Computer Science and Information Systems 18(4):1219–1238 https://doi.org/10.2298/CSIS210208036H

SimAndro-Plus: On Computing Similarity of Android Applications

Masoud Reyhani Hamedani[1] and Sang-Wook Kim[2,⋆]

[1] Department of Computer Science,
BK21PLUS Program for Advanced AI Research and Education,
Hanyang University, Seoul, Korea, 04763
[2] Department of Computer and Software, Hanyang University,
Seoul, Korea, 04763
{masoud,wook}@hanyang.ac.kr

Abstract. In this paper, we propose *SimAndro-Plus* as an *improved variant* of the state-of-the-art method, SimAndro, to compute the similarity of Android applications (apps) regarding their functionalities. SimAndro-Plus has two *major differences* with SimAndro: 1) it exploits *two* beneficial features to similarity computation, which are totally *disregarded* by SimAndro; 2) to compute the similarity score of an app-pair based on strings and package name features, SimAndro-Plus considers *not* only those terms co-appearing in both apps but *also* considers those terms appearing in one app while missing in the other one. The results of our extensive experiments with three *real-world* datasets and a dataset constructed by human experts demonstrate that 1) each of the two aforementioned differences is really *effective* to achieve better accuracy and 2) SimAndro-Plus *outperforms* SimAndro in similarity computation by 14% in average.

Keywords: android applications, apps data mining, feature extraction, API calls, manifest information, similarity computation

1. Introduction

Android applications (in short, apps) are rapidly growing in the number and variety [5] [17] distributed via the official Google Play Store[3] and other third-party stores such as Amazon App Store[4] and APKPure[5]. Google Play Store contains a *huge* number of apps divided into various categories such as game, communication, and business [11] [24]. As the number of apps in app stores increases dramatically, even if they are divided into various categories, smartphone users face a serious problem to find relevant apps providing their required functionalities [5] [13]. Therefore, there is an important *demand* for app search engines or recommender systems to alleviate this problem where employing an accurate *similarity method* is one of the most challenging issues [13] [15].

In the literature, some methods have been proposed for similarity computation of apps where we aim to find similar apps regarding their functionalities [4–7] [13] [22]. To do

⋆ Corresponding author
[3] http://play.google.com/apps
[4] https://www.amazon.com
[5] https://apkpure.com

this, proposed methods in references [4], [6], [22], SimApp [5], and DNADroid [7] extract the required features from the app stores; these feature might be *inaccurate*, *varied* in different app stores, *unavailable*, affected by *language barrier*, and *unappropriated* for similarity computation. Therefore, exploiting these features may lead us to *inaccurate* similarity computation [13]. On the contrary, SimAndro [13], an effective state-of-the-art method, exploits the features extracted (i.e., mined) from apps and the Android platform itself to compute the similarity of apps. The motivation behind SimAndro is that apps contain *helpful* and *unique* features for similarity computation such as API calls and manifest information that clearly capture the apps functionalities.

In this paper, we propose *SimAndro-Plus* as an *improved variant* of SimAndro to compute the similarity of apps *more* effectively. As SimAndro does, SimAndro-Plus performs feature extraction and similarity computation steps; however, it has *two major differences* with SimAndro in the *both* steps as follows. First, instead of API-method and manifest-complete, SimAndro-Plus exploits *two* new beneficial features to similarity computation named as *API-full-method* and *manifest-partial*, respectively. API-method *implicitly* capture the app's functionalities in some cases; for example, overloaded APIs are regarded as an *identical* feature, while they somehow perform different tasks. However, our API-full-method considers the API's signature (i.e., API's fully qualified name and its parameter list) as a feature, thereby capturing the app's functionalities *explicitly*. Manifest-complete considers the app components (i.e., extracted from the AndroidManifest.xml file) as part of the feature, which are *not* predefined entities in the Android platform and exploiting them in similarity computation may be *misleading*; for example, declaring an activity component by two different apps *cannot* imply the similarity between the two apps since these activity components can be implemented to perform totally different tasks in each of the two apps. However, SimAndro-Plus does *not* consider app components as part of the feature by exploiting the manifest-partial feature. Second, in computing the similarity score between two apps based on their corresponding strings and package name features, SimAndro considers *only* the terms co-appearing in both apps; however, it has been shown that in computing the similarity between two objects of terms (e.g., documents), the number of those terms appearing in one object while missing in the other one is important as well [16]. Therefore, by following [16], SimAndro-Plus considers those terms *appearing* in one app while *missing* in the other one along with those terms co-appearing in both apps. The results of our extensive experiments with three *real-world* datasets and a dataset constructed by *human experts* (i.e., authors) demonstrate that SimAndro-Plus *outperforms* SimAndro.

The contributions of this paper are as follows:

- We extract two new helpful features from apps.
- In computing the similarity based on strings and package name feature, we not only consider those terms appearing in both apps but also consider those terms appearing in only one of the apps.
- We verify the last two contributions help to improve the accuracy of original SimAndro.

The rest of the paper is organized as follows. In Section 2, we briefly explain the existing methods. In Section 3, we present our SimAndro-Plus and its two orthogonal steps. Section 4 explains our experimental setup and analyzes the results of our experiments. In Section 5, we conclude our paper.

2. Related Work

In this section, we explain SimAndro and other existing methods. However, since we mainly focus on SimAndro in this paper, the other methods are explained briefly; the complete explanations about them can be found at [13, Section 2].

Reference [4] proposes a method for app recommendation where the similarity score of an app-pair is computed based on their titles and user comments. Reference [22] proposes a method to invoke users for replacing an already installed app a with a new app b where the similarity score between a and b is computed by exploiting their descriptions. In SimApp [5], the similarity between two apps is computed individually based on multiple features such as description, rating, permissions, and size; then, the obtained individual similarity scores are combined into a single value as the final similarity score. In reference [6], a method is proposed for automatically tagging apps where the similarity score of an app-pair is computed as in SimApp. DNADroid [7] detects app cloning by computing the similarity between apps based on different features such as title, developer, and description. *All* of the aforementioned methods extract the required features from the *app stores*, which might incur the problems of being *inaccurate* (e.g., permission list), *varied* in different app stores (e.g., description and user comment), *unavailable* (e.g., user comment and rating), affected by *language barrier* (e.g., description and user comments), and *unappropriated* for similarity computation (e.g., size and rating); exploiting these features may lead us to inaccurate similarity computation. These methods highly depend on the human explanations and descriptions of apps and *neglect* the useful features that can be *mined* from apps themselves and the Android platform [13].

SimAndro [13] is an effective state-of-the-art method to compute the similarity of apps by exploiting features extracted (i.e., mined) from apps and the Android platform itself; it is an easy-to-understand and straightforward similarity method for apps that can be applied to a wide range of applications such as app search engines, app recommendation, and app clustering. The motivation behind SimAndro is that apps contain *helpful* and *unique* features for similarity computation that clearly capture the apps functionalities without depending on the human explanations or descriptions of apps. SimAndro performs the two orthogonal steps of feature extraction and similarity computation. In the former step, *API-methods*, *manifest-complete*, *strings*, and *package name* are extracted as four different features from the *classes.dex*, *AndroidManifest.xml*, *strings.xml*, and *AndroidManifest.xml* files, respectively. We note that a typical app is an archive file type called Android Package (APK); this file is easily extractable by any archiving software and contain different folders (e.g., assets, lib, and META-INF) and files (e.g., AndroidManifest.xml, classes.dex. and strings.xml) [9] [13]. In the latter step, four similarity scores of an app-pair (a, b) are calculated based on the aforementioned heterogeneous features *separately*. Then, by utilizing TreeRankSVM [1], the weighted linear combination of the above four scores is regarded as the *final* similarity score of (a, b).

3. Proposed Method

Figure 1 illustrates an overview of our SimAndro-Plus. The overall process in both feature extraction and similarity computation steps are *somehow* similar to the ones in SimAndro; however, in order to make the paper self-contained, we briefly explain the two steps in this section along with the two major differences between SimAndro-Plus and its predecessor.

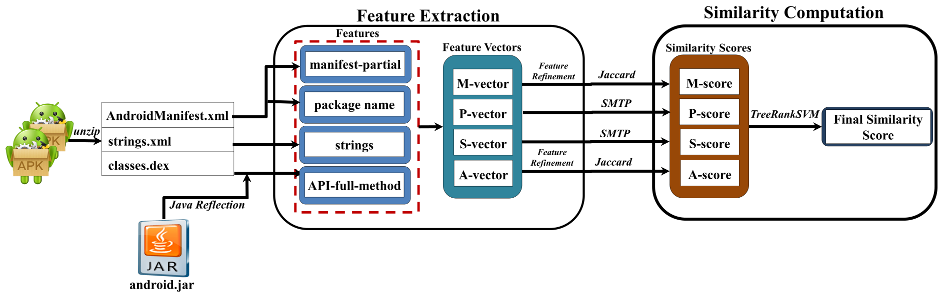


Fig. 1. An overview of SimAndro-Plus

3.1. Feature Extraction

In this step, we extract *API-full-method*, *manifest-partial*, *strings*, and *package name* from apps as four heterogeneous features where API-full-method and manifest-partial are two new features disregarded by SimAndro, while strings and package name are same as the ones exploited by SimAndro.

API-full-method Feature APIs in the Android platform are utilized by apps to interact with the underlying Android system and the device [8] [13]; for example, by calling the "android.os.Handler. removeMessages (int what)" API, an app removes pending messages with code "what" from the message queue. More specifically, API calls can clearly capture the app's behaviors and functionalities [12] [13] [23]. Therefore, *we extract the API calls as a feature to understand what operations an app executes*. SimAndro considers the API's fully qualified name as a feature called API-method (e.g., "android.os.Handler.removeMessages" for the above API). Let us consider the "android.os. Handler.removeMessages (int what, object obj)" API that removes pending messages with code "what" whose object is "obj" from the message queue. Although these two APIs are *different*, they are regarded *identical* by the API-method feature. To solve this problem, SimAndro-Plus exploits a new feature called *API-full-method* that considers the *API's signature* (i.e., API's fully qualified name and its parameter list) instead of only the fully qualified name. As an example, for the two aforementioned APIs, "android.os.Handler.removeMessages (I)"[6] and "android.os.Handler.removeMessages (I, L)" are considered as the API-full-method feature, respectively. API-full-method captures the apps functionalities *more* accurate that API-method since it considers the API's parameter list as the part of the feature. In Section 4.2, we show that API-full-method is *more* beneficial than API-method to similarity computation.

To extract the API-full-method feature, we utilize *both* APK file and Android platform as follows. First, we mine the DEX file via it's different sections such as the *header*, *method_ids*, *string_ids*, *type_ids*, *proto_ids*, and *data*. The method_ids section contains identifiers for all the app's methods; the string_ids section contains identifiers for all the strings (e.g., classes, methods, parameters, etc.) in the app; the type_ids section contains

[6] For simplicity, we use Dalvik symbols to represent parameters.

identifiers for all the types (classes and primitive types) defined by the app; proto_idx contains identifier for the return type and parameters of each method in the app; the header section defines the offset and the size of each of the aforementioned sections. Through the starting address of the method_ids section in the header, we read all entries in the section. Each entry in this section is a *data structure* that contains various kinds of information about a method including an index (*class_idx*) to an offset in the type_ids section, an index (*name_idx*) to an offset in the string_ids section, and an index (*proto_idx*) to an offset in the proto_ids section. The offset pointed by class_idx has an index to another offset in the string_ids section where we obtain the name of the method's owner class. We also extract the name of the method itself through name_idx. The offset pointed by proto_idx has an index to an offset in other list contains number of parameters and their types. For each entry in the method_ids section, we *concatenate* its class name, method name, and parameter list to construct a candidate API-full-method. Then, we apply the Java reflection to the "android.jar" file to obtain a list of all API descriptions in the Android platform; if a candidate API-full-method does *not* belong to this list, we *ignore* it.

Finally, based on the API-full-method feature, an app is represented as a *binary* vector, *A-vector*, where each dimension corresponds to a feature value and the content of a dimension indicates the presence (i.e., value as 1) or absence (i.e., value as 0) of its corresponding feature value in the app [19]. In order to clarify it, suppose that $\{a_1, a_2, ..., a_n\}$ is a set of n distinctive API-full-methods extracted from *all* the apps in a dataset; then, A-vector of app a is represented as $< v'_0, v'_1, ..., v'_{n-1} >$ with n dimensions where $v'_i = 1$ $(0 \leq i \leq n-1)$ if a contains the feature value a_{i+1}; otherwise $v'_i = 0$.

Manifest-partial Feature The AndroidManifest.xml file holds useful *meta* information (i.e., manifest information) about an app such as permissions, hardware/software components, app components (i.e., activity, service, broadcast receiver, and content provider), and intent filters (i.e., action and category); these information supports *both* installation and execution of the app [8] [13]. The permissions are *required* to perform critical tasks such as network access, the hardware/software components indicate either an essential or optional hardware (e.g., GPS) and software (e.g., VoIP) components that the app requires, the activity component implements a task with UI (user interface), the service component implements a background task without UI, the broadcast receiver component enables the app to receive events broadcast by the Android system or other apps, the content provider component supplies data access interface, and intent filters facilitates communication between the app's components and also between different apps. This information *can* capture the app's behaviors and functionalities as API calls do [2] [12] [13]; thus, we extract the manifest information as a feature for similarity computation.

SimAndro considers all the aforementioned information *including* the four app components as a feature called manifest-complete. An app component is defined by *developers* as a subclass of its specific standard class in the Android platform to implement the app's *specific* functionalities. For example, although two activity components c_1 and c_2 from two different apps *a* and *b* are both defined as subclasses of the "android.app.Activity" class, they are developed with their own arbitrary names and *under* specific functionalities of *a* and *b*, respectively; even if c_1 and c_2 are both activity components, they may *not* implement similar functionalities. More specifically, contrary to permissions, hardware/software components, action, and category, app components are *not* predefined entities in the Android platform and are developed independently for each app under the app's specifications, thereby considering them as a feature may provide us *inaccurate* similarity scores. To solve this problem, SimAndro-Plus exploits a new feature called *manifest-partial* where *only* permissions, hardware/software components, action, and category are considered.

Based on the manifest-partial feature, an app is represented as a binary vector, *M-vector*, which is similar to *A-vector*. In Section 4.2, we show that manifest-partial is *more* beneficial than manifest-complete to similarity computation.

Strings and Package Name Features Furthermore, we consider strings and package name as two other features as SimAndro does. The strings.xml file is a single reference for various strings in an app where each string has a *name attribute* as its unique identifier [13, Fig. 3]; we extract both the string and its name attribute since the name attribute also represents some semantic information about the app. As an example, the following line in the strings.xml file of "Weather Forecast", a free app for weather forecasting, defines a string:

$$<string\ name = "weather\_sunny" > Sunny </string>$$

The package name located in the AndroidManifest.xml file is a *unique* identifier for the app and follows Java package naming convention. It is a combination of multiple terms (i.e., simple term or compound one) concatenated by dot and normally provides us abstract information about the app's functionalities; for example, the "weather.widget. weatherforecast" is the package name of the "Weather Forecast" app. For each of these two features, we remove non-alphabetical characters, split compound strings (e.g., weatherforecast), remove stop words, perform stemming, and calculate the TF-IDF score [19] for each term. Finally, based on strings and package name features, an app is represented as two *non-binary* vectors *S-vector* and *P-vector*, respectively, where each dimension corresponds to a term and the content of the dimension is the TF-IDF score of the term.

Feature Refinement In order to obtain better accuracy in similarity computation, we need to perform a feature refinement. The reason is that some of the feature values are widely used in a large number of apps *regardless* of their functionalities, thereby exploiting them in similarity computation leads to *inaccurate* similarity scores. An as examples, consider the two following cases; the "android.os.Message.sendToTarget()" API used by an app to send a message to a specific handler is invoked in more than 90% of apps in our datasets, and the "INTERNET" permission allowing the Internet access is requested by more than 95% of apps in our datasets. We apply a feature refinement similar to the one in SimAndro to our new features, API-full-method and manifest-partial, as follows.

To refine the API-full-method feature with a dataset, we consider a *threshold*, *T*, from 10% to 70% of the dataset size in step of 10% and a feature value is *neglected* if the number of apps in the dataset containing it is *higher* than *T*; in other words, we do *not* consider those feature values that are common among more than *T* of apps. Then, we compute the apps similarity based on *only* the API-full-method feature refined with each value of *T* and compare the accuracy of these seven different cases; the *T* value of the case providing us the better accuracy is selected as the best value of *T*. To refine the manifest-partial feature, we perform the same process.

3.2. Similarity Computation

As explained before, an app is represented by four different vectors as A-vector, M-vector, S-vector, and P-vector corresponding to its API-full-method, manifest-partial, strings, and package name features, respectively. As in SimAndro, to calculate the similarity score of an app-pair (a, b) based on the API-full-method feature, $A\text{-}score(a, b)$, and the manifest-partial feature, $M\text{-}score(a, b)$, we apply Jaccard Coefficient (Jaccard) [19] to corresponding A-vectors and M-vectors of a and b, respectively. In the case of $A\text{-}score(a, b)$, it is calculated as follows:

$$A\text{-}score(a,b) = \frac{\Sigma_i A_i^a \cdot A_i^b}{\Sigma_i A_i^a + \Sigma_i A_i^b - \Sigma_i A_i^a \cdot A_i^b} \tag{1}$$

where A_i^a and A_i^b denote the contents (i.e., 0 or 1) of the i^{th} dimensions in A-vector of a and A-vector of b, respectively.

We note that $M\text{-}score(a, b)$ is also calculated in the same way. We employed Jaccard to calculate these two scores since in the literature, it is a well-known similarity measure widely used to calculate the similarity of binary vectors (i.e., sets) in various topics such as image segmentation [10], document summarization [20], and similarity computation [14].

On contrary to SimAndro, to compute the similarity score of an app-pair (a, b) based on the strings feature, $S\text{-}score(a, b)$, and the package name feature, $P\text{-}score(a, b)$, we apply SMTP (similarity measure for text processing) [16] *instead of* Cosine [19] to corresponding S-vectors and P-vectors of a and b, respectively, for the following reasons. S-vector and P-vector are *non-binary* vectors where each dimension corresponds to a term (i.e., a feature value) and the content of the dimension is set as its weight (i.e., the TF-IDF score). To calculate the similarity between two non-binary vectors, *not* only the proximity between the weights of co-appearing terms in both vectors but *also* the number of those terms *appearing* in one vector while *missing* in the other one is important as well. More specifically, as have been shown in [16], 1) the presence or absence of a term is more important in similarity computation than the difference between the weights of a co-appearing term in both vectors; 2) the similarity score should increase when the difference between the weights of a co-appearing term decreases; 3) the similarity score should decrease when the number of terms appearing in one vector but missing in the other one increases. Let us consider three sample vectors *i*=$<2, 0, 3, 0>$, *j*=$<2, 1, 3, 1>$, and *k*=$<2, 4, 2, 2>$. Although there are two missing terms in *i*, the Cosine similarity score between *i* and *j* (i.e., 0.93) is *higher* than that between *j* and *k* (i.e., 0.78) where there is *not* any missing terms; Cosine does *not* acknowledge the third aforementioned case.

SMTP is an effective measure that considers *all* the three aforementioned cases in similarity computation. To calculate $S\text{-}score(a, b)$, we apply SMTP to the corresponding S-vectors of a and b as follows:

$$S\text{-}score(a,b) = \frac{\frac{\Sigma_i N_*(S_i^a, S_i^b)}{\Sigma_i N_\cup(S_i^a, S_i^b)} + \lambda}{1+\lambda}, \tag{2}$$

$$N_*(S_i^a, S_i^b) = \begin{cases} 0.5 \cdot \left(1 + exp\left(-(\frac{S_i^a - S_i^b}{\sigma_i})^2\right)\right), & S_i^a, S_i^b \neq 0, \sigma_i \neq 0 \\ 0.5, & S_i^a, S_i^b \neq 0, \sigma_i = 0 \\ 0, & S_i^a, S_i^b = 0 \\ -\lambda, & otherwise \end{cases}$$

$$N_\cup(S_i^a, S_i^b) = \begin{cases} 0, & S_i^a, S_i^b = 0 \\ 1, & otherwise \end{cases}$$

where S_i^a and S_i^b denote the weights of the i^{th} terms in S-vector of *a* and S-vector of *b*, respectively. λ denotes a constant and σ_i does the standard deviation of all non-zero weights of the i^{th} term in the dataset. Note that we regard an extra condition "$S_i^a, S_i^b \neq 0, \sigma_i = 0$", which is *not* considered in the SMTP original formulation; if $\sigma_i = 0$, the SMTP definition is incorrect and the similarity score is *undefined* since the division by zero happens. $N_\cup$ sums up the number of terms contributing in similarity computation.

The following three cases are considered through the four conditions in calculating N_*: 1) those terms co-appearing in *both* apps contribute *positively* to the similarity computation where the *amount* of contribution depends on the proximity of their corresponding weights in two apps and their standard deviations in the dataset (i.e., if $S_i^a, S_i^b \neq 0, \sigma_i \neq 0$). when the standard deviation is zero, the amount of contributions is less than the former case (i.e., if $S_i^a, S_i^b \neq 0, \sigma_i = 0$). 2) Those terms missing in *both* apps, do *not* contribute to the similarity computation (i.e., if $S_i^a, S_i^b = 0$). 3) Those terms *appearing* in one app but *missing* in the other one *adversely* affect the similarity score (i.e., fourth condition).

The similarity score of (a, b) based on their package name features, $P\text{-}score(a, b)$, is also calculated in the same way as $S\text{-}score(a, b)$. In Section 4.2, we show that SMTP is *more* beneficial than Cosine to similarity computation of apps. Finally, as in SimAndro, we apply a *weighted* linear combination to combine the four scores into a *single* value as the *final* similarity score of (a, b) as follows:

$$\begin{aligned} S(a, b) = {} & w_1 \cdot A\text{-}score(a, b) + w_2 \cdot M\text{-}score(a, b) \\ & + w_3 \cdot P\text{-}score(a, b) + w_4 \cdot S\text{-}score(a, b) \end{aligned} \quad (3)$$

where w_1, w_2, w_3, and w_4 are weights to control the degree of *importance* of each score in the combination. We automatically find the best value of these four weights by utilizing TreeRankSVM [1] as a machine learning technique; more details can be found in [13, Section 3.3].

It has been shown that instead of considering all the above scores equally significant and simply summing up them into a single value as the final similarity score, applying a weighted linear combination to combine them contributes to obtain better accuracy in similarity computation [13]. We note that it also could be an option to simply combine our four heterogeneous features into a single one (i.e., each app is represented by a single binary vector) and then compute apps similarity based on this single feature; however, it has been shown that considering each of the four heterogeneous features separately in similarity computation is beneficial to obtain better accuracy [13].

3.3. Overall Process: Review

In this section, we present a *simple* review of the overall process required to compute the similarity between two apps as follows.

Feature extraction and refinement First, we use an archiving software (e.g., ark[7]) to unzip all the apps in the dataset. Then, we extract the features for *each* app as follows. We mine the app's classes.dex file through its different sections to extract API-full-method (in the case of SimAndro-Plus) and API-method (in the case of SimAndro); the mining process of the classes.dex file is described in Section 3.1 and [13, Section 3.2.1] in detail. As an example, for the "WhatsApp Messenger" app, we extracted 5,398 feature values for API-full-method and 5,301 features values for API-method. Note that the API-full-method feature has *more* values since it considers the API's parameter list as part of the feature. As an example, "WhatsApp Messenger" calls both of the two following APIs: the "android.media.MediaCodec.releaseOutputBuffer (int index, boolean render)" API is called to return an unnecessary buffer to the codec or to render it on the output surface, while the "android.media.MediaCodec.releaseOutputBuffer (int index, long renderTimestampNs)" API is called to update the surface timestamp of an unnecessary buffer and return it to the codec to render it on the output surface; API-full-method considers *two various* feature values for the above two APIs as "android.media.MediaCodec.releaseOutputBuffer (I, Z)" and "android.media.MediaCodec.releaseOutputBuffer (I, J)", respectively; however, API-method considers an *identical* feature value for both cases as "android.-media.MediaCodec.releaseOutputBuffer". Next, we apply the feature refinement to both API-full-method and API-method features as explained in Section 2.1 where the best values of T are 30% (i.e., refer to Table 2) and 20% (i.e., refer to [13, Table 4]), respectively, with the google dataset. As an example, we have 1,907 and 1,561 feature values for API-full-method and API-method, respectively, with "WhatsApp Messenger" after refining them as its *final* features.

Now, we exploit the AndroidManifest.xml file to extract manifest-partial (in the case of SimAndro-Plus) and manifest-complete (in the case of SimAndro); since this file is in the XML format, the feature extraction is straightforward and not tedious on contrary to that of the classes.dex file. For example, for the "WhatsApp Messenger" app, we extracted 85 and 97 values for manifest-partial and manifest-complete features, respectively. Note that the manifest-partial feature has *less* number of values since it does *not* take into account the app components (i.e., activity, service, broadcast receiver, and content provider). Now, we apply the feature refinement to both manifest-partial and manifest-complete features where the best values of T are 30% (i.e., refer to Table 2) and 20% (i.e., refer to [13, Table 6]), respectively, with the google dataset. As an example, we have 64 and 74 values for manifest-partial and manifest-complete features, respectively, with "WhatsApp Messenger" after refining them as its *final* features.

Next, we extract the package name (e.g., "com.whatsapp" for our sample app), decompound it into its constituent terms (e.g., "com whats app" for above case) by utilizing the Levenshtein algorithms [3], remove non-alphabetical characters and stop words, perform stemming on the terms, and measure the TF-IDF score of each term to obtain the package name feature. Finally, we extract the name attributes and their unique identifiers from the strings.xml file, remove non-alphabetical characters, split the strings, remove stop words including the Java reserved keywords as well, perform stemming on the remaining terms, and calculate the TF-IDF score of each term to obtain the strings feature.

[7] `https://apps.kde.org/ark/`

Automatic weight tuning Each app is represented by four vectors as A-vector, M-vector, S-vector, and P-vector; in the case of SimAndro-Plus, these vectors correspond to the API-full-method, manifest-partial, strings, and package name features of the app, respectively, while in the case of SimAndro, they correspond to the API-method, manifest-complete, strings, and package name features, respectively. Now, we utilize TreeRankSVM to find the best values of w_1, w_2, w_3, and w_4 in Equation (3) automatically as follows; these values are later used to compute the similarity score of any app-pairs. We randomly choose 75% of the apps in the dataset to make a training set where each of the chosen apps is regarded as a *query* app. For *each possible* app-pair (a, q) regarding to a query app q, we make a *hyperplane vector* (see [13, Section 3.3] for more detail) as follows:

$$\{r, qid, A\text{-}score(a, q), M\text{-}score(a, q), P\text{-}score(a, q), S\text{-}score(a, q)\} \quad (4)$$

when r is set as 1 if a is relevant to q (i.e., a belongs to the same category of q), otherwise 0 and qid is a real number started from 1 denoting a query number. For both SimAndro-Plus and SimAndro, $A\text{-}score(a, q)$ and $M\text{-}score(a, q)$ are calculated by applying Jaccard to the corresponding A-vectors and M-vectors of a and q, respectively. For SimAndro-Plus, $P\text{-}score(a, q)$ and $S\text{-}score(a, q)$ are calculated by applying SMTP to the corresponding P-vectors and S-vectors of a and q, respectively; in these two cases, for SimAndro, Cosine is utilized instead of SMTP.

Similarity computation Let us consider two apps a as "WhatsApp Messenger" and b as "TalkU". To compute the similarity score of app-pair (a, b), SimAndro-Plus employs Jaccard to calculate $A\text{-}score(a, b)$ and $M\text{-}score(a, b)$, employs SMTP to calculate $P\text{-}score(a, b)$ and $S\text{-}score(a, b)$, and finally applies the best values of w_1, w_2, w_3, and w_4 (i.e., obtained in the previous step) to Equation (3) to compute $S(a, b)$. SimAndro performs the same process; however, 1) $A\text{-}score(a, b)$ and $M\text{-}score(a, b)$ are calculated based on API-method and manifest-complete, respectively; 2) $P\text{-}score(a, b)$ and $S\text{-}score(a, b)$ are calculated by employing Cosine; 3) consequently, the best values of w_1, w_2, w_3, and w_4 are also obtained by a separate automatic weight tuning than the one performed with SimAndro-Plus.

To compute the similarity score between a *new* app and the existing ones in the dataset, we utilize the already identified values of w_1, w_2, w_3, and w_4. To update these values, we can follow some strategies; for example, if the number of new apps added to the dataset is 25% of the *original* dataset size (i.e., *identical* to our test set for the *last* automatic weight tuning), we perform a *new* automatic weight tuning on the dataset.

4. Evaluation

In this section, we carefully evaluate the effectiveness of our two contributions (i.e., exploiting the two new features and applying SMTP instead of Cosine) and compare the accuracy of SimAndro-Plus with that of SimAndro.

4.1. Experimental Setup

In order to conduct a fair evaluation, we employed the *same* datasets with SimAndro; *google*, *apkpure*, and *amazon* are three *real-world* datasets constructed based on the data

Table 1. Statistics of our datasets

	google	apkpure	amazon	manual
# apps	8903	11068	20570	444
# categories	74	43	204	37

obtained by crawling Google Play Store, APKPure, and Amazon App Store, respectively. We constructed the *manual* dataset by selecting few apps from the three real-world datasets and carefully dividing them into various categories based on their functionalities. Table 1 shows the statistics of our datasets.

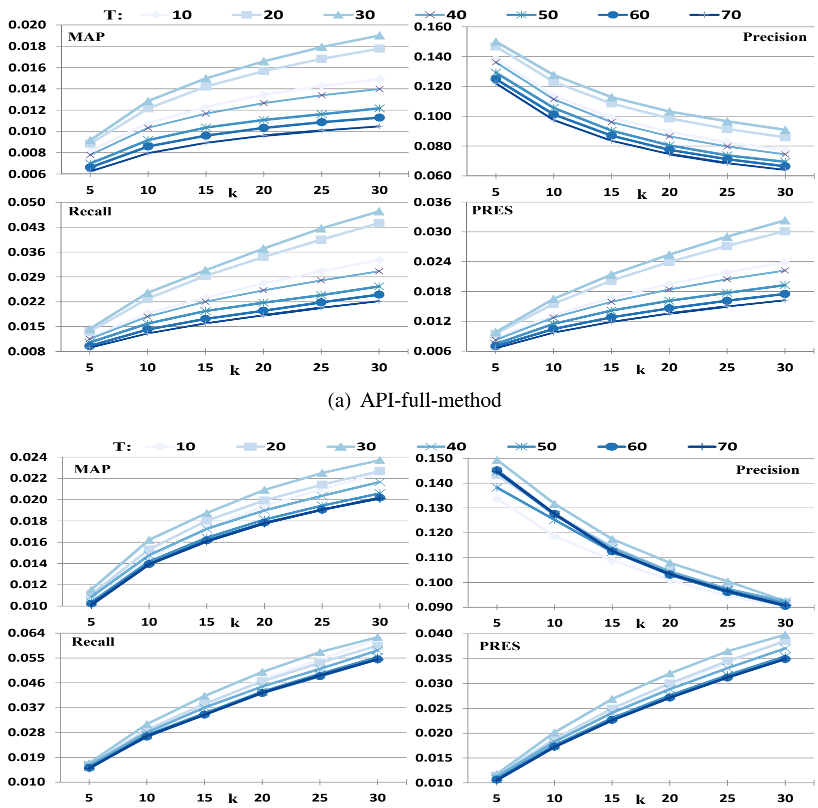
For the manual dataset, we can regard the categories as a ground truth set since the precise categorization is performed by humans expert (i.e., authors). For real-world datasets, their original categories are regarded as the ground truth sets since it is difficult and time-consuming to categorize them by humans expert (i.e., performing user studies); however, in order to conduct accurate and reliable evaluations, we consider a *fine-grained* categorization in our real-world datasets. For example, in our google dataset, the "Tools" category contains six sub-categories as "Alarm", "Flashlight", "Calculator", "Input", "Wi-Fi", and "Recommended"; instead of considering *all* the apps in these six sub-categories under a single category as "Tools", we consider these sub-categories as six *distinct main* categories as "Tools_Alarm", "Tools_Flashlight", "Tools_Calculator", "Tools_Input", "Tools_Wi-Fi", and "Tools_Recommended".

To evaluate the effectiveness, MAP, precision, recall [19], and PRES [18] are utilized as our evaluation metrics. In Equation (2), we set the value of λ as 1 by following [16].

4.2. Results and Analyses

In this section, we refine our new features, compare the effectiveness of applying API-full-method, manifest-partial, and SMTP to similarity computation with those of API-method, manifest-complete, and Cosine, respectively. Finally, we compare the accuracy of SimAndro-Plus with that of SimAndro.

Feature Refinement As explained in Section 3.1, we perform a feature refinement for API-full-method and manifest-partial features with our four datasets through the same process as in SimAndro. Figure 2(a) illustrates the result of our feature refinement for API-full-method with the google dataset on top *k* (*k*=5, 10, 15, 20, 25, 30) results; in the top of the figure, different values of T and their corresponding line patterns are shown (e.g., $T = 30$ and $T = 60$ are represented with triangle and circle marked lines, respectively). As shown in Figure 2(a), the best accuracy in terms of MAP, precision, recall, and PRES is observed when the value of T is set as 30% *regardless* of *k*; by setting T to smaller values than 30% (i.e., 10% and 20%) or to larger values than 30% (i.e., 40%, 50%, 60%, and 70%), we would get lower accuracy. Figure 2(b) illustrates the result of the feature refinement for the manifest-partial feature with the google dataset on top *k* results where the best accuracy is observed when the value of $T = 30\%$ regardless of *k*. Table 2 summarizes the complete results of the feature refinement with all datasets.



(a) API-full-method

(b) Manifest-partial

Fig. 2. Feature refinement with the google dataset.

Table 2. Results of feature refinement

	google	apkpure	amazon	manual
API-full-method	30%	30%	20%	20%
manifest-partial	30%	30%	40%	40%

Effectiveness Comparison of API-full-method and API-method As explained in Section 3.1, we exploit API-full-method instead of API-method, which is one of the major differences between SimAndro-Plus and SimAndro. Now, we compare the effectiveness of API-full-method with that of API-method in similarity computation with our four datasets as follows. For each dataset, we employ the best values of T for API-full-method from Table 2 and for API-method from [13, Table 4]; for example, with the google dataset, we set the best value of T as 30% and 20% for API-full-method and API-method, respectively. Then, with each dataset, we apply Jaccard to compute the similarity of apps by exploiting *only* API-full-method and API-method features *separately*; in other words, we do *not* consider the other three features in similarity computation. Finally, we compare the results of these two similarity computations for each dataset where for simplicity,

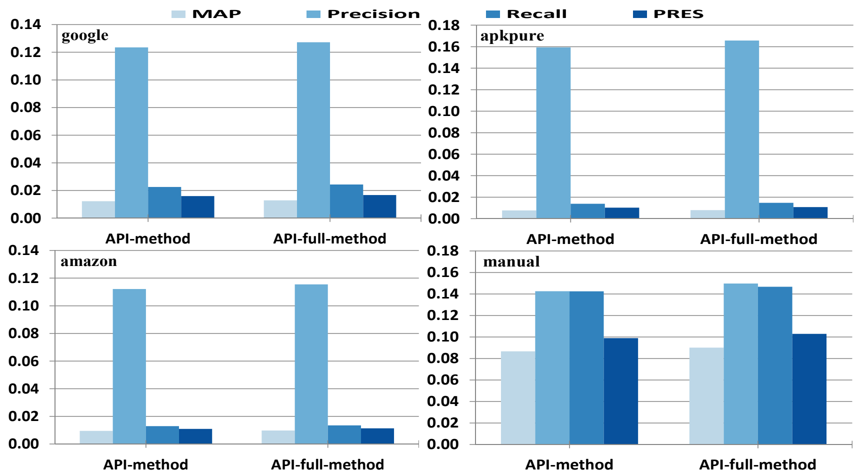


Fig. 3. Accuracy of API-full-method and API-method.

Table 3. Accuracy improvements(%) by API-full-method over API-method

	MAP	precision	recall	PRES
google	5	3	7	5
apkpure	4	4	6	7
amazon	3	3	4	4
manual	4	5	3	4

the effectiveness is considered as the *average* of MAP, precision, recall, and PRES on different values of k[8]. Figure 3 shows the results of this comparison; with *all* datasets, API-full-method shows *better* accuracy in terms of MAP, precision, recall, and PRES since it captures the apps functionalities *more* accurate than API-method by considering the API's signature, while API-method considers only the API's fully qualified name and neglects its parameter list. Table 3 represents the *percentage* of improvements in accuracy obtained by API-full-method over API-method with each dataset.

Effectiveness Comparison of Manifest-partial and Manifest-complete As explained in Section 3.1, SimAndro-Plus exploits the manifest-partial feature, while SimAndro exploits manifest-complete; this is another major difference between SimAndro-Plus and SimAndro. Now, we compare the effectiveness of these two features in similarity computation with our four datasets as follows. We employ the best values of T for manifest-partial from Table 2 and for manifest-complete from [13, Table 6] regarding to the target dataset; for example, with the apkpure dataset, we set the best value of T as 30% and 20% for manifest-partial and manifest-complete, respectively. Then, with each dataset, we apply Jaccard to compute the similarity of apps by exploiting *only* manifest-partial and manifest-complete features separately. Finally, we compare the results of these two

[8] As an example, we compute MAP for k=5, 10, 15, 20, 25, 30; then, the average of these six values is considered as MAP.

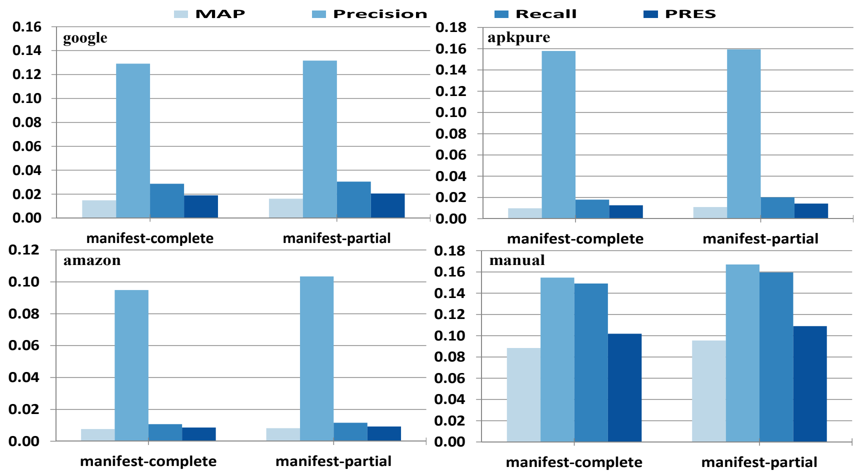


Fig. 4. Accuracy of manifest-partial and manifest-complete.

Table 4. Accuracy improvements(%) by manifest-partial over manifest-complete

	MAP	precision	recall	PRES
google	9	2	6	8
apkpure	11	1	12	12
amazon	7	9	8	7
manual	7	8	7	7

similarity computations for each dataset. Figure 4 illustrates the results of this comparison; with *all* datasets, manifest-partial shows *better* accuracy in similarity computation than manifest-complete in terms of all evaluation metrics. The reason is that manifest-complete considers app components in similarity computation; app components are *not* predefined entities and are developed independently in an app by developers under their own arbitrary names and functionalities, thereby considering them in similarity computation leads to inaccurate similarity scores. Table 4 represents the percentage of improvements in accuracy obtained by manifest-partial over manifest-complete with each dataset.

Effectiveness Comparison of SMTP and Cosine As explained in Section 3.2, SimAndro-Plus applies SMTP instead of Cosine to compute the similarity between two apps based on their strings and package name features. We compare the effectiveness of these two measures in similarity computation as follows. For each dataset, we compute the similarity of apps by applying Cosine and SMTP to *only* each of strings and package name features separately (i.e., four different cases); then for each feature, we compare the results of two similarity computations obtained by employing SMTP and Cosine. Figure 5 illustrates the results of this comparison; in the case of *both* strings and package name features, with *all* datasets, SMTP shows *better* accuracy than Cosine. The reason is that, on contrary to Cosine, SMTP considers not only the terms (i.e., feature values) co-appearing in both apps but also takes into account those terms appearing in one app while missing in the other

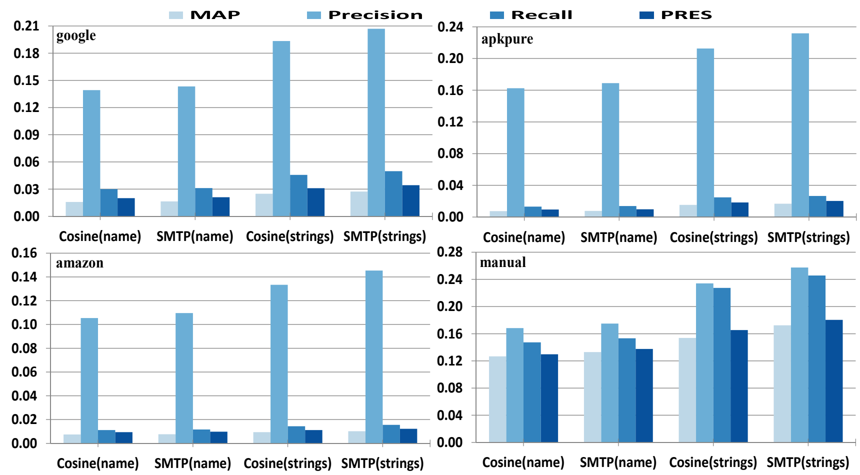


Fig. 5. Accuracy of SMTP and Cosine.

Table 5. Accuracy improvements(%) by SMTP over Cosine

	strings				package name			
	MAP	precision	recall	PRES	MAP	precision	recall	PRES
google	10	7	9	11	4	3	4	5
apkpure	10	9	7	10	4	4	5	3
amazon	9	9	8	9	3	4	4	5
manual	12	10	8	9	5	4	4	6

one. Table 5 represents the percentage of improvements in accuracy obtained by SMTP over Cosine with each dataset for both features; as observed in the table, SMTP shows higher improvements over Cosine with the strings feature than those with the package name feature since the latter feature for an app contains very few number of terms than the former one.

Effectiveness Comparison of SimAndro-Plus and SimAndro As shown in the last three sub-sections, considering API-full-method and manifest-partial as new features instead of API-method and manifest-complete, respectively, are effective; also, applying SMTP instead of Cosine to strings and package name features provides us better accuracy. These results imply that our *both* contributions are beneficial to similarity computation. As shown in reference [13], SimAndro outperforms existing methods proposed in references [5], [4], [22], [6], and [7]; therefore, here, we only compare the accuracy of SimAndro-Plus with that of SimAndro. More specifically, SimAndro-plus exploits API-full-method, manifest-partial, strings, and package name as four features, and applies Jaccard to the first two features and SMTP to the last two ones, while SimAndro exploits API-method, manifest-complete, strings, and package name as four features, and applies Jaccard to the first two features and Cosine to the last two ones. Figure 6 illustrates the results of this comparison with the four datasets; SimAndro-Plus *outperforms* SimAndro in terms of MAP, precision, recall, and PRES with *all* datasets. Table 6 represents the per-

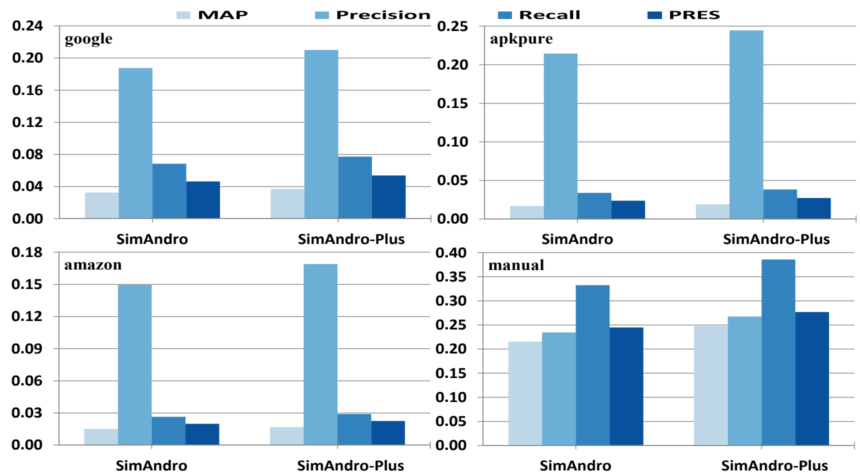


Fig. 6. Accuracy comparison between SimAndro-Plus and SimAndro.

Table 6. Accuracy improvements(%) by SimAndro-Plus over SimAndro

	MAP	precision	recall	PRES
google	14	12	13	16
apkpure	13	14	13	15
amazon	11	13	10	13
manual	15	14	16	13

centage of improvements in accuracy obtained by SimAndro-Plus over SimAndro with each dataset; in average over all datasets, SimAndro-Plus outperforms its predecessor by 14%.

As another evaluation, we perform the same queries in reference [13] by employing SimAndro-Plus and compare their results with those of SimAndro as follows. We consider two well-known apps in the google dataset as "WhatsApp Messenger" with the package name "com.whatsapp" and "Opera Browser" with the package name "com.opera.browser" from categories "Social_Messenger" and "Communication_WebBrowser", respectively. Then, we find out the 10 most similar apps to each of these query apps (i.e., result sets) by applying SimAndro-Plus as the similarity method. Table 7 shows the results where the Relevant column contains ✓sign if the retrieved app is in the same category as the query; otherwise contains ✗ sign. Table 8 borrowed from reference [13] shows the results of the same queries with SimAndro. As shown in both tables, some apps are repeated under different signs in the Relevant column; the reason is that the google dataset assigns multiple categories to some apps. As an example, in the top result set of Table 7, the "Viber" app[9] with the package name "com.viber.voip" is repeated three times where it is marked as *relevant* in rank 8 since it belongs to the same category as the query, and it is marked as *irrelevant* in ranks 9 and 10 since it belongs to other categories than the query's category.

[9] Viber is a cross-platform voice over IP and instant messaging software application provided by Japanese multinational company Rakuten.

Table 7. Result sets obtained by SimAndro-Plus for sample queries

	Rank	Package Name	Category	Relevant
WhatsApp Messenger	1	me.talkyou.app.im	Social_Messenger	✓
	2	me.talkyou.app.im	Communication_Message	✗
	3	kik.android	Social_Messenger	✓
	4	com.bbm	Social_Messenger	✓
	5	com.bbm	Communication_MovieChatting	✗
	6	me.dingtone.app.im	Communication_Message	✗
	7	me.dingtone.app.im	Social_Messenger	✓
	8	com.viber.voip	Social_Messenger	✓
	9	com.viber.voip	Communication_MovieChatting	✗
	10	com.viber.voip	Communication_Message	✗
Opera Browser	1	com.opera.mini.native	Communication_WebBrowser	✓
	2	com.apusapps.browser	Communication_WebBrowser	✓
	3	com.superapps.browser	Personalization	✗
	4	com.fsecure.ms.dc	Tool_Recommended	✗
	5	com.explore.web.browser	Social	✗
	6	com.explore.web.browser	Communication_WebBrowser	✓
	7	com.idotools.browser	Comics	✗
	8	mobi.mgeek.TunnyBrowser	Communication_WebBrowser	✓
	9	org.mozilla.firefox	Communication_WebBrowser	✓
	10	com.chrome.beta	Productivity	✗

As observed by comparing tables 7 and 8, SimAndro-Plus provides us *more* accurate results than SimAndro with the *both* queries. In the case of the first query (i.e., “WhatsApp Messenger”) in Table 7, the “Kik Messenger” app[10] with the package name “kik.android” in rank 3 and the “Viber” app in ranks 8, 9, and 10 are both messenger apps as “WhatsApp Messenger”, while they are absent in the result set obtained by SimAndro in Table 8. In the case of the second query (i.e., “Opera Browser”) in Table 7, the “Firefox Browser” app with the package name “org.mozilla.firefox” in rank 9 is a web browser as “Opera Browser”, while it is absent in the result set obtained by SimAndro in Table 8. More specifically, SimAndro-Plus fetches five similar apps for the both first and second query apps, while SimAndro does three and four similar apps for the first and second query apps, receptively.

5. Conclusions

In this paper, we proposed SimAndro-Plus to effectively compute the similarity of apps. SimAndro-Plus is an improved variant of SimAndro, the state-of-the-art method; however, it has two following major differences with SimAndro. First, SimAndro-Plus ex-

[10] Kik is a freeware instant messaging mobile app from the Canadian company Kik Interactive.

Table 8. Result sets obtained by SimAndro for sample queries

	Rank	Package Name	Category	Relevant
WhatsApp Messenger	1	net.mobileinnova.whatsmon	Tool_Recommended	✗
	2	me.talkyou.app.im	Social_Messenger	✓
	3	me.talkyou.app.im	Communication_Message	✗
	4	com.bbm	Social_Messenger	✓
	5	com.bbm	Communication_MovieChatting	✗
	6	me.dingtone.app.im	Communication_Message	✗
	7	me.dingtone.app.im	Social_Messenger	✓
	8	com.contapps.android	Communication_PhoneNumberBlocking	✗
	9	com.bsb.hike	Social	✗
	10	com.popularapp.fakecall	Productivity	✗
Opera Browser	1	com.opera.mini.native	Communication_WebBrowser	✓
	2	com.apusapps.browser	Communication_WebBrowser	✓
	3	com.fsecure.ms.dc	Tool_Recommended	✗
	4	com.superapps.browser	Personalization	✗
	5	com.explore.web.browser	Social	✗
	6	com.explore.web.browser	Communication_WebBrowser	✓
	7	com.idotools.browser	Comics	✗
	8	nh.smart.opensign	Finance	✗
	9	mobi.mgeek.TunnyBrowser	Communication_WebBrowser	✓
	10	com.chrome.beta	Productivity	✗

ploits two new features as API-full-method and manifest-partial, which are completely disregarded by SimAndro. Second, in similarity computation based on strings and package name features, SimAndro-Plus considers those terms appearing in one app but missing in the other one along with those terms appearing in both apps by employing the SMTP measure instead of Cosine. The results of our extensive experiments with four datasets of apps demonstrated that 1) the both new features are beneficial to similarity computation, 2) employing SMTP provides us better accuracy than Cosine, 3) SimAndro-Plus outperforms SimAndro.

As a future research direction, we plan to investigate the effectiveness of applying SimAndro-Plus to the app recommendation systems. SimAndro-Plus can be regarded as a reasonable solution to address the *item cold start problem* [21] in app recommendation where new released apps (i.e., items) with *no/few* related information in the app store cannot be recommended to the users. The reason is that SimAndro-Plus compute the similarity between apps only based on the features extracted from apps themselves.

Acknowledgments. This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2020-0-01373, Artificial Intelligence Graduate School Program, Hanyang University), the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2020R1A2B5B03001960), and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No.2018R1A5A7059549).

References

1. Airola, A., Pahikkala, T., Salakoski, T.: Training linear ranking svms in linearithmic time using redblack trees. Pattern Recognition Letters 32(9), 1328–1336 (Jul 2011)
2. Arp, D., Spreitzenbarth, M., Gascon, H., Rieck, K.: Drebin: Effective and explainable detection of android malware in your pocket. In: Proc. of NDSS, pp. 1–12 (2014)
3. Backurs, A., Indyk, P.: Edit distance cannot be computed in strongly subquadratic time (unless seth is false). In: Proceedings of the 47th Annual ACM Symposium on Theory of Computing, pp. 51–58 (2015)
4. Bhandari, U., Sugiyama, K., Datta, A., Jindal, R.: Serendipitous recommendation for mobile apps using item-item similarity graph. In: Proc. of AIRS, pp. 440–451 (2013)
5. Chen, N., Hoi, S., Li, S., Xiao, X.: Simapp: A framework for detecting similar mobile applications by online kernel learning. In: Proc. of ACM WSDM, pp. 305–314 (2015)
6. Chen, N., Hoi, S., Li, S., Xiao, X.: Mobile app tagging. In: Proc. of ACM WSDM, pp. 63–72 (2016)
7. Crussell, J., Gibler, C., Chen, H.: Attack of the clones: Detecting cloned applications on android markets. In: Proc. ESORICS, pp. 37–54 (2012)
8. Crussell, J., Gibler, C., Chen, H.: Andarwin: Scalable detection of android application clones based on semantics. IEEE TMC 14(10), 2007–2019 (Oct 2016)
9. Do, Q., Martini, B., Choo, K.K.: Exfiltrating data from android devices. Computers and Security 48(C), 74–91 (Feb 2015)
10. Dutta, B., Shinde, J.: Intuitionistic fuzzy clustering based segmentation of spine mr image. International Research Journal of Engineering and Technology 4(7), 790–794 (July 2017)
11. Faruki, P., Laxmi, V., Bharmal, A., Gaur, M., Ganmoor, V.: Androsimilar: Robust signature for detecting cariants of android malware. Information Security and Applications 22, 66–80 (2015)
12. Feizollah, A., Anuar, N.B., Salleh, R., Abdul Wahab, A.: A review on feature selection in mobile malware detection. Digital Investigation 13(C), 22–37 (Jun 2015)
13. Hamedani, M.R., Gyoosik, K., Seong-je, C.: Simandro: An effective method to compute similarity of android applications. Soft Computing pp. 1–22 (Jan 2019)
14. Hamedani, M.R., Kim, S.w.: Jacsim: An accurate and efficient link-based similarity measure in graphs. Information Sciences 414, 203–224 (November 2017)
15. Hamedani, M.R., Kim, S.W., Kim, D.J.: Simcc: A novel method to consider both content andcitations for computing similarity of scientific papers. Information Sciences 334-335(C), 273–292 (Mar 2016)
16. Lin, Y.S., Jiang, J.Y., Lee, S.J.: A similarity measure for text classification and clustering. IEEE TKDE 26(7), 1575–1589 (Jul 2014)
17. Ma, Z., Ge, H., Liu, Y., Zhao, M., Ma, J.: A combination method for android malware detection based on control flow graphs and machine learning algorithms. IEEE ACCESS 7, 21235–21245 (Feb 2019)
18. Magdy, W., Gareth, J.: Pres: A score metric for evaluating recall-oriented information retrieval applications. In: Proc. of ACM SIGIR, pp. 611–618 (2010)
19. Manning, C., Raghavan, P., Schutze, H.: Introduction to Information Retrieval. Cambridge University Press (2008)
20. Motta, J.M., Ladouceur, J.: A crf machine learning model reinforced by ontological knowledge for document summarization. In: Proceedings of the International Conference Artificial Intelligence, pp. 127–135 (2017)
21. Wei, J., He, J., Kai, C., Zhou, Y., Tang, Z.: Collaborative filtering and deep learning based recommendation system for cold start items. Expert Systems with Applications 69(1), 29–39 (March 2017)
22. Yin, P., Luo, P., Lee, W.C., Wang, M.: App recommendation: A contest between satisfaction and temptation. In: Proc. of ACM WSDM, pp. 395–404 (2013)

23. Zhang, M., Duan, Y., Yin, H., Zhao, Z.: Semantics-aware android malware classification using weighted contextual api dependency graphs. In: Proc. of ACM CCS, pp. 1105–1116 (2014)
24. Zhang, Y., Ren, W., Zhu, T., Ren, Y.: Saas: A situational awareness and analysis system for massive android malware detection. Future Generation Computer Systems 95, 548–559 (Jan 2019)

Masoud Reyhani Hamedani received the B.S. degree in computer science from Shahid Bahonar University, Kerman, Iran, in 2004, and the M.S. degree in software engineering from Payame Nour University, Tehran, Iran, in 2009, and the PhD degree in computer science from Hanyang University, Seoul, Korea in 2016. He worked as a postdoc researcher in Dankook University, Yongin, Korea until February 2018. In March 2018, he joint Hanyang University and currently is working as aresearch assistant professorin the Industry-University Cooperation Foundation, Program for Advanced AI Research and Education. His current research interests include data science, feature representation learning, similarity computation in social network, and deep learning.

Sang-Wook Kim received the B.S. degree in computer engineering from Seoul National University, in 1989, and the M.S. and Ph.D. degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST), in 1991 and 1994, respectively. In 2003, he joined Hanyang University, Seoul, Korea, where he currently is a professor at the Department of Computer Science and Engineering and the director of the Brain-Korea21-Plus research program. He is also leading a National Research Lab (NRL) Project funded by the National Research Foundation since 2015. From 2009 to 2010, he visited the Computer Science Department, Carnegie Mellon University, as a visiting professor. From 1999 to 2000, he worked with the IBM T. J. Watson Research Center, USA, as a postdoc. He also visited the Computer Science Department at Stanford University as a visiting researcher in 1991. He is an author of more than 200 papers in refereed international journals and international conference proceedings. His research interests include databases, data mining, multimedia information retrieval, social network analysis, recommendation, and web data analysis. He is a member of the ACM and the IEEE.

Received: February 08, 2021; Accepted: May 15, 2021.