3-2019

Automatic, highly accurate app permission recommendation

Zhongxin LIU

Xin XIA

David LO
Singapore Management University, davidlo@smu.edu.sg

John GRUNDY

Citation

LIU, Zhongxin; XIA, Xin; LO, David; and GRUNDY, John. Automatic, highly accurate app permission recommendation. (2019). *Automated Software Engineering*. 1-34.
Available at: https://ink.library.smu.edu.sg/sis_research/4366

Published in Automated Software Engineering,
March 2019, Pages 1-34
DOI 10.1007/s10515-019-00254-6

Automatic, highly accurate app permission recommendation

Zhongxin Liu[1] · Xin Xia[2] · David Lo[3] · John Grundy[2]

Received: 7 March 2018 / Accepted: 8 March 2019

Abstract
To ensure security and privacy, Android employs a permission mechanism which requires developers to explicitly declare the permissions needed by their applications (apps). Users must grant those permissions before they install apps or during runtime. This mechanism protects users' private data, but also imposes additional requirements on developers. For permission declaration, developers need knowledge about what permissions are necessary to implement various features of their apps, which is difficult to acquire due to the incompleteness of Android documentation. To address this problem, we present a novel permission recommendation system named PerRec for Android apps. PerRec leverages mining-based techniques and data fusion methods to recommend permissions for given apps according to their used APIs and API descriptions. The recommendation scores of potential permissions are calculated by a composition of two techniques which are implemented as two components of PerRec: a collaborative filtering component which measures similarities between apps based on semantic similarities between APIs; and a content-based recommendation component which automatically constructs profiles for potential permissions from existing apps. The two components are combined in PerRec for better performance. We have evaluated PerRec on 730 apps collected from Google Play and F-Droid, a repository of free and open source Android apps. Experimental results show that our approach significantly improves the state-of-the-art approaches $APRec^{CF_{correlation}}$, $APRec^{TEXT}$ and Axplorer.

Keywords Android security model · Permission recommendation · Collaborative filtering · Content-based recommendation

✉ Xin Xia
xin.xia@monash.edu

Extended author information available on the last page of the article

1 Introduction

Android is a popular mobile platform which enjoyed the largest market share (85.0%) in the first quarter of 2017 (IDC 2017). In order to prevent apps from accessing device resources and user's sensitive data in an unwanted way, Android employs a unique permission mechanism. This explicit permission mechanism requires a developer to declare permissions that are needed by an app in a specific file *AndroidManifest.xml*. Users must explicitly grant the permissions requested in the manifest file before installation or at runtime. In this way, users can mitigate and manage potential security risks.

Unfortunately, the permission mechanism introduces new requirements to developers. While developing an Android app, a developer has to learn not only the APIs used to implement the features but also the corresponding permissions. Moreover, Android official documentation (Android 2017b) advises developers to minimize the number of declared permissions in order to reduce security risks and improve user experience. This is compounded the fact that there are now more than 150 permissions in Android 7.1 (Android 2017c) and that the correct use of these permissions requires much expert knowledge. Developers usually learn the knowledge from the official Android documentation for API classes and permissions. Nevertheless, the documentation has been found to be incomplete (Felt et al. 2011; Au et al. 2012). Explicit links between APIs and permissions are not always well documented. These facts make it difficult for developers to find suitable permissions. So, a permission recommendation system is needed to help developers declare permissions correctly.

Some program-analysis-based tools can be utilized to meet the above-mentioned need, such as *Stowaway* (Felt et al. 2011), *PScout* (Au et al. 2012), *Androguard* (Desnos and Gueguen 2011) and AXPLORER (Backes et al. 2016). Each of them builds an Android permission map that identifies the permissions required for each API call. Permissions for an app can be recommended according to its used APIs and these permission maps. Most of these approaches usually require some manual work. For example, there are four permissions that are checked in native C/C++ code in Android 4.0, but *PScout* can not handle them automatically. These require manual inspection of the source code where these permission strings are used to extract a complete permission map (Au et al. 2012). Many of the permission maps extracted by these approaches are also related to specific Android versions, which means we must update these maps with the evolution of Android. Moreover, there is an empirical evidence that highlights the need for analysis beyond Android APIs for accurate permission recommendations (Karim et al. 2016). For example, there is an Android app named *Mp3 Voice Recorder* (GitHub 2017) in F-Droid (F-Droid 2017). It uses 18 APIs, including an API directly related to voice recording (i.e., *android.media.AudioRecord*) and only requires one permission: *RECORD_AUDIO*. In the permission map extracted by AXPLORER, we can not find any mapping which contains API *android.media.AudioRecord* or permission *RECORD_AUDIO*. This means AXPLORER is not able to recommend permissions correctly for such app.

In this paper, we describe a new automatic and more accurate approach to perform app permission recommendations. We propose PERREC, a novel mining-based permission recommendation system that performs better than existing approaches. PERREC has two main components, namely SEM and CBR. Both of them are leveraged to calculate recommendation scores for permissions. Sem uses a nearest-neighbor-based collaborative filtering technique. It utilizes similarities between apps, which are captured through both apps' commonly used APIs and semantic similarities between APIs, to select nearest neighbors. Inspired by content-based recommendation (Gunawardana and Shani 2009), Cbr first constructs the profile of an app using its used APIs. Then, it proposes a way to build permission profiles, which share the same vector space with the app profiles, from existing apps. Given a permission, Cbr computes its recommendation score according to the similarity between its profile and the target app's profile. PerRec combines Sem and Cbr using data fusion techniques (Wu 2012; Lucia et al. 2014) to achieve better performance.

We evaluate PerRec on 730 apps collected from Google Play and F-Droid. We use the Mean Average Precision (MAP) and two evaluation metrics which we refer to as Necessary Recall (NR) and Total-Recall Ratio (TRR) to measure the performance and effectiveness of PerRec. We learn from the experiment results that PerRec outperforms the state-of-the-art approaches.

The two main contributions of this work are:

1. We propose a new, automatic and accurate Android permission recommendation system, PerRec, which blends similarities between apps and content-based recommendation through data fusion methods to recommend permissions for Android apps.
2. We demonstrate by experiments on 730 apps the effectiveness of PerRec. The experiment results show that PerRec outperforms state-of-the-art baselines by substantial margins.

The remainder of this paper is organized as follows. In Sect. 2, we describe our motivation. In Sect. 3, we present the overall framework of PerRec. The three components of PerRec, i.e., Sem, Cbr and the data fusion component, are elaborated in Sects. 4–6, respectively. Section 7 illustrates the implementation details of PerRec. Our experimental results are provided in Sect. 8. In Sect. 9, we discuss the impacts of the amount of training data and the number of nearest neighbors in Sem, the customization of PerRec, the false positive results of PerRec and the cases where our approach performs badly. After brief discussions about threats to validity (Sect. 10) and related work (Sect. 11), we conclude this work in Section 12.

2 Motivation

2.1 Background

Android applications run inside a *sandbox* that restricts the system-level operations the app can make use of. An app has to request permission to make use of resources residing outside of this sandbox via a *permission*. These permissions are specified explicitly and declaratively inside a *manifest* file shipped with the app. For example, to allow an app to send an SMS message and access the internet:

```
<manifest xmlns:android="http://schemas.android.com/apk/
    res/android"
          package="com.example.XYZ">
        <uses-permission android:name="android.permission
            .SEND_SMS"/>
        <uses-permission android:name="android.permission
            .INTERNET"/>

    <application ...>
        ...
    </application>
</manifest>
```

The developer checks for permission inside the app code when needed e.g.:

```
if (ContextCompat.checkSelfPermission(thisActivity,
    Manifest.permission.SEND_SMS)
        == PackageManager.PERMISSION_GRANTED) {
    ...
    SmsManager smsManager = SmsManager.getDefault();
    smsManager.sendTextMessage(phoneNo, null, msg, null,
        null);
    ...
}
```

These permission manifests can become large and complicated. A further challenging problem is for developers to correctly and accurately identify and specify the set of required permissions in the app and maintain these over time.

To understand developers' experience on Android permissions, we conduct a survey by sending emails to 556 developers/organizations who have at least one app distributed in F-Droid. We received 87 responses. Over 72% of the respondents acknowledge that they once felt confused about what permissions their apps should declare when developing their Android apps. 56% of the respondents think a tool, which can automatically recommend permissions for their apps according to the used APIs, will be useful or very useful, and 30% of the respondents think this tool may be useful. Moreover, we asked the respondents if they would try such a tool. 78% of them state that they are willing to try it. Detailed responses are publicly available at https://goo.gl/neXPjZ. The results of our survey highlights the need for a permission recommendation tool. To address this need, we propose a novel mining-based permission recommendation system named PerRec in this work. Our approach recommends permissions for Android apps by learning from high-quality Android apps and experienced developers, hence could be very useful for new Android developers.

2.2 Usage scenario

The usage scenarios of PerRec are as follows:

Without tool Bob is developing an Android app. To determine the necessary set of Android permissions for his app, he needs to read the documentation of the APIs which may require permissions or test his app on a device and learn from the error messages, both of which affect the efficiency of Bob's development. During the development, Bob removes a feature, and some declared permissions become unnecessary. But Bob forgets to remove such permissions from the manifest file, which increases the risks for Bob's app to be compromised (Bartel et al. 2012). In addition, after Bob releases his app, users complain that Bob's app requires some unnecessary permissions, hence Bob's app gets less popular reviews than it deserves (Felt et al. 2012; Android 2017b).

With tool Bob makes use of PerRec during and after his development. To determine the necessary permissions for his app, Bob first reviews the permissions recommended by our tool. Then, Bob adds/ignores the permissions which he can make sure is necessary/unnecessary quickly. For other permissions, Bob can check their documentation or wait for test results to make decisions. In this way, Bob avoids reading unnecessary documents and reduces the times of tests, hence improves his efficiency. Before releasing, Bob uses our tool to detect extraneous permissions. By checking the recommended permissions, Bob finds and removes unnecessary permissions, which reduces the attack surface (Manadhata and Wing 2010) of Bob's app. After re-releasing, users do not complain about the permissions problems.

In our survey mentioned in Sect. 2.1, some respondents want our approach to be implemented as an IDE plug-in, but some other respondents think our tool should be online. Based on these suggestions, we think that PerRec can be deployed as a cloud service, and provides both a web app and an IDE plug-in as the front-end. To use the cloud service, developers can upload their APKs (i.e., Android application packages) through the web tool, and the recommended permissions will be displayed on the web page. Alternatively, they can leverage the IDE plug-in to automatically handle the whole procedure on the client side, i.e., extracting and uploading API lists as well as receiving and displaying recommended Android permissions. Developers only need to review the displayed permissions to make decisions.

2.3 Mining-based permission recommendation

Recently, a few mining-based approaches have been proposed to recommend permissions for Android apps. Karim et al. (2016) presented *ApMiner* to map given Android APIs to permissions by combining static analysis and association rule mining. Experimental results show that *ApMiner* outperforms *PScout* and *Androguard*. In a latter work, Bao et al. (2016, 2017) proposed $APRec^{CF_{correlation}}$ and $APRec^{TEXT}$ to improve the performance of *ApMiner* further. In the remainder of this paper, for simplicity sake, we refer to $APRec^{CF_{correlation}}$ and $APRec^{TEXT}$ as Cor and Text, respectively.

Given a target app, Cor leverages nearest-neighbor-based collaborative filtering to recommend permissions for it based on its neighbors, i.e., the apps that call some APIs which are also used by the target app. Text builds a text mining model to analyze the readme files of apps in an attempt to make permission recommendations. Bao et al. (2017) compared Cor and Text with *ApMiner*, and the experimental results showed that Cor and Text both perform better than *ApMiner*, and the performance difference between Cor and Text is small. In practice, Cor ignores the fact that APIs with similar

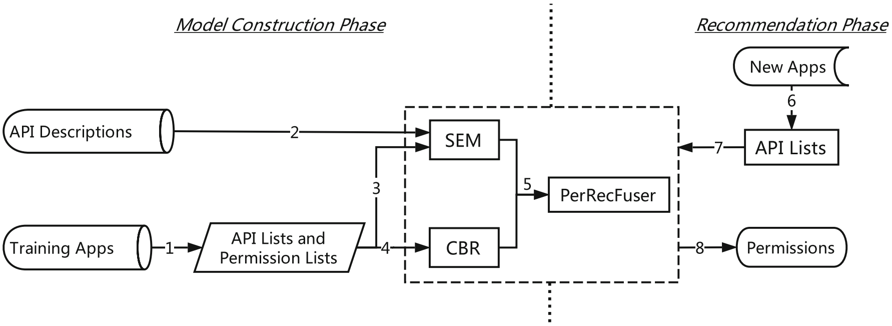


Fig. 1 Overall framework of PerRec

usage may require the same permissions. For example, API *java.net.URLConnection* and API *java.net.HTTPURLConnection* can be used in similar contexts. Given two apps APP_1 and APP_2, if APP_1 only uses *java.net.URLConnection* and APP_2 only uses *java.net.HTTPURLConnection*, Cor will not treat them as neighbors. However, since the two APIs both allow an app to access the Internet, both of them require permission *INTERNET*, and the two apps should be regarded as neighbors for collaborative filtering. Moreover, Text recommends permissions for an app according to its readme file, but not every app has a readme file.

3 PerRec

To address the issues of existing permission recommendation tools, we develop a new composite approach and framework, PerRec, shown in Fig. 1. PerRec contains two phases: a model construction phase and a recommendation phase. In the model construction phase, we aim to build a recommendation model from existing apps with known permissions and API descriptions. In the recommendation phase, given a new app APP_t, the model is used to recommend permissions required by APP_t according to APP_t's used APIs.

Our framework first extracts *API Lists* and *Permission Lists* from existing apps (Step 1). Given an app, its *API List* is the list of its used APIs, and its *Permission List* is the list of permissions required by this app. Then, to build a recommendation model, *API Lists*, their corresponding *Permission Lists* and API descriptions are input into Sem (Step 2 and 3), while Cbr does not need API descriptions (Step 4). The specific building processes of the two components are described in Sects. 4 and 5 below. Next, we construct a composite recommendation model $PerRecFuser$ by blending Sem and Cbr (Step 5).

After $PerRecFuser$ is built, it can be used to recommend permissions in the recommendation phase. Given a new app, we first extract its *API List* (Step 6). Then, this *API List* is input into $PerRecFuser$ (Step 7), which will output an ordered list of permissions that may be required by the new app (Step 8). These can then be reviewed and used by the developer.

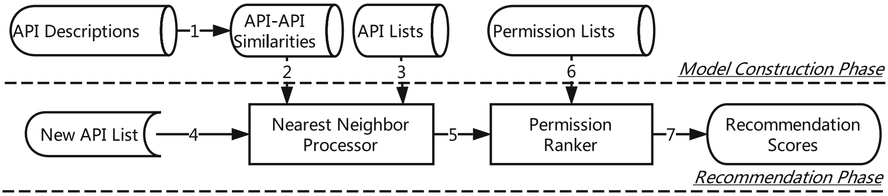


Fig. 2 Overall framework of SEM

4 Semantic-based similarities (SEM)

In order to overcome the shortcomings of COR and TEXT, we propose a new approach named SEM in PERREC to calculate recommendation scores for permissions. SEM also uses the collaborative filtering algorithm, but inspired by the intuition that APIs with similar usage may require the same permissions, SEM calculate the similarities between apps based on not only their commonly used APIs but also the similarities between APIs used by them.

Figure 2 presents the overall structure of our SEM framework. SEM uses the *API Lists* and *Permission Lists* of the training apps as well as API descriptions as its input. In the model construction phase, SEM first calculates the similarities between APIs (Step 1), and then outputs these *API-API similarities* into the *NearestNeighborProcessor* component for the following calculation (Step 2). In the recommendation phase, given a new app APP_t, we first input the *API Lists* of APP_t and all training apps as well as the *API-API similarities* into the the *NearestNeighborProcessor* (Step 2, 3 and 4). The *NearestNeighborProcessor* will calculate the similarities between APP_t and all training apps to find the nearest neighbors for APP_t (Step 5). Finally, based on APP_t's nearest neighbors and their *Permission Lists*, the *PermissionRanker* component calculates a recommendation score for each permission (Step 5, 6 and 7).

In the following paragraphs we first present the calculation of *API-API similarities*, and then elaborate the two key components of SEM: *NearestNeighborProcessor* and *PermissionRanker*:

4.1 API–API similarities

By investigating the API descriptions in Android API documentation (Android 2017a), we find that the descriptions of APIs with similar usage are semantically similar. So in SEM we measure how similar two APIs are using the semantic similarity between their descriptions in API documentation. Given two APIs A_1 and A_2 and corresponding API descriptions d_1 and d_2, we calculate the semantic similarity between A_1 and A_2 according to Mihalcea et al. (2006) and Ye et al. (2016), as follows:

$$sim(A_1, A_2) = sim(d_1, d_2) = \frac{\sum\limits_{w \in d_1} (maxSim(w, d_2) \times idf(w))}{\sum\limits_{w \in d_1} idf(w)} \tag{1}$$

where $idf(w)$ is w's inverse document frequency (idf), and $maxSim(w, d_i)$ can be regarded as the similarity between word w and the API description d_i. $idf(w)$ is defined as:

$$idf(w) = log\left(\frac{n_d}{1 + df_w}\right) \tag{2}$$

where n_d refers to the total number of API descriptions in the corpus, and df_w is the document frequency of word w, i.e., the number of API descriptions which contain w. $maxSim(w, d_i)$ is defined as:

$$maxSim(w, d_i) = \max_{w' \in d_i} sim(w, w')$$

where $sim(w, w')$ refers to the semantic similarity between two words w and w'.

To compute the semantic similarities between words, we first map each word into a real-value vector using the word embeddings pre-tained through GloVe (Pennington et al. 2014). Then we define two words' semantic similarity as the cosine similarity between their word vectors, i.e.:

$$sim(w_1, w_2) = sim_{cos}(\boldsymbol{w_1}, \boldsymbol{w_2}) = \frac{\boldsymbol{w_1}^T \boldsymbol{w_2}}{||\boldsymbol{w_1}||\ ||\boldsymbol{w_2}||} \tag{3}$$

where $\boldsymbol{w_i}$ represents the word embedding of word w_i. According to the steps mentioned above, we can prepare the *API-API similarities* for SEM.

4.2 *NearestNeighborProcessor*

We leverage the *NearestNeighborProcessor* to find the k nearest neighbors of APP_t in the recommendation phase. The *NearestNeighborProcessor* first needs to calculate the similarity between APP_t and each training app. Given a training app APP_i, the similarity between APP_t and APP_i is calculated as follows: First, we construct the feature vectors of APP_t and APP_i respectively. For APP_i, we use the *API vector* as its feature vector, which is defined as:

$$\boldsymbol{v}(APP_i) = (ind(A_1, AL_i), ind(A_2, AL_i), \ldots, ind(A_n, AL_i)) \tag{4}$$

where n is the total number of used APIs, A_i is the i_{th} API in the whole API set, AL_i is the *API List* of APP_i, and $ind(A_i, AL_i)$ indicates whether A_i belongs to AL_i, i.e., whether A_i is used by APP_i. For APP_t, we construct its feature vector based on its and APP_i's used APIs and the *API-API similarities*. The feature vector of APP_t with respect to APP_i is defined as follows:

$$\begin{aligned} \boldsymbol{fv}_i(APP_t) =& (apiSim(A_1, AL_t, AL_i), apiSim(A_2, AL_t, AL_i), \ldots, \\ & apiSim(A_n, AL_t, AL_i)) \end{aligned} \tag{5}$$

where AL_t is the *API List* of APP_t. $apiSim(A_j, AL_t, AL_i)$ is a special function used to measure the similarities between the APIs used by APP_t and APP_i, which is defined as:

$$apiSim(A_j, AL_t, AL_i) = \begin{cases} 0, & \text{if } A_j \notin (AL_t \cup AL_i); \\ 1, & \text{if } A_j \in AL_t; \\ \max\limits_{A_k \in AL_t} sim(A_j, A_k), & \text{otherwise} \end{cases} \tag{6}$$

In the above equation, $sim(A_j, A_k)$ is the similarity between API A_j and API A_k, which is defined in Eq. 1. The idea behind the $apiSim$ function is that for the APIs which are not used by both APP_t and APP_i, it is useless for calculating the app-app similarity; for the APIs used by APP_t, we set the values of their corresponding features as 1, just like the *API vector*; for the APIs which are used by APP_i but not APP_t, we set their corresponding feature values according to the *API-API similarities* described in Sect. 4.1 to help find nearest neighbors.

Then, we regard the cosine similarity between the feature vectors of APP_t and APP_i as their app-app similarity, which is calculated as follows:

$$sim(APP_t, APP_i) = sim_{cos}(\boldsymbol{fv}_i(APP_t), \boldsymbol{v}(APP_i))$$

where sim_{cos} is defined in Eq. 3.

Finally, after we calculate the similarities between APP_t and all the training apps, the training apps can be ranked according to their similarity scores. The top-k apps with highest similarity scores are picked as the k nearest neighbors ($kNNs$) of APP_t. By default, we set k as 10.

4.3 *PermissionRanker*

With the *Permission Lists* of APP_t's $kNNs$ as input, the *PermissionRanker* can compute a recommendation score for each permission. Given a permission P_i, its recommendation score (RS) is the sum of the similarity scores of apps which are in the $kNNs$ and require P_i:

$$RS^{SEM}(P_i) = \sum_{APP_i \in kNNs,\ P_i \in PL_i} sim(APP_t, APP_i) \tag{7}$$

where PL_i is the *Permission List* of APP_i. The higher a permission's recommendation score is, the more likely this permission is to be required by APP_t.

5 Content-based recommendation (Cbr)

Given an app APP_t, Sem recommends permissions based on its nearest neighbors. If there are some permissions which are required by APP_t but not needed by its nearest neighbors, Sem will perform badly. To mitigate this shortcoming, we propose another

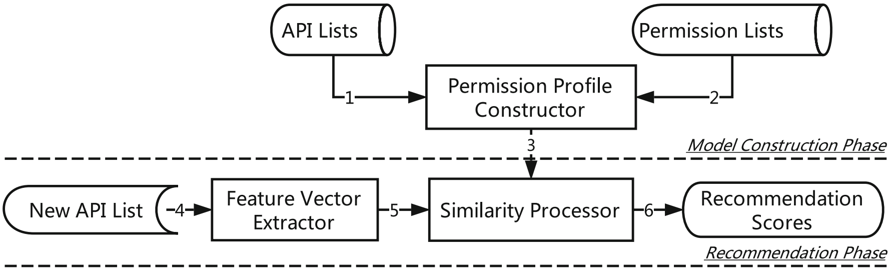


Fig. 3 Overall framework of CBR

component named CBR, which recommends permissions through permission profiles learned from all training apps instead of only nearest neighbors, to complement SEM.

CBR was inspired by content-based recommendation (Gunawardana and Shani 2009), which builds a user's profile based on the profiles of those items which have been rated by this user. In this way, users and items are presented in the same vector space, hence the similarities between them can be easily calculated. For app permission recommendation, we can treat each app as a user and each permission as an item. The action that an app declares a permission is similar to the action that a user rates an item. Therefore, given a permission P_i, CBR constructs its profile based on the profiles of those apps requiring P_i.

The structure of our CBR framework is shown in Fig. 3. In the model construction phase, the *API Lists* and *Permission Lists* of training apps are input into the *PermissionProfileConstructor* component to construct permission profiles (Step 1, 2 and 3). In the recommendation phase, APP_t's profile is extracted by the *FeatureVectorExtractor* based on its *API List* (Step 4 and 5). After that, the *SimilarityProcessor* calculates the similarities between APP_t's profile and each permission's profile. The resultant similarity scores are considered as the recommendation scores of the permissions (Step 6).

CBR contains three components: *FeatureVectorExtractor*, *PermissionProfileConstructor* and *SimilarityProcessor*, which we describe in the following paragraphs.

5.1 *FeatureVectorExtractor*

This component is responsible for constructing APP_t's profile based on its *API List*. In this approach, we use APP_t's *API vector*, which is defined in Eq. 4, as its profile.

5.2 *PermissionProfileConstructor*

This component leverages a tf-idf-like method to build profiles for permissions. Tf-idf, short for term frequency–inverse document frequency, can reflect how important a term is to a document in a corpus. Tf-idf scores are often used as the weights of terms in the field of information retrieval. For a given term t and a document d, the *tf* of term t in document d is defined as:

Table 1 A tiny dataset to illustrate how *PermissionProfileConstructor* Works

App	API list	Permission list
APP_1	A_1, A_2	P_1, P_3
APP_2	A_2, A_3	P_2, P_3
APP_3	A_1, A_3	P_1, P_2

$$tf(t,d) = \frac{f_{t,d}}{n_d}$$

In the above equation, $f_{t,d}$ is the times that t occurs in d, and n_d refers to the total number of terms in d.

Idf, defined in Eq. 2, is incorporated in tf-idf to diminish the weight of terms that occur very frequently in the document set, e.g., “the”, and increase the weight of terms that occur rarely. The *tfidf* of term t in document d is computed as:

$$tfidf(t,d) = tf(t,d) \times idf(t)$$

For each permission P_i, we create its *permission document* PD_i. PD_i is built by concatenating the *API Lists* of those apps which require P_i. For example, with the data in Table 1, the *permission document* of P_1 is “A_1, A_2, A_1, A_3”.

We first build *permission documents* for all permissions from the training apps. A corpus is created from these *permission documents* and API names are regarded as terms. Then, given an API A_i and a permission P_i, we can get $tfidf(A_i, PD_i)$ according to the corpus and the tf-idf equations mentioned above. Finally, the *PermissionProfileConstructor* can build P_i's profile $profile(P_i)$ as follows:

$$profile(P_i) = norm(\boldsymbol{v}'(P_i)) \tag{8}$$

$$\boldsymbol{v}'(P_i) = (tfidf(A_1, PD_i), tfidf(A_2, PD_i), \ldots, tfidf(A_n, PD_i)) \tag{9}$$

In the above equation, $norm$ is the normalization function, and $\boldsymbol{v}'(P_i)$ is P_i's *tf-idf vector*, which is similar in structure to an *API vector*. It is obvious that a permission profile and an app profile (i.e., *API vector*) share the same vector space.

5.3 *SimilarityProcessor*

This component is used to calculate recommendation scores for all permissions. Its input is all the permission profiles and APP_t's *API vector*. Given a permission P_i, we use the cosine similarity between APP_t's *API vector* and P_i's permission profile as P_i's recommendation score (RS):

$$RS^{CBR}(P_i) = sim_{cos}(\boldsymbol{v}(APP_t),\ profile(P_i)) \tag{10}$$

Table 2 An example to illustrate how data fusion methods work

Permission	RS^{SEM}	RS^{CBR}
1	0.55	0
2	0.83	0.92
3	0	0.39

In the above equation, sim_{cos} is used to calculated cosine similarity, which is defined in Eq. 3.

6 *PerRecFuser*

In the model construction phase, we build the models of SEM and CBR from training apps. Then, with APP_t's *API List* as input, both SEM and CBR can calculate recommendation scores for all permissions in the recommendation phase. The *PerRecFuser* component is then used to compose the two sets of recommendation scores together to make the final permission recommendation.

First, *PerRecFuser* leverages $l1$ norm to normalize the two sets of scores outputted by SEM and CBR separately. Then, by adapting an unsupervised data fusion method proposed in the information retrieval field, we combine the normalized scores from SEM and CBR to produce a new recommendation score for every permission. Based on the new scores, the *PerRecFuser* can provide an ordered list of permissions as the recommendation made by PERREC.

In this work, we investigate six well-known unsupervised data fusion methods: Max, Min, CombSUM (Fox et al. 1993; Fox and Shaw 1994), CombANZ (Fox et al. 1993; Fox and Shaw 1994), CombMNZ (Fox et al. 1993; Fox and Shaw 1994) and Borda Count (Aslam and Montague 2001). Based on the data fusion method used by *PerRecFuser*, there are six variants of PERREC, and we call them PERREC^{Max}, PERREC^{Min}, PERREC^{SUM}, PERREC^{ANZ}, PERREC^{MNZ} and PERREC^{BC} respectively. The following paragraphs elaborate how the six data fusion methods work using the example shown in Table 2.

6.1 Max

For each permission, this method selects the maximum score as its new recommendation score.

Example Based on Table 2, the set of new recommendation scores for Permission 1 to 3 would be {0.55, 0.92, 0.39}.

6.2 Min

Contrary to Max, this method selects the minimum score for each permission as its new recommendation score.

Example Based on Table 2, the set of new recommendation scores for Permission 1 to 3 would be {0, 0.83, 0}.

6.3 CombSUM

This method assumes that each approach is equally important. For each permission, this method simply sums up its recommendation scores from SEM and CBR as the new score.

Example Based on Table 2, the set of new recommendation scores for Permission 1 to 3 would be {0.55, 1.75, 0.39}

6.4 CombANZ

This method combines the recommendation scores from different approaches by computing the average of the non-zero scores. Suppose there are n_i approaches that assign non-zero scores to permission P_i, the new score of P_i is computed as follows:

$$Score(P_i) = 1/n_i \times (RS^{SEM}(P_i) + RS^{CBR}(P_i)) \tag{11}$$

Example Based on Table 2, the set of new recommendation scores for Permission 1 to 3 would be {$\frac{0.55}{1}$, $\frac{1.75}{2}$, $\frac{0.39}{1}$}={0.55, 0.88, 0.39}

6.5 CombMNZ

Given a permission P_i, CombMNZ multiplies the summation of all the recommendation scores of P_i with the number of approaches that assign non-zero scores to P_i. Assume that n_i denotes this number, CombMNZ calculates the new recommendation score for P_i as follows:

$$Score(P_i) = n_i \times (RS^{SEM}(P_i) + RS^{CBR}(P_i)) \tag{12}$$

Example Based on Table 2, the set of new recommendation scores for Permission 1 to 3 would be {1×0.55, 2×1.75, 1×0.39}={0.55, 3.5, 0.39}

6.6 Borda count

To combine the recommendation scores from different approaches, Borda Count first converts each recommendation score set into a rank set—permissions with higher scores would obtain smaller ranks. Then, for each permission, this method computes its *ranking points* given by different approaches. The *ranking point* of a permission given by an approach is defined as the subtraction of the permission's rank from the total number of permissions, i.e.:

$$RP_i^j = |P| - rank_i^j \tag{13}$$

Table 3 Example of ranks and ranking points

Permission	Ranks	Ranking points
1	{2, 3}	{1, 0}
2	{1, 1}	{2, 2}
3	{3, 2}	{0, 1}

In the above equation, $rank_i^j$ is the rank of P_i that is outputted by approach j, $|P|$ is the total number of permissions and RP_i^j is the corresponding *ranking point*. Finally, the new recommendation score of a permission is the sum of its *ranking points* given by all approaches. In this work, the new score can be computed as follows:

$$Score(P_i) = RP_i^{SEM} + RP_i^{CBR} \tag{14}$$

Example The ranks and *ranking points* of Permission 1 to 3 are shown in Table 3. Based on these *ranking points*, the set of new recommendation scores for Permission 1 to 3 would be {1, 4, 1}.

7 Implementation

In this section, we present the implementation details of PerRec.

PerRec is implemented on top of scikit-learn (Pedregosa et al. 2011), a well-known machine learning library in Python. Our approach takes *API Lists* and *Permission Lists* of apps as well as API descriptions as input.

API List extraction APIs which are used by an app are declared in import statements in Java source code files. By using srcML (Collard et al. 2003), a lightweight static analysis tool which transforms source code to a single XML file, we extract the *API List* of an app from such XML file. Considering user-defined classes, we only select the API classes which are contained in the Android software stack and Java standard libraries, such as *android.app.Notification* and *java.net.URL*.

Permission List extraction The *Permission List* of an app is extracted from its *AndroidManifest.xml* file directly.

API description extraction To obtain API descriptions, we first download Java API docs from the official website of Java (Oracle 2017b) and collect Android API docs from Android SDK (Android 2017d). Then we extract the description of each API class from its corresponding HTML file. We only collect the textual descriptions of API class, and code snippets and method descriptions are removed through XPath expression.

Pre-Trained Word Embeddings. To calculate the semantic similarities between words, we leverage the word embeddings pre-trained by GloVe (Pennington et al. 2014) in Sect. 4.1. The authors of GloVe publish several pre-trained sets of word embeddings in GloVe's official website (Pennington et al. 2017). We use the one whose training corpus is Wikipedia 2014 and Gigaword 5 and vector dimension is 50. This set of word embeddings are powerful enough for SEM to perform well, and vectors with 50 dimensions can reduce the time needed to calculate semantic similarity.

In addition, to enable others to use PERREC, we have published our source code and dataset on GitHub.[1]

8 Evaluation

In this section, we first describe our key research questions and experimental setup. Then, we present the details of the baseline approaches, i.e., AXPLORER, COR and TEXT, followed by the evaluation metrics used to measure the recommendation performance. Finally, we present the results of our experiments..

8.1 Research questions

We are interested in answering the following key research questions:

- RQ1: How effective are the 6 variants of PERREC?
- RQ2: How much improvement can PERREC achieve over the baseline approaches?
- RQ3: Can PERREC outperform its own components, i.e., SEM and CBR?
- RQ4: Do word embeddings learned from project-specific corpora change the performance of SEM and PERREC?
- RQ5: Do different similarity metrics affect the performance of CBR and PERREC?

8.2 Experimental setup

To answer these research questions, we collected 730 open source Android apps from F-Droid, which is a free and open source Android app repository, and Google Play.

To collect Android apps from F-Droid, we first crawled the list of apps in F-Droid, which contains 2899 Android apps. Since GitHub provides convenient REST APIs and the stargazers on GitHub can help us identify high-quality projects (Munaiah et al. 2017), we only kept those apps which are hosted in GitHub, and obtained 2107 apps. Then, we removed the apps with no more than 50 stargazers on GitHub to filter out potential low-quality apps, and 719 apps remained.

We also make use of GitHub to help us collect apps distributed by Google Play. We first obtained the list of all Java projects in GitHub using GHTorrent (Gousios and Spinellis 2012, 2018). The list contains over 1,600,000 projects. Next, we crawled the readme files of these projects through GitHub's REST API, and parsed each readme file to identify whether it contains Google Play links. We only found 1304 Java projects of which the readme files contain Google Play links. Then, we extracted the Google Play links of each project, and filter out the projects with more than one distinct Google Play links. There were 1217 projects left. We treated the 1217 projects as Android apps, and assumed their distinct Google Play links directed to their Google Play pages.

[1] https://github.com/Tbabm/PerRec.

Finally, we extracted the score and score number of each app from its Google Play page. We only reserved the apps which obtain a score of no less than 3.5 and are scored by at least 10 users, obtaining 334 apps.

We merged the two app lists obtained from F-Droid and Google Play, removed 87 duplicate apps and got 966 apps in total. Given an app, its source code was downloaded by us from GitHub as follows: If the app leveraged GitHub Releases to release its source code, we downloaded its latest release. If the app did not use GitHub releases, but its GitHub repository contained tags, we downloaded its source code at the latest tag. If the app used neither GitHub Releases nor tags, we downloaded its source code in the master branch. In this work, we only consider the Android apps written in Java, hence the non-Java Android apps, such as apps written in Kotlin and C++, were deleted. Since we use srcML (Collard et al. 2003) to parse the source code and extract *API Lists*, the apps of which the source code can not be parsed by srcML were removed. The apps which do not require any system permissions were also filtered out. Ultimately, we collected 730 apps to form our dataset.

For the 730 apps, we construct our ground truth from the permissions that are actually used by them, which are declared in their *AndroidManifest.xml* files. The *API lists* and *Permission Lists* of the 730 apps are extracted using the methods discussed in Sect. 7. On average, the 730 collected apps require 5.37 ± 4.64 (mean ± standard deviation) permissions and use 147.83 ± 99.14 APIs. The average number of required permissions is close to that reported by Wu et al. (2012) (4.67 permissions on average) in their dataset.

We used a tenfold cross-validation procedure to evaluate the effectiveness of a permission recommendation approach. To perform tenfold cross-validation, the 730 apps in our dataset are first shuffled and evenly divided into tenfold, i.e., each fold has 73 apps. Training and testing are performed 10 times (i.e., in 10 runs). For the i_{th} run, the i_{th} fold is regarded as testing set, and other ninefold are combined as training set. A permission recommendation system is built from the training set, and then recommends permissions for apps in the testing set. For each app in the testing set, we calculate several evaluation metrics (which will be described in Sect. 8.4) for it. For each run, the means of these metrics for the testing set are then calculated as the test result of this run. After 10 runs, every fold has been used as testing set, and we can get 10 test results. We compute the mean value for every metric using the 10 test results, and regard these mean values as the final result.

8.3 Baseline approaches

In this study, we use a program-analysis-based approach AXPLORER (Backes et al. 2016) and two mining-based approaches, i.e., COR and TEXT (Bao et al. 2016, 2017) as our baseline approaches. The following paragraphs briefly describe how these baseline approaches work.

Table 4 A tiny permission map to illustrate how Axplorer works

API	Permissions
A_1	P_1, P_3
A_2	P_2, P_3
A_3	P_1

8.3.1 Axplorer

Axplorer (Backes et al. 2016) is an Android application framework analysis tool, which does not directly recommend permissions for apps. However, Axplorer can be used to extract Android permission maps, which are leveraged to make permission recommendations. The permission maps extracted by other program-analysis-based tools, e.g., *Stowaway* (Felt et al. 2011) and *PScout* (Au et al. 2012), can also be utilized in the same way. But *Stowaway* is now out-of-date (Felt et al. 2017), and *PScout* has not updated its permission map for 3 years (Au et al. 2017). Compared to them, the permission maps extracted by Axplorer are still maintained. Moreover, Backes et al. (2016) have shown that Axplorer improves over *PScout* in terms of precision. Therefore, we choose Axplorer as our experimental program-analysis-based baseline.

Since Android apps are usually developed for several or a range of Android OS/API levels, we first combine all the permission maps extracted by Axplorer and published in its official website (Backes et al. 2017) into one. Then, given a test app APP_i, we take all API classes in its *API List* and query their mappings from the permission map to find corresponding permission sets as query results. Finally, we merge these permission sets into one. This permission set, in which permissions are ranked according to the number of times they appear in the query results, is the final recommendation.

For example, with the permission map in Table 4, if APP_i's *API List* is $\{A_1, A_3\}$, we first query the mappings of A_1 and A_3. The query results are two permission sets. Then we merge the two sets into one and rank these permissions according to their frequencies of occurrence.

8.3.2 Cor

Cor recommends permissions for APP_t based on the *Permission Lists* of the apps which are similar to APP_t. In this approach, similarity between two apps are measured by their jointly used APIs; more specifically, by the *Pearson correlation* similarity between their *API vectors* (defined in Eq. 4). Based on the similarities between APP_t and each of the training apps, Cor can find the k nearest neighbors ($kNNs$) of APP_t from the training apps. The permission recommendation is then made according to the permissions required by APP_t's $kNNs$. To align with Bao et al. (2016, 2017)'s approach, we set Cor to use 10 nearest neighbors to make recommendations.

8.3.3 Text

Text builds a text mining model based on multinomial Naive Bayes to recommend permissions. This model is constructed from apps' readme files, which are collected from GitHub. All the readme files are first preprocessed and represented in the form of "bags-of-words". In bag-of-words model, a readme file is represented as a multiset of its words, disregarding grammar and word order but keeping multiplicity. Each processed word becomes a feature. After using a feature selection method, Text forms a classifier for each permission based on selected text features.

In the recommendation phase, Text first extracts the text features of APP_t, and then the probability that APP_t requires permission P_i is calculated by the i_{th} classifier in the text mining model. Finally, permission recommendations are made based on these probabilities. According to Bao et al. (2017), Text selects 1000 term features for each multinomial Naive Bayes classifier.

8.4 Evaluation metrics

In order to evaluate the effectiveness of these approaches, we use Mean Average Precision (MAP) and two metrics we refer to as Necessary Recall (NR) and Total-Recall Ratio (TRR).

8.4.1 Mean average precision (MAP)

MAP provides a single-figure measure of recommendation quality (Christopher et al. 2008), and it has been shown to have good discrimination and stability to evaluate ranking techniques (Baeza-Yates et al. 1999). An app usually requires more than one permissions. MAP considers all correct results, hence we use it to measure the average performance of PerRec to recommend all the required permissions.

Given a single query (in our case: an app APP_t), its average precision is defined as the mean of the precision values obtained for different sets of top-k documents (in our case: permissions) that were retrieved before every relevant document is retrieved, which is computed as:

$$AvgP(APP_t) = \frac{\sum_{k=1}^{M} P(k) \times Rel(k)}{|APs|}$$

where M is the number of candidate permissions in a ranked list, $Rel(k)$ indicates whether the permission at position k is actually required by APP_t or not, and APs refers to the permissions which are actually needed by APP_t. $P(k)$ is the precision at the given cut-off position k, and is computed as:

$$P(k) = \frac{|APs\ in\ top\text{-}k|}{k}$$

where $top\text{-}k$ is the set of top-k permissions returned by an approach. The MAP for a set of apps S is then the mean of the average precision scores for all apps in S:

$$MAP(S) = \frac{\sum_{APP_t \in S} AvgP(APP_t)}{|S|}$$

8.4.2 Necessary recall (NR)

A metric, we refer to as Necessary Recall (NR), is used to measure the recall of our approach. Given a test app APP_t, NR is the percentage of actually required permissions out of the first n permissions returned by a recommender system for APP_t, where n is the number of the necessary permissions of APP_t. NR is defined as:

$$NR(APP_t) = \frac{|APs\ in\ top\text{-}n|}{n}$$

The APs are APP_t's necessary permissions, and $top\text{-}n$ is the set of top-n permissions returned by an approach. For a set of apps, NR is the mean of the NR values for all apps in it.

8.4.3 Total-recall ratio (TRR)

To measure the effort our approach requires to achieve total recall, i.e., to recommend all the correct permissions for an app, we propose an evaluation metric Total-Recall Ratio (TRR). Given a training set and a test app APP_t, an approach can recommend a list of permissions RPs for APP_t, and APP_t's TRR is then calculated as follows:

$$TRR(APP_t) = \begin{cases} \frac{n_{min}}{|APs|}, & \text{if } APs - RPs = \emptyset; \\ \frac{n_{all}}{|APs|}, & \text{otherwise} \end{cases} \tag{15}$$

In a nutshell, TRR measures that if we want to recall all the correct permissions of APP_t, how many permissions on average will be recommended by an approach for one correct permission. More specifically, if the approach can achieve total recall for APP_t, we simply compute the ratio of the minimal number of recommended permissions to achieve total recall, i.e., n_{min}, and the number of APP_t's correct permissions, i.e., $|APs|$, as APP_t's TRR. Otherwise, we penalize TRR by replacing n_{min} with the total number of permissions captured by the training set, i.e., n_{all}. The closer TRR is to one, the better it is. For a set of apps, TRR is the mean of the TRR values for all apps in it.

For example, using the examples shown in Table 5, the TRR value of APP_1 is $\frac{3}{2} = 1.5$ since we can achieve total recall using the first three permission in APP_1's RPs.

Table 5 Examples to illustrate how TRR is calculated

APP	APs	RPs
APP_1	P_1, P_2	P_3, P_2, P_1, P_4
APP_2	P_3, P_4	P_2, P_3, P_1

Assuming the training set captured 50 distinct permissions in total, i.e., $n_{all} = 50$, the TRR value of APP_2 is $\frac{50}{2} = 25$ since APP_2's RPs do not contain all the correct permissions.

8.5 Results

RQ1: How effective are the 6 variants of PERREC?

Motivation We want to investigate the effectiveness of the 6 variants of PERREC, i.e., PERREC^{Max}, PERREC^{Min}, PERREC^{SUM}, PERREC^{ANZ}, PERREC^{MNZ}, PERREC^{BC}, and find the variant with best performance.

Approach We evaluate the performance of each variant by computing MAP, TRR and NR for it on our dataset using tenfold cross validation. Then, we select the best approach according to all the three metrics. We also conducted a Wilcoxon signed-rank test at the confidence level of 95% to check whether the performance differences between the selected best variant and the other variants are significant. For each variant, we collected 10 effectiveness values in terms of each metric. The Wilcoxon signed-rank tests are conducted on the 10 pairs of effectiveness values. Moreover, to analyze the magnitude of the observed differences, we used an effect size measure named Cliff's Delta (δ). Following the guidelines in Grissom and Kim (2005), the Cliff's Delta values are interpreted according to Table 6.

Results Table 7 presents the means of MAP, TRR and NR for each PERREC's variant, and we also plot the MAP, TRR and NR values of the tenfold using boxplots, as shown in Fig. 4. We can see that in terms of MAP, the performance of PERREC^{SUM},

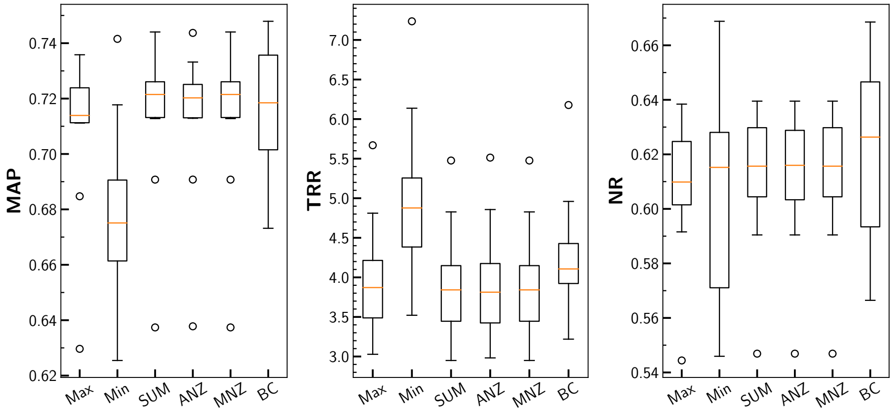
Table 6 The effectiveness level of Cliff's delta

Cliff's delta ($\|\delta\|$)	Effectiveness level
$\|\delta\| < 0.147$	Negligible
$0.147 \leq \|\delta\| < 0.33$	Small
$0.33 \leq \|\delta\| < 0.474$	Medium
$0.474 \leq \|\delta\|$	Large

Table 7 Means of MAP, TRR and NR for PERREC's variants

Metrics	Max	Min	SUM	ANZ	MNZ	BC
MAP	0.707	0.680	0.713	0.712	0.713	0.717
Rank	5	6	2	4	2	1
TRR	3.99	5.01	3.93	3.94	3.93	4.26
Rank	4	6	1	3	1	5
NR	0.607	0.607	0.611	0.611	0.611	0.620
Rank	5	5	2	2	2	1
Rank sum	14	17	5	9	5	7

Max, Min, SUM, ANZ, MNZ and BC refers to PERREC^{Max}, PERREC^{Min}, PERREC^{SUM}, PERREC^{ANZ}, PERREC^{MNZ} and PERREC^{BC} respectively



(a) MAP **(b)** TRR **(c)** NR

Fig. 4 MAP, TRR and NR for PerRec's variants

Table 8 *p* value and Cliff's delta (δ) for PerRec^{SUM} compared with the other variants

Metrics	S versus Max	S versus Min	S versus ANZ	S versus MNZ	S versus BC
	p value (δ)	*p* value (δ)	*p* value (δ)	*p* value (δ)	*p* value (δ)
MAP	0.001 (0.22)	0.010 (0.58)	0.138 (0.02)	1.000 (0.00)	0.278 (−0.10)
TRR	0.002 (−0.10)	0.001 (−0.64)	0.246 (0.00)	1.000 (0.00)	0.002 (−0.28)
NR	0.007 (0.18)	0.461 (0.10)	0.295 (0.01)	1.000 (0.00)	0.161 (−0.16)

S refers to PerRec^{SUM}, and Max, Min, ANZ, MNZ and BC refers to PerRec^{Max}, PerRec^{Min}, PerRec^{ANZ}, PerRec^{MNZ} and PerRec^{BC} respectively

PerRec^{ANZ}, PerRec^{MNZ} and PerRec^{BC} is close, and is better than that of the other two variants. On TRR, PerRec^{Max}, PerRec^{SUM}, PerRec^{ANZ} and PerRec^{MNZ} perform similarly, and outperform PerRec^{Min} and PerRec^{BC}. As for NR, PerRec^{BC} outperforms other variants, but its performance is less stable than that of PerRec^{SUM}, PerRec^{ANZ} and PerRec^{MNZ}. Since we need to select the best variant according to all the metrics, for each variant, we calculate its rank according to the means of the three metrics, and add the three ranks together as *Rank Sum*. We can learn from Table 7 that PerRec^{SUM} and PerRec^{MNZ} have the highest *Rank Sum*, and their performance on all the metrics is the same. Therefore, each of them can be chosen as the best variants. In this work, we choose PerRec^{SUM} to answer the subsequent research questions.

We then compute the *p* values of the Wilcoxon signed-rank tests and the Cliff's Delta (δ) for PerRec^{SUM} compared with the other variants, as shown in Table 8. We consider the performance difference between PerRec^{SUM} and another variant on a metric to be statistically significant at the confidence level of 95% if the corresponding *p* value is less than 0.05. We can see from Table 8 that in terms of MAP, the performance of PerRec^{SUM} is significantly better than PerRec^{Max} and PerRec^{Min}, while PerRec^{BC} does not statistically significantly improve PerRec^{SUM}. On TRR, PerRec^{SUM} achieves significantly better performance than PerRec^{Max}, PerRec^{Min} and

PerRec^{BC}. As for NR, PerRec^{SUM} only significantly improves PerRec^{Max}. We can also know that the performance difference between PerRec^{SUM} and PerRec^{ANZ} in terms of all the metrics is not significant.

In summary, PerRec^{SUM} and PerRec^{MNZ} perform equally, and both can be chosen as the best variant. The variants which consider two components outperform those which only use the scores from one component to rank, i.e., PerRec^{Max} and PerRec^{Min}. Such finding highlights the benefit of combining the two components. In the following research questions we use PerRec^{SUM} to conduct experiments, and PerRec is used to represent PerRec^{SUM} for brevity.

RQ2: How much improvement can PerRec achieve over the baseline approaches?

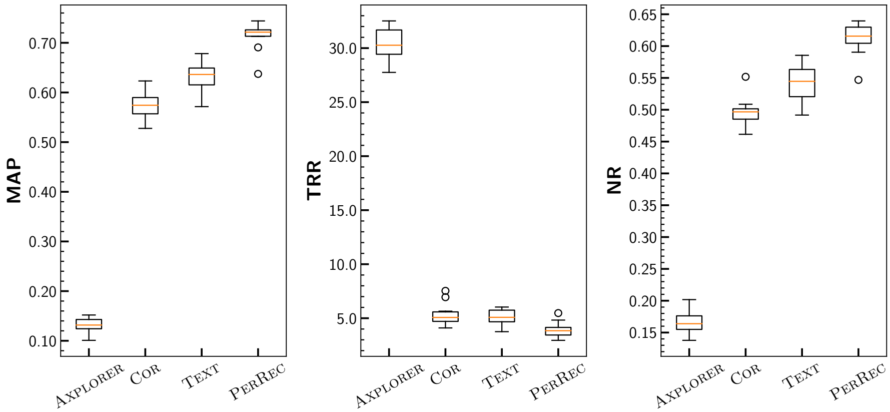
Motivation We want to investigate whether PerRec can provide more accurate permission recommendations than the three baseline approaches (Axplorer, Cor, Text).

Approach We compare PerRec with the three baselines in terms of MAP, TRR and NR. The Wilcoxon signed-rank test and the Cliff's Delta (δ) are also conducted as described in RQ1.

Results Table 9 presents the means of MAP, TRR and NR for Axplorer, Text, Cor and PerRec, and the MAP, TRR and NR values of the tenfold are plotted in Fig. 5.

Table 9 Means of MAP, TRR and NR for PerRec and the baselines

Metrics	Axplorer	Cor	Text	PerRec
MAP	0.132	0.574	0.632	0.713
Improved	440.2%	24.2%	12.8%	–
TRR	30.41	5.34	5.10	3.93
Ratio	0.13	0.74	0.77	–
NR	0.168	0.496	0.541	0.611
Improved	263.7%	23.2%	12.9%	–



(a) MAP **(b)** TRR **(c)** NR

Fig. 5 MAP, TRR and NR for PerRec and the baselines

Table 10 *p* value and Cliff's delta (δ) for PerRec compared with the baselines

Metrics	PerRec versus Axplorer	PerRec versus COR	PerRec versus TEXT
	p value (δ)	*p* value (δ)	*p* value (δ)
MAP	0.001 (1.00)	0.001 (1.00)	0.001 (0.90)
TRR	0.001 (− 1.00)	0.001 (− 0.76)	0.001 (− 0.70)
NR	0.001 (1.00)	0.001 (0.98)	0.001 (0.92)

Table 11 Means of MAP, TRR and NR for Sem, Cbr, and PerRec

Metrics	Sem	Cbr	PerRec
MAP	0.701	0.679	0.713
Improved	1.7%	5.0%	–
TRR	5.01	4.03	3.93
Ratio	0.78	0.98	–
NR	0.606	0.600	0.611
Improved	0.8%	1.8%	–

We can see that in terms of the three metrics, PerRec performs better than all the baseline approaches. On MAP, PerRec outperforms Axplorer, Text and Cor by 12.8% to 440.2%. The ratios between the TRR value of PerRec and the TRR values of Axplorer, Text and Cor are all less than 0.8, which means to recall all the necessary permissions for an app, PerRec only requires less than 80% of the efforts that are needed by the baseline approaches. PerRec also improves over the three baselines in terms of NR by 12.9% to 263.7%.

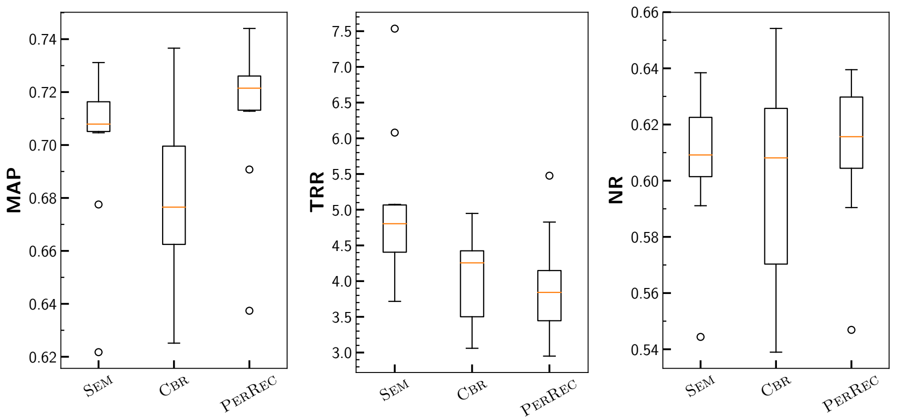
Table 10 shows the *p* values and the Cliff's Delta values when we compare PerRec with the baseline approaches in terms of MAP, TRR and NR. We can see that all the *p* values are less than 0.05, and the absolute values of Cliff's Delta ($|\delta|$) are all greater than 0.474. These values means that the improvements achieved by PerRec over the baselines are statistically significant and substantial.

These results demonstrate that PerRec outperforms the baseline approaches i.e., Axplorer, Text and Cor. The key differences between PerRec and the baselines is that PerRec takes relationships between APIs into consideration and leverage different components to capture such relationships. Therefore, we think the good performance of PerRec indicates that capturing relationships between APIs matters for permission recommendation tasks.

RQ3: Can PerRec outperform its own components, i.e., Sem and Cbr?

Motivation Sem and Cbr are important components of PerRec. We want to investigate whether the composition of the two components of PerRec is actually beneficial or not.

Approach Following the process used to answer RQ2, we evaluate the effectiveness of Sem, Cbr and PerRec based on their MAP, TRR and NR. The Wilcoxon signed-rank test and the Cliff's Delta (δ) are also used.



(a) MAP **(b)** TRR **(c)** NR

Fig. 6 MAP, TRR and NR for PerRec and its components

Table 12 *p* value and Cliff's delta (δ) for PerRec compared with its components

Metrics	PerRec versus Sem	PerRec versus Cbr
	p value (δ)	*p* value (δ)
MAP	0.001 (0.46)	0.014 (0.62)
TRR	0.001 (−0.64)	0.188 (−0.14)
NR	0.002 (0.22)	0.188 (0.16)

Results Table 11 shows the means of MAP, TRR and NR for PerRec and its two components. The MAP, TRR and NR values of the tenfold are plotted in Fig. 6, and Table 12 presents the *p* values and the Cliff's Delta (δ) when we compare PerRec with Sem and Cbr in terms of the three metrics. We can see that Sem performs better than Cbr in terms of MAP and NR, while Cbr outperforms Sem on TRR. By combining them together, PerRec improves Sem on all the three metrics. The improvements are statistically significant since the three corresponding *p* values are less than 0.05. PerRec also shows significant improvement over Cbr in terms of MAP with large effect size, and achieves similar performance to Cbr on TRR and NR.

The reason of Sem's better performance compared to Cbr in terms of MAP and NR may be that Sem recommends permissions for a new app by learning from similar apps, and tends to recommend the permissions that are frequently used by these similar apps. It is often the case that such permissions are also required by the new app, hence Sem can be accurate when recommending the top-k permissions, which leads to better MAP and NR. Cbr constructs permission profiles from all the apps in the training set instead of only considering similar apps. It is more accurate to predict whether a permission that is not frequently used by similar apps is required by a new app or not. This may make Cbr achieve total recall more easily, and hence obtain better TRR. The reason that PerRec outperforms Sem and Cbr on the three metrics may be that the two components complement each other in PerRec.

Table 13 Means of MAP, TRR and NR for SEM$_{wiki}$, SEM$_{docs}$, PERREC$_{wiki}$ and PERREC$_{docs}$

Metrics	SEM$_{wiki}$	SEM$_{docs}$	PERREC$_{wiki}$	PERREC$_{docs}$
MAP	0.701	0.699	0.713	0.712
Improved	–	0.3%	–	0.1%
TRR	5.01	5.06	3.93	3.95
Ratio	–	0.99	–	0.99
NR	0.606	0.604	0.611	0.610
Improved	–	0.3%	–	0.2%

In summary, combining SEM and CBR together can significantly improve the MAP of CBR without reducing its TRR and NR, and significantly improve the SEM on all the metrics with at least small effect size. Therefore, the composition of the two components in PERREC is beneficial.

RQ4: Do word embeddings learned from project-specific corpora change the performance of SEM and PERREC?

Motivation For SEM, we use the pre-trained word embeddings which are learned from Wikipedia 2014 and Gigaword 5. Recently, some researchers (Ye et al. 2016; Xu et al. 2016) learn word embeddings used in Software Engineering (SE) tasks from project-specific corpora, and show that the project-specific word embeddings perform well even if the size of the corpus is limited. Therefore, we want to determine if there exists a performance difference between the variants of SEM which use different sets of word embeddings, and whether these variants of SEM impact the performance of PERREC.

Approach We refer to the pre-trained word embeddings as *wiki embeddings*, and the project-specific embeddings as *docs embeddings*. In order to learn *docs embeddings*, we first create a project-specific corpus from Android docs and JDK 7 docs. Android docs are contained in Android SDK, and JDK 7 docs can be downloaded from its official website (Oracle 2017a). Since SEM does not consider code snippets, we also delete all the code snippets, which are usually placed between special HTML tags such as $<$ pre $>$ and $<$ code $>$ from these docs. The GloVe model is used to train *docs embeddings*, and the parameters we use are nearly the same with those used to train the *wiki embeddings*, except that we do not specify the maximum size of the vocabulary (max-vocab). When training *wiki embeddings*, the maximum size of the vocabulary is set to 400,000. However, our project-specific corpus, which only contains about 23 million tokens, is much smaller than Wikipedia 2014 and Gigaword 5 – in which there are more than 6 billion tokens. Hence there is no need for us to specify max-vocab.

After we get the *docs embeddings*, we use them in SEM. The variant of SEM based on *docs embeddings* is referred to as SEM$_{docs}$, and the old approach is named SEM$_{wiki}$. The variants of PERREC which use different variants of SEM are referred to as PERREC$_{docs}$ and PERREC$_{wiki}$. We first evaluate the MAP, TRR and NR of the two variants of SEM, and then evaluate the performance of PERREC$_{docs}$ and PERREC$_{wiki}$ on the same metrics.

Results We can see from Table 13 that the performance differences between SEM$_{wiki}$ and SEM$_{docs}$ on the three metrics are very small. The performance of PERREC$_{wiki}$ and PERREC$_{docs}$ is also nearly the same in terms of the used metrics. These results show that project-specific word embeddings do not improve the effectiveness of SEM. Making use of different variants of SEM in PERREC also does not result in substantial difference in performance either.

RQ5: Do different similarity metrics affect the performance of CBR and PERREC?

Motivation We want to investigate whether it is possible to improve the performance of CBR and PERREC by simply changing the similarity measure used in CBR.

Approach In addition to *cosine similarity*, we investigate *Euclidean similarity* and *Pearson correlation similarity* in CBR, and the three variants of CBR are named CBR$_{cos}$, CBR$_{euc}$ and CBR$_{cor}$ respectively. The corresponding variants of PERREC are referred to as PERREC$_{cos}$, PERREC$_{euc}$ and PERREC$_{cor}$. *Cosine similarity* is defined in Equation 3. The *Pearson correlation* between two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ is:

$$Cor(\boldsymbol{x}, \boldsymbol{y}) = \frac{COV(\boldsymbol{x}, \boldsymbol{y})}{\sigma_{\boldsymbol{x}}\sigma_{\boldsymbol{y}}}$$

where $COV(\boldsymbol{x}, \boldsymbol{y})$ is the covariance between $\boldsymbol{x}$ and $\boldsymbol{y}$, and $\sigma_{\boldsymbol{x}}$ and $\sigma_{\boldsymbol{y}}$ are the standard deviation of $\boldsymbol{x}$ and $\boldsymbol{y}$. Then, *Pearson correlation similarity* is calculated as:

$$sim_{cor}(\boldsymbol{x}, \boldsymbol{y}) = \frac{Cor(\boldsymbol{x}, \boldsymbol{y}) + 1}{2}$$

Euclidean similarity is defined as follows:

$$sim_{euc}(\boldsymbol{x}, \boldsymbol{y}) = 1/\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{16}$$

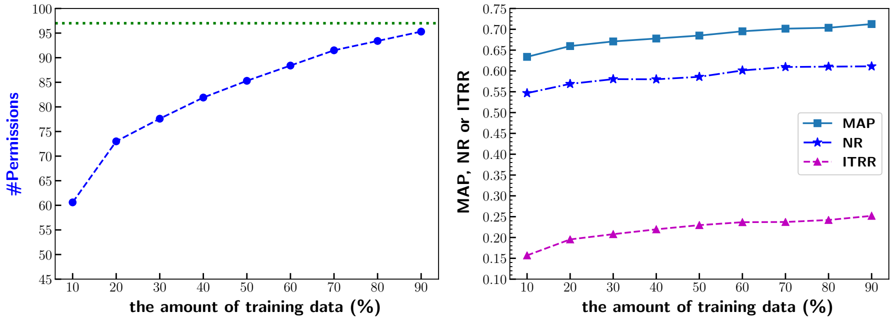
where n is the size of the vector.

We evaluate the performance of these variants of CBR and PERREC by computing their MAP, TRR and NR values.

Result Table 14 presents the means of MAP, TRR and NR for the variants of CBR and the corresponding variants of PERREC. We can see that CBR$_{cor}$ performs better than the other two variants of CBR on the three metrics. But the performance differences are very small. As for PERREC, PERREC$_{cos}$ only slightly outperforms PERREC$_{euc}$ and PERREC$_{cor}$. There results show that using different similarity metrics in CBR does not substantially affect the performance of CBR and PERREC.

Table 14 Means of MAP, TRR and NR for CBR$_{cos}$, CBR$_{euc}$, CBR$_{cor}$, PerRec$_{cos}$, PerRec$_{euc}$ and PerRec$_{cor}$

Metrics	CBR$_{cos}$	CBR$_{euc}$	CBR$_{cor}$	PerRec$_{cos}$	PerRec$_{euc}$	PerRec$_{cor}$
MAP	0.679	0.679	0.688	0.713	0.708	0.711
Improved	1.3%	1.3%	–	–	0.7%	0.3%
TRR	4.03	4.03	3.94	3.93	3.98	3.94
Ratio	0.98	0.98	–	–	0.99	1.00
NR	0.600	0.600	0.609	0.611	0.608	0.609
Improved	1.5%	1.5%	–	–	0.5%	0.3%



(a) Average numbers of captured permissions

(b) MAP, NR and ITRR

Fig. 7 Evaluating PerRec with varying amount of training data

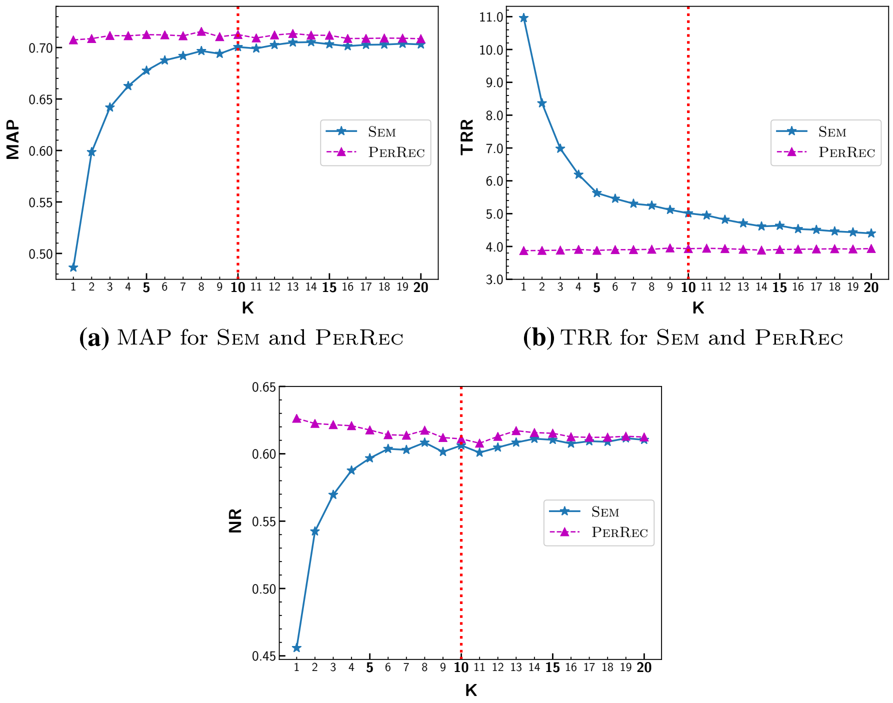
9 Discussion

9.1 Sensitivity study

Our dataset captures 97 Android system permissions, while the total number of system permissions in Android 7.1 (Android 2017c) is 151. To explore the saturating speed of our dataset with respect to the captured permission number, we conduct a sensitivity study with varying amount of training data.

We first shuffled and evenly divided our dataset into tenfold. Then we performed 10 experiments. In the i_{th} experiment, we used the i_{th} fold as the test set, shuffled the remaining ninefold and selected the first consecutive 1, 2,..., 9 folds to construct 9 training sets which contain different amount of data. 9 models were trained using the 9 training sets, and were evaluated on the same test set in terms of MAP, NR and TRR. After finishing the 10 experiments, for each amount of training data (e.g., 10% data), we calculated the average number of distinct permissions captured by the training sets and the means of MAP, NR and TRR for PerRec.

Figure 7a presents the average numbers of distinct permissions captured by different amount of training data. We can see that the more training apps we use, the more permissions the training set can capture. We also plot our evaluative results in Fig. 7b.



(a) MAP for SEM and PERREC **(b)** TRR for SEM and PERREC

(c) NR for SEM and PERREC

Fig. 8 MAP, TRR and NR for SEM and PERREC when using K nearest neighbors in SEM

Different from MAP and NR, the lower TRR is, the better the performance is. To better visualize our results, we plot inverse Total-Recall Ratio (ITRR, i.e., $1/TRR$, the greater the better) instead of TRR. We can see that MAP, NR and ITRR increases as the amount of training data increases. But compared to the growth rate of captured permission numbers, the ascending speed of MAP, NT and ITRR is slow. In addition, we conduct these experiments by randomly selecting data as training set. In practice, we can pick a small amount (e.g., 20%) of data which is the most diverse as the training set. Such strategy may help our approach achieve the same performance as using 90% of data.

9.2 The number of nearest neighbors in SEM

The SEM component select k nearest neighbors to make recommendations. In our evaluative experiments, we set k to 10 by default. To explore how the number of nearest neighbors in SEM affects the recommendation performance, we evaluate the MAP, TRR and NR for SEM and PERREC with different ks. The results are shown in Fig. 8. We can see from Fig. 8a that when k is less than 10, the greater k is, the better performance SEM can achieve in terms of MAP. When k is greater than or equal to 10, the MAP value of SEM becomes stable. Different from MAP, the TRR value of

SEM increases as k increases, but the increasing rate becomes very small after k is greater than 14. In terms of NR, the performance of SEM increases with k when k is less than 7, fluctuates when k is between 7 and 16, and then becomes stable. As for PERREC, its performance is stable on MAP and TRR with k increasing. On NR, although PERREC's performance fluctuates when k is less than 16, the performance differences are small, and the NR values become stable when k is greater than 16. These results show that when k is small, the more nearest neighbors we consider in SEM, the better SEM's performance is. But with k increasing, SEM's performance will become stable. Moreover, by combining two components together, PERREC is not sensitive to the number of nearest neighbors in SEM.

9.3 The customization of PERREC

Although the ranked permission list returned by PERREC usually contains a number of permissions, in practice, our tool can be easily adapted to only recommend a limited number of permissions for users to review according to some heuristic rules and the accuracy of PERREC. For example, when recommending potentially missing permissions for an app, if there are no permissions declared by this app, the front end of our tool can only display the top 6 permissions returned by PERREC for users to review. If the app already declares n permissions, our tool can only display the permissions which are in the top 6 or the top $0.7 \times n$ returned permissions but are not declared yet. We limited the permission number to 6 since in our dataset apps require 5.37 permissions on average, and to $0.7 \times n$ due to the MAP value of PERREC on our dataset. When detecting extraneous permissions for an app, if the app has declared n permissions in the manifest file, the front end of our tool can only display the permissions which have been declared by the app but are not in the set of the top $4 \times n$ permissions returned by PERREC. We set such number to $4 \times n$ since the TRR value of PERREC on our dataset is 3.93. These strategies can make our approach more user-friendly.

9.4 False positive results of PERREC

As a mining-based approach, a shortcoming of PERREC is that it may recommend false positive permissions, i.e., predict unnecessary permissions to be required. We do not claim that PERREC can automatically identify a perfect set of permissions for an app. Instead, PERREC aims to assist developers, especially new developers, in making quick and accurate decisions about what permissions their apps should declare. As mentioned in Sects. 2.2 and 9.3, our approach can achieve this goal by recommending a limited number of permissions for developers to review during and after their development. After reviewing the recommended permissions of our approach, developers may, for example, find necessary but undeclared permissions or extraneous permissions before releasing. Therefore, we argue that although there are false positive results, PERREC is still useful for helping developers.

9.5 Where PerRec performs badly

By investigating our test results, we find two types of apps where PerRec performs badly. First of all, PerRec can not recommend permissions which do not appear in the training set. If an app requires one or more such permissions, PerRec can not achieve a good performance on it. For example, there is an app named *RemoteDroid* (Khan 2018) in our dataset which can stream an Android device's display to another Android device. To provide users with this function, *RemoteDroid* declares 4 permissions, but two of them, i.e., *CAPTURE_VIDEO_OUTPUT* and *CAPTURE_SECURE_VIDEO_OUTPUT*, are not required by any app in the training set. Therefore, our approach fails to recommend the two permissions for *RemoteDroid* and performs badly on it.

In addition, we find PerRec also performs badly on apps requiring *unpopular permissions*. Given an app, the *unpopular permissions* refer to the permissions which are seldom required by both training apps and the app's nearest neighbors. For example, an app named *Applications Info* (Majeur 2018) can monitor available information of all installed apps, and requires only one permission *GET_PACKAGE_SIZE*. However, such permission is only required by 2 out of 657 training apps, and none of the nearest neighbors found by Sem declare it. Hence *GET_PACKAGE_SIZE* is an *unpopular permission*. The rank of *GET_PACKAGE_SIZE* recommended by Sem is 94 with 96 permissions used by apps in the training set. Since Cbr recommends permissions through permission profiles learned from all training apps rather than only similar apps, it performs better than Sem in this case and recommends *GET_PACKAGE_SIZE* as the 47th permission. By combining Sem and Cbr, PerRec recommends *GET_PACKAGE_SIZE* as the 48th permission, and does not perform well on *Applications Info* either.

10 Threats to validity

In this section, we discuss several threats that may affect the validity of our experiment results.

– *Mistakes in our code and experiment bias* Although we have checked our code many times, there may still exist some analytical errors of which we were not aware.
– *Properness of the measures* This study makes use of the MAP, TRR and NR to evaluate the performance of PerRec. MAP is widely used in many studies (Karim et al. 2016; Bao et al. 2016, 2017; Xia et al. 2014). TRR and NR are proposed by us according to the properties of this Android permission recommendation task.
– *Incorrect permission declaration* Android apps may declare permissions incorrectly. To mitigate this threat, we build our dataset by only using the Android apps which either get more than 50 stargazers on GitHub or obtain an average score of no less than 3.5 and are scored by no less than 10 users in Google Play. We believe that most of the apps in our dataset are of high quality and declare permissions correctly.

- *Permissions coverage* First, PerRec can not recommend permissions which are not captured by the training set. Second, our approach can not handle dynamic permissions. Besides, our evaluation makes use of 97 out of 151 Android system permissions without considering customized permissions. Therefore, the experimental results may not reflect all the system permissions.
- *Generalization* Our dataset contains a limited number of Android apps and there may exist bias in our dataset. Thus the relationships between Android APIs and permissions learned by our approach from our dataset may not generalize to some new apps. To mitigate this threat, we carefully collect 730 high-quality Android apps from Google Play and F-Droid to build our dataset. Nonetheless, additional replication studies on larger high-quality datasets may prove fruitful.

11 Related work

To the best of our knowledge, the most related works to our study are *ApMiner* proposed by Karim et al. (2016), and $APRec^{CF}$ as well as $APRec^{TEXT}$ proposed by Bao et al. (2016, 2017). These approaches leverage data mining techniques to recommend permissions for Android apps. As explained in the motivation (see Sect. 2), *ApMiner* combines static analysis and association rule discovery to make app permission recommendations. $APRec^{CF}$ and $APRec^{TEXT}$ are based on collaborative filtering and text mining techniques respectively. $APRec^{CF}$ has three variants, and $APRec^{CF_{correlation}}$, which is used as one of the baselines in this work, is the one with best performance. PerRec is also based on data mining techniques. However, PerRec is able to capture the relationships between APIs to make more accurate recommendations than these approaches.

Many authors have proposed diverse approaches to build mappings between APIs and Android app permissions or relate an app's resources to permissions. Felt et al. (2011) proposed a tool named *Stowaway*, which builds an Android permission map by using automated testing tools on the Android API. Since *Stowaway* extracts the permission map for only one version of Android and is now out-of-date, Au et al. (2012) proposed *PScout*, a tool which performs static reachability analysis to recover control dependencies between API calls and permission checks and is able to produce a permission map for each Android version. *Androguard* (Desnos and Gueguen 2011) is a reverse engineering tool which exploits the API to permission mappings extracted by *PScout* for permission checking. Nevertheless, Karim et al. (2016) have shown that *ApMiner* performs better than *PScout* and *Androguard* in terms of app permission recommendations. Backes et al. (2016) built an Android application framework analysis tool, namely Axplorer, which can be used to create a permission map of Android, and improves over *PScout* in terms of precision.

Our approach is a mining-based approach, which is less sensitive to Android evolution and does not require manual inspection. In order to compare our approach with the state-of-the-art program-analysis-based approach, we use Axplorer as one of the baselines. In addition, Vidas et al. (2011) created one-to-many permission-API mappings by manually parsing the API documentation. However, the usability of their work is limited by incomplete Android documentation. Qu et al. (2014) presented a system named AutoCog which applies natural language processing (NLP) techniques to build links between app descriptions in Android market and permissions. Different from Qu *et al.*'s work, our work only leverages API and API descriptions to recommend permissions. Moreover, in practice, not every app provides a comprehensive app description.

Compared to program-analysis-based approaches, PerRec by using data mining techniques can capture more contextual information, which contributes much to accurate permission recommendations. Compared to other mining-based approaches, the technical novelty of our approach is reflected in the following aspects: First, PerRec only takes as input APIs that are called by apps and API descriptions that can be collected from Android documentation easily. Second, our approach can capture similarities between APIs, which help to improve the accuracy of app permission recommendations. In order to capture these similarities, Sem utilizes API descriptions and semantic similarities between APIs to select nearest neighbors. Given a permission, its profile constructed by Cbr contains the information about which APIs may require this permission. Hence similarities between APIs are also captured by permission profiles in Cbr. Finally, PerRec is a composite framework, which combines two data mining techniques using data fusion methods to achieve high performance.

12 Conclusion

In this study, we propose a new permission recommendation system named PerRec, which contains two recommender components, i.e., a semantic-based similarity analyzer (Sem) and a content-based recommendation approach (Cbr). By leveraging data fusion methods, PerRec combines the two sets of recommendation scores outputted by Sem and Cbr to make better Android permission recommendations. According to our evaluation on 730 Android apps collected from F-Droid and Google Play, PerRec improves the effectiveness of its two components, i.e., Sem and Cbr. Moreover, PerRec performs better than the state-of-art approaches proposed by Bao et al. (2016, 2017) and Backes et al. (2016) significantly and substantially in terms of MAP, TRR and NR.

References

Android: Android API reference. https://developer.android.com/reference/classes.html (2017a). Accessed 10 Aug 2017

Android: Android security tips. https://developer.android.com/training/articles/security-tips.html (2017b). Accessed 10 Aug 2017

Android: Android security tips. https://developer.android.com/reference/android/Manifest.permission.html (2017c). Accessed 10 Aug 2017

Android: The download page of android SDK. https://developer.android.com/sdk/download.html (2017d). Accessed 10 Aug 2017

Aslam, J.A., Montague, M.: Models for metasearch. In: Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 276–284. ACM (2001)

Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: Pscout: analyzing the android permission specification. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 217–228. ACM (2012)

Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: The website of PScout. https://github.com/zyrikby/PScout (2017). Accessed 10 Aug 2017

Backes, M., Bugiel, S., Derr, E., McDaniel, P.D., Octeau, D., Weisgerber, S.: On demystifying the android application framework: re-visiting android permission specification analysis. In: USENIX Security Symposium, pp. 1101–1118 (2016)

Backes, M., Bugiel, S., Derr, E., McDaniel, P.D., Octeau, D., Weisgerber, S.: The website of axplorer. http://www.axplorer.org/ (2017)

Baeza-Yates, R., Ribeiro-Neto, B., et al.: Modern Information Retrieval, vol. 463. ACM Press, New York (1999)

Bao, L., Lo, D., Xia, X., Li, S.: What permissions should this android app request? In: International Conference on Software Analysis, Testing and Evolution (SATE), pp. 36–41 . IEEE (2016)

Bao, L., Lo, D., Xia, X., Li, S.: Automated android application permission recommendation. Sci. China Inf. Sci. **60**(9), 092,110 (2017). https://doi.org/10.1007/s11432-016-9072-3

Bartel, A., Klein, J., Le Traon, Y., Monperrus, M.: Automatically securing permission-based software by reducing the attack surface: an application to android. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 274–277. ACM (2012)

Christopher, D.M., Prabhakar, R., Hinrich, S.: Introduction to information retrieval. Introd. Inf. Retr. **151**(177), 5 (2008)

Collard, M.L., Kagdi, H.H., Maletic, J.I.: An XML-based lightweight C++ fact extractor. In: 11th IEEE International Workshop on Program Comprehension, pp. 134–143 . IEEE (2003)

Desnos, A., Gueguen, G.: Android: from reversing to decompilation. In: Proceedings of Black Hat Abu Dhabi, pp. 77–101 (2011)

F-Droid: F-Droid. https://f-droid.org (2017). Accessed 10 Aug 2017

Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 627–638. ACM (2011)

Felt, A.P., Ha, E., Egelman, S., Haney, A., Chin, E., Wagner, D.: Android permissions: user attention, comprehension, and behavior. In: Proceedings of the Eighth Symposium on Usable Privacy and Security, p. 3. ACM (2012)

Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: The website of stowaway. http://android-permissions.org/ (2017). Accessed 10 Aug 2017

Fox, E.A., Shaw, J.A.: Combination of Multiple Searches, vol. 243. NIST Special Publication SP, Gaithersburg (1994)

Fox, E.A., Koushik, M.P., Shaw, J., Modlin, R., Rao, D., et al.: Combining evidence from multiple searches. In: The First Text Retrieval Conference (TREC-1), vol. 500, p. 319. US Department of Commerce, National Institute of Standards and Technology (1993)

GitHub: The repository of MP3 voice recorder in github. https://github.com/yhirano/Mp3VoiceRecorderSampleForAndroid (2017). Accessed 10 Aug 2017

Gousios, G., Spinellis, D.: Ghtorrent: Github's data from a firehose. In: 9th IEEE Working Conference on Mining Software Repositories (MSR), pp. 12–21. IEEE (2012)

Gousios, G., Spinellis, D.: The website of ghtorrent. http://ghtorrent.org/ (2018). Accessed 10 Aug 2017

Grissom, R.J., Kim, J.J.: Effect Sizes for Research: A Broad Practical Approach. Lawrence Erlbaum Associates Publishers, London (2005)

Gunawardana, A., Shani, G.: A survey of accuracy evaluation metrics of recommendation tasks. J. Mach. Learn. Res. **10**(Dec), 2935–2962 (2009)

IDC: Smartphone vendor market share. https://www.idc.com/promo/smartphone-market-share/ (2017). Accessed 10 Aug 2017

Karim, M.Y., Kagdi, H., Di Penta, M.: Mining android apps to recommend permissions. In: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 427–437. IEEE (2016)

Khan, U.: RemoteDroid in F-Droid. https://f-droid.org/wiki/page/in.umairkhan.remotedroid (2018). Accessed 10 Aug 2017

Lucia, D.L., Xia, X.: Fusion fault localizers. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, pp. 127–138. ACM (2014)

Majeur: Applications info in Google Play. https://play.google.com/store/apps/details?id=com.majeur.applicationsinfo (2018). Accessed 10 Aug 2017

Manadhata, P.K., Wing, J.M.: An attack surface metric. IEEE Trans. Softw. Eng. **3**, 371–386 (2010)

Mihalcea, R., Corley, C., Strapparava, C., et al.: Corpus-based and knowledge-based measures of text semantic similarity. AAAI **6**, 775–780 (2006)

Munaiah, N., Kroh, S., Cabrey, C., Nagappan, M.: Curating GitHub for engineered software projects. Empir. Softw. Eng. **22**(6), 3219–3253 (2017)

Oracle: Java API documentation. https://docs.oracle.com/javase/7/docs/api/ (2017a). Accessed 10 Aug 2017

Oracle: The official website of Java. http://www.oracle.com/technetwork/java/javase/downloads/index.html (2017b). Accessed 10 Aug 2017

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al.: Scikit-learn: machine learning in python. J. Mach. Learn. Res. **12**(Oct), 2825–2830 (2011)

Pennington, J., Socher, R., Manning, C.D.: Glove: global vectors for word representation. EMNLP **14**, 1532–1543 (2014)

Pennington, J., Socher, R., Manning, C.D.: The website of glove. https://nlp.stanford.edu/projects/glove/ (2017). Accessed 10 Aug 2017

Qu, Z., Rastogi, V., Zhang, X., Chen, Y., Zhu, T., Chen, Z.: Autocog: measuring the description-to-permission fidelity in android applications. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1354–1365. ACM (2014)

Vidas, T., Christin, N., Cranor, L.: Curbing android permission creep. Proc. Web **2**, 91–96 (2011)

Wu, S. (ed.): Ranking-based fusion. In: Data Fusion in Information Retrieval, pp. 135–147. Springer, Berlin (2012)

Wu, D.J., Mao, C.H., Wei, T.E., Lee, H.M., Wu, K.P.: Droidmat: android malware detection through manifest and API calls tracing. In: Seventh Asia Joint Conference on Information Security (Asia JCIS), pp. 62–69. IEEE (2012)

Xia, X., Lo, D., Wang, X., Zhang, C., Wang, X.: Cross-language bug localization. In: Proceedings of the 22nd International Conference on Program Comprehension, pp. 275–278. ACM (2014)

Xu, B., Ye, D., Xing, Z., Xia, X., Chen, G., Li, S.: Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp. 51–62. ACM (2016)

Ye, X., Shen, H., Ma, X., Bunescu, R., Liu, C.: From word embeddings to document similarities for improved information retrieval in software engineering. In: Proceedings of the 38th International Conference on Software Engineering, pp. 404–415. ACM (2016)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Zhongxin Liu[1] · Xin Xia[2] · David Lo[3] · John Grundy[2]

Zhongxin Liu
liu_zx@zju.edu.cn

David Lo
davidlo@smu.edu.sg

John Grundy
john.grundy@monash.edu

[1] College of Computer Science and Technology, Zhejiang University, Hangzhou, China

[2] Faculty of Information Technology, Monash University, Melbourne, Australia

[3] School of Information Systems, Singapore Management University, Singapore, Singapore