organizes all UI elements in the file into a tree structure by using *View* and *ViewGroup* objects: *View*s are leaf nodes of the tree structure, and each *View* is a UI element; *ViewGroup*s are objects that hold other *View* objects in order to define the layout of the interface. For example, Fig. 2 gives the structure diagram of the *res folder* and further shows the tree structure of the xml file which is responsible for controlling the screenshot on the left

[1] https://ibotpeaches.github.io/Apktool/.

Table 1
The attributes related to binding text to UI elements.

android:text	*android:title*
android:textOn	*android:textOff*
android:contentDescription	*android:hint*
android:label	

in Fig. 1, leaf nodes (*ImageView*, *TextView* and *View*) of this tree structure represent UI elements of the product and they are organized by *ViewGroups* (*LinearLayout*).

By analyzing the tree structure of each xml file in Set^{xml} in turn, we can identify the UI elements in each file and get all UI elements of an App. Note that there are two types of UI elements: one type can serve one feature of the App, such as the leaf node in the blue box in Fig. 2; and the other type can render content for users, such as the leaf node in the red box. Only UI elements serving features can be used as data basis to establish mapping relationships between features and APIs, and we will filter out useless elements in our subsequent work.

For each available UI element, we can analyze relevant files and Java code to get its feature information as well as APIs called to realize its functionality, so that the mapping relationships between features and APIs can be gained.

2.2. Getting the feature information of UI elements

Developers can give text labels of UI elements through the attributes shown in Table 1, and these text labels can be used for showing the feature related information on the interface of the App, so that users can understand functionalities of corresponding UI elements. Thus, we first use text labels of UI elements as basis to get the texts related to their features. There are two kinds of ways to get the content of text labels for UI elements as follows:

- **Getting the content of text labels based on label assignments.** Developers can assign the content to text labels of UI elements in the *layout folder*, and we can get the content of text labels based on these label assignments in two ways. On one hand, developers usually give the content of text labels in the xml files of *layout folder*, and we can get the label content from the *layout folder* directly. Refer to Listing 1, developers directly assign the text label of the button, which id is *take*, with *take photo* in the *layout folder*. On the other hand, when the text label of a UI element is assigned with the form *@string/index* in the *layout folder*, we need to identify the content of the text label in the *string file* of *values folder*. Also refer to Listing 1, the label assignment of the button, which id is *send*, satisfies this form, and we can identify the content *send photo* of its text label in the *values folder* based on the index *label_send*.

layout folder

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent">
    <Button
        android:id="@+id/take"
        android:text="take photo" />
    <Button
        android:id="@+id/send"
        android:text="@string/label_send"
        android:onClick="click_send"/>
    <ImageView
        android:id="@+id/edit_text"
        android:src="@drawable/logo_text"/>
    <ImageView
        android:id="@+id/cropPhoto"
        android:src="@drawable/crop"/>
</LinearLayout>
```

Values folder

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="label_send">Send photo</string>
</resources>
```

Listing 1: The simple design example in the *res folder*.

- **Getting the content of text labels from Java code.** Developers can entirely or partially reuse the layouts in multiple ways [9], and the text information of UI elements in such layouts usually differs depending on purposes. Thus, developers

Java code

```
Button b1 = (Button)findViewById(R.id.camera);
b1.setText("make collage");

Button b2 = (Button)findViewById(R.id.take);
b2.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        //action after click button
    }
});
```

Listing 2: The simple code example in Java code.

often use the function *setText*() to re-assign text labels for these UI elements in Java code. Refer to Listing 2, by utilizing the function *setText*() in Java code, developers re-assign the text label of the button, which id is *camera*, with *make college*. For the UI element which text label has been modified in Java code, we take the modified text as the content of its text label.

For UI elements which text label shows the feature related information, we can understand their functionalities by analyzing the text labels. For example, for a button which *android:text* is assigned with *send photo*, we can understand from its text label that users can send photos to others by clicking this button. However, some UI elements do not have text labels or their text labels cannot reflect the information related to their features, such as UI elements which text label is assigned with *Yes* or *OK*. For these UI elements, we cannot get the texts related to their features through text labels. By observing and summarizing more than 2000 UI elements, we established a dictionary with the words which often appear in the interface of Apps but are unrelated to the feature information. In this process, we have invited three App developers with more than one year of industrial experience (do not include the authors) to observe words of each UI element separately, and only the words considered to have no feature information by all of them were added into the dictionary. With the enrichment of dictionary, it was difficult to discover new words, the process ended and the dictionary was gained. When analyzing a UI element, if it does not have the text label or the text of its text label is the word in this dictionary, our approach gets the text related to its feature according to the following two ways:

Firstly, we can get texts related to the feature information of UI elements from their sibling nodes in tree structures of the xml files. Around UI elements in the App interface, developers often explain what will happen if they are triggered, so we can understand the feature information of the UI elements through texts around them. In addition, the texts around UI elements usually can be gained by analyzing their sibling nodes in tree structures of the xml files. In this way, for the UI element which feature information cannot be obtained by analyzing the text label, we find out its sibling node in the tree structure that can express the text information, such as *TextView*, and use the content of the text label of this sibling node as the text related to the feature of UI element. Refer to the UI element in the dotted box in Fig. 1, it has no text label, but we can easily understand that its functionality is *camera* by observing the text around it, and we can get this information from its sibling node in the tree structure of xml file as shown in Fig. 2.

Secondly, we can also utilize the *idname* to analyze the feature information of UI elements. We notice that many developers assign the *idname* of UI elements with the text reflecting their features, and there are usually two assignment forms at this time. On one hand, the *idname* with the assignment form *word_word...word* usually contains useful information for understanding features of UI elements. Refer to the first *ImageView* in Listing 1, we can infer that this UI element provides the functionality *edit text* for users by analyzing its *idname*. On the other hand, the *idname* can be used to analyze the features of UI elements if developers use camel-case to assign the content for the *idname*. Refer to the second *ImageView* in Listing 1, it can be seen from its *idname* that users can crop photos by interacting with this UI element. Based on this observation, for the UI element which have no sibling nodes expressing the text information, we use the content of *idname* as the text related to the feature if its *idname* assignment satisfies the above two forms.

According to the above processing, we can get texts related to the feature for UI elements. Texts with feature related expression forms can be used as features of UI elements directly, such as *verb+noun* and *verb* [10]. However, the feature related texts of many UI elements are only descriptions about the functionalities and they contain the redundant information. For example, the text obtained from the text label of a UI element is *You can backup folders like Downloads and Screenshots*, we cannot use it as the feature. In our previous work [10], a method mining features from texts was provided, and we introduce it into this paper to get the feature information for texts without feature related expression forms. Here, we only give a brief introduction on the method as follows.

The kernel of our method is a set of feature extraction rules, which are gained by summarizing the relationship between structures of sentences and features. The rules can be classified into two kinds: one kind is tree transform rules, they aim at overall analyzing and transforming the parsing tree by replacing the pronouns with their anaphors and eliminating useless grammar structures; the other kind is information extraction rules, they define the operations to gain features from the transformed parsing tree.

Table 2
The important callback methods.

onClick	*onItemSelected*	*onKey*
onTextChanged	*afterTextChanged*	*onScroll*
onItemClick	*onEditorAction*	*onDrag*
onLongClick	*onHover*	*onChronometerClick*

Based on these extraction rules, we extract the information from texts without feature related expression forms as the features of corresponding UI elements. For the UI element mentioned above (the content of text label is *You can backup folders like Downloads and Screenshots*), we can extract the feature *backup folder* from the text label, and use it as the feature that this UI element serves.

2.3. Getting APIs used to realize functionalities of UI elements

The way for users to interact with an App is using touchscreen gestures such as touching and sliding, and Android Apps react to gestures by registering event listeners to corresponding UI elements [9]. After registering, UI elements will keep monitoring gestures and be triggered if matched. When a UI element is triggered, the UI element realizes its action by calling the *callback* method. Thus, by summarizing APIs called by the *callback* method and APIs called by custom methods that run when calling the *callback* method, we can get all APIs used to realize the functionality of a UI element. There are many *callback* methods in the Android development, and we show some important ones in Table 2.

In order to support the work of getting related APIs for a UI element, we establish a custom method call graph based on custom methods related to its functionality realization and call relationships of these methods. When a UI element is triggered, the application runs the *callback* method to further call other relevant custom methods, so the custom method call graph of UI element is a tree structure and the *callback* method is the root node of this tree. For establishing the custom method call graph of UI element, we first need to identify its *callback* method.

In Android Apps, each xml file in the *layout folder* corresponds to a piece of Java code, the *callback* methods of UI elements controlled by the xml file are written on it. *Public file* in the *values folder* shows the mapping relationship between the file name of each xml file as well as its *id*, and we can identify the *id* of corresponding xml file for each piece of Java code through the function *setContentView*(*xmlFileId*) in Java code. Thus, we can find out the corresponding piece of Java code for each xml file according to the file name, which is shown directly in the *layout folder*. In addition, there are two ways for a UI element to define the *callback* method:

- **Defining *callback* methods in the *layout folder*.** Developers can define the *callback* methods for UI elements in the xml files of *layout folder* directly. Refer to Listing 1, the *callback* method of the button, which id is *send*, is defined in the xml file, and the method *click_send* in Java code is its *callback* method.
- **Defining *callback* methods in Java code.** As shown by the button *b*2 in Listing 2, developers can use the functions *findViewById*(*uiElementId*) and *setOnClickListener*() to define *callback* methods of UI elements in Java code.

According to the above two definition ways, we identify the *callback* method in the corresponding piece of Java code for the UI element, and use it as the root node of the custom method call graph. Note that some UI elements do not have the *callback* method and we cannot establish the custom method call graph for them. These UI elements only render text information for users, such as advertisements, rather than serving functionalities of Apps, so we do not use them as the data to mine mappings between features and APIs.

By using *androguard*,[2] we can get call relationships between custom methods. For example, the call relationships $< m_1 \rightarrow m_2 >$ and $< m_1 \rightarrow m_3 >$ denote that the custom method m_1 calls m_2 and m_3 when m_1 runs. Combining this information with the *callback* method, we establish the custom method call graph of each UI element, a tree structure which uses custom methods related to the functionality realization as nodes and uses call relationships of these custom methods as edges. Fig. 3 shows part of the custom method call graph for a UI element (its feature is *rotate photo*) of *Pixlr*, it can be seen that we can get an API set $Set^{methodAPI}$ for each node in the tree structure according to used APIs. Based on API sets of all nodes, we further get a API set Set^{uiAPI} for each UI element as follows:

$$Set^{uiAPI} = \cup_{N \in Set^{node}} N.Set^{methodAPI}$$

where Set^{node} is the set of nodes in the custom method call graph, and $N.Set^{methodAPI}$ is $Set^{methodAPI}$ of the node N. Note that our method for extracting the information of APIs is not affected by code obfuscation because it does not rely on the semantic information in the code. For example, it can be seen from the name of methods in Fig. 3 that the code of *Pixlr* has been obfuscated, but we can still establish call relationships between methods and identify the API information in methods to get APIs used by UI elements.

[2] https://pypi.org/project/androguard/.

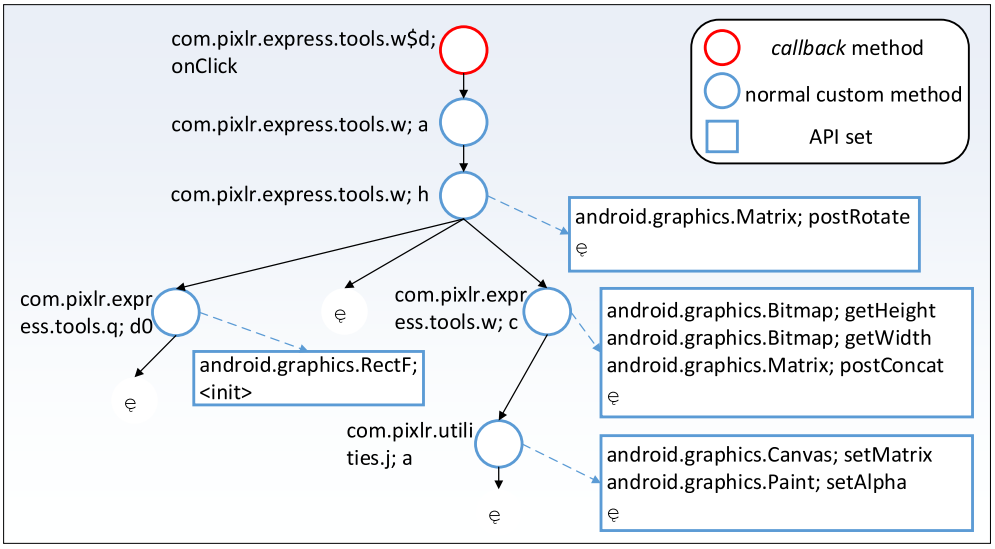

Fig. 3. Part of custom method call graph of a UI element of *Pixlr*.

After the process described above, we get the features of UI elements as well as APIs used to realize UI elements' functionalities. The UI elements, which can get the feature information and which Set^{uiAPI} is not null, can be used to establish mappings between features and used APIs. The feature F and the set Set^{uiAPI} of a UI element construct a mapping $M = (F, Set^{uiAPI})$, and all mappings of the category establish a mapping set Set^{M}.

3. Constructing the API knowledge system

After getting the mappings between features and APIs used to realize them, we further summarize the results to construct a knowledge system for the API recommendation. This section describes the construction process of API knowledge system in detail.

3.1. Filtering the mappings in Set^{M}

Although different products could have similar features, they may achieve the features with different quality. Considering that the low-quality features have the possibility of using APIs unreasonably [4], we filter their related mappings from Set^{M} and construct the API knowledge system only by analyzing the features with high completion quality.

In our previous work [11], we have provided an effective method to evaluate the completion quality of features by using user reviews. We make some improvements to this method and introduce it into this paper to complete the task of filtering the mappings. The reviews of each App are used to establish a review set Set^{Rev}, and we pre-process reviews in it by: modifying the typos and contractions in reviews based on the method given in [12]; and splitting reviews into review sentences. We evaluate the completion quality for the feature of each mapping in Set^{M} based on Set^{Rev} of the corresponding App, and this process consists of two steps: associating reviews with the feature and mining the sentiments in associated reviews to evaluate the completion quality of the feature.

As nouns and verbs are very important for expressing the information in texts [10], we identify reviews related to features by comparing nouns and verbs in them. For associating reviews with features, we tag the POS of words in reviews with NLTK [13], and nouns and verbs of each review are used to establish a set $Set^{Rev\_keyword}$. In this process, we treat continuous nouns and the verbal phase as a noun and a verb. There are two reasons: on one hand, continuous nouns in reviews, such as *album preview*, express a specific semantic meaning, and one of words cannot reflect it; on the other hand, the verb in a verbal phase cannot describe the accurate semantic meaning alone, such as the verb *turn* in the verbal phase *turn off*.

Analogously, all nouns and verbs of a feature also construct a word set $Set^{F\_keyword}$, and continuous nouns and verbal phases in features are processed in the same way as reviews. In this process, the POS of words in features is tagged based on App descriptions: for a word, we search it in App description texts, and identify its POS in the sentences of descriptions with NLTK, the POS with the most frequency is used as the POS of this word. The reason why we utilize App descriptions to tag the POS of words in features rather than tagging them with NLTK directly is that: NLTK tag the POS of words in sentences based on the contexts in sentences, but some features gained from UI elements only consist of a few word; moreover, both App descriptions and texts gained from UI elements are given by developers, so they are consistent in the usage of most words: according to our observation on the practice data, the POS tag of about 80% of words appearing in both App descriptions and UI elements is the same.

After gaining the keywords from reviews and features, we identify their relationships by calculating the relevancy. Suppose there are a review Rev and a feature F, the relevancy between them is determined by two parameters S_{noun} and S_{verb}, the calculation formulas for them are as follows:

$$S_{noun} = \max_{\substack{word_i \in Set^{Rev\_keyword},\ POS = noun \\ word_j \in Set^{F\_keyword},\ POS = noun}} similarity\left(word_i, word_j\right),$$

$$S_{verb} = \max_{\substack{word_i \in Set^{Rev\_keyword},\ POS = verb \\ word_j \in Set^{F\_keyword},\ POS = verb}} similarity\left(word_i, word_j\right),$$

where $similarity\left(word_i, word_j\right)$ represents the Euclidean distance of vectors between $word_i$ and $word_j$, the vectors of words are obtained by using word2vector [14], and the vector average of words in continuous nouns (or verbal phase) is used as its vector. Based on S_{noun} and S_{verb}, we give the formula to calculate the relevancy between the review and the feature as follows:

$$Relevancy\left(Rev, F\right) = w_n S_{noun} + w_v S_{verb},$$

where w_n and w_v are the weights of S_{noun} and S_{verb} respectively, and $w_n + w_v = 1$. We can adjust weights of S_{noun} and S_{verb} according to the character of different categories. The weight of S_{noun} is usually higher than S_{verb} because the nouns are more important for expressing the information than verbs.

For a review Rev and a feature F, we think Rev should be associated with F if their relevancy is bigger than the set threshold $\theta_{relevancy}$. Based on this method, we can get related reviews for a feature from the corresponding Set^{Rev}.

After getting related reviews for a feature, we mine sentiments in reviews associated with it to evaluate its completion quality. Each review has a rating value, but it is the comment of the user on the whole App rather than a specific feature, so we cannot use it as the sentiment value of the review. For getting a value that can reflect the sentiment of each review, we use the function *sentiment*() in the natural language processing package *pattern.en* provided by *Python* to analyze the sentiment in the review, and get a sentiment value $Senti$, which value range is $[-1, 1]$ from the most negative to the most positive.

Based on the sentiment values of associated reviews, we give calculation formula to evaluate the completion quality of each feature F:

$$Quality\left(F\right) = \frac{\sum_{Rev \in Set^{association}} Rev.Senti}{\mid Set^{association} \mid},$$

where $Set^{association}$ is a set consisting of reviews associated with F, $\mid Set^{association} \mid$ is the number of reviews in $Set^{association}$, and the value range of $Quality\left(F\right)$ is $[-1, 1]$ from the worst to the best.

We set a threshold $\theta_{quality}$, and mappings which feature has the completion quality value lower than $\theta_{quality}$ are removed from Set^M. Note that if one feature is only associated with a few number of reviews (less than the threshold θ_{review}), we cannot identify its completion quality reasonably because the value would be affected by the basis of a few users seriously, so we just keep the mapping related to it in Set^{M} directly.

3.2. Integrating the information contained in mappings of Set^M

For achieving the goal of recommending APIs, we construct the API knowledge system by integrating the information contained in mappings of Set^M, and this process includes: organizing features into a hierarchical feature framework, which nodes in different levels describe features with different granularities; and summarizing the API knowledge for nodes in the framework based on mappings between features and APIs.

3.2.1. Establishing the feature framework

We organize features gained from UI elements into a three-level tree structure and use it as the index of API knowledge. Fig. 4 shows part of the feature framework of photograph Apps, we use it as the example to illustrate the meaning and construction process of each level in the feature framework. Note that the features shown in the framework are common for better readability.

Category level

The category level is the root node of feature framework, it shows the category that feature framework belongs to. We can use the category given in App stores (such as Google Play) as this level. For example, it can be seen from the category level in Fig. 4 that this feature framework organizes features of Apps in the category “Photograph”.

Object level

Nodes in the object level describe more high-level features and show the objects that features operate on. For example, the node *photo* of the object level in Fig. 4 denotes that features related to this node give operations on the photo.

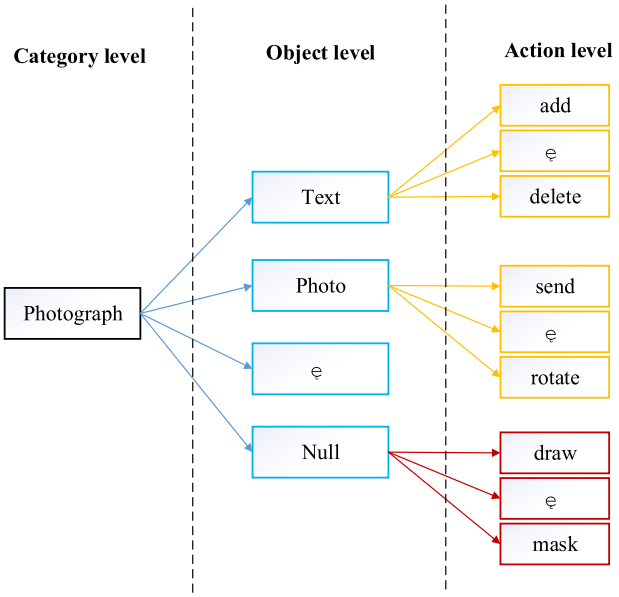

Fig. 4. The feature framework of Photograph Apps.

Nouns in features of mappings in Set^M are the basis of constructing the object level and we use them establish a noun set Set^{noun}. Based on vectors of words in Set^{noun} which are gained by using word2vector, we utilize k-means [15] to cluster them to group nouns with the similar meaning together. The center word of each cluster is used as a node in the object level, and all words in a cluster construct a word set $Vocabulary$ for corresponding object node. The $Vocabulary$ is responsible for summarizing various expressions of nodes. To identify the best solution, we use elements silhouette discussed in [16] to identify the best number of clusters. Besides these noun nodes, we also define a special node *null* and the features related to it do not have the operation object.

Action level

Nodes in the action level represent operations on objects in the object level, and there are two kinds of nodes in this level as shown in Fig. 4. On one hand, each node in the yellow box with its parent node together can generate a feature, for example, the node *rotate* and its parent node *photo* give a feature *rotate photo*. On the other hand, the parent node of nodes in the red box is the node *null*, and they are features in themselves without considering the parent node, such as *draw* and *mask*.

Verbs in features of mappings in Set^M are the basis of constructing nodes in the action level. For each node in the object level other than the node *null*, verbs in features, which contain the nouns in its $Vocabulary$, are used to establish the child nodes in the action level. To make sibling nodes different from each other, we cluster them with k-means and use the center word of each cluster as the node in the action level. All words of a cluster construct a word set $Vocabulary$ for the corresponding action node. For the node *null* in the object level, we use verbs in features, which do not contain nouns, as the basis to get its child nodes. Same as other nodes in the object level, we also cluster these verbs and use center words as child nodes of the node *null*.

3.2.2. Summarizing the API knowledge for nodes in the feature framework

By analyzing the relationships between features in mappings and functionalities represented by nodes in the feature framework, we associate mappings with nodes to get the related API knowledge. We analyze each mapping in Set^M in turn to identify their association nodes in the feature framework. For a mapping M, we process it in three different conditions according to the expression form of its feature:

- If the feature contains verbs but not nouns, the association node of M is the node which parent node is the node *null*, and its $Vocabulary$ includes the verb of this feature;
- If the feature contains nouns but not verbs, the node in the object level, which $Vocabulary$ includes the feature, is the association node of M;
- If the feature contains both nouns and verbs, the association node of this mapping is the node which $Vocabulary$ includes the verb part of the feature, and the $Vocabulary$ of its parent node includes the noun part of this feature.

Note that we define the associated mappings of the nodes in the action level of the framework are also associated with their parent nodes to support the work of summarizing the API knowledge for nodes in the object level.

After analyzing all mappings in Set^{M}, each node in the feature framework can be associated with relevant mappings, and these mappings construct a set Set^{nodeM}. Based on Set^{nodeM}, we establish a API set $Set^{nodeAPI}$ for each node as follows:

$$Set^{nodeAPI} = \cup_{M \in Set^{nodeM}} M.Set^{uiAPI}$$

Completing the above work, all the nodes in the feature framework gain their relevant API information, and the API knowledge system is constructed.

4. Recommending APIs based on the knowledge system

Based on the API knowledge system, we give the process of API recommendation, which consists of two steps. Firstly, we identify the query nodes in the feature framework according to keywords and expression forms of features in queries. Then, for helping developers use the information recommended by us effectively, we summarize the API knowledge of query nodes into recommendation lists.

4.1. Identifying the query nodes for queried features

When developers use our method, we identify the related nodes according to the features in their queries. Considering the expression form of the queried feature, there are three conditions in this process.

If the feature contains verbs but not nouns, we search the query node among child nodes of the node *null*. We calculate the relevancy between the feature and each node in turn, and the node which relevancy with the feature is the biggest is the query node. The calculation formula of the relevancy between a verb feature F and an action node $Node$ is as follows:

$$Relevancy^{verb}(F, Node) = \max\left(similarity(F.word, Node.word) \mid Node.word \in Node.Vocabulary\right),$$

where $F.word$ is the word of F, the vectorization of the verbs is the same as the processing when constructing the API knowledge system, particularly verbal phrases.

If the feature contains nouns but not verbs, we search the query node in the object level. The node with the most relevance to the feature is the query node, and the calculation formula of relevancy between a noun feature F and an object node $Node$ is shown as follows:

$$Relevancy^{noun}(F, Node) = \max\left(similarity(F.word, Node.word) \mid Node.word \in Node.Vocabulary\right),$$

where vectors of nouns are also gained by using word2vector, and the average of vectors of continuous nouns is used as their vector.

If the feature contains nouns and verbs, we search the query node in the action level (the child nodes of the node *null* are not considered). Similar as above, the node which relevancy with the feature is the biggest is the query node, and the calculation formula of the relevancy between the node in the action level and the feature containing nouns as well as verbs is:

$$Relevancy^{verb+noun}(F, Node) = w_v Relevancy^{verb}(F.verb, Node) + w_n Relevancy^{noun}(F.noun, parent(Node)),$$

where $F.verb$ and $F.noun$ are the verb part and the noun part of feature F respectively; $parent(Node)$ is the parent node of $Node$; w_v and w_n are the weights of the verb part and noun part respectively, $w_v + w_n = 1$, and we can adjust them according to the character of different categories.

4.2. Showing the API knowledge of query nodes

To enhance the usability of the recommended information in the development process, we summarize the API knowledge of query node into a recommendation list and provide it to developers. Each recommendation list consists of two parts: $List^{common}$ and $List^{special}$. We give the meaning and the establishment process of them as follows.

$List^{common}$ is an API list which is gained based on the usage frequency of APIs available for realizing the feature. For getting the $List^{common}$ of a node, we count the number of each API in its $Set^{nodeAPI}$, and then establish an ordered API list according to the obtained values.

$List^{common}$ takes usage frequency as the only factor for ranking APIs, and the ones that rank high in it may be not special APIs available for realizing the functionality represented by the node, but the common APIs which can be used to support a lot of features, such as APIs related to exceptions. Although $List^{common}$ can provide the useful information for developers when they develop App products, it is better if we can give special APIs that are important for the feature reflected by the

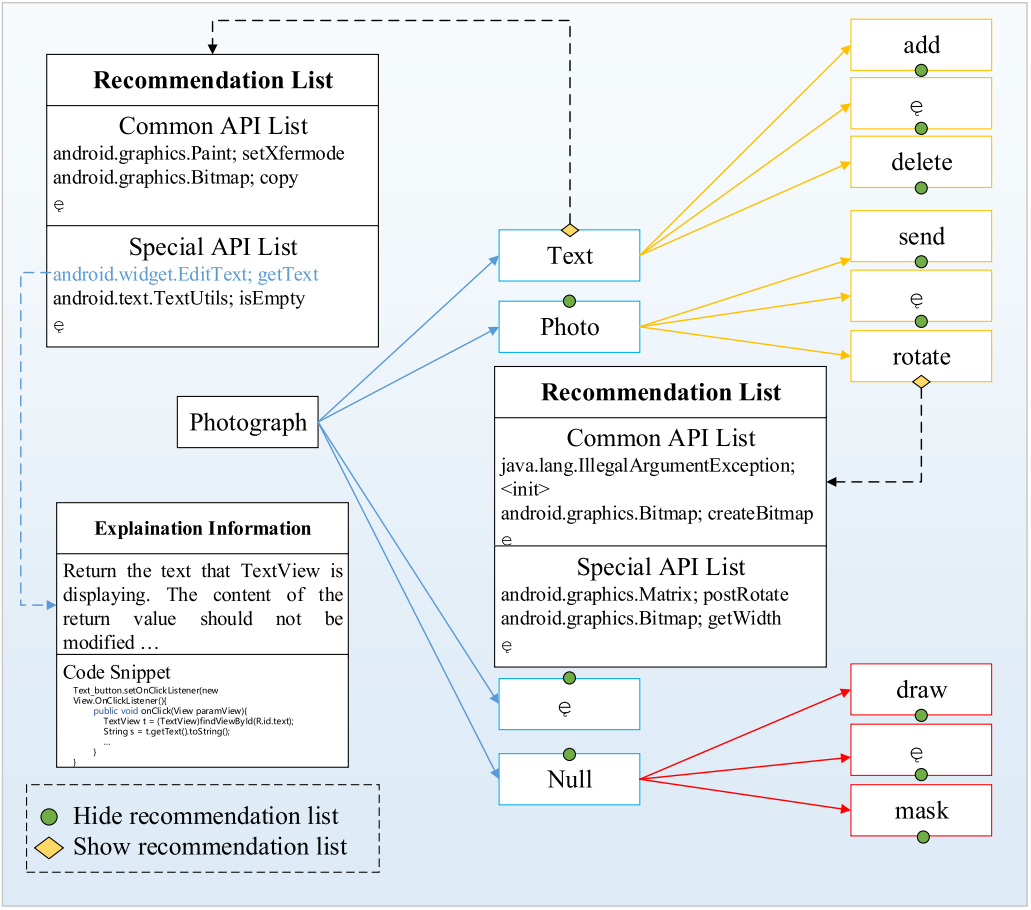

Fig. 5. The example of recommendation lists of features in Photograph category.

node. For achieving this goal, we utilize tf-idf [17] to calculate the importance of each API in $Set^{nodeAPI}$, and then rank them based on the importance values to establish an API list. We define this API list as $List^{special}$.

In order to evaluate the importance of APIs for the node in the object level, we first use $Set^{nodeAPI}$s of all nodes in the object level to establish a set Set^{API_Set}, and then use tf-idf to calculate the importance value of each API *api* based on Set^{API_Set}:

$$tf\ (api) = \frac{Set^{nodeAPI}\_Num(api)}{|\ Set^{nodeAPI}\ |};$$

$$idf\ (api) = log\left(\frac{\left|Set^{API\_Set}\right|}{Set^{API\_Set}\_Num\ (api) + 1}\right);$$

$$Importance\ (api) = tf\_idf\ (api) = tf\ (api) \times idf\ (api),$$

where $|\ Set^{nodeAPI}\ |$ is the number of APIs in $Set^{nodeAPI}$ of the corresponding node; $Set^{nodeAPI}\_Num(api)$ is the number of *api* in $Set^{nodeAPI}$; $\left|Set^{API\_Set}\right|$ is the number of API sets in Set^{API_Set}; and $Set^{API\_Set}\_Num\ (api)$ is the number of API sets, which contain *api*, in Set^{API_Set}.

The $Set^{nodeAPI}$s of all nodes in the action level also construct a set Set^{API_Set} to support calculating the importance values of APIs for the node in the action level, and the calculation method is the same as nodes in the object level.

For helping developers understand and use recommended APIs, we also give the explanation information for each API in the recommendation lists, and the explanation information consists of two parts: on one hand, we give the explanation text about the API's functionality by mining Android official documents; on the other hand, we show the usage demonstration of API, which is gained from Java code of existing Apps. Fig. 5 shows part of the recommendation lists of features *text* and *rotate photo*, and it also gives the presentation about the explanation information of a recommended API *android.widget.EditText; getText*.

Table 3
The information of the experimental data.

Category	Number of Apps used to construct API knowledge system
Photograph	3287
Navigation	2660
Social	3521
Music&Audio	3153

Table 4
Accuracy scores of the feature supplement data for UI elements.

Category	*Accuracy* of texts gained from sibling nodes	*Accuracy* of *idnames*
Photograph	0.92	0.79
Navigation	0.90	0.84
Social	0.89	0.91
Music&Audio	0.95	0.86
Average	0.92	0.85

5. Experiments and results

We aim to answer the following two questions:

- Question 1: Whether our method can gain the required information from the data effectively?
- Question 2: Whether our method can recommend suitable APIs to developers of Android Apps according to queried features?

For answering the above questions, we chose 4 categories of Apps on Google Play as experimental subjects, including: "Photograph", "Navigation", "Social" and "Music&Audio". There are two main reasons for choosing these categories: on one hand, the features of products in these categories are quite different from each other, so we believe that using them as experimental subjects can validate the generalization of our method; on the other hand, choosing these categories as subjects is conducive to evaluating the performance of our method, because we want to analyze the performance of our API recommendation method based on the information in StackOverflow, in which there are many discussions about realizing the functionalities of selected categories.

For each category, we firstly selected about 3000 products with a rating score greater than 4.5, and wrote a crawler[3] to get their APK files from *APK Download*.[4] The specific number of Apps in each category is shown in Table 3. Then, we analyzed each APK file according to our method to get mappings between features and used APIs. Our code for processing APK files was provided online,[5] and we also showed several APK files and the results gained after processing them with our code.[6] Finally, we filtered mappings with user reviews (the threshold of filtering mappings $\theta_{quality}$ is set to -0.5 in our experiment) and constructed the API knowledge system based on the remaining mappings. Reviews of selected products were gained from Google Play directly by using our crawler.[7]

The participants in our experiments were doctors from Jilin University in China, they majored in software engineering, especially requirements engineering and data mining, and all of them had more than one year of industrial experience in the App development. They were responsible for labeling the samples of Question 1 and giving the features used in Question 2.

5.1. The experiment for Question 1

In the previous papers [9–11], many parts of the method that we use to get the required information have been validated, so we no longer do experiments to measure their effectiveness. For UI elements which cannot gain the feature information by analyzing text labels, our method proposes to use text labels of sibling nodes in the tree structures of xml files and *idname*s as the supplementary data to mine their features, and we only constructed an experiment to validate the effectiveness of this part for answering Question 1.

5.1.1. Experimental design for Question 1

In the process of constructing the API knowledge system, we used text labels of sibling nodes and *idname*s as the supplementary data to get the feature information for some UI elements. For answering Question 1, we randomly chose

[3] https://github.com/gaoshanquan/CodeAndData/tree/master/Crawler/APK.
[4] https://apps.evozi.com/apk-downloader/.
[5] https://github.com/gaoshanquan/CodeAndData/tree/master/AnalyzeAPK.
[6] https://github.com/gaoshanquan/CodeAndData/tree/master/example.
[7] https://github.com/gaoshanquan/CodeAndData/tree/master/Crawler/review.

Table 5
The number of features in each category after filtering.

	Number of features before filtering	Number of features after filtering
Photograph	80	52
Navigation	80	47
Social	80	43
Music&Audio	80	57
Total	320	199

100 text labels of sibling nodes and 100 *idname*s from the data used to mine features of UI elements to establish two test data sets for each selected category. Then, 3 raters labeled samples in test data sets separately: the label is *correct* if the rater thinks the text of sample can reflect the feature information and *incorrect* otherwise. The consensus labels were used as labels of samples directly. For disagreements, the doctors discussed together and then voted, only the samples which reach an agreement by all raters were used in the experiment. After the first round, the Fleiss' kappa value [18] of labeling samples is 0.65, and this indicates that the result of inter-rater agreement is acceptable. Based on the labels of samples in test data sets, we evaluate whether our supplementary data can be used to analyze features of UI elements as follows:

$$Accuracy = \frac{|correctSample|}{|testDataSet|},$$

where $|testDataSet|$ is the number of samples in the test data set; $|correctSample|$ is the number of samples, which label is *correct*, in the test data set.

5.1.2. Experimental result for Question 1

Table 4 shows the experimental result of Question 1, and it can be seen that our supplementary data can provide feature related texts for UI elements which cannot gain the feature information by analyzing text labels: the accuracy values of text labels gained from sibling nodes are from 0.89 to 0.95, and the average accuracy value of *idname*s is 0.85.

As mentioned earlier, developers usually give feature related explanations around UI elements without textual information, so it is reasonable to utilize the text labels of sibling nodes in tree structures of xml files to get the feature information for these UI elements. In fact, many *idname*s that developers give do not contain the information related to the features, but our method only chooses the *idnames*, which meet the expression forms given by us, as the supplemental data, so we can get a high accuracy. In addition, there are 8% of text labels gained from sibling nodes and 15% of *idname*s cannot be used to analyze the features of UI elements, and we summarized two main reasons for the invalid data through observation. On one hand, some UI elements without the textual information can express their functionalities through images, and UI elements of their sibling nodes in the tree structures of xml files render the information unrelated to them, such as advertisements. On the other hand, some *idname*s meet the expression forms given by us, but developers add contents unrelated to features in them, such as the *idname dialog_01_outer_space*.

In summary, our method can gain the feature information and API information from the relevant data effectively.

5.2. The experiment for Question 2

In this section, we constructed an experiment to evaluate the performance of our method for recommending APIs according to queried features. The experimental design and the result are introduced in detail as follows.

5.2.1. Experimental design for Question 2

In each category, three doctors randomly selected 200 Apps together to identify features in their descriptions. In this process, we only considered the products with a rating score greater than 4.5, and this is because we believe that the descriptions of these Apps are of higher quality, which is conducive to the work of identifying the features. Only when a feature is identified by all doctors can we use it as the sample in our experiment. In the end, we randomly selected 80 different features from the samples for each category to evaluate the performance of our recommendation method. The evaluation process consists of two steps:

Firstly, we constructed a standard API set for each feature based on questions and answers on StackOverflow. We searched the features on StackOverflow to get questions related to their realization, and the expression form of input is *[android] + feature* for ensuring that the returned results are about the Android development. For example, for the feature *send message*, we searched *[android] send message* to get questions for it. By reading the descriptions of questions, we identified related questions for each feature, and features that cannot get any related question were filtered because we cannot evaluate the recommendation result for these features. The number of remaining features in each category is shown in Table 5. There were many answers under a question, and the answers, which is accepted by the questioner or which vote is bigger than 0, were considered valid. For each feature, all APIs in valid answers of related questions were used to establish a standard API set.

Table 6
The Precision scores of different categories.

	Precision@N					
	List^{special}			***List^{common}***		
	P@3	*P@5*	*P@10*	*P@3*	*P@5*	*P@10*
Photograph	0.50	0.52	0.73	0.27	0.29	0.35
Navigation	0.38	0.47	0.60	0.32	0.36	0.43
Social	0.63	0.65	0.74	0.21	0.26	0.33
Music&Audio	0.44	0.46	0.67	0.25	0.30	0.39
Average	0.49	0.53	0.69	0.26	0.30	0.38

Table 7
The Mean Average Precision scores of different categories.

	Mean Average Precision@N					
	List^{special}			***List^{common}***		
	MAP@3	*MAP@5*	*MAP@10*	*MAP@3*	*MAP@5*	*MAP@10*
Photograph	0.31	0.36	0.35	0.19	0.20	0.15
Navigation	0.25	0.29	0.30	0.22	0.27	0.19
Social	0.37	0.40	0.37	0.19	0.18	0.19
Music&Audio	0.29	0.31	0.34	0.15	0.23	0.09
Average	0.31	0.34	0.34	0.19	0.22	0.16

Secondly, we evaluated the performance of our method in each category by comparing the APIs that our method recommends and ones in the standard API sets. Our method can recommend two API lists $List^{special}$ and $List^{common}$ for a feature, and we compared APIs in lists and ones in the standard API set to evaluate the recommendations of two lists respectively. Based on the obtained results of all features, we can get the evaluation result for a category. Our experiment utilized two metrics to measure the performance of our method: *Precision@N* (*P@N*) and *Mean Average Precision@N* (*MAP@N*).

- The calculation formula for *Precision@N* is:

$$Precision@N = \frac{|\ Set^{matchF}\ |}{|\ Set^{F}\ |},$$

  where Set^{matchF} is a set of features, which top N recommended APIs in our list have at least one correct API (we stipulate that a recommended API is the correct if it is also in the standard API set); Set^{F} is the set of all features in the category; and $|\ Set^{F}\ |$ is the number of features in Set^{F}.
- *Mean Average Precision@N* is a popular metric for measuring recommendation algorithms in information retrieval. For calculating the value of $MAP@N$ of a category, we first need to calculate the value of $AP@N$ for each feature F as follows:

$$AP@N = \frac{\sum_{k=1}^{N}(P(k) \times rel(k))}{number\ of\ relevant\ APIs},$$

  where $P(k)$ is the precision at a cut-off rank k in the API list, the value of $rel(k)$ is 1 if the API at the rank k in the API list is also in the standard API set and 0 otherwise. The *number of relevant APIs* should be the number of APIs that can be used in the realization process of the feature, but we cannot get this value. Same as [4], for solving this problem, we also optimistically assume that all the top N APIs in our recommendation lists that match an answer from StackOverflow for a query question form the total set of relevant APIs for the query. Based on the $AP@N$ values of all features, we calculate the value of *MAP@N* for a category:

$$MAP@N = \frac{\sum_{F \in Set^{F}} F.AP@N}{|\ Set^{F}\ |}.$$

5.2.2. Experimental result for Question 2

Table 6 and 7 show the experimental result about evaluating the performance of our method for recommending APIs according to queries. According to the evaluation result, the precisions of $List^{special}$ in the recommendation lists can be up to 0.53 at top 5, and 0.69 at top 10, while the precision score of $List^{common}$ is 0.30 at top 5, and 0.38 at top 10 respectively. The $MAP@10$ scores of $List^{special}$ and $List^{common}$ are 0.34 and 0.16 respectively. It can be seen that the evaluation of $List^{special}$ gains a good result, but the performance of $List^{common}$ is unsatisfactory. We think this is reasonable, and it does not mean

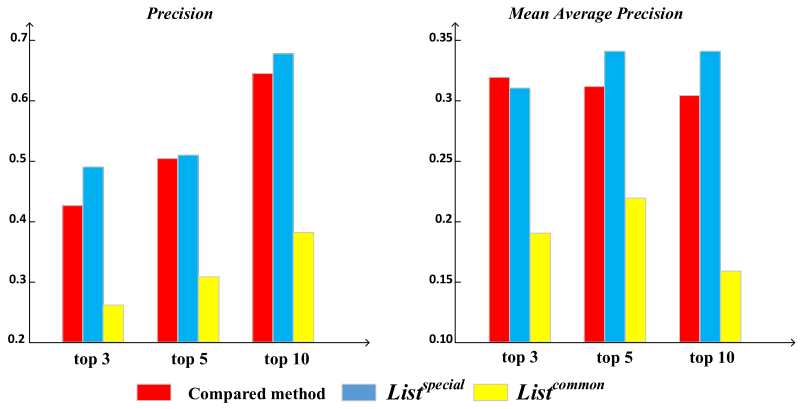

Fig. 6. The comparison results.

that APIs given in $List^{common}$ are not useful for the realization of queried features. The answers on StackOverflow usually give the core but incomplete code snippets, APIs in them are mainly special APIs related to the functionality realization, and the common APIs, which can also be used to support realizing queries, are usually not discussed on StackOverflow. For example, when a developer of social Apps asks the question about how to send message, answerers on StackOverflow usually only focus on the realization process of sending texts from a user to others without considering how to get texts that users enter, the API *android.widget.EditText; getText* recommended by $List^{common}$ can help developers complete this job, but it is not mentioned on StackOverflow.

From the experimental result, we can see that there are many inappropriate recommendations in both $List^{special}$ and $List^{common}$. We analyzed it and summarized three main reasons. Firstly, our method utilizes App descriptions to tag the POS of words in the features gained from UI or given by developers. Although this method can accurately tag the POS for most of words in features, it also gives some incorrect results. Secondly, the vectors of some words, which are gained by using word2vector, are unreasonable, and this leads to errors in calculating similarity values between different words. Thirdly, limited by the information on StackOverflow, API standard sets cannot summarize all APIs available for realizing queried features, which makes some correct recommendations considered inappropriate.

In order to further analyze the performance of our method, we compared our method with the API recommendation method proposed by [4], which also utilizes *Precision@N* and *Mean Average Precision@N* to evaluate the recommendation result. We quoted the performance of the compared method from its paper directly, and Fig. 6 shows the comparison results. As previously analyzed, the evaluation result of $List^{common}$ is unsatisfactory because the answers on StackOverflow usually do not discuss the common APIs, so we only considered the performance of $List^{special}$ when comparing the recommendation of two methods. When N=5, the precisions of two methods are similar, but the precision scores of ours are higher than the compared method if N is 3 or 10. Moreover, except that N=3, the mean average precision values of our method are higher than the compared method. We believe that our method gains better results by benefiting from three advantages:

Firstly, the features that our method can recommend for are more extensive. The compared method cannot recommend APIs for many features because the information gained from *Android Tutorial* and *Android SDK Reference* is limited, while our method can give relevant APIs to support the realization of almost all features by summarizing the API usage of App products with the same or similar features.

Secondly, the functionality information of our method is more accurate. Both our method and the compared method establish mapping relationships between functionalities and APIs to recommend APIs for developers. However, the functionality information of our method is more accurate, because we gain it from UI directly while the compared method uses tf-idf to extract keywords from relevant texts as functionalities of corresponding APIs.

Thirdly, the index of API knowledge that we use summarizes various expressions for features. Our method organizes features, which are gained from UI, into a hierarchical framework, and use it as the index of API knowledge. The framework summarizes various expressions for features by establishing the *Vocabulary* for nodes, and this makes our method can better match relevant APIs with developers' queries.

Note that we did not evaluate the performance of compared method on features used in our experiment to compare two methods on a common data set, this is because the research purpose of compared method is not completely consistent with our method. Compared method mainly provides suggestions for problems in code programming, and its effect on recommending APIs for realizing features is not ideal, so it is unfair for compared method 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 method.

5.3. Threats to the validity of experiments

Despite the encouraging results, this work has some potential threats to validity. We give a discussion on them from the following two aspects.

The reliability of labels of samples used in experiment. In the experiment answering Question 1, doctors give the labels of samples in test data sets, and we evaluate whether texts gained from our supplementary data can represent feature related texts of UI elements based on these labels. Labeling the samples is a very subjective job, and raters may give incorrect results. However, our labels were given by multiple doctors independently, and their results discussed to converge when discrepancies occurred. Thus, we believe the labels of samples in our experiment are reliable.

Standard API sets cannot summarize all valuable APIs for features used to evaluate our method. To evaluate the performance of our method for recommending APIs according to queried features, we compared APIs recommended by us and the ones in standard API sets, which were gained by analyzing questions and answers on StackOverflow. Limited by the information on StackOverflow, we cannot summarize all APIs available for realizing features used in the experiment in the standard API sets, and this makes some reasonable recommendations of our method considered inappropriate. However, even in such cases, the evaluation result of our method is still acceptable, so we can draw a conclusion: our method has the ability to recommend suitable APIs for queried features.

6. Related work

In our approach, we firstly analyze the UI of Apps, and use the elements in it as the bridge to establish the relationships between features and APIs; then, we further summarize the obtained results to construct the API knowledge system for API recommendation. Thus, the related work is discussed from the following two directions.

6.1. Studies on UI analysis of mobile Apps

There is a lot of valuable information in UI and many researchers use them to solve different problems. Here, we summarize the work related to us from two main topics.

On one hand, many studies analyze the relationship between UI and APIs (or permissions) to evaluate the security state of Apps. For example: Avdiienko Vitalii et al. firstly extract the Android APIs Apps invoke as well as the text shown on their screen, and then detect *outliers* (anomaly Apps) based on the association between APIs and texts [9]; Xi Shengqu et al. use deep learning techniques that jointly model icons and their contextual texts to learn an icon-behavior model, and detects intention-behavior discrepancies by computing the outlier scores based on the learned model [8]; Li Yuanchun et al. introduce the permission-UI mapping as an easy-to-understand representation to illustrate how permissions are used by different UI components within a given application, so that they can help users understand the purpose of permission requests [19]; AsDroid identifies the permissions related to UI components using static analysis and find out malwares by detecting mismatches between the permissions and the text extracted from UI components [20]. Similar to these studies, our method also considers the relationship between UI and APIs, but we apply it to the research of recommending APIs instead of analyzing the security state of Apps. We firstly extract the features UI elements can serve from relevant texts, and then identify APIs employed by UI elements to establish mapping relationships between features and APIs. These mapping relationships provide the data basis for us to get APIs available for realizing features.

On the other hand, many researchers provide the useful reference information for developers to support the development or evolution of their products by mining UI and relevant data. For example: Kevin Moran et al. leverages computer vision techniques and natural language generation to accurately and concisely summarize changes made to the GUI of a mobile App between successive commits or releases [21]; Sen Chen et al. extract the activity transition graph and leverages static analysis techniques to render UI pages to visualize the storyboard with the rendered pages [22]; Kevin Mran et al. introduce an approach for verifying whether the GUI of a mobile App was realized according to its intended design, this approach resolves GUI-related information from both realized Apps and mock-ups and uses computer vision techniques to identify common errors in the realizations of mobile GUIs [23]. With the similar pursuits as these methods, we want to provide effective support for developers during their development process. Using APIs prevents developers from reinventing the wheel, but it is hard for them to understand which APIs can help their development because there are thousands of APIs have been developed to cater for various purposes and developers are often unaware of the existence of APIs suitable for a particular feature of the product they are developing. For helping developers solve this problem, we further summarize relationships between features and APIs, which are gained by using UI elements as the bridge, to construct an API knowledge system, and recommend APIs for developers based on this knowledge system.

6.2. Studies on API recommendation

For helping developers improve the efficiency of the product development, many researchers propose methods to recommend suitable APIs for developers. We divide API recommendation methods into two types based on the analyzed data resource, and discuss them separately.

Firstly, many studies recommend APIs for developers by analyzing relevant texts combining natural language descriptions and code, and there are two important types of texts. On one hand, many researchers mine official documents, such as Android tutorials and SDK documents, to get recommendation information. For example: Weizhao Yuan et al. first build Android-specific API databases to contain the correlations among various functionalities and APIs, based on customized parsing of code snippets and natural language processing of texts in Android tutorials and SDK documents, and then textual and code similarity metrics are adapted for recommending relevant APIs [4]; For helping developers selecting suitable APIs on API documents, Haibo Yu et al. provide an API recommendation method named APIBook, this method combines semantic relevance, type relevance and the extent of degree that API method is used to sort these API methods and rank those that are highly relevant and widely used in the top positions [24]. On the other hand, many studies recommend suitable APIs for developers by analyzing questions and answers on StackOverflow. For example: Rahman Mohammad Masudur et al. propose a novel API recommendation technique-RACK that recommends a list of relevant APIs for a natural language query for code search by exploiting keyword-API associations from the crowdsourced knowledge of StackOverflow [25]; Thanh Nguyen et al. provide a method to derive the API elements relevant to the task described in the input by statistically learning from a StackOverflow corpus of text descriptions and corresponding code, and the inferred API elements with their relevance scores are ensemble into an API usage by their API usage synthesis algorithm that learns the API usages from a large code corpus via a graph-based language model [26]; Miltiadis Allamanis et al. provide a method to recommend APIs for developers based on the information gained by leveraging data that has short natural language utterances paired with source code snippets, like titles of questions along with source code found in answers from StackOverflow [27]. These methods rely on the information in relevant texts to recommend APIs for developers, but features that they can give recommendations for are limited, this is because that these texts mainly discuss how to solve the problems in code programming, such as *matrix rotation*, rather than giving the information about realizing features of products. Different from them, our method takes providing valuable recommendations about APIs for features that developers query as the primary pursuit, and can recommend suitable APIs for extensive features.

Secondly, much research provides the recommendations for developers by analyzing the products themselves. For example: Ferdian Thung et al. propose a method to recommend APIs for developers, this method firstly learns from a training dataset of changes made to a software system recorded in repositories (each change in the dataset has three parts: the textual description describing the change, the code before the change and the code after the change), and then recommends APIs to developers according to queried features [28]; Ferdian Thung et al. also propose an automated approach called WebAPIRec that takes as input a project profile and outputs a ranked list of web APIs that can be used to realize the profile, WebAPIRec learns a model that minimizes the incorrect ordering of web APIs based on the historical data of web API usages [29]; Anh Tuan Nguyen et al. propose an API recommendation method that taps into the predictive power of repetitive code changes to provide relevant API recommendations for developers, this method is based on statistical learning from fine-grained code changes and from the context in which those changes were made [30]. Similar as these research, our method also analyzes products themselves to achieve the goal of recommending APIs for developers, but the data resource we use is App stores. This makes our method can give recommendations for more extensive features because we can get relevant information by mining the API usage of products with the same or similar features. In addition, by summarizing App features provided by App stores, we can get various expressions on similar features and features with different abstract levels to construct a comprehensive index of API knowledge, and it helps us better match relevant APIs with queries that developers give.

7. Conclusion

Products in App stores embody the experience and wisdom of developers, and we summarize the API usage of Apps with the same or similar features as the reusable knowledge for API recommendation. The method firstly utilizes UI elements as the bridge to establish mapping relationships between features and APIs used to realize them. Then, we construct the API knowledge system by integrating the information contained in mapping relationships between features and APIs. Finally, based on the keywords and expression forms of features in the queries, we identify the corresponding API knowledge and summarize it into recommendation lists to help developers use the information effectively. The results of experiments showed that our method can gain the feature information and API information from relevant data accurately (the average accuracy of supplementary data used for mining features of UI elements without textual information is 88.50%). In addition, by designing an experiment based on questions and answers on StackOverflow, we validated that the proposed method can give valuable APIs for queried features (*Precision@10* and MAP@10 of the $List^{special}$ in our recommendation lists can be up to 0.69 and 0.34 respectively).

In the future, we will consider mining the relevant data, such as API official documents and Java code of existing products, to recommend the combinations of APIs for features to be realized, so that developers can develop their products more effectively.

CRediT authorship contribution statement

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

Acknowledgements

The work is funded by National Key Research and Development Program of China 2017YFB1003103, Natural Science Research Foundation of Jilin Province of China under Grant Nos. 20190201193JC.

References

[1] W. Martin, F. Sarro, Y. Jia, et al., A survey of App store analysis for software engineering, IEEE Trans. Softw. Eng. 99 (2017) 817–847.
[2] Keng-Pei Lin, Yi-wei Chang, Chih-Ya Shen, Mei-Chu Lin, Leveraging online word of mouth for personalized App recommendation, IEEE Trans. Comput. Soc. Systems 5 (2018) 1061–1070.
[3] G. Hu, X. Yuan, Y. Tang, et al., Efficiently, effectively detecting mobile App bugs with AppDoctor, in: European Conference on Computer Systems, ACM, 2014.
[4] W. Yuan, H.H. Nguyen, L. Jiang, et al., API recommendation for event-driven Android application development, Inf. Softw. Technol. 107 (MAR.) (2019) 30–47.
[5] L. Cai, Haoye Wang, Q. Huang, Xin Xia, Zhenchang Xing, D. Lo, BIKER: a tool for Bi-information source based API method recommendation, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019.
[6] Federica Sarro, Afnan A. Al-Subaihin, M. Harman, Yue Jia, W. Martin, Yuanyuan Zhang, Feature lifecycles as they spread, migrate, remain, and die in App stores, in: 2015 IEEE 23rd International Requirements Engineering Conference (*RE*), 2015, pp. 76–85.
[7] Jiang He, Jingxuan Zhang, Xiaochen Li, Zhilei Ren, David Lo, Xindong Wu, Zhongxuan Luo, ACM Trans. Softw. Eng. Methodol. 28 (4) (2019) 1–29.
[8] Shengqut Xi, et al., DeepIntent: deep icon-behavior learning for detecting intention-behavior discrepancy in mobile Apps, in: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, 2019.
[9] Vitalii Avdiienko, Konstantin Kuznetsov, Isabelle Rommelfanger, Andreas Rau, Alessandra Gorla, Andreas Zeller, Detecting behavior anomalies in graphical user interfaces, in: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), 2017, pp. 201–203.
[10] Yuzhou Liu, Lei Liu, Huaxiao Liu, Xiaoyu Wang, Hongji Yang, Mining domain knowledge from app descriptions, J. Syst. Softw. 133 (2017) 126–144.
[11] Yuzhou Liu, et al., Combining goal model with reviews for supporting the evolution of apps, IET Softw. 14 (2020) 39–49.
[12] P.M. Vu, T.T. Nguyen, H.V. Pham, et al., Mining user opinions in mobile App reviews: a keyword-based approach, in: 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, pp. 749–759.
[13] E. Loper, S. Bird, NLTK, the Natural Language Toolkit, 2002, pp. 63–70.
[14] L. Long, Accurate delivery analysis of distributed E-commerce based on, in: Word2vector. International Conference on Machine Learning, Big Data and Business Intelligence (MLBDBI), 2019.
[15] Jin Hua Xu, Hong Liu, Web user clustering analysis based on KMeans algorithm, in: International Conference on Information, NETWORKING and Automation, IEEE, 2010, V2-6-V2-9.
[16] Peter Rousseeuw, Silhouettes: a graphical aid to the interpretation and validation of cluster analysis, J. Comput. Appl. Math. (1987).
[17] A. Aizawa, An information-theoretic perspective of tf-idf measures, Inf. Process. Manag. 39 (1) (2003) 45–65.
[18] R. Falotico, P. Quatto, Fleiss' kappa statistic without paradoxes, Qual. Quant. 49 (2015) 463–470.
[19] Yuanchun Li, Yao Guo, Xiangqun Chen, PERUIM: understanding mobile application privacy with permission-UI mapping, in: UbiComp '16, 2016.
[20] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, Bin Liang, AsDroid: detecting stealthy behaviors in Android applications by user interface and program behavior contradiction, in: Proceedings of the 36th International Conference on Software Engineering (ICSE 2014), 2014, pp. 1036–1046.
[21] Kevin Moran, Cody Watson, John Hoskins, George Purnell, Denys Poshyvanyk, Detecting and summarizing GUI changes in evolving mobile apps, in: ASE 2018, 2018.
[22] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, Lihua Xu, StoryDroid: automated generation of storyboard for Android apps, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 596–607.
[23] Kevin Moran, Boyang Li, Carlos Bernal-Cárdenas, Dan Jelf, Denys Poshyvanyk, Automated reporting of GUI design violations for mobile Apps, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), 2018, pp. 165–175.
[24] Haibo Yu, Wenhao Song, Tsunenori Mine, APIBook: an effective approach for finding APIs, in: Internetware '16, 2016.
[25] Mohammad Masudur Rahman, Chanchal Kumar Roy, David Lo, RACK: automatic API recommendation using crowdsourced knowledge, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016, pp. 349–359.
[26] Thanh van Nguyen, Peter C. Rigby, Anh Tuan Nguyen, Mark Karanfil, Tien N. Nguyen, T2API: synthesizing API code usage templates from English texts with statistical translation, in: FSE, 2016.
[27] Miltiadis Allamanis, Daniel Tarlow, Andrew D. Gordon, Yi Wei, Bimodal modelling of source code and natural language, in: ICML, 2015.
[28] Ferdian Thung, Shaowei Wang, David Lo, Julia L. Lawall, Automatic recommendation of API methods from feature requests, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2013, pp. 290–300.
[29] Ferdian Thung, Richard Jayadi Oentaryo, David Lo, Yuan Tian, WebAPIRec: recommending Web APIs to software projects via personalized ranking, in: IEEE Transactions on Emerging Topics in Computational Intelligence, vol. 1, 2017, pp. 145–156.
[30] Anh Tuan Nguyen, Michael C. Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, Danny Dig, API code recommendation using statistical learning from fine-grained changes, in: FSE, 2016.