The Journal of Systems & Software 193 (2022) 111435

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Missing standard features compared with similar apps? A feature recommendation method based on the knowledge from user interface☆

Yihui Wang [a,b], Shanquan Gao [c], Xingtong Li [a,b], Lei Liu [a,b], Huaxiao Liu [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] Colleague of Information Science and Engineering, Yanshan University, Qinhuangdao, China

ARTICLE INFO

Article history:
Received 10 October 2021
Received in revised form 8 June 2022
Accepted 13 July 2022
Available online 17 July 2022

Keywords:
Android apps
Feature recommendation
User interface

ABSTRACT

To attract and retain users, deciding what features should be added in the next release of apps becomes very crucial. Different from traditional software, there are rich data resources in app markets to perform market-wide analysis. Considering that capturing key features that apps lack compared with its similar products and making up for them can be conducive to enhance the competitiveness, we propose a method to establish the feature relationships from the level of UI pages and recommend missing key features for the pages of apps based on these relationships. Firstly, we utilize the UI testing tool to collect UI pages for apps in the repository, and give the method to gain the feature information in them. Then, we identify the products similar to the analyzed app based on topic modeling technique. Finally, we establish the relationships between features by analyzing UI pages gained for the analyzed app as well as its similar products, and identify suitable features recommended to UI pages of the analyzed app based on these relationships. The experiment based on Google Play shows that our method can recommend features for apps from the level of UI pages effectively.

1. Introduction

With the rapid development of information technology, the smart mobile application (app) becomes more and more prevalent and the number of mobile applications gets explosive growth (Martin et al., 2016; Hassan et al., 2018; Eler et al., 2019; Ouzzani et al., 2016; Sasaguri et al., 2017): up to 2020, two well-known app markets (Google Play and Apple App Store) host over 2 million apps respectively (Jiang et al., 2019). However, the prosperity of app market not only brings huge economic benefits, but also generates increasingly fierce competition (McIlroy et al., 2015; Häring et al., 2021; Shah et al., 2019; Xu et al., 2016; Frey et al., 2017): less than half of products released in the app stores since 2010 have been retained until 2020.

Facing such cruel competition, app developers have to evolve their products rapidly to attract and retain users. In this process, deciding what features should be added in the next release becomes very crucial because the selection of app features is one of the most important factors that people consider when choosing apps (Chen et al., 2018). To identify valuable new features, developers need to pay attention to the features served by similar products. Consider the example in Fig. 1(a), compared with other two popular Chinese shopping apps, the search page of the app *PinDuoDuo* lacks a key feature to search for goods through pictures. As a result, users are unable to search for products effectively when they only have picture information but cannot give accurate text input, and this causes user dissatisfaction as shown in Fig. 1(b). The example reflects that when an app lacks the features provided by many products similar to it (we define this type of features as apps' missing standard features), users are prone to be dissatisfied and think that the capacity of this app is inadequate.

To identify new features for apps, some researches (e.g., Jiang et al. (2019), Liu et al. (2019) and Yu et al. (2016)) use description texts of their similar products as data resource to perform the market analysis. Although these methods can help app developers to obtain valuable feature information from other products to a certain extent, they cannot effectively capture standard features that are missing from apps for two reasons. For one thing, developers tend to show only a fraction of what apps actually do in descriptions (Gao et al., 2021), so lots of key features of apps cannot be gained by analyzing their description texts. For another, the feature information in app descriptions is coarse-grained (Yu et al., 2016), which leads to the inability to establish

☆ Editor: W. Eric Wong.

* Correspondence to: College of Computer Science and Technology, Jilin University, 2699 Qianjin Street, Changchun, Jilin, 130012, China.
E-mail address: liuhuaxiao@jlu.edu.cn (H. Liu).

https://doi.org/10.1016/j.jss.2022.111435
0164-1212/

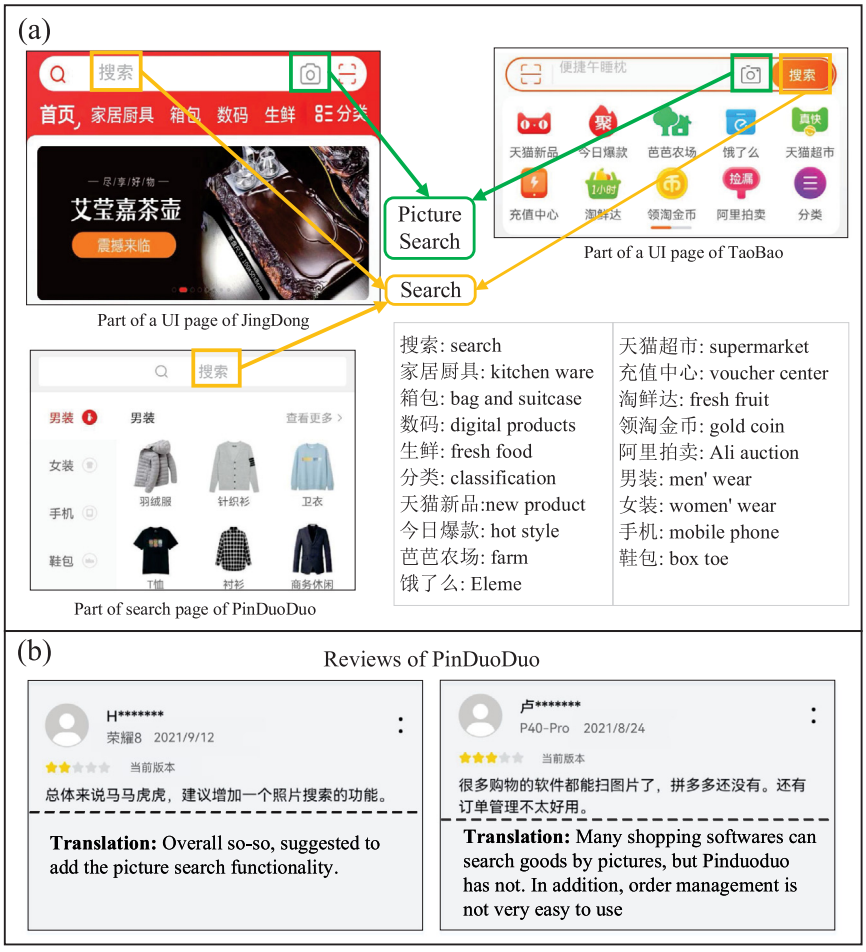

Fig. 1. Part of UI pages of three Chinese shopping apps and user reviews.

effective association relationships between related features by analyzing such texts. For instance, description texts of *JingDong* and *TaoBao* only introduce that users can search for goods they want but do not mention how users can search, so we cannot gain the association relationship between features *picture search* and *search* based on them.

Considering the defects of description texts, Chen et al. proposed UI COMPARER to mine the UI to gain key features missed by apps (Chen et al., 2018). UI COMPARER identifies UI pages that are similar in features to the pages of apps analyzed, and then compares them to explore products' missing features. Although its result is encouraging, there are two key threats in the method:

First, missing standard features of analyzed apps are indeed highly correlated with some of their existing features, but in other products, UI pages where these standard features are located may be not similar to any page of analyzed apps because the UI design style of products is diverse (Chen et al., 2020a). Refer to the above example, whether *JingDong* or *TaoBao*, the features provided by UI page containing *picture search* are obviously different from the ones of *PinDuoDuo*'s search page.

Second, UI COMPARER uses visible texts in the UI as data resources to gain app features, but previous work (Chen et al., 2020b; Xi et al., 2019) has demonstrated that a large number of UI components express the feature information through icons rather than texts. Incomplete feature mining makes a lot of valuable information omitted. Refer to the above example again, both *JingDong* and *TaoBao* display the feature *picture search* through icons, so even if UI COMPARER can gain the corresponding UI pages for *PinDuoDuo*'s search page, it still does not have the ability to capture this feature.

In this paper, we introduce a novel feature recommendation method for apps. Considering that app developers usually realize closely related features in the same UI pages, we analyze UI pages of target app (the analyzed app) as well as its similar products to measure the relevancy between features, thereby establishing their relationships. Based on the gained feature relationships, we identify the features from similar products that are relevant to existing features (or part of existing features) in the UI pages of target app. These features are thought to be the ones UI pages of target app might be missing and are recommended to software developers. To establish reliable feature relationships, we leverage a UI testing tool (Su et al., 2017), which average line coverage can exceed 60.00%, to collect UI pages for app products, and then comprehensively mine the features in these pages by analyzing XML files corresponding to UI pages and training a deep learning model. The line coverage of UI testing tool is gained by using EMMA (2006), which is an open-source toolkit for measuring Java code coverage. The line coverage of a program is the number of executed lines divided by the total number of lines.

As far as we know, this paper is the second study that takes the UI of apps as the data resource of feature recommendation besides UI COMPARER. However, benefiting from more comprehensive feature mining and more flexible recommendation strategy, our method is able to recommend a greater number of reliable features than UI COMPARER and thus is more capable of capturing key features missed by apps. For instance, our method can

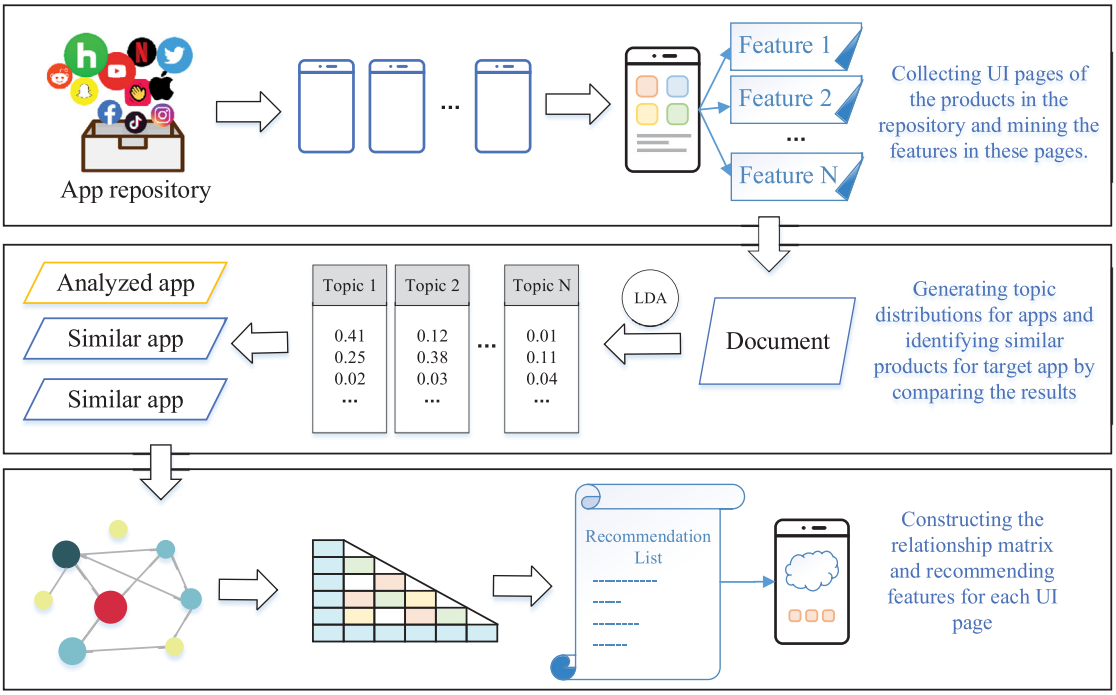

Fig. 2. Overview of our approach.

gain the feature *picture search* for *JingDong* and *TaoBao* through the trained model, and can establish a very positive relationship between *search* and *picture search* by analyzing UI pages to measure the relevancy between features. Based on this positive relationship, the method has the ability to recommend the feature *picture search* for *PinDuoDuo*'s search page. In summary, the contributions of this paper are as follows:

- We establish the rules used for analyzing XML files corresponding to UI pages and train a deep learning model to mine the features provided by components in the UI, including both UI components that express the feature information through texts and the ones that express such information through icons, thereby gaining the comprehensive feature information for UI pages of apps.
- We use the features gained from the UI to characterize apps well, so as to complete the task of identifying similar products for apps effectively.
- Conduct experiments to validate that each important step of our recommendation method is effective and the features recommended by us to UI pages of apps are reasonable (the average Hit Ratio value can reach 67.00%).

This paper is organized as follows. Section 2 presents the overview of our feature recommendation method. Section 3 shows the process of collecting UI pages of app products and mining the feature information in them. Section 4 gives the method for identifying the similar products of target app. Section 5 details how we establish the feature relationships and recommend suitable features for target app from the level of UI pages. We describe our experiment design and results in Sections 6 and 7 respectively. Threats to validity are discussed in Section 8. Section 9 discusses related work, and Section 10 shows the conclusion and future work.

2. Overview of our approach

To help app developers understand the standard features missing from their products, we propose a feature recommendation method, which consists of three main steps as shown in Fig. 2:

Firstly, we use a UI testing tool (Su et al., 2017) to collect UI pages for the products in the app repository, and mine the feature information in these pages in two ways: on one hand, we give the rules to mine the features provided by UI components from XML files corresponding to UI pages; on the other hand, for the components that cannot gain such information by analyzing XML files, we adopt the CNN (Sun et al., 2020; Strauch and Grundy, 2021) and transformer encoder decoder (Takushima et al., 2019; Yang et al., 2020) to train a model to generate features for them.

Then, to identify similar products for target app, we establish feature documents for apps in the repository based on features gained from the UI, and leverage Latent Dirichlet Allocation (LDA) (Blei et al., 2003) to process the gained documents for generating topic distributions of apps, thereby identifying similar products for target app by comparing these distributions.

Finally, we establish the feature relationships by analyzing UI pages of target app as well as its similar products, and construct a matrix to store these relationships. By utilizing this matrix, we evaluate the recommendation value of features from similar products for each UI page of target app, so as to use the features that are of high value and meet the criteria in quantity to establish a recommendation list. Considering the number of features is to ensure that the features in the recommendation list are standard.

3. Collecting UI pages and mining the feature information for apps in the repository

To support our recommendation work, we need to collect the UI pages for apps in the app repository (e.g., a category of Google Play) and mine the features in these pages. The features refer to the capabilities provided by app products (Johann et al., 2017),

such as *send message*, *share*, *back to top*, and *back*. Note that, some capabilities are described by non-verb words, for example, *photo album* and *playlist*. Below, we introduce in detail how we complete these two tasks.

To collect the UI pages of apps, we leverage a UI testing tool (Su et al., 2017) to automatically execute apps by simulating the interaction between users and products, so as to explore their UI and collect the UI screenshots. The tool utilizes UI Automator (Lämsä, 2017; Patil et al., 2016; Zelenchuk, 2019) to export an XML file for the current UI, and identifies executable UI components in the UI as well as their operation types (e.g., *click* and *scroll*) based on the gained file. As illustrated in Fig. 3, UI Automator can export an XML file for UI-1, and the XML file stores the attributes of UI components in the UI-1 as well as the hierarchical structure of these UI components. It can be seen from this XML file that the clickable attribute of the UI component in the red box is true, so this component is executable and its operation type is *click*. Based on the operation types of executable UI components, the tool simulates the corresponding human actions to emit the UI events and enter other UIs. Taking a UI as the starting point, the tool can explore different parts of the UI of apps in this way and retain the screenshots for the UI that has been reached. Refer to Fig. 3 again, the tool jumps from the UI-1 to UI-2 and UI-3 in turn by simulating the actions *click* and *scroll*, and can collect three UI screenshots in this process.

To ensure the coverage of explored UI, this tool determines which UI components to execute by observing three rules: (a) when a component is more frequently executed than others, its priority should be lowered so as to increase the probability of interacting with executable UI components that have not been operated; (b) if a component exhibits more subsequent UIs after its execution, it should be prioritized in future so that more new components can be visited; (c) the UI components about *back* or *scroll* should be executed at right time, because executing these components too early can cause the tool to discard the current UI and thus prevents the execution of other executable components that exist in the current UI. Although these rules are designed to improve the coverage of UI exploration, the tool still omits many UI pages, which may result in some missing features of target app not being captured by our method.

In general, a UI screenshot is a UI page. However, if the transfer action between UI screenshots is *scroll*, we treat the collection of them as a UI page. Note that, there may be duplicate parts between different UI screenshots used to form a UI page, but this does not affect our work because we only consider whether UI pages serve the features when mining UI pages to establish the relationships for features.

As mentioned above, the UI testing tool can export an XML file for each gained UI screenshot by utilizing UI Automator, and it is by analyzing these XML files that we mine the feature information in the UI screenshots to further gain features in the UI pages. To establish the rules used for analyzing XML files, we selected about 200 apps from 4 popular categories (i.e., Music, Navigation, Social, and Education), and actually experienced these products to collect the components, which provide the features to users, as observation data. The reason we selected these four categories is that we believe that these categories provide enough samples to cover kinds of practice situations and can be used to summarize our rules. For mitigating subjectivity concerns, three doctors with app development experience participated in this process, and a UI component was used as observation data only if it was considered to provide the feature by all participants. We found that almost all of UI components collected achieve the functionalities by being clicked: of the 500 UI components, 97.00% of them rely on users' *click* action to emit the corresponding UI events. Thus, we mine the feature information in the UI screenshots by analyzing clickable UI components. Correspondingly, we only consider the UI components which clickable attribute is true when analyzing the XML files. There are four rules used for gaining the features of clickable UI components, which are summarized from observation data:

- (***Rule*$_1$**) **If the text label of a UI component is not null, we use the content of its text label as the feature served by this component.** Some UI components show their feature information in the UI of app products directly by assigning the text labels with effective texts. Refer to Fig. 4(a), the text label of UI component in the red box is assigned with *Send*, which is displayed in the UI to tell users what functionality will be achieved after clicking the component. Meanwhile, in the process of collecting UI components for our observation data, we found that clickable components with text greater than 8 words generally do not provide features for users, but are related to other aspects such as advertisements and mails. Thus, we only consider the components which gained texts are shorter than (or equal to) 8 words when mining the features in the UI screenshots by analyzing clickable UI components' text labels.
- (***Rule*$_2$**) **If the type of a UI component is related to the framework, we search out its internal component which text label is not null, and use the content of this text label as the feature served by the UI component.** Many app developers like to use a UI component as the framework to organize other components and treat the whole framework as a unit to serve a feature for users. At this time, the icon information and text information of the framework are displayed by its internal components. Refer to Fig. 4(b), the UI component in the red box is clickable and its component type is related to the framework (*android.support.v7.widget.LinearLayoutCompat*). The text label of its second internal component is assigned with *Collage*, which shows the feature served by the whole framework.
- (***Rule*$_3$**) **If a UI component only gives the icon information and the text label of its sibling component is assigned with the effective text, we use this text as the feature of UI component.** Many components express the feature information through their surrounding texts. Moreover, such texts can be gained by analyzing the text labels of their sibling components. Refer to Fig. 4(c), the UI component in red box and the one in yellow box are sibling components, the component in the red box is clickable and we can easily understand that the feature it provides is *templates* through the text label of its sibling component.
- (***Rule*$_4$**) **If a UI component only gives the icon information and its feature cannot be gained by analyzing sibling components, we use the content of content-description as its feature.** Some UI components do not express the feature information by displaying the texts in the UI but through their icons, and we cannot gain the features for them by utilizing the above rules. Developers often give the feature information in the content-descriptions of UI components, and users with vision impairment can adopt the screen reader embedded in the mobile operating systems to read content-descriptions to understand the components' functionalities. Thus, we use content-descriptions as a data resource to gain UI components' feature information. Refer to Fig. 4(d), we can understand that the feature served by the UI component in the red box is *flip vertical* by observing its content-description.

Fig. 5 summarizes the way in which clickable UI components in our observation data express the feature information. It can be seen that 64.12% of components express their features through

Fig. 3. The example used to illustrate the process of collecting UI pages.

Fig. 4. Some UI components and their corresponding codes. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

texts in the UI, while the remaining components use icons to display this information. Unfortunately, of UI components that make use of icons to present features, 28.16% have null content-descriptions. For such components, we cannot gain their features by utilizing the above rules. Inspired by Chen et al. (2020b), we adopt the CNN and transformer encoder decoder to train a model to generate the feature texts for UI components that cannot get such information from XML files. Firstly, we establish icon-feature mappings as the training data based on the UI components which features can be gained through XML files: for UI components which type is related to the layout, we crop the icons of their internal components based on the bounds attributes and treat the feature information gained for the whole frameworks as the features of icons, so as to establish icon-feature mappings; for UI components which features can be gained from sibling nodes or content-descriptions, we crop the icons of components and combine them with the obtained features to establish icon-feature mappings. Then, we use ResNet-101 architecture (He et al., 2016) pretrained on MS COCO dataset (Lin et al., 2014) as our CNN module and train the Transformer module based on training data. By utilizing the model, we generate the features for UI components that cannot gain the feature information by analyzing XML files. As shown in Fig. 6, the model extracts the icon features to form embedding by using CNN, and further generates feature texts for icons based on the transformer module. CNN-based model can automatically learn latent features from a large image database, which has outperformed hand-crafted features in many computer vision tasks. Moreover, there are two reasons for selecting Transformer model as the encoder and decoder. First, it overcomes the challenge of long-term dependencies, since it concentrates on all relationships between any two input vectors. Second, it supports parallel learning because it is not a sequential

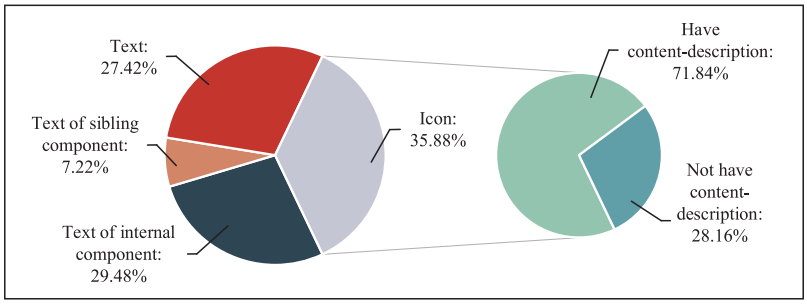

Fig. 5. The way UI components express feature information.

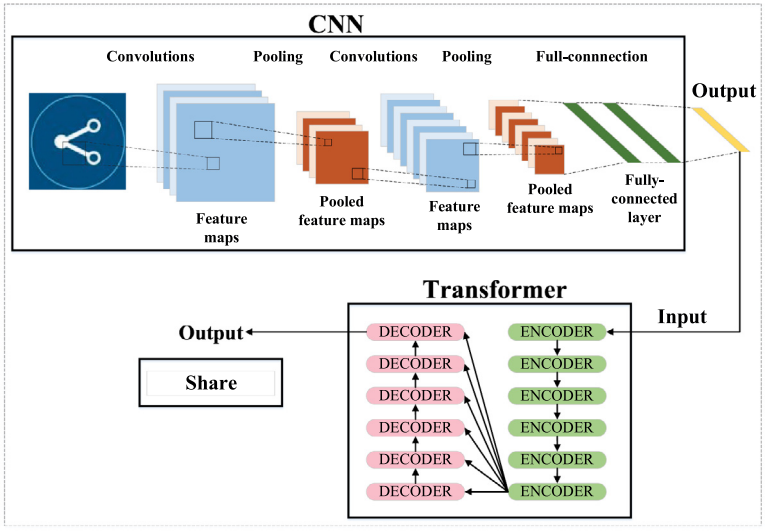

Fig. 6. Overview of the model used to generate features of UI components.

learning and all latent vectors could be computed at the same time, resulting in the shorter training time than RNN model.

Relying on the above method, we collect UI pages and mine the feature information in them for apps in the repository. Note that, some UI interaction-related features (i.e., the features about *back*, *next page*, *close*, *more option*, *home*, and *menu*) are present in a large number of UI pages and are combined with a variety of features. These features can cause a lot of interference to our subsequent recommendation work, so we remove them. Fig. 7 gives a UI page and further shows the features gained for it using our method.

We adopt dynamic analysis technology to mine the features for app products rather than static analysis technology in this paper because dynamic analysis technology can directly gain the feature information with UI pages as organizational units.

4. Identifying similar apps for target app

There are maybe a few hundred thousand products in the app repository, and the development purpose of many products in them is different from target app, so we do not use all apps in the repository but select the products with similar functionalities to target app from them as the reference apps of feature recommendation. Inspired by Jiang et al. (2019) and Hindle et al. (2012), we generate the topic distributions about features for apps and compare the gained results to identify the products similar to target app.

App descriptions are given by developers to introduce the features of products to users, and they are used to build the documents about features for apps in many studies (e.g., Jiang et al. (2019), Liu et al. (2018) and Gorla et al. (2014)). However, app descriptions usually only show the important features of

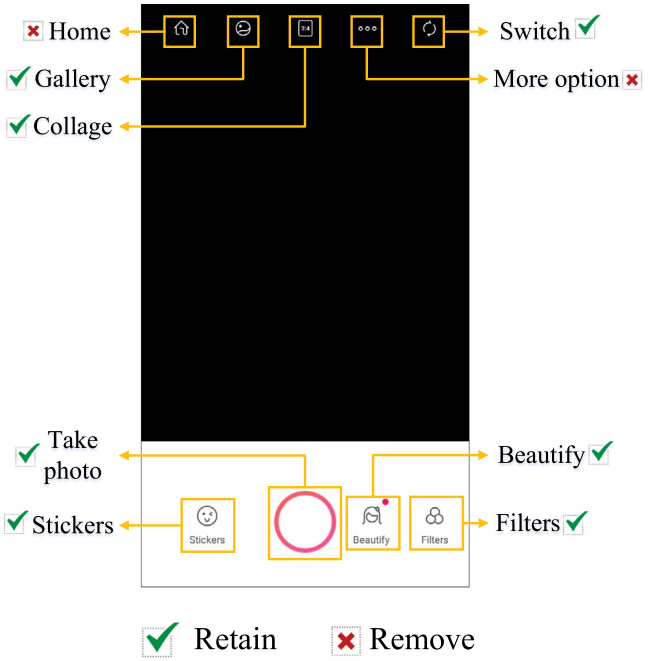

Fig. 7. A UI page and the features gained for it.

products (some products even only give a few features in their description texts). Thus, it is difficult to obtain satisfactory results using description texts as the data resource to generate feature documents of apps.

Considering that features gained from the UI of app products are far richer than the ones in their description texts, we establish feature documents for apps in the repository based on this data resource, thereby identifying similar products for target app. Firstly, we establish a feature document for each app in the repository based on the features gained from UI pages. Then, we leverage LDA to process the gained documents and generate a topic distribution about features for each product. Finally, we evaluate the similarity between target app and other products, and top K products similar to target app are used as the reference apps for feature recommendation (K can be adjusted according to developers' needs). The similarity between two apps is gained by calculating the cosine similarity of their topic distributions. Suppose that there is an app a, the similarity between it and target app t is calculated with the formula as below:

$$Similarity(a,t) = \frac{\sum_{\forall k} P_{a\in k} \times P_{t\in k}}{\sqrt{\sum_{\forall k} P_{a\in k} \times P_{a\in k}} \times \sqrt{\sum_{\forall k} P_{t\in k} \times P_{t\in k}}},$$

where $P_{a\in k}$ and $P_{t\in k}$ represent the probabilities of apps a and t belonging to a specific topic k. For instance, suppose the topic distribution of an app is (a_1, a_2, a_3) and the one of target app is (t_1, t_2, t_3), their similarity value can be calculated according to the above formula, namely $\frac{a_1\times t_1+a_2\times t_2+a_3\times t_3}{\sqrt{a_1^2+a_2^2+a_3^2}\times\sqrt{t_1^2+t_2^2+t_3^2}}$.

5. Constructing the relationship matrix and recommending features based on it

After gaining similar products for target app, we take UI pages of target app and its similar products as data basis to construct a relationship matrix of features. Based on this matrix, we evaluate the recommendation value of features from similar products for UI pages of target app. The features that are of high value and meet the criteria in quantity are treated as missing standard features of UI pages and recommended to developers. This section details the feature recommendation process.

Developers of different app products may give different descriptions for the same functionality, so we cluster the features in the gained UI pages with DBSCAN (Schubert et al., 2017; Sheridan et al., 2020). The clustering process consists of the following three main steps:

Firstly, we use all features in the gained UI pages to establish a feature set, and calculate the similarity values between features in this set by utilizing the method proposed in our previous work (Gao et al., 2021). For two features F_i and F_j, the similarity calculation formula between them is as follows:

$$FeatureSimilarity(F_i, F_j) = w_n S_{noun} + w_v S_{verb},$$

where:

$$S_{noun} = \max_{\substack{word_i \in Set^{F_i\_keyword},\ POS = noun \\ word_j \in Set^{F_j\_keyword},\ POS = noun}} \times\ Similarity(word_i, word_j);$$

$$S_{verb} = \max_{\substack{word_i \in Set^{F_i\_keyword},\ POS = verb \\ word_j \in Set^{F_j\_keyword},\ POS = verb}} \times\ Similarity(word_i, word_j);$$

$Set^{F\_keyword}$ is the set of words in the feature F; $Similarity(word_i, word_j)$ is the cosine similarity between vectors of $word_i$ and $word_j$, the word vectors are gained by using word2vector (Goldberg and Levy, 2014); w_n and w_v are weights of S_{noun} and S_{verb} respectively, and the values of weights can be adjusted according to the characteristics of different categories. Note that, if both features have only nouns (verbs), then $w_n = 1$ and $w_v = 0$ ($w_n = 0$ and $w_v = 1$). The POS of words in features is tagged by utilizing NLTK (Bird, 2004). In addition, consecutive nouns (verb phrase) are treated as one noun (verb), and the vector mean of all words is used as the overall vector.

Then, we use the reciprocal of similarity values between features as their distances, and the reason why we consider the reciprocal rather than directly using similarity values is that the greater the similarity value between features, the more similar their meanings.

Finally, we utilize DBSCAN to cluster the features of the same aspect together. In this process, we set two important parameters *epsilon* and *minPoints* to 1.25 and 1 respectively for two reasons: one is that the similarity value greater than (or equal to) 0.8 gained by our method represents that the meanings of features are consistent, and its reciprocal is exactly 1.25; and the other is that our clustering work is to cluster different features describing the same aspect together but not filter out rare features, so we assign *minPoints* with 1. Since the features in a cluster are about one aspect, we use feature clusters as units recommended to target app.

Features with high correlation are usually realized in the same UI pages (Chen et al., 2018), so we mine UI pages to analyze the relevancy between feature clusters to establish their relationships. We gain the relevancy value of two feature clusters based on the lift (Li et al., 2019; Hong et al., 2020) between them, which can reflect the influence between samples. The value range of the lift is $[0, +\infty)$ from the most negative to the most positive. Suppose that there are two feature clusters C_i and C_j, the specific formula used to calculate their relevancy value is as follows:

$$Relevancy(C_i, C_j) = \frac{Lift(C_i \to C_j) + Lift(C_j \to C_i)}{2},$$

where:

$$Lift(X \to Y) = \frac{P(Y/X)}{P(Y)} = \frac{P(X \cap Y)}{P(X) \times P(Y)};$$

$P(X)$ represents the ratio of UI pages serving the features in the cluster X to all UI pages; $P(X \cap Y)$ is the ratio of the pages serving both the features in the cluster X and the ones in the cluster Y to all UI pages. Note that, in fact the value of $Lift(C_i \to C_j)$ is equal to $Lift(C_j \to C_i)$, but we still consider average values when calculating the relevancy values for feature clusters because we want to express more clearly that the relationships between these clusters are non-directional.

We calculate the relevancy values between feature clusters gained after the clustering work by using our method, and store the results in the corresponding locations to construct the relationship matrix: for two clusters C_i and C_j, their relevancy value is stored in the location (C_i, C_j) or (C_j, C_i). Taking the relationship matrix in Fig. 8 as an example, there are totally 7 feature clusters after clustering the features of one aspect together. The lift value between cluster 4 and cluster 5 is 1.9, so the relevancy value between them is 1.9 ($\frac{(1.9)+(1.9)}{2}$), and this value is stored in the location (C_5, C_4) of relationship matrix. Of course, other elements in the matrix can be gained in the same way.

To capture the missing features for a UI page of target app, we utilize the relationship matrix to evaluate the recommendation value of feature clusters that are not served by target app (feature clusters not containing the features served by target app).

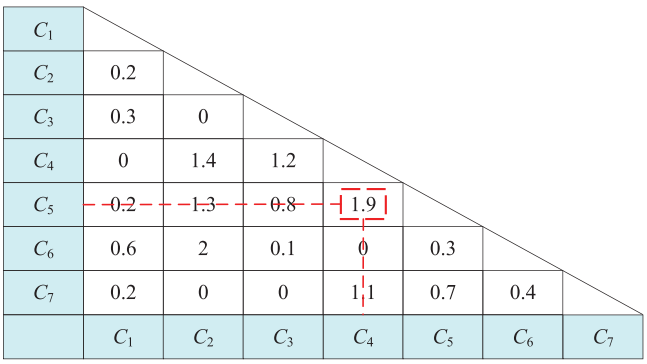

Fig. 8. Example of the relationship matrix.

For a feature cluster, we gain the relevancy values between it and clusters containing the features served by the UI page from the relationship matrix, and calculate its recommendation value based on these values. The specific formula used to calculate the recommendation value of a feature cluster C for a UI page U is as follows:

$$RecommendationValue(C, U) = \sum_{r \in set^{RelevancyValue(C,U)}} r,$$

where: $set^{RelevancyValue(C,U)}$ is the set of relevancy values greater than 1 obtained from relationship matrix. The reason why we only consider relevancy values greater than 1 is that only they reflect positive relationships and we want to evaluate the recommendation value of feature clusters based on positive relationships.

We still use our example to more clearly illustrate how to utilize the relationship matrix to calculate the recommendation value for feature clusters. Suppose that three features served by a UI page are contained in clusters 2, 4, 7 respectively, and we need to evaluate the recommendation value of cluster 5 for this UI page (cluster 5 does not contain features served by the analyzed app). By observing the relationship matrix in Fig. 8, we can see that the relevancy values between cluster 5 and clusters 2, 4, 7 are 1.3 (C_5, C_2), 1.9 (C_5, C_4) and 0.7 (C_7, C_5) respectively. Based on this result, the recommendation value of cluster 5 can be gained, that is 3.2 $(1.3 + 1.9)$.

For each UI page of target app, we evaluate the recommendation value for feature clusters not served by target app and rank the clusters according to the gained results. The center nodes of top-K clusters are used to establish an ordered feature list recommended to the product. Note that, not all the center nodes of feature clusters can be considered into the recommendation list. We set a threshold (the default value is 0.05, and developers can adjust it according to their needs), and only when the ratio between the number of samples in a feature cluster and the number of reference apps is greater than the threshold, its center node can be considered. This process can better ensure that the features in our recommendation lists are standard.

As discussed in Jiang et al. (2019), recommending a small list is a common practice in software engineering so that developers can check them from the top to the bottom sequentially without taking much time. Thus, we set K of recommendation list to 5 by default. Naturally, this value can be adjusted according to developers' needs. Fig. 9 shows a UI page of the app *Likee* and it also gives a recommendation list gained by utilizing our method. We can see that the features in our recommendation list have a correlation with existing features in the UI page and adding them to the page can greatly enrich the features of *Likee*. Many studies (e.g., Khalid et al., 2015; Weichbroth, 2020; Soui et al., 2020) focus on the quality of app UI design, but same as Jiang et al. (2019) and Chen et al. (2018), our method only provides potential new features for app developers without considering the impact of these features on the UI design of products. Developers need to conduct relevant analysis manually.

Table 1
The information of the experimental data.

Category	Number of apps
Music	3029
Navigation	2537
Social	3406
Education	2853

6. Experimental design

6.1. Research questions

We aim to answer the following three questions:

- **Question 1:** Whether our method can gain the feature information in UI pages of apps effectively?
- **Question 2:** Whether our method can identify similar products for apps reasonably?
- **Question 3:** Whether our method can recommend suitable features for mobile apps from the level of UI pages?

6.2. Dataset and participants

We chose 4 categories of apps on Google Play as the objects of our experiments, including: Music, Navigation, Social, and Education. There are two main reasons for choosing these categories: on the one hand, 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; on the other hand, there are a large number of products in these categories, which provide enough samples to cover kinds of practice situations. The specific number of apps in each category is shown in Table 1.

For apps in the experiment, we firstly used our crawler to gain their APK files from *APK Download*.[1] The reason why we crawl the APK files from a third-party website rather than Google Play is that it is difficult for us to get these files from Google Play directly. Then, we collected description texts of apps from Google Play by using our crawler. Finally, we collected the UI pages of apps by using the UI testing tool to process their APK files, and mined the feature information in them according to our method. The important code in our experiment was provided online.[2]

The participants in our experiment 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 evaluating the experimental results of Question 1 and Question 3.

6.3. Experimental design for Question 1

For each rule used to analyze XML files, we first randomly selected 100 clickable UI components, which can gain the feature text through the rule, from each category. Then, three experimental participants evaluated whether the texts obtained by the rule were related to feature information. For a text, only when

[1] https://apps.evozi.com/apk-downloader/.
[2] https://figshare.com/s/b4786e18dee406cf5ec0.

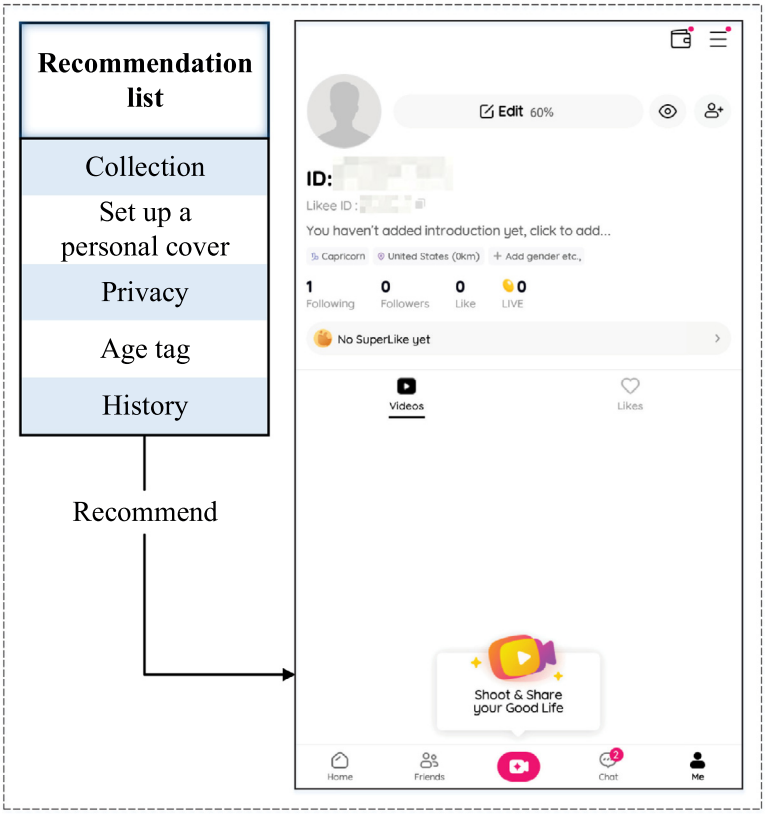

Fig. 9. Example of information recommendation.

all participants agreed that it was related to feature information could we consider the text to be correct. For $Rule_1$ to $Rule_3$, the purpose of constructing this experiment is to analyze whether these rules would extract the non-feature information that exists in the UI. For $Rule_4$, carrying the experiment aims at evaluating the reliability of feature texts gained by using the rule.

Some clickable UI components express the feature information through icons and their content-description attributes are null. These components cannot gain feature texts through our rules. We train a model based on CNN and transformer encoder decoder, and use it to generate feature texts for UI components that cannot gain such texts through our rules. To evaluate the effectiveness of our model, we firstly established the mappings between icons and features based on the experimental dataset. Secondly, we randomly selected 14,000 icon-feature mappings from the ones gained in step 1 to establish a training set. Based on this training set, we trained a model used for generating feature texts. Thirdly, we randomly selected 100 icon-feature mappings, which were considered correct by all experimental participants, for each category and used them to establish a test set. To ensure the reliability of the experiment, the mappings in test sets were different from the ones in the training set. Finally, we used our model to generate feature texts for the icons in test sets. By comparing these texts and the ground truth, we evaluated whether they were correct. If feature texts generated were exactly the same as the ground truth, we directly thought that they were correct. If the texts were not the same as the ground truth, three participants compared them, and only when all participants agreed that generated texts can express consistent information could we consider the results given by our model to be correct.

6.4. Experimental design for Question 2

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 (Al-Subaihin et al., 2019). Thus, we designed an experiment to evaluate the method indirectly.

As discussed in Al-Subaihin et al. (2019) and Vakulenko et al. (2014), app categories are manually created and populated by domain experts, and this makes app category assignment reputable and high-quality. Thus, we can optimistically assume that an app and its similar products should come from the same category. We evaluated our method based on this assumption and the evaluation process consists of three steps:

Firstly, we randomly selected 2000 apps from our dataset for each category as data basis for answering Question 2. 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.

Secondly, we used the features gained from UI pages 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 leveraged the module provided by Rehurek and Sojka (2011) to implement LDA and adopted perplexity (Huang et al., 2017) 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 an app and its similar products should come from the same category, so we evaluated the accuracy (accuracy@N) of identifying similar products for each app by calculating the percentage of apps in the same category as this app in top N similar products. Based on the accuracy values of

all apps in a category, we can further gain the accuracy for a category.

6.5. Experimental design for Question 3

As discussed in the researches (Jiang et al., 2019; Hariri et al., 2013), evaluating the feature recommendation method is a difficult task because this process is highly subjective. Same as these researches, we also adopted the *elimination-recovery* evaluation method to indirectly analyze the performance of proposed method. The evaluation process consists of three steps:

Firstly, we established test UI pages and their golden features. For each category, we randomly selected 400 UI pages from apps with rating higher than 4 in the experimental dataset. For each UI page, we eliminated a feature, which was considered appropriate for the page by three participants. Note that, if participants could not identify such a feature for a selected UI page, we did not use this page as the data basis for answering Question 3 and replaced it with another UI page. The changed UI pages were used as test UI pages in this experiment. Eliminated features were treated as golden features (missing standard features) of the corresponding test UI pages. The reason for selecting UI pages from high-rated apps is that these products usually organize the features reasonably, which can better ensure that golden features should be recommended to the corresponding test UI pages.

Secondly, we constructed the feature recommendation list for test UI pages. For each test UI page, we utilized our method to identify the products similar to its corresponding app from the same category (we set K to 120 in our experiment), and mined the UI pages of app products to establish the relationships between feature clusters, thereby constructing a relationship matrix. Note that, to guarantee the fairness of our experiment, we replaced the original page with test UI page in the process of constructing the relationship matrix. Based on the gained relationship matrix, we constructed a feature list recommended for test UI page (Same as Jiang et al. (2019),15 features were reserved in a recommendation list).

Thirdly, we calculated the Hit Ratio value (Hariri et al., 2013) **for each category.** For each test UI page, three participants checked the feature list recommended by our method to explore whether its golden feature exists in the list: only when all participants agree that there is at least a feature can reflect golden feature does the golden feature exists in the list. Based on the results of all test UI pages, we calculated the Hit Ratio value for each category as follows:

$$Hit\ Ratio = (Hit\_number)/400 \times 100\%,$$

where: *Hit_number* is the number of test UI pages which golden feature exists in the recommendation list.

We treated SAFER (Jiang et al., 2019) and UI COMPARER (Chen et al., 2018) as experimental baselines for two reasons: on one hand, SAFER is an excellent study of feature recommendation based on app descriptions, so comparing our method with it can explore the difference between two data resources (the UI and description texts); on the other hand, UI COMPARER is the first study to introduce the UI of apps to the work of feature recommendation, and we compare our method with it to analyze whether we can achieve better results.

7. Experimental results and analysis

7.1. Experimental result and analysis for Question 1

Fig. 10 shows the evaluation result of our rules, it can be seen that almost all texts gained through our rules are related to feature information: the accuracy values of $Rule_1$, $Rule_2$, $Rule_3$, and $Rule_4$ are 91.00%, 89.75%, 90.50%, and 97.00% respectively. Texts unrelated to feature information that are gained by using $Rule_1$ to $Rule_3$ are mainly about three aspects, including: phone numbers, meaningless words (e.g., OK and Yes), time and date.

When analyzing XML files to mine the features for UI pages, we ignore clickable components which text label is longer than 8 words. To explore the impact of ignoring these UI components, we randomly selected 100 clickable components with text label longer than 8 words from each category and evaluated whether their text labels were about feature information. To ensure the reliability of the evaluation result, three participants participated in this process. Fig. 10 gives the evaluation result, it can been seen that almost all text labels gained are unrelated to feature information. By observing the experimental data, we found that UI components with text label longer than 8 words were mainly about four aspects, including: user reviews, advertisements, mails, and the interaction between developers and users. Because of the very low accuracy, we believe it is reasonable to ignore clickable UI components with long text, even if this might make us miss some features in the UI pages.

Fig. 10 also shows the experimental result about our model, and we can see that the model has the ability to generate feature texts for UI components: the average accuracy in four categories is 64.75%. Although the model achieves a good result, it still generates incorrect feature texts for some icons. We manually studied the differences between generated texts and the ground truth, and summarized two main reasons for incorrect results. On one hand, similar icons may express different features in different scenarios, leading that our model generates wrong texts for some icons in test sets. Refer to Fig. 11, our model gives the feature *search* for the icon on the left, which corresponding feature is actually *image zoom graph*, but as shown in this figure that many apps do use the icons similar to it to represent the feature *search*. On the other hand, the features of some icon-feature mappings in test sets do not exist in the training set, such as *private radar*, so our model cannot generate accurate feature texts for icons corresponding to them.

The experimental result indicates that the redundant information obtained by $Rule_1$, $Rule_2$ and $Rule_3$ is very little. Meanwhile, the result also shows that $Rule_4$ and the model can help us effectively obtain the features texts for UI components expressing the feature information through icons, especially considering that the proportion of UI components that need to rely on the model to obtain features is not high. Thus, we believe that the task of mining the feature information for UI pages can be completed by utilizing our rules and model.

7.2. Experimental result and analysis for Question 2

Fig. 12 shows the experimental result of Question 2, we can see that our method can identify similar products for apps effectively: the average accuracy of our method can be up to 75.41% at top 10, 76.65% at top 50, and 78.13% at top 100. Of top N similar products that we gained for apps by using our method, many are not in the same category as them. According to our analysis, this is mainly because even products in different categories provide some of the same features, which causes LDA to generate similar topic distributions for these apps. In addition, Fig. 12 also indicates that similar product identification method is less effective in the Education category. By observing the experimental data, we summarized a main reason for such result: the collection of UI pages of some education apps does not gain the desired results and this makes the topic distributions that LDA generates for them not very reliable.

Many studies (Gorla et al., 2014; Zhang et al., 2018) use description texts as feature documents of apps, and then process them by using LDA to generate topic distributions, thereby

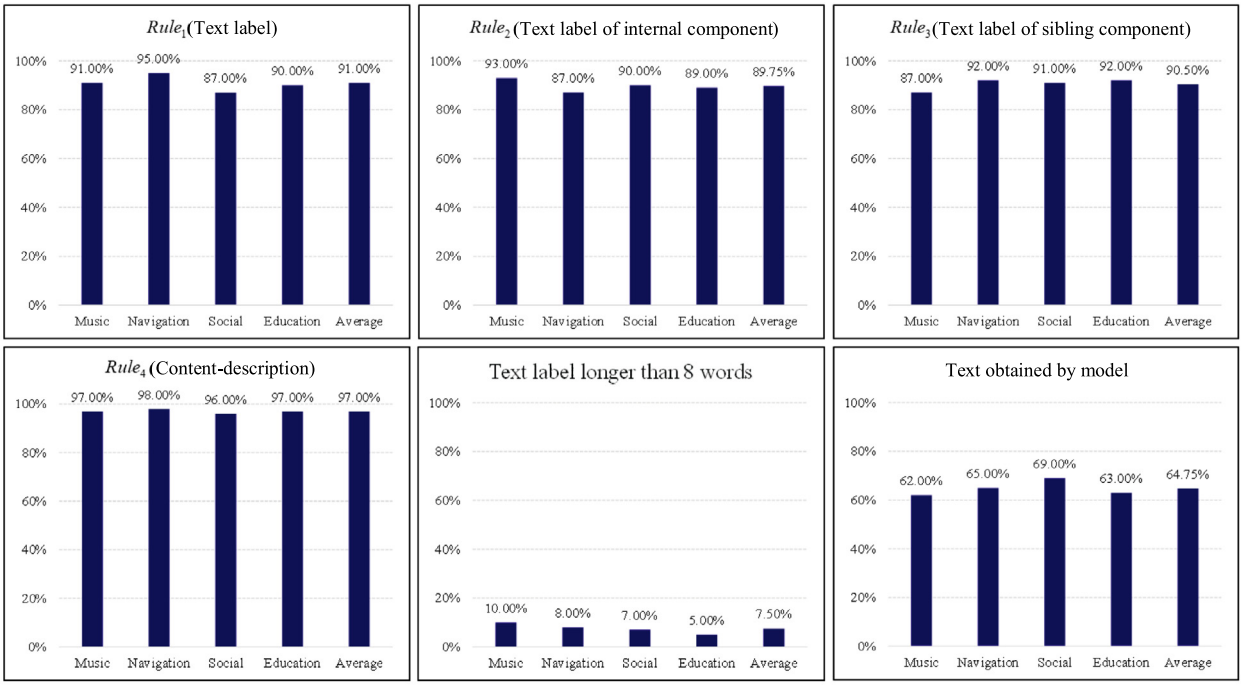

Fig. 10. The accuracy of our rules and model.

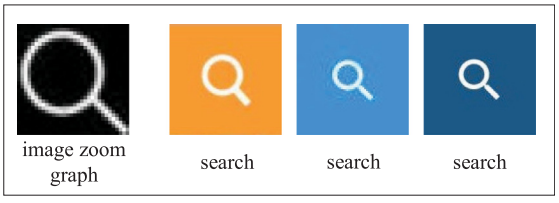

Fig. 11. Some icons and their ground truth.

identifying the similar products for apps. We treated this method as the comparison method to further explore the performance of our method. We applied the comparison method to the data set of this experiment to identify top-N similar products for each app. By using the same evaluation method, we analyzed the performance of this method. Fig. 13 gives the experimental result of comparison method, we can see that all three metrics of comparison method are lower than our method. This result indicates that using the features gained from UI pages as the data basis can indeed establish more reliable feature documents for apps, which can improve the effect of similar product identification.

When mining the recommendation information for an app, our method usually makes relevant analysis based on at least dozens of products, so we pay more attention to accuracy@50 and accuracy@100. Since the method achieves good results on both metrics, we believe that our method is capable of identifying similar products from an app repository for the analyzed app, thereby supporting the work of feature recommendation.

7.3. Experimental result and analysis for Question 3

Fig. 14 shows the evaluation result of our method, and it can be seen that the features recommended by us are reasonable: the average Hit Ratio value in all categories is 67.00%. Many golden features cannot be recommended to the corresponding test UI pages by our method, we analyzed the practice data and summarized two main reasons. On one hand, some calculation results of the similarity between features are not accurate, and this leads to the errors in the clustering process, which can affect the effect of feature recommendation. On the other hand, some UI pages useful for recovering golden features are not captured when utilizing the UI testing tool to collect the pages of app products. In addition, Fig. 14 also indicates that our method achieves better evaluation results in the categories of Music and Navigation. By comparing the experimental data in different categories, we found that this is mostly because there is a stronger correlation between the features in the same UI pages of apps in these two categories.

To further analyze the performance of our recommendation method, we treated test UI pages as recommended objects and utilized SAFER as well as UI COMPARER to establish recommendation lists for them, and then evaluated the lists based on the same metric for comparing different methods. To guarantee the fairness of comparison experiment, apps used by comparison methods to gain the recommendation information were the same as the ones used to construct the relationship matrix of our method and 15 features were reserved for each recommendation list. Note that, many recommendation lists provided by UI COMPARER do not have enough features because of the lack of UI pages that meet the condition.

Fig. 15 shows the experimental result of SAFER. We can see that Hit Ratio value of SAFER is worse than our method (19.38% versus 67.00%). SAFER establishes feature relationships by mining app descriptions and recommends new features based on these relationships. However, since the feature information in description texts is coarse-grained and incomplete, mining them cannot establish effective association relationships between related features and SAFER has no ability to recover golden features for test UI pages. The result verifies that mining description texts is indeed unable to capture the missing standard features for UI pages of app products as effectively as our method.

Fig. 15 also indicates that UI COMPARER cannot recover golden features for almost all test UI pages (Hit Ratio value is 3.94%). By analyzing the experimental data, we summarized two main reasons why the result of UI COMPARER was far worse than our

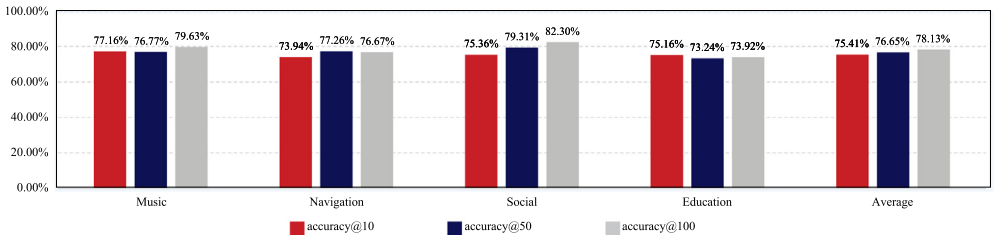

Fig. 12. The experimental result of similar app identification.

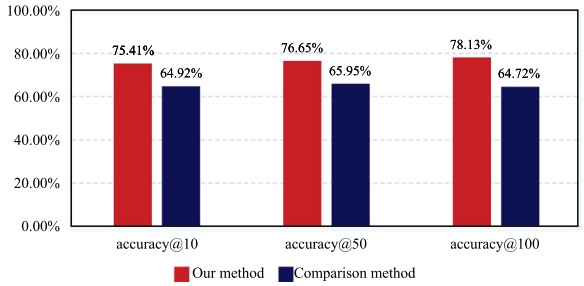

Fig. 13. The comparison result of Question 2.

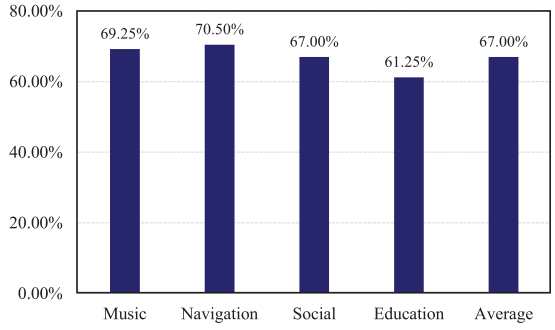

Fig. 14. The experimental result of feature recommendation.

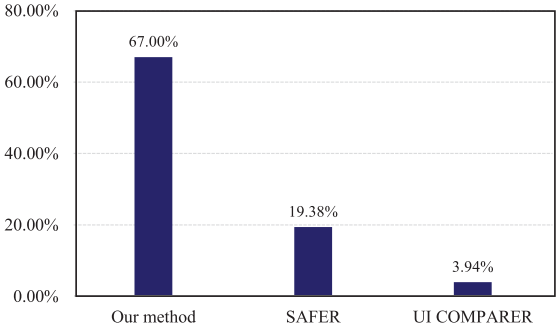

Fig. 15. The comparison result of Question 3.

method. On one hand, many test UI pages mainly express the feature information through icons, such as test UI page shown in Fig. 16(a). UI COMPARER only considers texts in the UI, so it cannot effectively gain the features provided by these test UI pages and thus cannot recover golden features for them. In contrast, our method is able to mine the features of icon-related UI components based on $Rule_4$ and deep learning model. On the other hand, UI COMPARER identifies similar UI pages for the analyzed pages and mines the recommendation information from them, while our method uses extensive UI pages as data basis to establish the relationships between features and recommends the features that have a correlation with existing features (or part of existing features) of analyzed UI pages. Benefiting from more flexible recommendation strategy, the features recommended by our method are far richer than UI COMPARER, which makes our method more capable of recovering golden features for test UI pages. For example, UI COMPARER cannot recover the golden feature for test UI page in Fig. 16(b) because in other products, UI pages with *All songs*-related features (e.g., the UI page in Fig. 16(c)) are not similar to it. However, our method can establish very positive relationships between the feature cluster related to *All songs* and two feature clusters (feature cluster related to *Playlists* as well as feature cluster related to *My favorites*), and can recover the golden feature for test UI page based on these two relationships.

In summary, our method can recommend suitable features added into the next release for apps from the level of UI pages and thereby preventing them from missing key standard features as much as possible.

8. Threats to the validity of experiments

Despite the encouraging results, this work has some potential threats to validity. We analyzed them from the following two aspects:

Internal validity. The construction of the experimental dataset may be a threat. Employing different apps into the dataset, our method may have different performance. It is unknown how our method performs when app products in the dataset are changed. In the future, we will explore the performance of our method when providing different apps in the dataset.

In the experiment answering Question 2, we only considered top 10, top 50, and top 100 because we were concerned that apps do not have many similar products in our dataset. In the future work, we will try to construct a larger-scale experimental dataset, thereby exploring the effectiveness of our method more comprehensively.

In the experiments answering Question 1 and Question 3, the number of test cases we selected for each category was not very large because the results need to be evaluated artificially. However, considering that we evaluated the performance of proposed method in multiple categories, we think the number of test cases is sufficient.

In the experiments answering Question 1 and Question 3, we invited experimental participants to evaluate the results gained from our method or comparison methods. This process is relatively subjective, and participants may give incorrect results. However, the results were given by multiple participants, and they were discussed to converge when discrepancies occurred. Thus, we believe the evaluation results in our experiment are reliable.

Since no dataset of golden features is available, we manually established test UI pages and their golden features in the experiment answering Question 3. The backgrounds and personal opinions of participants may influence the quality of these data.

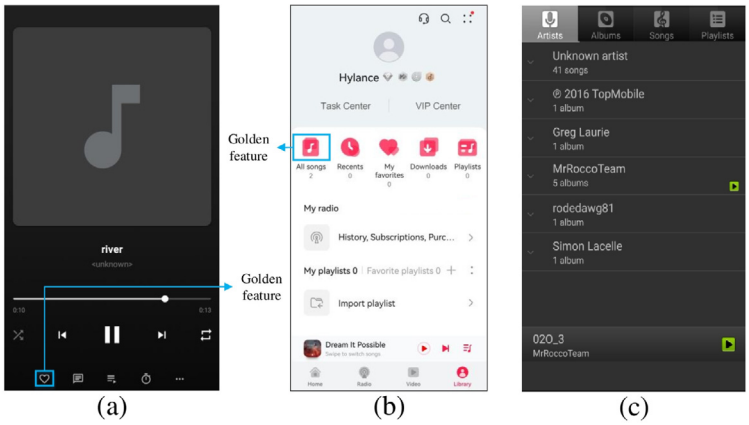

Fig. 16. The example of used to compare our method and UI COMPARER.

Table 2
A report of computation methods for feature similarity.

Type	Evaluation criteria of feature similarity	Techniques for analyzing semantics	Example references
1	Comparing the words in features directly	NULL	Harman et al. (2012)
2	Analyzing the semantics of features	Wordnet	Uddin et al. (2021)
		Word2vector	Yu et al. (2018)
		Deep leaning model	Wang et al. (2022)

To reduce the bias of participants, we had three distinct participants process each UI page together and discussed for each result. In such a way, we think that this threat is reduced as much as possible.

Due to different research directions and lack of professional knowledge, we did not evaluate the coverage of UI testing tool in our data set, resulting in incomplete experimental verification. However, the good experimental result of Question 3 can indirectly indicate that the tool has the ability to achieve good results on apps in our experiment.

External validity. The experiment was constructed based on the data from four different categories, and it is still uncertain how well our method performs over other apps of other app categories. Still, we think that four categories that are significantly different from each other are enough to illustrate the good generalizability of our method. In the future, we plan to evaluate the performance of our method on more app categories, thereby further exploring its generalizability.

9. Related work

Our method establishes the relationships between features by mining and summarizing the feature information in the UI pages of apps, and helps software developers gain suitable features based on the gained relationships. Thus, the related work is discussed from the following two directions.

9.1. Mining the feature information of app products

Feature information contained in various data resources could provide helps for many app development or evolution tasks (Hariri et al., 2013; Harman et al., 2012; Liu et al., 2017), and a number of researches have given methods to mine the features. For example: Timo Johann et al. define a series of POS patterns to extract the features from app reviews and app descriptions (Johann et al., 2017); Claudia Iacob et al. design MARA to retrieve mobile app feature requests from online reviews automatically (Iacob and Harrison, 2013). In these researches, the ones most relevant to us are Chen et al. (2018) and Avdiienko et al. (2017), which also focus on the feature information in the UI. These researches gain the features in the UI of apps by analyzing the attributes of components. Despite promising results, these researches are not able to effectively mine the features of components in the UI that express the feature information through icons, especially (Chen et al., 2018). To gain the feature information in the UI as comprehensively as possible, we mine the features provided by UI components in two ways: on one hand, by analyzing component attributes such as text labels and content-descriptions, we gain the features of UI components from XML files; on the other hand, for UI components that cannot obtain such information from XML files, we adopt the CNN and transformer encoder decoder to train a model to generate their feature texts.

9.2. Measuring the similarity between features

Developers of different app products may give different natural language descriptions for the same features, and many researchers proposed methods to measure the similarity between features as shown in Table 2. For one thing, some methods directly compare the words in features to analyze their similarity. For example: Harman et al. evaluate the similarity between features by counting the number of words shared by them Harman et al. (2012). For another, some researchers calculate the similarity of features by analyzing their semantics. For example: Uddin et al. measure the semantic similarity between features by using WordNet, and thereby gaining the similarity of features (Uddin et al., 2021); Wang et al. convert features into semantic vectors through a transformer-based sentence embedding model, and then calculate the cosine similarity between these vectors to complete the task of feature similarity analysis (Wang et al., 2022). In these researches, the one most relevant to us is Yu et al. (2018), which leverages word2vector to obtain the semantic vectors for the words in features and then calculates the similarity between features by comparing these vectors. Different from it, our method assigns different weights to words based on their part of speech when comparing the semantic vectors of words, so that we can improve the performance of feature similarity calculation by adjusting these weights according to the characteristics of different categories.

9.3. Selecting suitable features for app products

To support the work of software development or evolution, many researchers proposed methods to select suitable features for app products, and there are two main types of methods based on the analyzed data resources. On one hand, many methods select suitable features from user reviews of software products (Scalabrino et al., 2019; Carreño and Winbladh, 2013). For example: Layan Etaiwi et al. give the method to prioritize app reviews for helping developers decide what features in user reviews are more suitable to be added to products (Etaiwi et al., 2020; Gao et al., 2020) mines the reviews related to user requirements from massive amounts of data by training a classifier and gives the method to prioritize the features contained in these reviews to gain suitable features for the product evolution. These methods usually judge the suitability of features to apps based on the number and attributes (e.g., rating and vote) of relevant reviews. On the other hand, many researches select suitable features for app products by performing market-wide analysis. For example: Negar Hariri et al. use a novel incremental diffusive algorithm to extract the features from online product descriptions, and then employ association rule mining and k-nearest neighbor machine learning method to gain suitable features of apps during the domain analysis process (Hariri et al., 2013; Jiang et al., 2019; Liu et al., 2019) mine description texts of app products to establish the relationships between features for further achieving the goal of selecting suitable features for app products. The methods related to market analysis determine whether the features are suitable to apps mainly by analyzing whether they have a correlation with existing features of apps. In these researches, the one most relevant to us is UICOMPARER (Chen et al., 2018), which also takes the UI of app products as the data resource for analysis. However, benefiting from more comprehensive feature mining and feature relationships established based on extensive UI pages, our method is more capable of capturing key features missed by apps.

10. Conclusion and future work

In this paper, we propose a method to recommend potential features added in the next release for android apps from the level of UI pages, and thereby helping developers understand the standard features missing from their products. Firstly, we use a UI testing tool to collect UI pages for the products in the app repository, and mine the features in them by analyzing XML files corresponding to UI pages and adopting the CNN and transformer encoder decoder to train a model. Then, we identify the products with similar features to the app being analyzed by comparing the features gained from the UI. Finally, we establish the feature relationships to further construct a relationship matrix by analyzing the UI pages with feature information, and utilize this matrix to recommend suitable features for the UI pages of the analyzed app.

We constructed experiments to evaluate important steps in our method. The experimental results show that the redundant information obtained by $Rule_1$, $Rule_2$ and $Rule_3$ is very little (90.42% of texts gained through these rules are related to feature information), and $Rule_4$ as well as our deep learning model can provide reliable feature texts for UI components that cannot gain such texts through $Rule_1$, $Rule_2$ and $Rule_3$ (the accuracy values of $Rule_4$ and model are 97.00% and 64.75% respectively), so we can mine the feature information in UI pages effectively by combining 4 rules and the model. The experimental results also show that most of similar products identified by our method for apps are in the same categories as them (accuracy@10, accuracy@50, and accuracy@100 are 75.41%, 76.65%, and 78.13% respectively), which indirectly indicates that the method can identify similar products for apps reasonably. In addition, the experiment based on the elimination-recovery strategy verifies that the proposed method is helpful for mobile applications to enrich the catalog of features (the method can recover features for 67.00% of UI pages).

Our future researches mainly focus on three aspects. Firstly, we want to develop a well-established tool to support our approach in practice. Secondly, some studies (e.g., Jeong et al., 2020; Park and Zahabi, 2021; Wesson et al., 2010; Kortum and Sorber, 2015) give the methods to measure the usability of mobile application UI. Due to the lack of relevant data, the method in this paper does not consider the impact of recommended features on the usability of app UI. In the future work, we will try to take the usability into account in feature recommendation. Thirdly, we will try to combine dynamic analysis technology with static analysis technology to obtain the UI pages of app products more comprehensively, so as to improve the effect of feature recommendation.

CRediT authorship contribution statement

Yihui Wang: Conceptualization, Methodology, Software, Writing – original draft. **Shanquan Gao:** Methodology, Writing – original draft, Conceptualization. **Xingtong Li:** Validation, Investigation. **Lei Liu:** Data curation. **Huaxiao Liu:** Writing – review & editing, Supervision, Project administration.

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, JLU, the National Natural Science Foundation of China (NSFC) No. 62102160, Science and Technology Research Project of Education Department of JiLin Province of China (JJKH20211104KJ).

References

Al-Subaihin, A., Sarro, F., Black, S., Capra, L., 2019. Empirical comparison of text-based mobile apps similarity measurement techniques. Empir. Softw. Eng. 24, 3290–3315.

Avdiienko, V., Kuznetsov, K., Rommelfanger, I., Rau, A., Gorla, A., Zeller, A., 2017. Detecting behavior anomalies in graphical user interfaces. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). pp. 201–203.

Bird, S., 2004. NLTK: The natural language toolkit. arXiv cs.CL/0205028.

Blei, D., Ng, A., Jordan, M.I., 2003. Latent Dirichlet allocation. J. Mach. Learn. Res. 3, 993–1022.

Carreño, L.V.G., Winbladh, K., 2013. Analysis of user comments: An approach for software requirements evolution. In: 2013 35th International Conference on Software Engineering (ICSE). pp. 582–591.

Chen, J., Chen, C., Xing, Z., Xia, X., Zhu, L., Grundy, J.C., Wang, J., 2020a. Wireframe-based UI design search through image autoencoder. ACM Trans. Softw. Eng. Methodol. (TOSEM) 29, 1–31.

Chen, J., Chen, C., Xing, Z., Xu, X., Zhu, L., Li, G., Wang, J., 2020b. Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE). pp. 322–334.

Chen, X., Zou, Q., Fan, B., Zheng, Z., Luo, X., 2018. Recommending software features for mobile applications based on user interface comparison. Requir. Eng. 24, 545–559.

Eler, M.M., Orlandin, L., Oliveira, A.D.A., 2019. Do Android app users care about accessibility?: an analysis of user reviews on the Google play store. In: Proceedings of the 18th Brazilian Symposium on Human Factors in Computing Systems.

2006. EMMA: a free java code coverage tool. http://emma.sourceforge.net/.

Etaiwi, L., Hamel, S., Guéhéneuc, Y.-G., Flageol, W., Morales, R., 2020. Order in chaos: Prioritizing mobile app reviews using consensus algorithms. In: 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC). pp. 912–920.

Frey, R.M., Xu, R., Ilic, A., 2017. Mobile app adoption in different life stages: An empirical analysis. Perv. Mob. Comput. 40, 512–527.

Gao, S., Liu, L., Liu, Y., Liu, H., Wang, Y., 2020. Updating the goal model with user reviews for the evolution of an app. J. Softw. Evol. Process. 32.

Gao, S., Liu, L., Liu, Y., Liu, H., Wang, Y., 2021. API recommendation for the development of android app features based on the knowledge mined from app stores. Sci. Comput. Program. 202, 102556.

Goldberg, Y., Levy, O., 2014. Word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. arXiv abs/1402.3722.

Gorla, A., Tavecchia, I., Gross, F., Zeller, A., 2014. Checking app behavior against app descriptions. In: Proceedings of the 36th International Conference on Software Engineering.

Häring, M., Stanik, C., Maalej, W., 2021. Automatically matching bug reports with related app reviews. arXiv abs/2102.07134.

Hariri, N., Castro-Herrera, C., Mirakhorli, M., Cleland-Huang, J., Mobasher, B., 2013. Supporting domain analysis through mining and recommending features from online product listings. IEEE Trans. Softw. Eng. 39, 1736–1752.

Harman, M., Jia, Y., Zhang, Y., 2012. App store mining and analysis: MSR for app stores. In: 2012 9th IEEE Working Conference on Mining Software Repositories (MSR). pp. 108–111.

Hassan, S., Tantithamthavorn, C., Bezemer, C., Hassan, A., 2018. Studying the dialogue between users and developers of free apps in the google play store. Empir. Softw. Eng. 23, 1275–1312.

He, K., Zhang, X., Ren, S., Sun, J., 2016. Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 770–778.

Hindle, A., Bird, C., Zimmermann, T., Nagappan, N., 2012. 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). pp. 243–252.

Hong, J., Tamakloe, R., Park, D., 2020. Discovering insightful rules among truck crash characteristics using apriori algorithm. J. Adv. Transp. 2020, 1–16.

Huang, L., Ma, J., Chen, C., 2017. Topic detection from microblogs using T-LDA and perplexity. In: 2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW). pp. 71–77.

Iacob, C., Harrison, R., 2013. Retrieving and analyzing mobile apps feature requests from online reviews. In: 2013 10th Working Conference on Mining Software Repositories (MSR). pp. 41–44.

Jeong, J., Kim, N., In, H.P., 2020. Detecting usability problems in mobile applications on the basis of dissimilarity in user behavior. Int. J. Hum. Comput. Stud. 139, 102364.

Jiang, H., Zhang, J., Li, X., Ren, Z., Lo, D., Wu, X., Luo, Z., 2019. Recommending new features from mobile app descriptions. ACM Trans. Softw. Eng. Methodol. (TOSEM) 28, 1–29.

Johann, T., Stanik, C., AlirezaM.Alizadeh, B., Maalej, W., 2017. SAFE: A simple approach for feature extraction from app descriptions and app reviews. In: 2017 IEEE 25th International Requirements Engineering Conference (RE). pp. 21–30.

Khalid, H., Shihab, E., Nagappan, M., Hassan, A., 2015. What do mobile app users complain about? IEEE Softw. 32, 70–77.

Kortum, P.T., Sorber, M., 2015. Measuring the usability of mobile applications for phones and tablets. Int. J. Hum.-Comput. Interact. 31, 518–529.

Lämsä, T., 2017. Comparison of GUI testing tools for android applications.

Li, C., Ding, N., Zhang, G., Li, L., 2019. Association analysis of serial cases based on apriori algorithm. In: ICMAI 2019.

Lin, T.-Y., Maire, M., Belongie, S.J., Hays, J., Perona, P., Ramanan, D., Dollár, P., Zitnick, C.L., 2014. Microsoft COCO: Common objects in context. In: ECCV.

Liu, Y., Liu, L., Liu, H., Li, S., 2019. Information recommendation based on domain knowledge in app descriptions for improving the quality of requirements. IEEE Access 7, 9501–9514.

Liu, Y., Liu, L., Liu, H., Wang, X., 2018. Analyzing reviews guided by app descriptions for the software development and evolution. J. Softw. Evol. Process. 30.

Liu, Y., Liu, L., Liu, H., Wang, X., Yang, H., 2017. Mining domain knowledge from app descriptions. J. Syst. Softw. 133, 126–144.

Martin, W., Sarro, F., Jia, Y., Zhang, Y., Harman, M., 2016. A survey of app store analysis for software engineering. IEEE Trans. Softw. Eng. 43 (9), 817–847.

McIlroy, S., Ali, N., Hassan, A., 2015. Fresh apps: an empirical study of frequently-updated mobile apps in the google play store. Empir. Softw. Eng. 21, 1346–1370.

Ouzzani, M., Hammady, H.M., Fedorowicz, Z., Elmagarmid, A., 2016. Rayyan—a web and mobile app for systematic reviews. Syst. Rev. 5.

Park, J., Zahabi, M., 2021. A novel approach for usability evaluation of mobile applications. In: Proceedings of the Human Factors and Ergonomics Society Annual Meeting, Vol. 65. pp. 437–441.

Patil, N., Bhole, D., Shete, P., 2016. Enhanced UI automator viewer with improved android accessibility evaluation features. In: 2016 International Conference on Automatic Control and Dynamic Optimization Techniques (ICACDOT). pp. 977–983.

Rehurek, R., Sojka, P., 2011. Gensim – statistical semantics in python.

Sasaguri, H., Nilsson, P., Hashimoto, S., Nagata, K., Saito, T., Strooper, B.D., Hardy, J., Vassar, R., Winblad, B., Saido, T., 2017. APP mouse models for Alzheimer's disease preclinical studies. EMBO J. 36, 2473–2487.

Scalabrino, S., Bavota, G., Russo, B., Penta, M.D., Oliveto, R., 2019. Listening to the crowd for the release planning of mobile apps. IEEE Trans. Softw. Eng. 45, 68–86.

Schubert, E., Sander, J., Ester, M., Kriegel, H., Xu, X., 2017. DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN. ACM Trans. Database Syst. 42, 19:1–19:21.

Shah, F.A., Sirts, K., Pfahl, D., 2019. Using app reviews for competitive analysis: tool support. In: Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics.

Sheridan, K., Puranik, T., Mangortey, E., Pinon-Fischer, O.J., Kirby, M., Mavris, D., 2020. An application of DBSCAN clustering for flight anomaly detection during the approach phase.

Soui, M., Chouchane, M., Mkaouer, M.W., Kessentini, M., Ghédira, K., 2020. Assessing the quality of mobile graphical user interfaces using multi-objective optimization. Soft Comput. 24, 7685–7714.

Strauch, Y., Grundy, J., 2021. Two novel performance improvements for evolving CNN topologies. arXiv abs/2102.05451.

Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., Pu, G., Liu, Y., Su, Z., 2017. Guided, stochastic model-based GUI testing of Android apps. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.

Sun, Y., Xue, B., Zhang, M., Yen, G., Lv, J., 2020. Automatically designing CNN architectures using the genetic algorithm for image classification. IEEE Trans. Cybern. 50, 3840–3854.

Takushima, H., Tamura, A., Ninomiya, T., Nakayama, H., 2019. Multimodal neural machine translation using CNN and transformer encoder.

Uddin, M.K., He, Q., Han, J., Chua, C., 2021. Mining cross-domain apps for software evolution: A feature-based approach. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 743–755.

Vakulenko, S., Müller, O., vom Brocke, J., 2014. Enriching itunes app store categories via topic modeling. In: ICIS.

Wang, Y., Wang, J., Zhang, H., Ming, X., Shi, L., Wang, Q., 2022. Where is your app frustrating users? arXiv preprint arXiv:2204.09310.

Weichbroth, P., 2020. Usability of mobile applications: A systematic literature study. IEEE Access 8, 55563–55577.

Wesson, J., Singh, A., van Tonder, B., 2010. Can adaptive interfaces improve the usability of mobile applications? In: HCIS.

Xi, S., Yang, S., Xiao, X., Yao, Y., Xiong, Y., Xu, F., Wang, H., Gao, P., Liu, Z., Xu, F., Lu, J., 2019. 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.

Xu, R., Frey, R.M., Fleisch, E., Ilic, A., 2016. Understanding the impact of personality traits on mobile app adoption - insights from a large-scale field study. Comput. Hum. Behav. 62, 244–256.

Yang, Y., Wang, L., Shi, S., Tadepalli, P., Lee, S., Tu, Z., 2020. On the sub-layer functionalities of transformer decoder. In: EMNLP.

Yu, L., Chen, J., Zhou, H., Luo, X., Liu, K., 2018. Localizing function errors in mobile apps with user reviews. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, pp. 418–429.

Yu, H., Lian, Y., Yang, S., Tian, L., Zhao, X., 2016. Recommending features of mobile applications for developer. In: ADMA.

Zelenchuk, D., 2019. Supervised monkey tests with espresso and UI automator.

Zhang, C., Wang, H., Wang, R., Guo, Y., Xu, G., 2018. Re-checking app behavior against app description in the context of third-party libraries. In: SEKE.

Yihui Wang is a Ph.D. student in College of Computer Science and Technology Jilin University, China. She received the bachelor's degree in Internet of Things Engineering from Jilin University, China. Her research interests include text mining, mobile app development and GUI improvement.

Shanquan Gao is a Ph.D. student in College of Computer Science and Technology Jilin University, China. He received the bachelor's degree in Geomatics Engineering from Taiyuan University of Technology, China in 2015 and the master's degree in Software engineering from Jilin University, China, in 2018. His research interests include user review mining, API recommendation, GUI development.

Xingtong Li is a postgraduate student in College of Computer Science and Technology Jilin University, China. Her research interests include review mining, mobile app development.

Lei Liu is a doctoral supervisor in College of Computer Science and Technology Jilin University, China. He received the Master degree in Computer Science from Jilin University, China in 1985. The central themes of his research are programming language and its realization technology, software security and cloud computing, the semantic web and ontology engineering, knowledge representation and reasoning, etc. At Jilin University, he has held responsibilities for more than 30 projects as a lead person in the area of computer science. He has authored numerous papers and technical reports on various international journals and conferences.

Huaxiao Liu is an associate professor in College of Computer Science and Technology Jilin University, China. He received the Ph.D. in Computer Science from Jilin University, China in 2013. The central theme of his research is improving software quality, and his recent research concerns the software requirements engineering, app accessibility and deep learning.