APIMatchmaker: Matching the Right APIs for Supporting the Development of Android Apps

Yanjie Zhao, Li Li, Haoyu Wang, Qiang He, and John Grundy

Abstract—Android developers are often faced with the need to learn how to use different APIs suitable for their projects. Automated API recommendation approaches have been invented to help fill this gap, and these have been demonstrated to be useful to some extent. Unfortunately, most state-of-the-art works are not proposed for Android developers, and the ones dedicated to Android app development often suffer from high redundancy and poor run-time performance, or do not target the problem of recommending API usage patterns. To address this gap we propose to the community a new tool, namely *APIMatchmaker*, to recommend API usages by learning directly from similar real-world Android apps. Unlike existing recommendation approaches, which leverage a single context to find similar projects, we innovatively introduce a multi-dimensional, context-aware, collaborative filtering approach to better achieve the purpose. Specifically, in addition to code similarity, we also take app descriptions (or topics) into consideration to ensure that similar apps also provide similar functions. We evaluate *APIMatchmaker* on a large number of real-world Android apps and observe that *APIMatchmaker* yields a high success rate in recommending APIs for Android apps under development, and it is also able to outperform the state-of-the-art.

Index Terms—Android, API, Recommendation, Collaborative Filtering, APIMatchmaker.

1 Introduction

The software community has invented powerful IDEs (Integrated Development Environment) featuring comprehensive facilities, such as automatic code completion, to help developers better manage their software projects. The community has also made available a diversified set of libraries that offer Application Programming Interfaces (APIs) incorporating readily reusable function implementations. Developers can hence directly embed these libraries, instead of developing the same functions from scratch, to facilitate the development of their software applications.

Being able to provide readily reusable functions, APIs have become one of the most important components in modern software development. Our community hence has provided hundreds of thousands of software libraries, including APIs ranging from navigating maps, supporting security, using device features, and processing images and voice, etc. to scanning for malicious software packages. However, while the large number of existing libraries provide convenience for experienced developers to implement software quickly, they also introduce significant burdens to developers. This is because they need to constantly spend significant time to learn the usage of each new APIs in detail to correctly deploy it. As argued by Robillard, some APIs are hard to learn, even for professional developers working at large software companies such as Microsoft [1]. Some have shown developers even spend up to 19% of their programming time on the internet to search for source code, especially API usage examples [2].

To mitigate this, much research effort has been spent to automatically recommend appropriate APIs and their usage patterns [3], [4], [5], [6], [7]. For example, Niu et al. [3] have proposed a clustering-based approach, which leverages the co-existence relations between object usages, to recommend API usages. Nguyen et al. [5] propose an approach leveraging predictive models such as Hidden Markov Model to recommend API usages. Gu et al. [6] and Kim et al. [7] propose to learn API usages from similar code examples through code search.

Unfortunately, most of the state-of-the-art works are either not targeted to Android developers, the main focus of our work, or are implemented based on techniques such as clustering or traditional predictive models that come with a number of drawbacks. Indeed, for the former case, existing works cannot be easily adapted to recommend APIs for Android developers because particular features need to be specifically fulfilled when developing Android apps [8], [9]. For example, since the Android framework evolves rapidly, it is non-trivial to develop Android apps supporting all the historically released framework versions, which nevertheless could still be used in outdated devices. To this end, developers usually develop apps targeting only a small range of Android frameworks (e.g., by specifying the minimal, targeted, and maximum SDK version the app is designed to support) rather than all the historical frameworks. As a result, when recommending APIs to Android app developers, the range of supported SDK versions need to be considered so as to recommend usable APIs. For the latter case, clustering-based approaches, which use the frequency of patterns to achieve recommendations, has been

- *Yanjie Zhao, Li Li, John Grundy are with the Faculty of Information Technology, Monash University, Australia. Li Li is the corresponding author*
- *Haoyu Wang is with School of Cyber Science and Engineering, Huazhong University of Science and Technology, China*
- *Qiang He is with Faculty of Science, Engineering and Technology Swinburne University of Technology, Australia*

Manuscript accepted 16 Jan 2022

demonstrated to be inefficient as *frequent patterns are often uninteresting patterns*[1], as demonstrated by Fowkes et al. [4]. Predictive model-based approaches often require users to manually label a training dataset and prepare features for learning, which are known to be time-intensive and prone to errors. Furthermore, as argued by Nguyen et al. [11], the existing pattern recommendation approaches also suffer from high redundancy and poor run-time performance.

To cope with these limitations, we propose a method for learning and recommending Android API usage patterns, based on concepts emerging from Collaborative-Filtering (CF) recommendation systems. These systems have been leveraged to recommend items for users to purchase. The recommendation is made by selecting items that have been bought by similar users in similar contexts. One core requirement to incorporate collaborative-filtering for recommendation is to assess the similarity of customers. As recommended by Nguyen et al. [11], the projects that are highly similar to the project under development should provide higher quality patterns than those of dissimilar ones.

In the context of recommending Android API usages, by considering the method intended to use them, i.e., the client code as "customers" and API methods as "products", the API recommendation problem can be reformulated as *"which API methods should the client code invoke in order to complete the method under editing, taking into account the fact that some APIs have already been invoked by the client code?"* Specifically, for the app under development, we would like to learn API usages from such apps that are most similar to it, so as to preferentially recommend APIs that are conjointly used by those similar apps.

Unlike existing approaches, which leverage code implementations to locate similar projects, in this work, we also consider app topics (which need to be provided by app developers), in addition to the pure code implementations. The rationale behind this is that apps implementing the same topics tend to share similar high-level features, indicating likely similar (or even the same) code implementations. To this end, on top of the traditional collaborative-filtering algorithm, we propose a multi-dimensional context-aware collaborative-filtering approach to support the implementation of API recommenders for Android app development.

In this work, we were inspired by the FOCUS approach [11], [12], which was initially designed to learn from code snippets to recommend API usages for Java. The FOCUS authors have recently extended their initial implementation to further support the development of Android apps. They achieved this by directly converting Android apps code to Java code so as to be able to reuse the original Java-focused design. This indirect support has, however, overlooked some Android-specific features as they are not available in general Java applications. Our approach is designed to support the development of Android apps and hence has explicitly addressed those Android-specific features.

We design and implement a prototype API recommendation system called *APIMatchmaker* to support the development of Android apps. Since Android apps come with many different features compared to general Java applications, *APIMatchmaker* goes beyond the FOCUS approach by leveraging two-dimensional data (i.e., code implementation and app topic) to locate similar Android apps for learning and recommending API usages. We take into account both Android framework APIs and third-party library APIs. For the sake of simplicity, in this work, we will refer to both of them as Android APIs. Furthermore, due to the huge fragmentation issue (e.g., because of the Android framework's fast evolution), i.e., different Android apps may access different Android framework versions as each of them might contain a slightly different set of APIs, *APIMatchmaker* also takes this into account when recommending APIs.

With 12,000 real-world Android apps downloaded from AndroZoo [13] (their descriptions are directly crawled from Google Play), we experimentally demonstrate that our approach is effective and useful in recommending API usages to Android developers. *APIMatchmaker* achieves over 80% (or even 90%) of the success rate at Result@20 and can outperform the state-of-the-art approach as well as a baseline approach. The performance of *APIMatchmaker* can be even higher if the development of the active project is at a later stage, or increasing the size of the training app dataset, or by varying the customizable parameters provided by the tool. We make the following key contributions:

- we introduce to the community a new multi-dimensional context-aware collaborative filtering approach to better locate the most similar apps to support the recommendation;
- we design and implement a prototype tool called *API-Matchmaker*, which takes as input a method under editing and outputs a list of APIs (and their usage samples) meeting the constraints of the SDK versions that could be leveraged to complete the implementation of the method; and
- we evaluate our approach on 12,000 real-world Android apps under different experimental settings. Experimental results show that our approach is promising in recommending API usages to Android app developers.

The rest of this paper is organized as follows: Section 2 presents a motivating example attempting to help readers better understand the problem targeted in this work. Section 3 then details the design and implementation of our approach, namely *APIMatchmaker*. Next, we present the experimental setup and the evaluation results of our approach in Section 4 and Section 5, respectively. After that, we discuss the sensitivity of our approach concerning several aspects in Section 6. Finally, Section 7 discusses related work, and Section 8 concludes this paper.

2 Motivating Example

The typical usage scenario of our work is to recommend APIs, including their usages to app developers who are actively implementing an Android app. The developer might have already completed the development of some methods and is now halfway through finishing the method under editing (hereafter referred to as the "active method"). Fig. 1 illustrates such an example (extracted from app *com.appsfreeinc.zebra.sounds*, hereinafter referred to as the "active app project"). The active method (i.e., *a()*) is divided

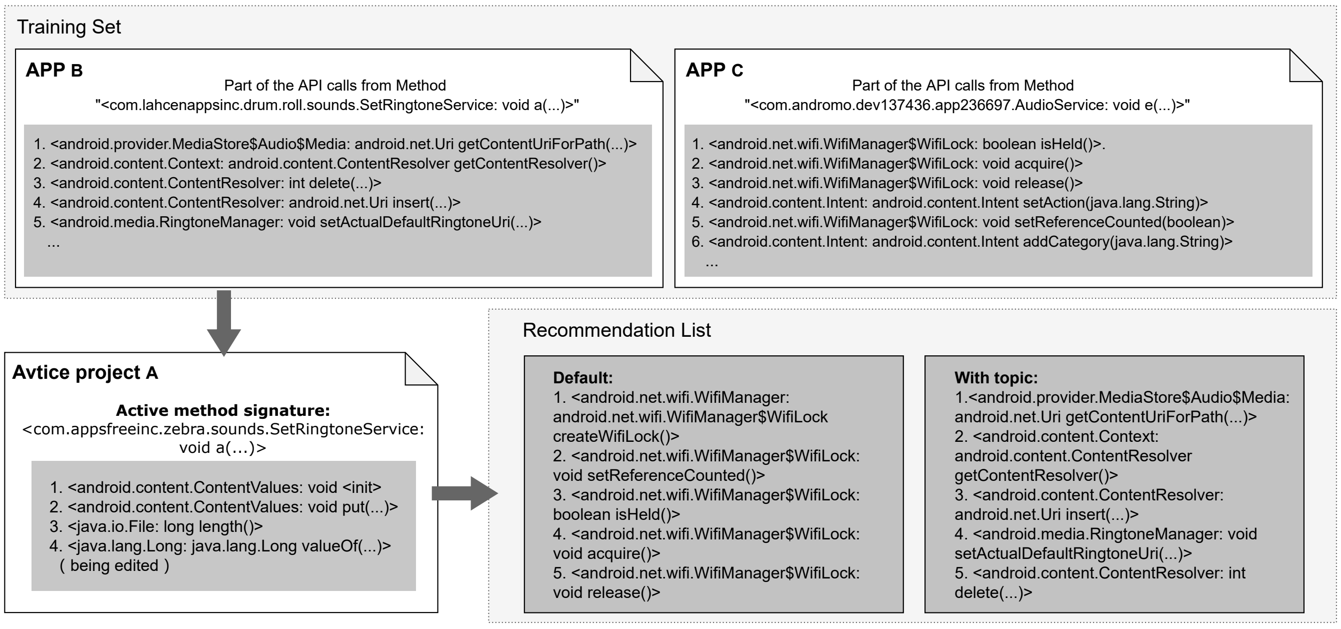
1. This problem is well known in the data mining literature [10].

```
1   private static void a(Context context, File file, String str, int i) {
2       if (context != null && file != null) {
3           String absolutePath = file.getAbsolutePath();
4           ContentValues contentValues = new ContentValues();
5           contentValues.put("_data", absolutePath);
6           ...
7           contentValues.put("_size", Long.valueOf(file.length()));

8           Uri contentUriForPath = MediaStore.Audio.Media.getContentUriForPath(absolutePath);
9           context.getContentResolver().delete(contentUriForPath, "_data=\"" + absolutePath + "\"", (String[]) null);
10          RingtoneManager.setActualDefaultRingtoneUri(context, i, context.getContentResolver().insert(contentUriForPath, contentValues));

11      }
12  }
```

Fig. 1: Code snippet extracted from app *com.appsfreeinc.zebra.sounds* (A Zebra sound player).



Part of the API calls from Method
"<com.lahcenappsinc.drum.roll.sounds.SetRingtoneService: void a(...)>"

1. <android.provider.MediaStore$Audio$Media: android.net.Uri getContentUriForPath(...)>
2. <android.content.Context: android.content.ContentResolver getContentResolver()>
3. <android.content.ContentResolver: int delete(...)>
4. <android.content.ContentResolver: android.net.Uri insert(...)>
5. <android.media.RingtoneManager: void setActualDefaultRingtoneUri(...)>
...

Part of the API calls from Method
"<com.andromo.dev137436.app236697.AudioService: void e(...)>"

1. <android.net.wifi.WifiManager$WifiLock: boolean isHeld()>.
2. <android.net.wifi.WifiManager$WifiLock: void acquire()>
3. <android.net.wifi.WifiManager$WifiLock: void release()>
4. <android.content.Intent: android.content.Intent setAction(java.lang.String)>
5. <android.net.wifi.WifiManager$WifiLock: void setReferenceCounted(boolean)>
6. <android.content.Intent: android.content.Intent addCategory(java.lang.String)>
...

Active method signature:
<com.appsfreeinc.zebra.sounds.SetRingtoneService: void a(...)>

1. <android.content.ContentValues: void <init>
2. <android.content.ContentValues: void put(...)>
3. <java.io.File: long length()>
4. <java.lang.Long: java.lang.Long valueOf(...)>
(being edited)

Default:
1. <android.net.wifi.WifiManager: android.net.wifi.WifiManager$WifiLock createWifiLock()>
2. <android.net.wifi.WifiManager$WifiLock: void setReferenceCounted()>
3. <android.net.wifi.WifiManager$WifiLock: boolean isHeld()>
4. <android.net.wifi.WifiManager$WifiLock: void acquire()>
5. <android.net.wifi.WifiManager$WifiLock: void release()>

With topic:
1.<android.provider.MediaStore$Audio$Media: android.net.Uri getContentUriForPath(...)>
2. <android.content.Context: android.content.ContentResolver getContentResolver()>
3. <android.content.ContentResolver: android.net.Uri insert(...)>
4. <android.media.RingtoneManager: void setActualDefaultRingtoneUri(...)>
5. <android.content.ContentResolver: int delete(...)>

Fig. 2: The details of active project A (under development), APP B, and APP C.

into two parts. The first part (lines 2-7) presents the code that has just been completed by the developer. The second part, highlighted in bold, is the actual implementation of this method (i.e., ground truth). In this motivating example, we consider the second part is unknown, and our objective of this work is to *recommend appropriate APIs for helping developers complete the second part*.

We plan to learn API usage patterns from existing apps because we hypothesize that similar apps may implement the same functionality using suitable, reusable libraries for the active app project. To this end, we use a large set of Android apps to locate similar apps of the active app project. Fig. 2 highlights two such example apps (B and C, respectively for apps *com.lahcenappsinc.drum.roll.sounds* and *com.andromo.dev137436.app236697*). Although both of them are very similar to the motivating example app in terms of code implementations (including the code in other methods that are not listed here), the API usages leveraged by these two apps are quite different, which could, in turn, provide noisy recommendation results.

Based on the recommendation algorithm proposed by Nguyen et al. [11] for recommending API usages for supporting the development of Java projects, for our motivating example, the recommended output would be the *default* list shown in Fig. 2. The result is however quite far from the ground truth highlighted in Fig. 1. We then look into the algorithm and find that in this example, app C has dominated the recommendation. Although both apps B and C are possible candidate apps to reuse API usage examples, a detailed analysis reveals that both app B and the active app project are players for offline sound resources while app C is for playing online music and news. Ideally, app B should be closer to the active project than app C and should contribute more to the recommendation. To this end, we refine our recommendation algorithm to also take into account the similarities of the app topic as codified in the app description text. The re-computed recommendation output, as highlighted in Fig. 2 (with topic), is now much more closer to our ground truth. This simple motivating example suggests that app topics should not be overlooked when learning implementations from existing Android apps. In this work, we leverage both other app implementations and other app topics to learn and recommend API usage patterns for the development of Android apps.

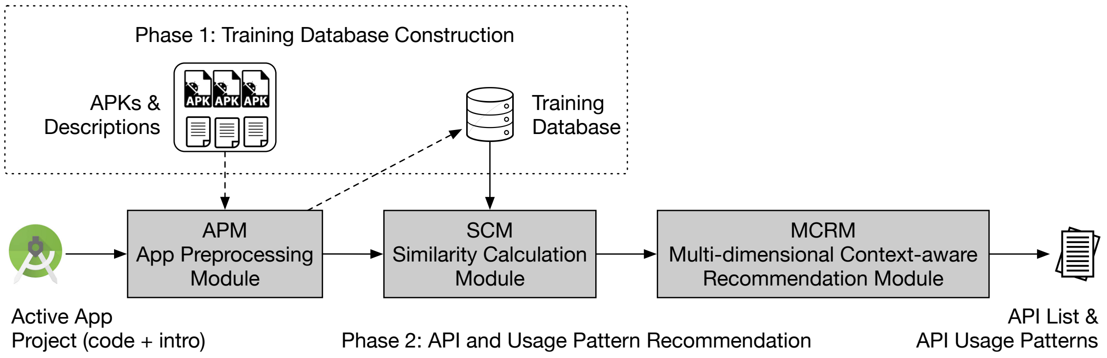

Phase 1: Training Database Construction
APKs & Descriptions
APK APK APK
Training Database
APM App Preprocessing Module
SCM Similarity Calculation Module
MCRM Multi-dimensional Context-aware Recommendation Module
Active App Project (code + intro)
Phase 2: API and Usage Pattern Recommendation
API List & API Usage Patterns

Fig. 3: The architecture of *APIMatchmaker*.

3 Our Approach: *APIMatchmaker*

Fig. 3 presents an overview of our approach *APIMatchmaker*, which leverages two phases to find the most likely best-fit existing Android APIs to assist in supporting the development of new Android apps. The two phases are (1) Training database construction and (2) API and usage pattern recommendation. The first phase prepares a training database based on the implementation of a large set of real-world Android apps. The training database will then be referenced and reused to support the implementation of the second phase, which aims to recommend APIs and API usage patterns to an active app project that is currently under development.

As shown in Fig. 3, *APIMatchmaker* is made up of three main modules: (1) App Pre-processing module (APM), (2) Similarity Calculation Module (SCM), and (3) Multi-dimensional Context-aware Recommendation Module (MCRM). The first module APM is leveraged by both of the two phases, while the remaining two modules are used to achieve the objectives of phase 2.

3.1 APM: App Preprocessing Module

APM is used to process three types of inputs in order to serve both of the aforementioned phases: (1) Android APKs, (2) Active App projects under development, and (3) App descriptions. The first type is needed to fulfill the requirement of phase 1 towards constructing a training database for supporting API recommendation. Compiled Android APKs are leveraged rather than open-source code because we want to build our training database based on real-world – likely high-quality – Android apps, which are usually only released as APKs. The second type is the ideal input for the second phase, for which the overall goal of this work is to help developers complete their active app projects under development. The last input type is considered because similar apps – those implementing the same goal, embedding the same features, etc. – may be more likely to use APIs similar to the active app project and the active method, as stated by Nguyen et al. [11]. This information can be taken into account when learning API usages from Android apps. We leverage app descriptions to identify similar apps and this input type is needed to fulfill the purposes of both phases.

This leads us to three key data sources for our pre-processing module:

- **Android APKs** are closed-source app versions distributed over popular app markets such as the official Google Play store. This module leverages Soot to parse closed-source compiled app code at the Jimple code level to extract all the methods implemented in the app and the Android APIs accessed by those methods. In other words, our approach does not require converting Android bytecode to Java source code to achieve its purpose. Soot is an optimization framework for supporting static program analysis of Java and Android apps, and Jimple is a typed 3-address intermediate representation provided by Soot to ease its code manipulations [14]. The output of this processing is the app's full list of method signatures and the set of Android APIs associated with each method signature.
- **Active App project under development.** The objective of *APIMatchmaker* is to match the right APIs for supporting the development of new Android apps. The input of *APIMatchmaker* is hence an app project that is under active development. In such a project, we expect that some methods have already been fully completed, while some others have not (might be half-completed or not even started). The goal of *APIMatchmaker* is hence to recommend the best suited APIs and their usage patterns that will help app developers quickly expand those incomplete methods.
  Since the active app projects come with source code, the pre-processing of the first module (APM) is achieved by directly parsing the source code. In practice, *APIMatchmaker* leverages JavaParser to ease the implementation. Given an active app project, this module will parse all its developer-defined methods and output those methods, along with their accessed Android APIs.
- **App descriptions** are provided by app developers to introduce and advertise the app. For closed-source apps, their descriptions can usually be collected from the app markets where the apps are hosted. For example, on the official Google Play store, all apps are provided with a dedicated page to describe their goals, functions and key features. For such apps that are still under development, we need their developers to provide such descriptions when using our tool-chain to help them implement the apps.
  App descriptions are provided in natural language. This module hence leverages natural language pro-

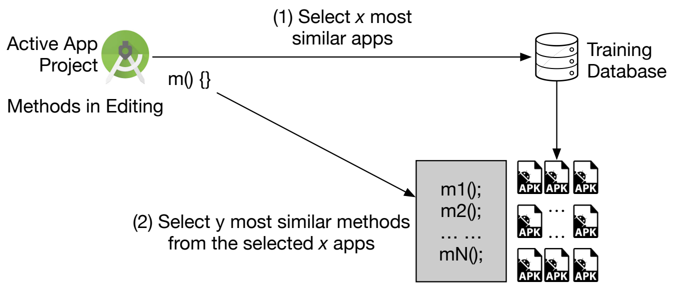

(1) Select x most similar apps
Active App Project
Methods in Editing
m() {}
Training Database
m1();
m2();
... ...
mN();
APK
(2) Select y most similar methods from the selected x apps

Fig. 4: The process of selecting x most similar apps and y most similar methods (x and y are parameters needed to be customized by the users of *APIMatchmaker*).

cessing (NLP) techniques to pre-process this type of input data. Specifically, *APIMatchmaker* will first cut the description text paragraphs into words. It then turns all the words into lower case, and removes punctuation and stop words (i.e., the most common words in a language). After that, *APIMatchmaker* performs a stemming step to further remove the morphological affixes of the remaining words to only retain their common base forms. Due to grammatical reasons, different forms of a word, such as *organize*, *organizes*, and *organizing*, could be used. However, these forms are essentially just different tenses of the same root word, i.e., *organize*, and hence should be treated as such. The last stemming step is introduced to achieve that, i.e., mapping words that are derived from one another to a common word.

3.2 SCM: Similarity Calculation Module

Given an Android app project under active development, our *APIMatchmaker*'s second module performs similarity analyses aiming to find similar apps from the training set. As shown by Zhong et al. [15], under certain usage scenarios (e.g., implementing the same functions or leveraging the same library modules), API methods are frequently called together and even follow some sequential calling rules. Taking this empirical evidence in mind, we hypothesize that similar apps under specific scenarios could share similar API usage patterns as well. Therefore, we believe that it is also possible to learn API usage patterns from existing apps.

To this end, in this Similarity Calculation module, we aim to identify and learn API usages from existing apps that are similar to the app projects under active development. As shown in Fig. 4, we rely on two steps to achieve this purpose. First, we leverage code implementation and app description to select x apps[2] that are most similar to the app project under development. Then, for such methods under editing in the active app project, we select top-y similar methods from the previously selected x apps. These y methods will be leveraged by the MCRM module (detailed in Subsection 3.3) to learn and recommend API usages for the methods under editing in the active app project.

3.2.1 Select x most similar apps.

For this we use the following two types of similar apps: (1) apps that are similar in terms of their targeted topics or features. The rationale behind this is that apps implementing the same topics could share similar code implementation, such as leveraging the same third-party libraries. (2) apps that are similar in terms of app implementation. These apps may share the same method signatures or access the same set of Android APIs compared with the app project under active development.

2. This parameter is configurable. Similarly, the upcoming parameter y is also configurable.

App Similarity (Topic). For calculating the topic similarity of Android apps, app descriptions are leveraged to fulfill this purpose. Using the first pre-processing module, the app descriptions have been transformed to clean and concise versions. We leverage a popular and classical algorithm called Term Frequency and Inverse Document Frequency (TF-IDF) to calculate the topic similarities between the active app project and the apps in the training database.

In the TF-IDF algorithm, Term Frequency (TF) refers to the frequency of keywords in a document, which can be calculated via the following formula, where $n_{i,j}$ is the number of times the word i appears in document j, and $\Sigma_k n_{k,j}$ is the total number of words in document j.

$$TF_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \tag{1}$$

Inverse Document Frequency (IDF) refers to the inverse text frequency, which is an index used to measure the weight of keywords. IDF can be calculated by the following formula, where $|D|$ is the total number of documents, and $|j : t_i \in d_j|$ is the number of documents where the word i presents.

$$IDF_i = log\frac{|D|}{|j : t_i \in d_j|} \tag{2}$$

Generally speaking, the higher frequency of words in a particular document (i.e., higher TF value), or the lower frequency of words in the entire document set (i.e., lower IDF value), the higher TF-IDF value can be achieved. In other words, TF-IDF tends to filter out common words, meanwhile retaining the important ones. The TF-IDF can be calculated via the following formula:

$$TF - IDF = TF_{i,j} \times IDF_i \tag{3}$$

Given two app descriptions p' and q' (of apps p and q) and their TF-IDF vectors $\vec{\lambda}$ and $\vec{\mu}$, respectively, *APIMatchmaker* leverages cosine similarity to calculate the distance of these two descriptions (cf. Formula 4).

$$sim_1(p,q) = \frac{\vec{\lambda} \cdot \vec{\mu}}{|\vec{\lambda}||\vec{\mu}|} = \frac{\sum_{i=1}^{n} \lambda_i \times \mu_i}{\sqrt{\sum_{i=1}^{n}(\lambda_i)^2} \times \sqrt{\sum_{i=1}^{n}(\mu_i)^2}} \tag{4}$$

App Similarity (Implementation). Similar to the approach used to calculate the topic similarity of Android apps, we leverage the same TF-IDF algorithm to calculate app similarities based code implementations. The only difference is that API calls are leveraged rather than descriptive natural language words. We represent an Android app as a vector $\vec{\phi} = \{\phi_i\}_{i=1,...,k}$, with ϕ_i being the TF-IDF value of each API call. Then, the similarity of apps p and q, with their feature vectors $\vec{\phi} = \{\phi_i\}_{i=1,...,k}$ and $\vec{\omega} = \{\omega_j\}_{j=1,...,l}$, can be calculated by Formula 5.

$$sim_2(p,q) = \frac{\vec{\phi} \cdot \vec{\omega}}{|\vec{\phi}||\vec{\omega}|} = \frac{\sum_{t=1}^{n} \phi_t \times \omega_t}{\sqrt{\sum_{t=1}^{n}(\phi_t)^2} \times \sqrt{\sum_{t=1}^{n}(\omega_t)^2}} \quad (5)$$

3.2.2 *Select y most similar methods.*

The reason why this step is conducted rather than simply taking into account all the methods declared in the x apps is that many of the declared methods may not access or only access a few Android APIs. Hence, it could introduce noise into our approach if those methods are taken into account. Therefore, in this work, we decide to only select y most similar methods to support further analyses.

Method Similarity. Given the method under editing for which we are about to recommend APIs and usage patterns to use, we leverage the Jaccard similarity coefficient to find its y nearest neighbors $M = \{m_i\}_{i=1,2,3,...,y}$. The Jaccard coefficient is a well-used metric that has been frequently leveraged to calculate the similarity or distance of different sets. Given two sets A, B, as shown in Formula 6, the *Jaccard Index* is expressed as the ratio of the size of the intersection of A and B to the size of the union of A and B:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \quad (6)$$

For our work, we calculate the similarity between the method under editing and a method in the x apps via Formula 7, where $\mathbb{F}(m)$ and $\mathbb{F}(n)$ are the sets of API calls extracted from method signatures m and n (the extraction is done in the APM module).

$$sim_\gamma(m,n) = \frac{|\mathbb{F}(m) \cap \mathbb{F}(n)|}{|\mathbb{F}(m) \cup \mathbb{F}(n)|} \quad (7)$$

3.3 MCRM: Multi-dimensional Context-aware Recommendation Module

We now present the last module of *APIMatchmaker*, which performs multi-dimensional context-aware API recommendation after filtering our incompatible APIs.

3.3.1 *Filtering out incompatible APIs*

Recall that certain Android APIs are only available in a number of SDK versions. When developing apps with a dedicated Android SDK version, developers can only take advantage of the APIs available in that SDK. However, the resulting app is expected to be able to run different devices running different Android frameworks (i.e., SDK versions). This mismatch has led to the well-known fragmentation issue in the mobile community, for which Android apps may crash on certain devices while running smoothly on others. Typically, there are two types of compatibility issues introduced by the fast evolution of the Android framework.

- **Forward Compatibility Issue** implies that a given app developed targeting a given API level may not execute seamlessly on devices running Android with higher API levels.
- **Backward Compatibility Issue** implies that a given app developed with a given API level may not perform normally on devices running Android systems with lower API levels.

Because of the aforementioned compatibility issues, we have to take SDK versions into account when recommending possible APIs for implementing an active method. Fortunately, Android apps have been provided with a manifest file to configure the supported SDK versions.

- *minSdkVersion*, i.e., the minimum API Level at which the app is intended to run. This attribute is leveraged by Google Play to filter out devices with SDK platform versions lower than the value declared in *minSdkVersion*.
- *targetSdkVersion*, i.e., the API Level that the app is targeting. If this attribute is not set explicitly, the default value will be set to the value of *minSdkVersion*.

In this module, before running the multi-dimensional context-aware recommendation algorithm, we take additional efforts to extract the supported SDK versions from the apps in the training dataset and subsequently filter out such APIs that may cause compatibility issues (i.e., the incompatible APIs should not be included in the recommended API list). We achieve this function by automatically taking into account the lifecycle of APIs (i.e., when they are introduced and when they are excluded in the framework), which are extracted by checking the evolution of the Android frameworks, following the CiD approach introduced by Li et al. [9].

3.3.2 *Multi-dimensional context-aware recommendation*

Using our *APIMatchmaker*'s second module, the search space is reduced from all the methods of the training apps to a set of methods selected from a set of apps in the training set. In this last step, we introduce our multi-dimensional context-aware recommendation approach. This aims to recommend APIs for the method under editing in the active app project.

In particular, *APIMatchmaker* first determines the number of Android APIs (k) accessed by the selected apps (x) and then models them into a $(y+1) * k$ matrix. As shown in Table 1, methods (selected ones $m_1 \rightarrow m_y$ plus the one under editing m_{edit}) are represented as rows, and APIs are represented as columns. For the elected y methods, each of their cells in the matrix will be set to either true (1) or false (0), representing whether the corresponding API has been accessed by the method or not. For example, cell (m_2, api_k) is set to be 1, indicating that method m_2 has accessed api_k. For the method under editing (i.e., m_{edit} in the last row), each of its cells will be set to either true (1) or unknown (-1). The *true* cells indicate the set of APIs that have already been leveraged by the method under editing (i.e., m_{edit}). The *unknown* ones are the possible candidates that could be needed to complete m_{edit}. The goal of our *APIMatchmaker*'s last module is hence to predict which of the possible candidates could be used by the method under editing so as to recommend them to app developers to assist them in completing the method code.

Collaborative filtering has been often used for recommendation algorithms to implement recommendation systems [16]. A typical application of collaborative filtering is to recommend items that a user is most likely to purchase based on his/her past shopping records or information about other users with similar purchasing behaviors in an e-commerce system. Chen et al. [17] introduce the concept of

TABLE 1: An example of the encoding matrix.

	api_1	api_2	api_3	api_4	api_5	...	api_k
m_1	1	1	0	1	0	1	1
m_2	1	1	0	0	0	1	1
m_3	1	1	0	0	1	1	0
...	1	1	1	0	0	0	1
m_y	1	1	1	1	1	0	1
m_{edit}	1	1	-1	-1	-1	-1	-1

context into the traditional collaborative filtering algorithm. The purpose of this is to recommend to the current user in the present context what other like-minded users do in a similar context.

Borrowing the idea of context-aware collaborative filtering for recommending items for users to purchase, in this work we leverage the same algorithm to recommend the candidate usage of Android APIs. In our work, the method plays the role of a **user**, the API plays the role of an **item** and the app project, to which the method belongs, plays the role of **context**. A **rating** (e.g., a numerical value) is further associated with a user and an item. The prospective outcome of a collaborative filtering system is a set of predicted ratings (aka. recommendations) for a specific user and a subset of items [18]. The recommendation system considers the most similar users to the active user (aka. neighbours) to calculate new ratings. Additionally, based on the traditional collaborative filtering approach, we propose a new algorithm called multi-dimensional context-aware collaborative filtering. The idea behind this new algorithm is to integrate different types of similarity metrics to fulfill the API usage recommendation.

Let us define the project, i.e., context C as a tuple of z different types of similarity metrics, where $c_t(t \in 1...z)$ is a type of context.

$$C = (c_1, c_2, ..., c_z) \tag{8}$$

Except the APIs already included in the active method m_{edit}, for each api accessed by the x APKs, *APIMatchmaker* computes a score for each cell representing an api, i.e., cells set as -1 in the encoding matrix [3] shown in Table 1. The probability of recommending a given API api to m_{edit} can then be calculated via the following formula [17], where M is the set of the y most similar method signatures, sim_γ is defined by Formula 7, and $\bar{r_{m_{edit}}}$ and $\bar{r_m}$ are the mean ratings of m_{edit} and m, respectively.

$$p_{m_{edit},api,C_{m_{edit}}} = \bar{r_{m_{edit}}} + \frac{\sum_{m \in M}(R_{m,api,C_{m_{edit}}} - \bar{r_m}) \cdot sim_\gamma(m_{edit}, m)}{\sum_{m \in M} sim_\gamma(m_{edit}, m)} \tag{9}$$

In our implementation, $\bar{r_m}$ can be obtained by the encoding matrix in Table. 1, i.e., calculating the average rating of the cells in the row corresponding to m. For $\bar{r_{m_{edit}}}$, we set its value to 0.8 following the general practice of the state-of-the-art [11].

3. The encoding matrices of different projects in the training set are essentially a simplified presentation of the context-aware 3-dimensional scoring matrix, as mentioned in FOCUS [11]. Interested readers are encouraged to read this paper for more details.

The weighted rating of each method $m \in M$, i.e., $R_{m,api,C_{m_{edit}}}$, with respect to API api and project context of the method under editing, can be calculated via the following formula, where w_t is the weight we assign to each type of context and $\sum_{t=1}^{z} w_t$ should equal to one, and sim_t is defined by Formula 4 and Formula 5.

$$R_{m,api,C_{m_{edit}}} = \sum_{t=1}^{z} w_t \cdot r_{m,api,C_m} \cdot sim_t(C_{m_{edit}}, C_m) \tag{10}$$

We rely on the similarity of two contexts – two-dimensions: topic and code implementation – to recommend API usages. In this case, project context C will be (c_1, c_2), respectively representing the contexts of app topic and code implementation. Similarly, (w_1, w_2) will be respectively the weights of app topic and code implementation. These two weights can be configured by the users of our approach. By default, we set their values to be $(0.2, 0.8)$.

Output. The output of *APIMatchmaker* will be a list of Android APIs that are ranked based on the scores calculated via Formula 9. This list will be continuously updated to adapt to the change of the method under editing. Moreover, apart from the top-N APIs recommended based on the already written code in the active method, *APIMatchmaker* will also generate API usage samples that are extracted from the selected most similar apps, to help developers make use of the recommended APIs.

4 Experimental Setup

To evaluate the effectiveness of *APIMatchmaker*, we need to answer the following research questions:

- **RQ1:** How effective is *APIMatchmaker* in recommending accurate APIs for Android app developers to complete their development?
- **RQ2:** How does *APIMatchmaker* compare with other state-of-the-art tools?
- **RQ3:** To what extent do different parameters affect the performance of *APIMatchmaker*?
- **RQ4:** How effective is the multi-dimensional context-aware collaborative filtering approach applied by *APIMatchmaker*?

4.1 Dataset

To answer these research questions, we crawled a large set of real-world Android apps from Google Play, including their descriptions advertised on Google Play. Since it is not straightforward to crawl Android apps from Google Play, we resorted to collecting the latest Google Play apps from the well-known AndroZoo dataset, which has already been leveraged by various research projects [13]. At the moment, AndroZoo contains over 10 million Android apps crawled from various sources, including the official Google Play app store.

Since our collected dataset will be used as a training set for *APIMatchmaker* to learn API usages, it would be great if the collected apps have significant usage of different Android APIs. To this end, when crawling apps from AndroZoo, we checked API usage of each downloaded app

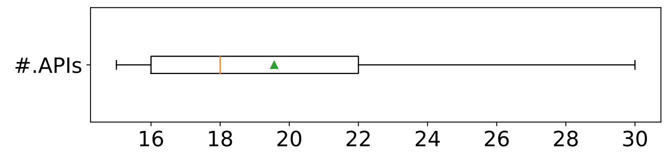


Fig. 5: Distribution of the numbers of API in the methods selected from the 12,000 apps.

and only retained the ones that have at least six (6) methods using at least fifteen (15) Android APIs.

Furthermore, when collecting apps from AndroZoo, we realize that not all the Google Play apps in AndroZoo are currently available on Google Play (e.g., those apps might be removed already), and not all the apps are described in English. As a result, we cannot obtain or parse those apps' descriptions, and thereby we have to ignore them when preparing our dataset. In addition, some apps, although different in terms of their hash digests (i.e., SHA256), are essentially the same app (i.e., share the same app package name but with different versions). Unfortunately, such apps share the same app description and thereby could introduce biases to our approach and experiments. To mitigate this, we only retain the latest app version in our dataset. Finally, because of some corner cases, some apps cannot be successfully parsed by our tool-chain to preprocess Android apps. Indeed, some apps do not contain the AndroidManfest configuration file, so that their app package names cannot be extracted. In this work, we simply ignore those apps. Eventually, we stopped the collection at 12,000 apps from roughly 40,000 apps randomly selected from AndroZoo. Our final dataset is made up of 12,000 unique Android apps, along with their descriptions. Fig. 5 presents the basic statistics of the numbers of unique APIs called per method in the 12,000 apps. The median and average numbers are 18 and 19.8, respectively. Our final dataset is made up of 12,000 unique Android apps, along with their descriptions. Fig.5 presents the basic statistics of the numbers of unique APIs called per method in the 12,000 apps. The median and average numbers are 18 and 19.8, respectively. It suggests that the methods used for evaluation roughly involve 16-22 API calls in most cases.

4.2 Experimental settings

Since it is hard to find active app projects under development, we use existing apps for simulation. Specifically, given an Android app with Δ methods declared, we propose to simulate its development at two different stages: **Stage 1:** App developers have completed the implementation of $\Delta/2-1$ methods and are now preparing to finish the $(\Delta/2)$-th method; **Stage 2:** App developers have completed the implementation of $\Delta - 1$ methods and are now preparing to finish the last method (Δ-th). In each stage, for the method under editing, we further take into account two scenarios: **Scenario 1:** The method has already accessed into one Android API; **Scenario 2:** The method has accessed into four Android APIs. Combining the stages with scenarios, we eventually set up four experimental settings to evaluate the performance of *APIMatchmaker* in recommending Android APIs for supporting the development of Android apps:

- **E1 (Stage 1, Scenario 1)**: The active app project has $\Delta/2 - 1$ method completed and the $\Delta/2$-th method under editing has already accessed into one Android API.
- **E2 (Stage 1, Scenario 2)**: The active app project has $\Delta/2 - 1$ method completed and the $\Delta/2$-th method under editing has already accessed into four Android APIs.
- **E3 (Stage 2, Scenario 1)**: The active app project has $\Delta - 1$ method completed and the Δ-th method under editing has already accessed into one Android API.
- **E4 (Stage 2, Scenario 2)**: The active app project has $\Delta - 1$ method completed and the Δ-th method under editing has already accessed into four Android APIs.

For each of these four settings, we use the standard procedure, i.e., 10-fold cross-validation, to evaluate the performance of our *APIMatchmaker*'s recommendations. To this end, we randomly divide our dataset (i.e., 12,000 apps) into ten sets (1,200 apps in each set). We then select nine sets to fulfill the training set and use the remaining set for testing. We repeat this process ten times to make sure that each of the ten sets has been regarded as a test set once. The average scores of the ten tests are then reported as the final performance of our approach. Since it is not easy to get the actual developing order of each Android project, the selection of the completed methods is random.

4.3 Evaluation Metrics

Given a method under editing in an active Android app project, the objective of our approach is to recommend a ranked list of API calls (e.g., N APIs) to help developers complete the implementation of the method. To help assess whether *APIMatchmaker* fulfills this objective, we leverage the following metrics to evaluate the effectiveness of our approach.

For each test sample, we only consider one active method. Given a set of projects P under testing, for the method under editing m in each project $p \in P$, *APIMatchmaker* generates N recommended APIs, i.e., $R_N(p)$, to fulfill m.

Success rate: We consider that a recommendation is successful for project p as long as at least one out of the N APIs hit the Ground-Truth set $GT(p)$. The success rate for the $|P|$ projects can then be calculated via Formula 11, where $GT(p)$ stands for the set of APIs actually accessed by m in p, and $match_N(p)$ is defined as the intersection of the recommended N APIs and $GT(p)$, i.e., $match_N(p) = R_N(p) \cap GT(p)$.

$$success\ rate@N = \frac{count_{p \in P}(|match_N(p)| > 0)}{|P|} * 100\% \tag{11}$$

Precision and Recall: For each test sample, $precision@N$ is the ratio of the top N recommended APIs matching $GT(p)$, and $recall@N$ is the ratio of APIs belonging to $GT(p)$ falling in the top N recommendations.

$$Precision@N = \frac{|match_N(p)|}{N} * 100\% \tag{12}$$

$$Recall@N = \frac{|match_N(p)|}{|GT(p)|} * 100\% \tag{13}$$

TABLE 2: Success rate of 10-fold cross-validation on the 12,000 randomly selected apps for *APIMatchmaker*, the state-of-the-art tool FOCUS, and the baseline of our approach. Result@N indicates the number of recommended APIs considered for evaluation.

N	*APIMatchmaker*				*Baseline 1* - FOCUS				*Baseline 2*			
	E1	**E2**	**E3**	**E4**	**E1**	**E2**	**E3**	**E4**	**E1**	**E2**	**E3**	**E4**
1	52.68%	60.34%	54.09%	61.1%	45.89%	51.92%	45.74%	54.01%	49.73%	49.71%	49.9%	49.1%
5	68.36%	77.33%	69.17%	78.08%	62.28%	71.14%	63.9%	72.43%	68.75%	66.99%	69.45%	66.58%
10	73.7%	82.8%	75.6%	83.04%	68.99%	77.41%	70.13%	78.45%	75.62%	73.27%	75.69%	73.48%
15	77.34%	88.03%	78.96%	88.36%	73.94%	83.13%	74.54%	84.31%	79.35%	78.4%	79.76%	79.39%
20	81.85%	91.61%	82.59%	91.64%	79.45%	87.43%	79.69%	89.14%	81.83%	81.84%	82.52%	83.1%

5 RESULTS

5.1 RQ1: Performance of *APIMatchmaker*

In this first research question, we investigated the performance and effectiveness of our approach *APIMatchmaker*. Following the settings described in Section 4.2, all the experiments are conducted based on the default parameters of *APIMatchmaker*, i.e., ten similar apps ($x = 10$), six similar methods ($y = 6$), 20% and 80% weights respectively for app topic and app code implementation($w_1 = 0.2, w_2 = 0.8$).

Table 2 summarizes our experimental results with respect to different situations when different numbers of APIs considered for evaluation. Specifically, we present our results in five different situations: 1, 5, 10, 15, 20. For example, when Result@5 is concerned, we leverage the top-5 recommended APIs to calculate the performance (success rate, precision, and recall) of our approach. Generally speaking, the more APIs considered, the higher the success rate our approach can achieve. Indeed, if only one API is considered, our approach can already hit the correct API at over 50% of success rate. When increasing the number of recommended APIs to 20, the success rate can exceed 80%. In other words, by checking at most 20 APIs and for more than 80% of cases (i.e., methods under editing), app developers can successfully find the right APIs to complete the implementation of the method. This experimental result shows that our approach is effective in recommending APIs for supporting developers to implement Android apps.

Furthermore, when different scenarios – comparing E1 to E2, or E3 to E4 – are considered, *APIMatchmaker* will achieve different performance. As shown in Table 2, *APIMatchmaker* always achieves better performance in Scenario 2, compared with the results yielded in Scenario 1. This result is expected by us because Scenario 2 provides more known APIs than Scenario 1. Indeed, the more known APIs are provided, the more close neighbor methods can be located, and thereby the higher performance can be achieved.

Even for the same scenario, when different stages are taken into account – Stage 1 to Stage 2 and comparing E1 to E3 and E2 to E4 – it is interesting to observe that the Stage 2 setting can always get a higher performance compared to that of the Stage 1 setting. This suggests that the success rate increases when the number of completed methods increases. In other words, *APIMatchmaker* will be more useful for app projects that are already at a later stage.

We now go one step further to check the sensitivity of our approach to the size of the training dataset. Ideally, we would hypothesize that the larger the training dataset, the higher the performance our approach would achieve. Indeed, a larger set of training dataset could potentially allow *APIMatchmaker* to select x most similar apps that are even closer to the active project under development than selecting from a small dataset. To this end, apart from the 12,000 training apps considered in this work, we conduct two new experiments, each considering 3,000 and 6,000 apps as the training set, respectively, where these apps are randomly selected from the 12,000 datasets. Fig. 6 illustrates the experimental results. Clearly, in all the four experimental settings (cf. E1-E4), these results confirm our hypothesis that the performance of *APIMatchmaker* increases when more apps are considered for the training set.

Fig. 7 further compares the experimental results obtained by excluding the app topic context (i.e., only taking code implementation into consideration, i.e., ($w_1 = 0, w_2 = 1$). The experimental results confirm our findings that the app topic context is helpful for finding the most similar apps. Interestingly, the benefit is even more significant when the size of the training set is small.

RQ1 Answer

With over 80% or even 90% (in scenario 2) success rate, *APIMatchmaker* achieves promising results in recommending API usages at Result@20, and yields acceptable results in an extreme case when only one recommended API is concerned. Furthermore, the more knowledge *APIMatchmaker* can learn from the active app project under development (in later stages), the higher performance *APIMatchmaker* can achieve in successfully recommending API usages.

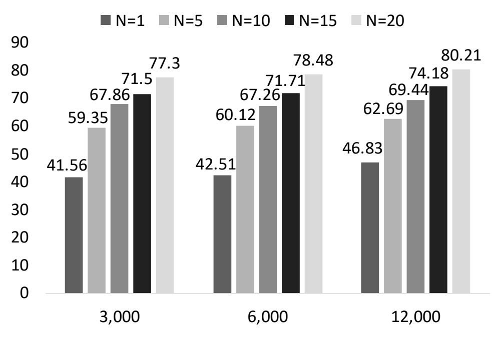
5.2 RQ2: Comparison with the State-of-the-art

We are interested in comparing the performance of our *APIMatchmaker* with that of other state-of-the-art approaches. Specifically, we compare our approach with the state-of-the-art FOCUS approach and a straightforward baseline approach.

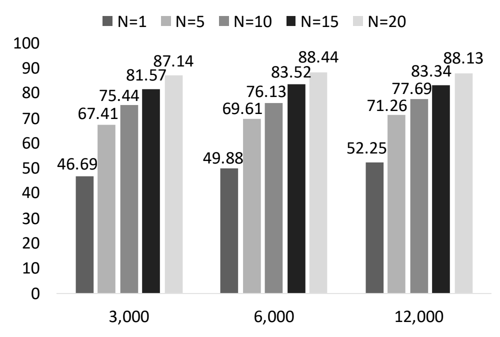
***Baseline 1* - FOCUS:** FOCUS was initially designed to learn from open-source Java projects to recommend APIs for supporting the development. The authors have then extended their work to also support the analysis of closed-source Android apps [12]. In this work, we use their extended version to compare against our *APIMatchmaker* tool.

***Baseline 2*:** In order to further verify the effectiveness of *APIMatchmaker*, we construct a baseline approach based on probability statistics. The baseline approach works as follows:

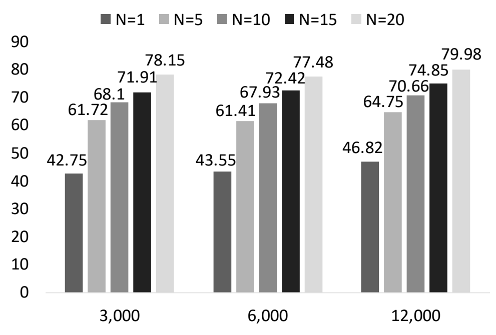
(1) Training Phase: For each app in the training set, we implement a static analyzer and leverage it to parse all the methods declared in the app. In each method, the static analyzer further extracts all its accessed Android APIs,



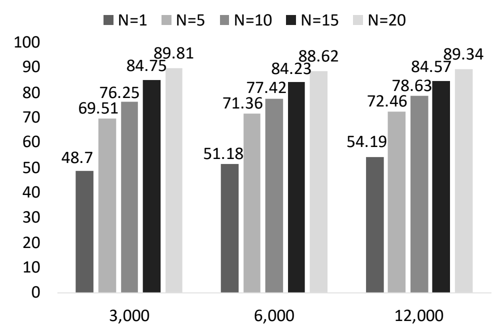
(a) E1



(b) E2

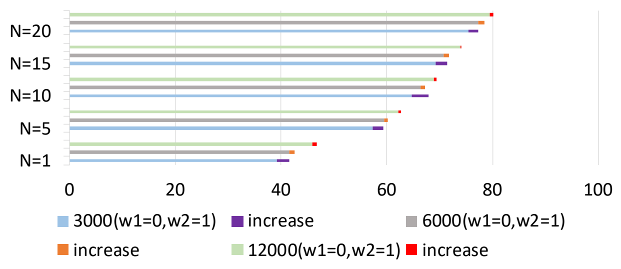


(c) E3

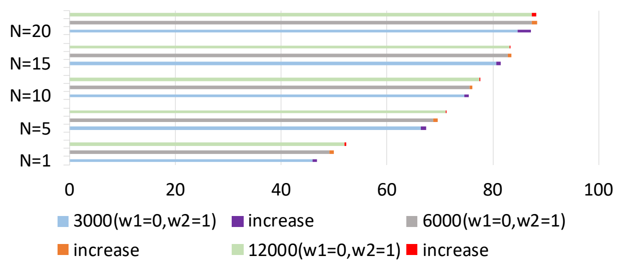


(d) E4

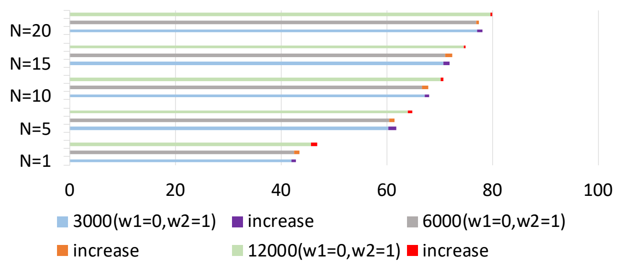
Fig. 6: Experimental results observed by varying the size of the training dataset.



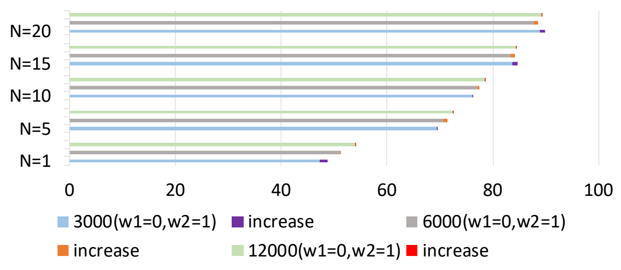
(a) E1



(b) E2



(c) E1



(d) E2

Fig. 7: Success rate increases brought by the app topic context (i.e., the default setting, $w_1 = 0.2, w_2 = 0.8$) compared to the results only observed via code implementation context (i.e., $w_1 = 0, w_2 = 1$).

along with their execution sequences. Then, based on the extracted information (API sequences), inspired by the idea of Function Call Graph raised by McMillan et al. [19], we construct a Weighted API Invocation Graph (WAIG). In WAIG, each Android API is recorded as a node, and API execution sequences are recorded as weighted edges. The weights of edges are simply set based on the number of times API execution sequences appear in the apps of the training set. For instance, suppose there is a method that has accessed two Android APIs (api_1 and api_2), and api_1 is called before api_2. In this example, the two APIs will be included as two nodes (will create new nodes if not

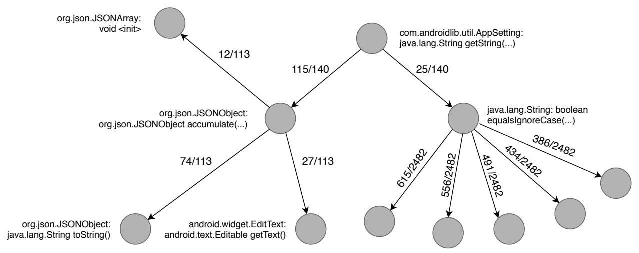

org.json.JSONArray: void <init>
com.androidlib.util.AppSetting: java.lang.String getString(...)
12/113
115/140
25/140
org.json.JSONObject: org.json.JSONObject accumulate(...)
java.lang.String: boolean equalsIgnoreCase(...)
74/113
27/113
615/2482
556/2482
491/2482
434/2482
386/2482
org.json.JSONObject: java.lang.String toString()
android.widget.EditText: android.text.Editable getText()

Fig. 8: An example of the constructed weighted API dependency graph applied to the *Baseline 2* approach.

already existed) in the WAIG. These two nodes will then be connected via a directed edge (i.e., $api_1 \rightarrow api_2$). If the directed edge already exists, there is no need to create the edge anymore but simply increase the weight by one to the existing edge. Fig. 8 portrays a sample WAIG (a sub-graph of the final WAIG built based on the training apps). In this case, "com.androidlib.util.AppSetting:java.lang.String getString()" is followed by "org.json.JSONObject: org.json.JSONObject accumulate()" or "java.lang.String: boolean equalsIgnoreCase()", but the former occurs more frequently, so the weight is higher than the latter.

(2) Testing Phase: Given a method under editing and the APIs calls already written in it, our baseline approach will try to locate the same API execution sequences on the WAIG (built based on a set of training apps). If an exact match cannot be achieved, a similar execution sequence will be considered. Starting from the located sequence (e.g., the last node), the baseline approach will simply take its succeeded N-nodes (with edge's weights taken into account) as the list of APIs for the recommendation.

To enable a fair comparison, we use the same setting to evaluate the effectiveness of FOCUS and this baseline approach: 10-fold cross-validation on the randomly selected 12,000 APKs with the same training and test set applied in each fold. The experimental results (i.e., success rate) of FOCUS and the baseline approach are shown in the last column of Table. 2. Generally, FOCUS and the baseline approach achieve more or less the same performance while *APIMatchmaker* outperforms both of them in all the experimental settings. Recall that FOCUS shares the same collaborative filtering algorithm with our approach, and its outputs, although lower, share the same pattern as well, i.e., the results of E2/E4 are much higher than that of E1/E3. This pattern, however, does not appear in the simple baseline approach, for which the experimental results are more or less the same across all the four settings. If only comparing the results of FOCUS and the baseline approach, FOCUS yields even lower performance when E1/E3 settings (with only one API written) are considered. This evidence further confirms our previous finding that the more knowledge our approach (or similar approach) can learn from the method being editing, the higher performance it can achieve. Moreover, the simple baseline approach could also be a suitable supplement for the complex learning-based approaches, especially when there is no sufficient pre-knowledge to supervise the learning.

Fig. 9 shows the distribution of precision and recall scores of *APIMatchmaker*, FOCUS, and our baseline approach, in the pre-defined four experimental settings (i.e., E1, E2, E3, and E4), respectively. In all experimental settings, both in terms of precision and recall, *APIMatchmaker* outperforms the two baselines. Mann-Whitney-Wilcoxon (MWW) tests additionally confirm that the differences between *APIMatchmaker* and the two counterparts are statistically significant. As highlighted in Table. 3, which summarizes the $p{-}value$s of MWW tests conducted between *APIMatchmaker* and FOCUS, as well as between *APIMatchmaker* and the baseline approach, the $p - value$s are always smaller than *0.005*[4]. The only exceptions are the comparison results in E2, E3 and E4 between *APIMatchmaker* and FOCUS when Top-20 APIs are considered.

RQ2 Answer

Under the same experimental settings, *APIMatchmaker* outperforms both FOCUS (i.e., *Baseline 1*) and *Baseline 2* in achieving significantly higher success rate, precision, and recall for recommending APIs for supporting the development of Android apps.

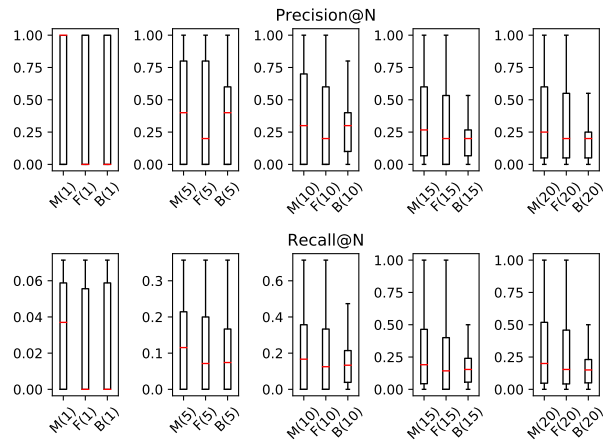
5.3 RQ3: Impact of parameter tuning on *APIMatchmaker*

We now explore the impact of altering the parameter values related to the number of similar apps and methods on the performance of *APIMatchmaker*. To this end, we designed multiple sets of experiments (considering different numbers of most similar projects and methods) to fulfill this objective. Specifically, we set $x = 5, 10, 20$ and $y = 3, 6, 12$. In total, we conduct nine (i.e., $3{*}3$) groups of experiments. The other parameters (i.e., context weights) are kept to the default value pre-configured in *APIMatchmaker*.

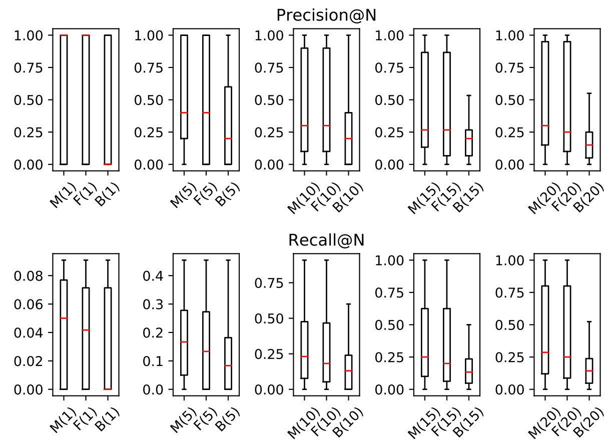
Table 4 presents the experimental results for the pre-defined four settings, respectively. Similar to our previous findings, no matter which group of experiments is concerned, when increasing the number of APIs (i.e., N) to be considered, the performance increases continuously. Furthermore, when looking at the increases in the number of most similar apps (with the same number of most similar methods), or the number of most similar methods (with the same number of most similar apps), in most of the cases, the success rate of *APIMatchmaker* also increases constantly. Indeed, with the largest number of most similar apps and methods ($x = 20$, $y = 12$), *APIMatchmaker* achieves the best performance in all the four settings, giving over 83% of success rate when only one API is known and at least 92% of success rate when four APIs are known.

We further explore the impact of increasing the number of most similar apps and methods. Specifically, given a group of parameters (x, y), we would like to check which of the following two settings (i.e., (2x, y) and (x, 2y)) contributes more to the increase of the performance of *APIMatchmaker*, and the results of (2x, 2y) setting are also analysed for reference. Fig. 10 illustrates the comparison of the improvements brought by the previous three settings:

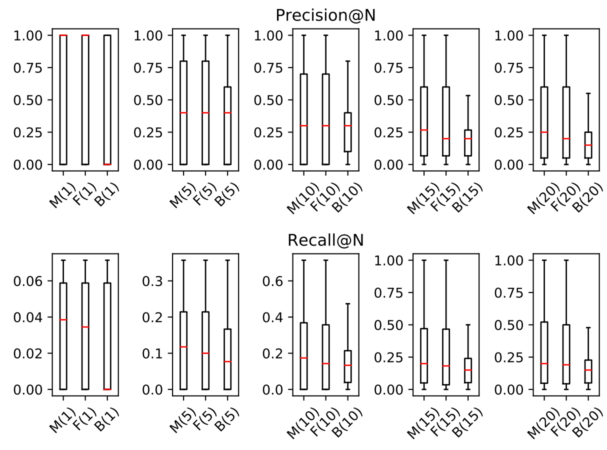
4. Given a significance level $\alpha = 0.005$, if $p - value < \alpha$, there is five chance in a thousand that the difference between the two datasets is due to a coincidence.



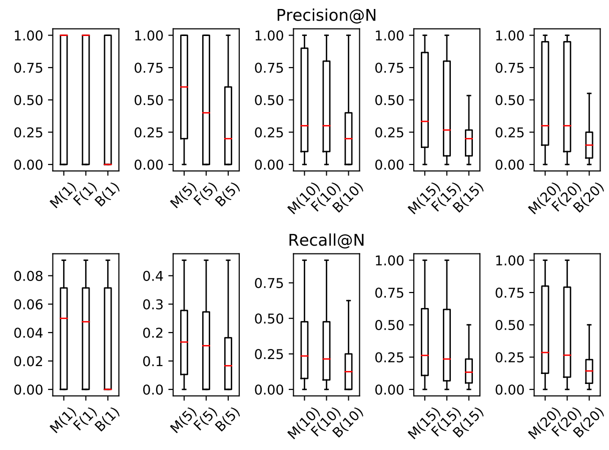
(a) E1



(b) E2



(c) E3



(d) E4

Fig. 9: Distribution of precision and recall for *APIMatchmaker*, FOCUS (*i.e., Baseline 1*) and the *Baseline 2* of our approach.

TABLE 3: The $p - values$ of Mann-Whitney-Wilcoxon Tests on the comparison results of between *APIMatchmaker* and Baseline 1 - FOCUS, as well as between *APIMatchmaker* and Baseline 2.

	E1				**E2**				**E3**				**E4**			
	FOCUS		Baseline		FOCUS		Baseline		FOCUS		Baseline		FOCUS		Baseline	
N	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
1	5.14E-18	6.73E-18	4.68E-4	1.93E-3	5.17E-22	1.26E-22	6.08E-44	1.09E-40	1.04E-6	4.44E-7	6.33E-8	5.17E-7	1.96E-3	5.01E-3	3.85E-51	2.62E-45
5	1.04E-23	1.16E-24	6.64E-60	6.29E-52	8.28E-17	1.65E-18	3.48E-255	2.19E-229	9.97E-8	5.3E-8	5.34E-65	1.45E-55	3.88E-6	2.39E-6	4.41E-290	1.19E-257
10	1.04E-20	1.05E-21	4.41E-73	2.84E-72	1.09E-13	1.18E-14	2.23E-308	2.23E-308	4.24E-7	3.64E-7	1.12E-92	1.3E-90	1.42E-5	1.51E-5	2.23E-308	2.23E-308
15	8.37E-15	1.86E-15	6.41E-85	1.39E-83	3.65E-7	1.71E-7	2.24E-230	6.79E-234	6.19E-6	5.96E-6	1.47E-113	1.98E-112	7.34E-4	7.2E-4	9.81E-263	1.53E-265
20	3.87E-6	2.46E-6	6.29E-51	5.16E-52	1.99E-2	1.84E-2	1.68E-111	4.11E-113	8.04E-3	8.31E-3	1.04E-60	1.577E-61	5.02E-2	5.13E-2	9.25E-120	1.07E-121

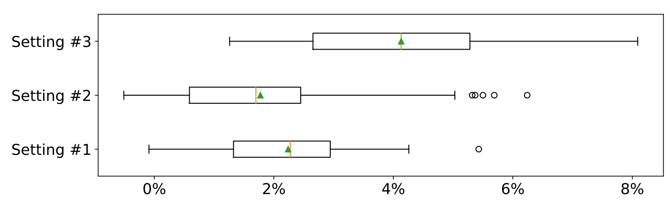


Fig. 10: Distribution of the performance increases by varying the size of most similar apps and methods (i.e., parameters x or y).

Setting #1: Double the number of most similar apps (2x, y), Setting #2: Double the number of most similar methods (x, 2y). Setting #3: Double the number of most similar apps and methods (2x, 2y). The improvements will be the difference between the new results and the original ones, i.e., (2x, y) - (x, y), (x, 2y) - (x, y) and (2x, 2y) - (x, y) respectively. Clearly, the improvements brought by increasing parameter x are more significant than that of increasing parameter y, and the performance of *APIMatchmaker* will be more significantly improved, if both x and y are increased, as indicated by the results of Setting #3. This significance has further been backed up by the MWW testing result.

TABLE 4: Success rate of 10-fold cross-validation on the 12,000 randomly selected apps by varying the size of most similar apps and methods (i.e., parameters x and y).

	E1									E2								
N	**(5, 3)**	**(10, 3)**	**(20, 3)**	**(5, 6)**	**(10, 6)**	**(20, 6)**	**(5, 12)**	**(10, 12)**	**(20, 12)**	**(5, 3)**	**(10, 3)**	**(20, 3)**	**(5, 6)**	**(10, 6)**	**(20, 6)**	**(5, 12)**	**(10, 12)**	**(20, 12)**
1	46.83%	47.68%	50.26%	49.01%	52.68%	54.83%	51.31%	55.19%	59.1%	52.25%	55.02%	55.98%	57.75%	60.34%	61.67%	59.46%	62.27%	64.93%
5	62.69%	65.2%	68.03%	64.49%	68.36%	70.61%	65.02%	69.27%	72.52%	71.26%	74.12%	75.43%	73.78%	77.33%	78.33%	74.11%	77%	79.29%
10	69.44%	71.3%	74.24%	70.38%	73.7%	76.74%	70.5%	74.34%	77.63%	77.69%	80.18%	81.45%	79.49%	82.8%	83.71%	80.16%	82.91%	85.15%
15	74.18%	74.92%	77.73%	74.38%	77.34%	79.92%	74.53%	77.59%	80.51%	83.34%	85.34%	86.53%	84.91%	88.03%	88.26%	85.27%	87.83%	89.61%
20	80.21%	80.31%	81.56%	80.76%	81.85%	83.41%	80.74%	81.48%	83.96%	88.13%	89.4%	90.17%	88.44%	91.61%	91.52%	88.92%	91.71%	92.87%
	E3									**E4**								
N	**(5, 3)**	**(10, 3)**	**(20, 3)**	**(5, 6)**	**(10, 6)**	**(20, 6)**	**(5, 12)**	**(10, 12)**	**(20, 12)**	**(5, 3)**	**(10, 3)**	**(20, 3)**	**(5, 6)**	**(10, 6)**	**(20, 6)**	**(5, 12)**	**(10, 12)**	**(20, 12)**
1	46.82%	49.39%	50.94%	50.29%	54.09%	55.97%	52.01%	55.57%	60.1%	54.19%	55.73%	56.34%	58%	61.1%	62.58%	60.67%	63.54%	65.78%
5	64.75%	66.44%	69.48%	65.24%	69.17%	71.68%	65.99%	69.87%	73.89%	72.46%	74.84%	76.17%	74.19%	78.08%	78.99%	75.88%	78.38%	80.6%
10	70.66%	72.62%	74.9%	71.34%	75.6%	77.03%	71.67%	75.45%	78.83%	78.63%	80.76%	82%	80.36%	83.04%	84.41%	81.65%	83.92%	86.17%
15	74.85%	76.48%	78.53%	75.3%	78.96%	80.29%	75.62%	78.45%	81.48%	84.57%	86.26%	87.19%	85.67%	88.36%	89.18%	87.47%	89.1%	90.74%
20	79.98%	81.23%	82.54%	80.29%	82.59%	83.9%	80.52%	82.12%	84.48%	89.34%	89.66%	90.65%	89.66%	91.64%	92.26%	90.98%	91.97%	93.25%

TABLE 5: Success rate of 10-fold cross-validation on the 12,000 randomly selected apps by varying the weights of the two contexts (i.e., app topic w_1 and code implementation w_2).

	E1										
N	**(0,1)**	**(0.1,0.9)**	**(0.2,0.8)**	**(0.3,0.7)**	**(0.4,0.6)**	**(0.5,0.5)**	**(0.6,0.4)**	**(0.7,0.3)**	**(0.8,0.2)**	**(0.9,0.1)**	**(1,0)**
1	45.89%	46.01%	**46.83%**	46.45%	46.14%	45.74%	44.92%	45.01%	45.65%	44.13%	39.64%
5	62.28%	62.3%	62.69%	62.29%	**63.3%**	63.2%	63.22%	63.12%	62.77%	61.74 %	54.97%
10	68.99%	69.12%	69.44%	**70.35%**	70.21%	70.11%	69.48%	68.74%	68.91%	66.35%	61.72%
15	73.94%	74.02%	**74.18%**	74.16%	74.15%	73.9%	73.73%	74.2%	74.24%	73.4%	67.06%
20	79.45%	79.68%	**80.21%**	80.1%	80%	79.75%	78.34%	78.99%	79.79%	77.35%	72.76%
	E2										
N	**(0,1)**	**(0.1,0.9)**	**(0.2,0.8)**	**(0.3,0.7)**	**(0.4,0.6)**	**(0.5,0.5)**	**(0.6,0.4)**	**(0.7,0.3)**	**(0.8,0.2)**	**(0.9,0.1)**	**(1,0)**
1	51.92%	52.12%	52.25%	52.21%	52.1%	52.46%	**52.52%**	52.15%	52%	51.75%	45.71%
5	71.14%	71.2%	71.26%	71.65%	**71.69%**	71.45%	71.17%	71.03%	70.64%	68.16%	62.33%
10	77.41%	77.51%	**77.69%**	77.49%	77.45%	77.4%	77.5%	77.33%	77.43%	69.76%	69.08%
15	83.13%	83.25%	83.34%	83.3%	83.28%	83.3%	83.36%	83.44%	**83.48%**	82.57%	75.17%
20	87.43%	87.74%	88.13%	**88.14%**	87.95%	87.63%	87%	87.56%	87.48%	86.1%	81.17%
	E3										
N	**(0,1)**	**(0.1,0.9)**	**(0.2,0.8)**	**(0.3,0.7)**	**(0.4,0.6)**	**(0.5,0.5)**	**(0.6,0.4)**	**(0.7,0.3)**	**(0.8,0.2)**	**(0.9,0.1)**	**(1,0)**
1	45.74%	45.89%	46.82%	46.73%	**46.89%**	46.73%	46.54%	46.44%	46.72%	45.16%	39.88%
5	63.9%	64.2%	**64.75%**	64.16%	63.98%	63.47%	62.79%	62.84%	63.65%	62.58%	56.98%
10	70.13%	70.25%	**70.66%**	70.33%	70.12%	70.1%	69.21%	70.12%	70.25%	68.16%	64.09%
15	74.54%	74.68%	74.85%	74.77%	**74.93%**	74.14%	73.58%	74.69%	74.77%	73.26%	68.33%
20	79.69%	78.5%	**79.98%**	79.43%	79.91%	79.21%	78.4%	78.98%	79.24%	77.37%	74.5%
	E4										
N	**(0,1)**	**(0.1,0.9)**	**(0.2,0.8)**	**(0.3,0.7)**	**(0.4,0.6)**	**(0.5,0.5)**	**(0.6,0.4)**	**(0.7,0.3)**	**(0.8,0.2)**	**(0.9,0.1)**	**(1,0)**
1	54.01%	54.12%	54.19%	**54.2%**	53.22%	53.87%	54.07%	53.89%	54.04%	53.74%	46.79%
5	72.43%	72.44%	**72.46%**	72.33%	71.95%	72.15%	72.02%	72.1%	72.18%	70.35%	63.29%
10	78.45%	78.55%	**78.63%**	78.41%	78.21%	78.37%	78.57%	78.46%	78.4%	77.25%	70.35%
15	84.31%	84.44%	**84.57%**	84.41%	84.21%	84.46%	84.52%	84.1%	84.02%	83.2%	77.19%
20	89.14%	89.2%	89.34%	**89.4%**	88.24%	89.17%	88.76%	88.21%	87.91%	86.53%	83.19%

RQ3 Answer

Generally, the more number of most similar apps, or the more number of most similar methods considered, the higher performance *APIMatchmaker* will yield. Nevertheless, the former case (i.e., increasing the number of most similar apps) contributes more to the performance of *APIMatchmaker* than the latter case (i.e., increasing the number of most similar methods).

5.4 RQ4: Effectiveness of the multi-dimensional context-aware collaborative filtering approach

Recall that we provide four parameters (x apps, y methods, (w_1, w_2) context weights) for users to customize the performance of *APIMatchmaker*. The previous section explores the sensitivity of *APIMatchmaker* on the number of similar apps and methods (i.e., x and y) to the final performance of recommending API usages to Android developers. Here we explore the impact of the other two parameters (i.e., w_1, w_2) on the recommendation performance of *APIMatchmaker*. Specifically, we are interested in evaluating the effectiveness of the multi-dimensional context-aware collaborative filtering algorithm applied by *APIMatchmaker*. Specifically, since we only take two-dimensional data (app code and app topic) into consideration, we evaluate the sensitivity of the weights we have assigned to these two dimensions. By default, the weights for app topic and code (w_1, w_2) are set to be (0.2, 0.8). In this work, we compare this default setting with another ten settings formed by altering the weights, i.e., (0, 1), (0.1, 0.9), (0.3, 0.7), (0.4, 0.6), (0.5, 0.5), (0.6, 0.4), (0.7, 0.3) (0.8, 0.2), (0.9, 0.1) and (1, 0). Weights (0, 1) and (1, 0) respectively stand for the cases where only app code or app topic is considered for locating similar methods in the training set. All the other parameters of *APIMatchmaker* are kept the same to ensure a fair comparison.

Table. 5 summarizes the experimental results. As shown in the seventh and thirteenth columns, *APIMatchmaker* yields significantly worse results when only app topic is considered, showing that the app code is a very important dimension for locating similar apps in learning and recommending API usage patterns. Nevertheless, by considering only the app code, *APIMatchmaker* does not achieve the best performance either. Indeed, in almost all the cases, the combination of app topic and code (e.g., (0.2, 0.8), (0.3, 0.7), (0.4, 0.6)) achieves better performance than that of app code alone (0, 1) or app topic alone (1, 0). This empirical evidence shows that our multi-dimensional context-aware collaborative filtering algorithm is indeed effective and useful for learning API usages from large-scale Android apps.

RQ4 Answer

Balancing the weights of two types of context can indeed achieve better performance than considering one type along. This empirical evidence demonstrates that our multi-dimensional context-aware approach is effective and useful in finding the closest apps for learning API usages and subsequently allowing our approach to recommending more suitable APIs for methods under editing.

5.5 Artifact and Data Availability

To foster open science, we have made available online our full implementation and the relevant datasets on the famous open-access repository Zenodo [20], aiming to help readers reproduce our experimental results. We further put the full implementation as an open-source project on Github (i.e., the subsequent updates will hence be also recorded and opened) via the following link.

https://github.com/SMAT-Lab/APIMatchmaker

5.6 Threats to Validity

The validity of our study may have been impacted by several threats below we highlight the key ones.

Threats to External Validity. One of the main threats to external validity lies in the random selection of our training dataset, which may not be generic and representative of all the apps available in the ecosystem. We have attempted to mitigate this threat by selecting more than 10,000 real-world apps that have been all released over the official Google Play store. Additionally, we also evaluate the effectiveness of our approach by varying the size of the training dataset. The fact that there is no significant difference observed by doing that shows that this threat to the external validity of our study could be small. Furthermore, at the moment, we do not take into account app obfuscation in this work, despite it has likely been applied to some of the apps in our training dataset. Nonetheless, since we are mainly interested in learning API usages, which would not be impacted by simple obfuscation strategies such as method renaming, our approach should remain valid and effective. Indeed, as revealed by Dong et al. [21], the majority of Android apps are only obfuscated via simple strategies. Advanced strategies such as altering the code implementation are rarely adopted by real-world Android apps.

We leverage the app description on Google Play to represent the app topic. We consider these descriptions are reliable because they are publicly advertised by the app owners. However, the app descriptions may not be well-written or representative of the actual app topic. Ideally, it would be great if automated approaches could be introduced to validate the app descriptions before applying to our approach. This is nonetheless out of the scope of this work. We take it as our future work.

The implementation of *APIMatchmaker* relies on two static analysis frameworks, Soot [22] and JavaParser[5], for which their reliability issues could propagate to *APIMatchmaker* so as to threaten the validity of our approach. Nevertheless, both Soot and JavaParser are well-known frameworks that have already been shown practical and useful by many of our fellow researchers [23], [24], [25]. We hence believe the threat brought by these two tools should not be significant.

So far, our *APIMatchmaker* only supports API recommendation for such app projects that are developed in Java, although there is nowadays a significant portion of Android apps developed using Kotlin [26], a new programming language introduced by Google to develop Android apps. Nonetheless, since Kotlin is designed to interoperate fully with Java, its syntax is quite similar to that of Java. Hence, it should be relatively straightforward to extend our approach to also support Kotlin-based app projects. We take this as our future work.

Threats to Internal Validity. The main threat is that we employ simulated experimental scenarios for evaluation rather than user studies. Drawing on the related work [11], we mitigate the threat by forming large-scale datasets with apps randomly collected from AndroZoo [13] and adopting 10-fold cross-validation to reduce the impact of contingency. We also attempt to simulate different development scenarios (at an earlier or later stage of the development) and examine the recommended results of *APIMatchmaker* with the ground-truth composed of the real APIs invoked by the method under editing. Nevertheless, we believe user studies are also necessary towards understanding if our approach has fulfilled the actual demand of Android app developers.

Since it is hard to guess and thereby simulate the sequence of methods developed by app developers, in our work, we resort to random selections to prepare the completed methods to fulfill our experiments. However, in the actual situation, the part that the developer has completed is usually related to the method under editing. As a result, the experimental results presented in this work may not reflect the actual capability of *APIMatchmaker* in practice. In order to explore the influence of the order of the methods belonging to the completed part of the test method on the experimental results, we conduct supplementary experiments by comparing the methods selected via a random order with methods selected based on their lexicographic order on the 12,000 apps. The final results suggest that the impact seems to be marginal. Indeed, the $p-value$s of MWW tests are always larger than 0.05, indicating that there is no statistically significant difference between the aforementioned two experiments.

6 Discussion

To the best of our knowledge, *APIMatchmaker* is the first work that combines both code implementation and app topic text to learn and recommend API usage. Unlike most state-of-the-art works, which mainly focus on recommending Android APIs based on known *Java objects*, in *APIMatchmaker* we aim to give suggestions without knowing such *objects* information. Below we discuss the performance sensitivities of our approach with respect to different experimental settings.

5. https://javaparser.org

Impact of the Training Dataset. As experimentally demonstrated in Section 5.1, when varying the training dataset size (e.g., in addition to the data set of 12,000 apps, we conducted two sets of comparative experiments on 6,000 and 3,000 datasets, respectively), the performance of *APIMatchmaker* also varies. Generally speaking, the larger the training dataset, the higher performance *APIMatchmaker* could achieve, as a larger training dataset could potentially allow *APIMatchmaker* to select most similar apps that are even closer to the active project under development than selecting from a small dataset. As also confirmed via experiments, the app descriptions (or topics) are also helpful for identifying the most similar apps. Nevertheless, in this work, we have not evaluated the impact of app qualities (neither the apps' code implementation nor their descriptions) selected for training the model. This, however, is outside of the scope of this work. We hence consider it as our future work.

Impact of the number of known accessed APIs in the active method under editing. Recall that in all the experiments conducted in answering the proposed research questions, we directly follow the settings of FOCUS and consider that the method under editing has already accessed four Android APIs, as described in Scenario 2 of E2 and E4 in Section 4.2. We now explore the influence of the numbers of APIs that have already been accessed by the method under editing on the recommendation performance of *APIMatchmaker*. Ideally, when the number of known accessed APIs increases (as the developer continuously edits the active method), the recommended results should be improved as well, as more information becomes available for training. Towards confirming this hypothesize, we conduct another round of experiments by replicating the E2 and E4 experiments with the number of known APIs changed, e.g., from four to half of the APIs leveraged by the active method. Our experimental results indeed confirm the previous hypothesis, i.e., the precision and recall are improved when half of leveraged APIs (rather than four) are considered. This experimental evidence further demonstrates the usability of our recommendation approach.

On the Importance of taking Android SDK versions into account. We would like to emphasize that the SDK version check of the recommended APIs is critically important since the evolution of the Android framework could cause compatibility issues. For example, API <*SentenceSuggestionsInfo: int getLengthAt(int)*> is added in API level 16 while the min sdk version of app *com.college.theking.christ* is 15. When the above API is recommended for the implementation of the app, compatibility issues will occur if the SDK version supported by the API is not checked and known to the developer. In other words, if the developer does not protect the API with condition checks, when the app runs on an Android device with the SDK version 15, it will crash. In order to provide developers with more help, if such an API appears in our recommended list, *APIMatchmaker* will remove it and inform the developer of the reason for the removal, which is to separately remark it according to the incompatibility compared to the SDK versions of the app currently being developed. As for future work, instead of excluding incompatible APIs, we would like to enhance our recommendation approach to provide better options to resolve this problem, including alternative ways to bypass the compatibility issues (e.g., by recommending its possible replacements).

Learning API usages from more fine-grained similar artifacts. In this work, to fulfill our multi-dimensional context-aware approach for recommending API usages, we have leveraged app descriptions to locate similar apps, aiming at learning API usages from apps that share similar functions with the one under development. Our experimental results have demonstrated that this information is indeed useful for improving the recommendation results. However, similarity analysis at the app level might be too coarse. More fine-grained similarity analysis could further improve the recommendation results. For example, one could leverage the methods' comments to compute method-level similarities when selecting relevant implementations for learning API usages. The methods' comments, however, are only available in source code projects, which are not the focus of this work. We hence regard it as our future work and encourage our fellow researchers to collaboratively explore this research direction.

7 Related Work

We summarize key related work from three aspects, i.e., recommendation in Android and in the field of software engineering, and collaborative filtering applied in software engineering.

7.1 Recommendation in Android Development

Since the development process of mobile apps relies heavily on API frameworks and libraries, in the Android community, researchers have devoted much effort to supporting Android API recommendation to better support mobile app development.

Some works try to give appropriate recommendation suggestions on third-party libraries [27], [28], [29], others knuckle down to the code level, that is, giving real-time suggestions during app development. Hence, many works have been done to extract parameters as recommendations in similar programming scenarios [30], [31]. Except for recommending third-party libraries or parameter values for APIs, there are a large number of researchers focusing on recommending Android APIs and their usage patterns. According to Wu et al. [32]'s defined categories, we introduce state-of-the-art research works related to API recommendations. Most integrated development environments (IDEs) haven been widely equipped with fundamental code completion features, which have been shown effective and useful by developers. Through the basic recommendation features of IDEs, developers can promptly complete an API by typing 'dot' subsequent an object instance to obtain a recommendation list generated based on the static information of the Android app under development.

At present, most of the advanced API recommendation mechanisms predict API usage patterns that are generated together with a given object instance. For example, Nguyen et al. [33] propose a sequence-based tool named DroidAssist to perform code completion for method calls based on Hidden Markov model of API usages (HAPI). They subsequently provide another approach, the key component of which is also HAPI, to train a statistical model of API usage from Dalvik Virtual Machine (DVM) bytecodes [5]. The objective of these approaches is to recommend the next method call as well as a more suitable method sequence. Users are required to provide the object instances being edited as query, which is not the same as the usage scenario of our approach. In our circumstance, even if the programmer does not enter an instance of the object, *APIMatchmaker* can also give suitable recommendations for the method currently under development.

Clustering techniques have also been employed to object usage-based Android API recommendation. Niu et al. [3] mine API usage patterns by clustering the data based on the associations of object usages, i.e., API sequences on a given class, while building the "gold set" manually based on human programming knowledge is time-consuming and a potential threat to the construct validity in terms of both establishment and evaluation. In contrast, *APIMatchmaker* utilizes the context information to extract API usage patterns from similar apps, i.e., the construction is completed without human intervention. Yuan et al. [34] initially focus on the need to recommend event callbacks in the environment of Android application development and introduce an approach to recommend both functional APIs and the event callbacks that need to be overridden. Later on, the authors extend their work by establishing Android-specific API description databases designating the associations among diverse functionalities and APIs [35]. Unlike their work, which is based on structured and fixed databases, *APIMatchmaker* learns from a training set that can be easily adjusted to fulfill different requirements.

Code search, as another research topic in software engineering, has also been leveraged as a means to recommend API usages for Android developers. Indeed, with the objective of code search, Gu et al. [6] introduce an approach to generate API usage sequences based on a natural language query through a deep learning-based approach. Similarly, Jiang et al. [36] generate recommended code snippets based on multi-aspect features, including text, topic, and the number of lines, etc. Different from the aforementioned code search methods, Nie et al. [37] propose an approach leveraging knowledge learned from Stack Overflow to grow the performance of code search algorithms. All the above three works are mainly based on querying natural languages to perform relevant-API recommendations, which are naturally different approaches compared to ours. Nevertheless, we believe those approaches, together with ours, could supplement each other and hence be combined to better serve app developers.

The work most close to ours is the one proposed by Nguyen et al. [11], who introduce a context-aware collaborative filtering based algorithm to recommend Java method invocations utilizing Rascal M^3 model [6]. Later, the authors extend their work by leveraging the algorithm to the Android platform [12]. In our work, we extend the context to multi-dimensions to make full use of other features besides code implementation.

6. https://www.eclipse.org/jdt/core/

7.2 Recommendation in software engineering

Researchers have consecrated many efforts in offering fundamental recommendation features as a primary recommendation for modern IDEs [38], [39], [40]. In recent years, concerning further improve the efficiency of developers, advanced research works based on recommendations are emerging [4], [41], [42], [43], [44], [45], [46], [47]. As another example, Gu et al. [48] present a graph kernel-based approach to the selection of API usage examples by representing source code as object usage graphs.

Mcmillan et al. [19], [49] utilize graph-based matching to retrieve and visualize associated functions and their usages. Chan et al. [50] propose an optimized algorithm to search in an API graph with the given text query phrases. Thung et al. [51] take as input a textual description of a feature request and recommend API methods based on the similarity between textual API descriptions. Rahman et al. [52] exploit the keyword-API association identified by crowd-sourced knowledge of StackOverflow to enrich the translation between natural language query and code search. Raghothaman et al. [53] propose a work to suggest code snippets given API-related natural language queries by learning structured API call sequences from open-source code repositories. Huang et al. [54] propose BIKER, which leverages both Stack Overflow posts and API documentation to extract candidate APIs for ranking with a programming task described in natural language. The above works frequently mention the utilization of text as query phrases. However, in our work, the topic text of apps is uniquely used as a type of context together with code implementation, which is significantly different from the traditional modes of code search with text queries.

7.3 Collaborative filtering applied in software engineering

Collaborative filtering techniques are widely utilized in software engineering in general to implement recommendation systems. To assist developers in taking advantage of available third-party libraries, Thung et al. [55] combine association rule mining and user-based collaborative filtering and propose a technique to recommend likely relevant libraries according to the libraries an application currently uses. Furthermore, Yu et al. [16] introduce an approach that combines collaborative filtering and Latent Dirichlet Allocation (LDA) to provide suggestions about third-party libraries for mobile apps. Moreover, He et al. [29] design a novel approach leveraging Matrix Factorization (a classic collaborative filtering based prediction approach) for predicting useful third-party libraries. Similarly, Xia et al. [56] employ an approach that combines a Matrix Factorization based latent factor model with a neighborhood-based method to capture implicit relations for improving the code reviewer recommendation, which has been acknowledged to be of great importance for software quality assurance. That is because due to the complexity of expertise and availability of candidate reviewers, it can be quite challenging to find appropriate reviewers.

Collaborative filtering techniques have also been introduced to recommend sampling methods to improve the performance of software defect prediction. For example, Sun et al. [57] present a collaborative filtering based sampling method recommendation algorithm to automatically suggest appropriate sampling methods for newly identified defect data.

8 Conclusion

In this paper, we have proposed a novel multi-dimensional context-aware collaborative filtering-based approach, *APIMatchmaker*, for recommending API usages to Android app developers. *APIMatchmaker* initially leverages both pure code implementations and app topics to identify high-level features and subsequently locate similar projects as well as methods that are closest to the active project's method under editing. Then, it utilizes the encoding matrix and rating algorithm to obtain the output of recommended APIs and extracts the code snippets (as API usage samples) from the original APKs files. Our experimental results on large-scale datasets demonstrate that *APIMatchmaker* is effective in learning and recommending API usages for Android developers and is able to outperform both state-of-the-art and our baseline approaches.

As key areas for future work we plan to integrate our approach into Android Studio (the default IDE recommended for app developers to implement Android apps) as a plugin and aim at continuously recommending API usages during the whole development phase of given Android apps. The recommended list of API usages as well as their sample code, will be continuously updated based on the code written by app developers. In addition to recommending API usages for Android projects developed in Java, we also plan to support API recommendation for Kotlin-based Android apps.

Acknowledgements

This work is supported by ARC Laureate Fellowship FL190100035, ARC Discovery Early Career Researcher Award (DECRA) project DE200100016, and a Discovery project DP200100020.

References

[1] M. P. Robillard, "What makes apis hard to learn? answers from developers," *IEEE software*, vol. 26, no. 6, pp. 27–34, 2009.

[2] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2009, pp. 1589–1598.

[3] H. Niu, I. Keivanloo, and Y. Zou, "Api usage pattern recommendation for software development," *Journal of Systems and Software*, vol. 129, pp. 127–139, 2017.

[4] J. Fowkes and C. Sutton, "Parameter-free probabilistic api mining across github," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 254–265.

[5] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Learning api usages from bytecode: a statistical approach," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 416–427.

[6] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 631–642.

[7] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. Le Traon, "Facoy - a code-to-code search engine," in *The 40th International Conference on Software Engineering (ICSE 2018)*, 2018.

[8] H. Cai, Z. Zhang, L. Li, and X. Fu, "A large-scale study of application incompatibilities in android," in *The 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*, 2019.

[9] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "Cid: Automating the detection of api-related compatibility issues in android apps," in *The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*, 2018.

[10] C. C. Aggarwal, M. A. Bhuiyan, and M. Al Hasan, "Frequent pattern mining algorithms: A survey," in *Frequent pattern mining*. Springer, 2014, pp. 19–64.

[11] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, and M. Di Penta, "Focus: A recommender system for mining api function calls and usage patterns," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1050–1060.

[12] P. T. Nguyen, J. Di Rocco, C. Di Sipio, D. Di Ruscio, and M. Di Penta, "Recommending api function calls and code snippets to support software development," *IEEE Transactions on Software Engineering*, 2021.

[13] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 468–471.

[14] S. Arzt, S. Rasthofer, and E. Bodden, "The soot-based toolchain for analyzing android apps," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2017, pp. 13–24.

[15] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *European Conference on Object-Oriented Programming*. Springer, 2009, pp. 318–343.

[16] H. Yu, X. Xia, X. Zhao, and W. Qiu, "Combining collaborative filtering and topic modeling for more accurate android mobile app library recommendation," in *Proceedings of the 9th Asia-Pacific Symposium on Internetware*, 2017, pp. 1–6.

[17] A. Chen, "Context-aware collaborative filtering system: Predicting the user's preference in the ubiquitous computing environment," in *International Symposium on Location-and Context-Awareness*. Springer, 2005, pp. 244–253.

[18] P. Brusilovski, A. Kobsa, and W. Nejdl, *The adaptive web: methods and strategies of web personalization*. Springer Science & Business Media, 2007, vol. 4321.

[19] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 111–120.

[20] APIMatchmaker, *The dataset for APIMatchmaker*, 2021. [Online]. Available: https://zenodo.org/record/5812605

[21] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding android obfuscation techniques: A large-scale investigation in the wild," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2018, pp. 172–192.

[22] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, vol. 15, 2011, p. 35.

[23] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[24] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 280–291.

[25] R. Coppola, M. Morisio, and M. Torchiano, "Evolution and fragilities in scripted gui testing of android applications," in *Proceedings of the 3rd International Workshop on User Interface Test Automation. ACM*, 2017.

[26] P. Liu, L. Li, Y. Zhao, X. Sun, and J. Grundy, "Androzooopen: Collecting large-scale open source android apps for the research community," in *The 2020 International Conference on Mining Software Repositories, Data Track (MSR 2020)*, 2020.

[27] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: fast and accurate detection of third-party libraries in android apps," in *Proceedings*

of the 38th international conference on software engineering companion, 2016, pp. 653–656.

[28] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "Libd: scalable and precise third-party library detection in android markets," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 335–346.

[29] Q. He, B. Li, F. Chen, J. Grundy, X. Xia, and Y. Yang, "Diversified third-party library prediction for mobile app development," *IEEE Transactions on Software Engineering*, 2020.

[30] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques." in *NDSS*, 2016.

[31] L. Li, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Parameter values of android apis: A preliminary study on 100,000 apps," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 584–588.

[32] J. Wu, L. Shen, W. Guo, and W. Zhao, "Code recommendation for android development: how does it work and what can be improved?" *Science China Information Sciences*, vol. 60, no. 9, p. 092111, 2017.

[33] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Recommending api usages for mobile apps with hidden markov model," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 795–800.

[34] W. Yuan, H. H. Nguyen, L. Jiang, and Y. Chen, "Libraryguru: Api recommendation for android developers," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 364–365.

[35] W. Yuan, H. H. Nguyen, L. Jiang, Y. Chen, J. Zhao, and H. Yu, "Api recommendation for event-driven android application development," *Information and Software Technology*, vol. 107, pp. 30–47, 2019.

[36] H. Jiang, L. Nie, Z. Sun, Z. Ren, W. Kong, T. Zhang, and X. Luo, "Rosf: Leveraging information retrieval and supervised learning for recommending code snippets," *IEEE Transactions on Services Computing*, vol. 12, no. 1, pp. 34–46, 2016.

[37] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li, "Query expansion based on crowd knowledge for code search," *IEEE Transactions on Services Computing*, vol. 9, no. 5, pp. 771–783, 2016.

[38] C. Omar, Y. S. Yoon, T. D. LaToza, and B. A. Myers, "Active code completion," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 859–869.

[39] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Effective smart completion for javascript," *Technical Report RC25359*, 2013.

[40] T. Omori, H. Kuwabara, and K. Maruyama, "Improving code completion based on repetitive code completion operations," *Information and Media Technologies*, vol. 10, no. 2, pp. 210–225, 2015.

[41] C. Chen and Z. Xing, "Similartech: automatically recommend analogical libraries across different programming languages," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 834–839.

[42] F. Thung, R. J. Oentaryo, D. Lo, and Y. Tian, "Webapirec: Recommending web apis to software projects via personalized ranking," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 1, no. 3, pp. 145–156, 2017.

[43] H. Zhang, A. Jain, G. Khandelwal, C. Kaushik, S. Ge, and W. Hu, "Bing developer assistant: improving developer productivity by recommending sample code," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 956–961.

[44] S. Azad, P. C. Rigby, and L. Guerrouj, "Generating api call rules from version history and stack overflow posts," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 4, pp. 1–22, 2017.

[45] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig, "Api code recommendation using statistical learning from fine-grained changes," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 511–522.

[46] P. Roos, "Fast and precise statistical code completion," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 757–759.

[47] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How can i use this method?" in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 880–890.

[48] X. Gu, H. Zhang, and S. Kim, "Codekernel: A graph kernel based approach to the selection of api usage examples," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 590–601.

[49] C. Mcmillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu, "Portfolio: Searching for relevant functions and their usages in millions of lines of code," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 4, pp. 1–30, 2013.

[50] W.-K. Chan, H. Cheng, and D. Lo, "Searching connected api subgraph via text phrases," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[51] F. Thung, S. Wang, D. Lo, and J. Lawall, "Automatic recommendation of api methods from feature requests," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 290–300.

[52] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Automatic api recommendation using crowdsourced knowledge," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 349–359.

[53] M. Raghothaman, Y. Wei, and Y. Hamadi, "Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 357–367.

[54] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "Api method recommendation without worrying about the task-api knowledge gap," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 293–304.

[55] F. Thung, D. Lo, and J. Lawall, "Automated library recommendation," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 182–191.

[56] Z. Xia, H. Sun, J. Jiang, X. Wang, and X. Liu, "A hybrid approach to code reviewer recommendation with collaborative filtering," in *2017 6th International Workshop on Software Mining (SoftwareMining)*. IEEE, 2017, pp. 24–31.

[57] Z. Sun, J. Zhang, H. Sun, and X. Zhu, "Collaborative filtering based recommendation of sampling methods for software defect prediction," *Applied Soft Computing*, vol. 90, p. 106163, 2020.