Hindawi
Wireless Communications and Mobile Computing
Volume 2018, Article ID 1250359, 21 pages
https://doi.org/10.1155/2018/1250359

Research Article

AndroClass: An Effective Method to Classify Android Applications by Applying Deep Neural Networks to Comprehensive Features

Masoud Reyhani Hamedani,[1] **Dongjin Shin**,[2] **Myeonggeon Lee**,[1] **Seong-Je Cho**,[1] **and Changha Hwang**[2]

[1]*Department of Software Science, Department of Computer Science and Engineering, Dankook University, Yongin, Republic of Korea*
[2]*Department of Applied Statistics, Department of Data Science, Dankook University, Yongin, Republic of Korea*

Correspondence should be addressed to Seong-Je Cho; sjcho@dankook.ac.kr

Received 17 May 2018; Revised 12 July 2018; Accepted 25 July 2018; Published 10 September 2018

Academic Editor: Gerardo Canfora

Android application (app) stores contain a *huge* number of apps, which are *manually* classified based on the apps' descriptions into various categories. However, the predefined categories or apps descriptions are usually *not* very accurate to reflect the real functionalities of apps, thereby leading to *misclassify* the apps, which may cause serious *security issues* and *unreliability* problem in the app store. Therefore, the automatic app classification is an *important* demand to construct a *secure*, *reliable*, *integrated*, and *easy to navigate* app store. In this paper, we propose an effective method called *AndroClass* to *automatically* classify apps based on their *real* functionalities by using *rich* and *comprehensive* features representing the *actual* functionalities of the apps. AndroClass performs *three* steps of *feature extraction*, *feature refinement*, and *classification*. In the feature extraction step, we extract 14 various features for each app by utilizing a *unified tool suite*. In the feature refinement step, we apply *Random Forest* algorithm to refine the features. In the classification step, we combine refined features into a *single* one and AndroClass is equipped with K-Nearest Neighbor, Naive Bayes, Support Vector Machine, and Deep Neural Network to classify apps. On the contrary to the existing methods, all the utilized features in AndroClass are *stable* and *clearly* represent the actual functionalities of the app, AndroClass does *not* pose any issues to the *user privacy*, and our method can be applied to classify *unreleased* or *newly released* apps. The results of *extensive* experiments with two *real-world* datasets and a dataset constructed by *human experts* demonstrate the effectiveness of AndroClass where the classification accuracy of AndroClass with the latter dataset is 83.5%.

1. Introduction

Nowadays, smartphones are playing an important role in our lives by providing *rich* functionalities that allow users to perform different activities such as playing games, browsing the Internet, using navigation services, doing online shopping, and checking the bank balance [1–4]. A smartphone is equipped with an operating system such as Android, BlackBerry, iOS, and Windows Mobile, while among these operating systems, Android is the *fastest* growing one [3, 5–8]. Due to the popularity of Android smartphones, Android applications (in short, *apps*) are also rapidly growing in the *number* and *variety* distributed via app stores such as the official Google Play Store (http://play.google.com/apps), Amazon App Store (https://www.amazon.com), and APKPure (https://apkpure.com) [4, 9–11].

In order to develop Android apps, Java programming language is usually used while other languages like C/C++ are also supported. Java code is compiled and converted into one or more Dalvik executable bytecode files (if the total number of methods referenced in an app exceeds 65536, the app is converted to multiple DEX files [12]), which is run on the Dalvik Virtual Machine (DVM) or the Android runtime (ART) implemented for limited resource environments such as smartphones. The structure of a typical app is as follows. *AndroidManifest.xml* is an XML file holds meta information about the app (hereafter, *manifest information*) such as the required security permissions. The *classes.dex* file contains the Dalvik executable bytecode of the app. The *resources.arsc* file contains precompiled application resources in a binary format. The *res* folder contains different resources required by the app to run. The *META-INF* folder contains the app's digital signature and the developer certificate. The *lib* folder contains the compiled code specific to a software layer of a processor. The *assets* folder contains app's assets such as license information and FAQ. All the aforementioned files and folders are packaged into a *single* Android Package (*APK*), which is an archive file type easily extractable by *any* archiving software [1, 6, 13].

In the literature, *significant* attentions have been paid to the topic of malware detection such as the studies in [1, 6, 7, 9, 14–25]. However, very *fewer* efforts have been devoted to the *classification of apps*. Google Play Store contains nearly a million apps providing different functionalities; to *facilitate* browsing and searching of the apps, they are classified into various categories such as tools, music, social media, and browsers [26–29]. This categorization is helpful for *both* users and developers in different aspects. The users can browse the appropriate category to find the *relevant* apps to their needs [27, 30]. In the case of developers, they can decide what *functionalities* they should provide in their own app and determine what common *problems* or *bugs* they should be aware of by considering the other apps belonging to the same category [31, 32].

In the app store, the appropriate category for an app is *manually* selected by the developers or store managers based on the app's description [27, 28, 31]. This is not only a *time-consuming* task [26, 31] but it may also lead to *misclassify* the apps since the predefined categories or descriptions in the store are usually *not* very accurate to reflect the real functionalities of apps [26, 29]. As a result, the manual classification may cause *different* problems in the app stores as follows. First, the serious *security issues* may affect the app store since the classification process can *easily* be manipulated by malware developers to *evade* the malware detection through assigning the malware to an *unrelated* category [26, 27]. Second, users have *difficulties* to find their required apps and also may be persuaded to install the apps or pay for the ones that do *not* provide users' expected functionalities [27]. Therefore, the automatic app classification is an *important* demand to construct an *integrated*, *easy to navigate*, *reliable*, and *secure* app store where the appropriate category is *automatically* assigned to an app based on its *real* functionalities *before* the app is officially released in the store.

We note that a category *defines* the functionalities of its belonging apps in an abstract way [3, 26, 27]; therefore, we need to take advantage of those features that *reflect* the actual functionalities of the apps in classification. As explained before, APK files contain *rich* sources of information such as the AndroidManifest.xml file that holds meta information about the app (e.g., permissions, hardware, and software components) and the DEX file that contains the app's bytecode (holding API calls, etc.). For app classification, the features can be extracted from this very *helpful* and *unique* information; as an example, an app *cannot* send SMS, read contacts, or access camera *without* requesting special permissions and invoking appropriate APIs. In other words, permissions and API calls can *clearly* capture the app's *behaviors* and *functionalities* [1, 22, 24, 33].

In this paper, we propose an effective method called *AndroClass* to *automatically* classify Android apps based on their *real* functionalities by taking advantage of *rich* and *comprehensive* features representing the *actual* functionalities of the apps. AndroClass performs *three* orthogonal steps of *feature extraction*, *feature refinement*, and *classification*. In the feature extraction step, we extract 14 various features for each app containing *API packages*, *API classes*, *API methods*, and *API-method full signatures* from the DEX file; *permissions*, *hardware or software components*, *activities*, *services*, *broadcast receivers*, *content providers*, *actions*, and *categories* from the Manifest.xml file along with *strings* and their *name attribute* from the strings.xml file. However, for simplicity without losing generality, we summarize all these features into the three types as *API calls*, *manifest information*, and *strings*. Consequently, an app is represented as three separate binary vectors, *A-vector*, *M-vector*, and *S-vector*, containing the aforementioned feature types, respectively. For each of the API calls and manifest information, we consider multiple *candidates* and select the one that shows the *best* effectiveness in classification as the *final* representative of the target feature type. Our AndroClass does *not* utilize any third-party tools to extract the features; instead, we developed a *unified tool suite* to mine APK files and the Android platform, thereby obtaining useful information and extracting features from them.

In the feature refinement step, we apply embedded models [34] by utilizing *Random Forest* algorithm [35] to refine the feature values in A-vectors, M-vectors, and S-vectors *separately* and select the feature values that are useful for classification. In the classification step, we combine features that are already refined into a *single* feature; consequently, an app is represented as a single binary feature vector, *F-vector*. Then, we apply *four* well-known classification algorithms as K-Nearest Neighbor (KNN), Naive Bayes (NB), Support Vector Machine (SVM), and Deep Neural Network (DNN) to classify apps into their appropriate categories; to the best of our knowledge, AndroClass is the *first* method which applies DNN to app classification.

On the contrary to the existing methods proposed in [4, 26–29, 36], our employed datasets have a *large number* of *fine-grained* categories, all the utilized features in AndroClass are *stable* and *clearly* represent the actual functionalities of the app, AndroClass does *not* need to access the users' smartphones for feature extraction and thereby does *not* pose any issues to the *user privacy*, our method can be applied to classify *unreleased* or *newly released* apps as well, and AndroClass does *not* utilize existing third-party tools for feature extraction. In Sections 2 and 3.5, we discuss these issues in detail. In addition to app classification, AndroClass can be applied to malware detection and malware classification (i.e., detecting the malware family) as well (we note that the malware detection and malware classification topics are out of the scope of our paper. As a part of our future work, we plan to extensively study and evaluate the effectiveness of applying AndroClass to the aforementioned topics). We demonstrate the *effectiveness* of our AndroClass by conducting *extensive* experiments with two *real-world* datasets, *Google* and *APKPure* (i.e., constructed based on Google Play Store and APKPure, respectively) and a dataset constructed by *human experts* and also employing evaluation measures of accuracy, precision, recall, and F-score [37]. We note that, in this paper, we focus on classification effectiveness than efficiency for the following reasons. First, app stores are seriously suffering from *misclassification* [3]; therefore, in this paper, we propose an automatic classification method with a reasonable effectiveness. Second, it is possible to improve the classification efficiency through different software and hardware solutions such as employing distributed computing techniques or using special machines. Third, constructing the app classification model is *not* a real-time task, and the app store manager does *not* have to reconstruct the model whenever a new app is released in the store; the model can be updated periodically while the previous model is still in use.

The contributions of this paper are summarized as follows:

(i) We propose *AndroClass* to *automatically* classify Android apps based on their *real* functionalities.

(ii) We take advantage of and evaluate 14 *rich* and *comprehensive* features in classification that represent the *actual* behaviors and functionalities of the apps.

(iii) We conduct *extensive* experiments with *three* different datasets of apps each of which contains *fine-grained* categories.

(iv) To the best of our knowledge, AndroClass is the *first* method applies DNN to app classification and employs a datasets constructed by human experts to *carefully* evaluate the effectiveness of classification.

(v) We develop a *unified tool suite* that covers our *entire* requirements in feature extraction without utilizing the existing third-party tools.

2. Related Work

Reference [4] proposes a method to classify apps where strings, permissions, rating, number of rating, and the size of apps are used as features for training different classification algorithms. The size of the employed dataset is very *small* (i.e., only 820 app and 7 categories) (in our manual dataset, we have smaller number of apps (i.e., 364) but bigger number of categories, which means we have tried to carefully evaluate AndroClass with a *fine-grained* categorized dataset). Also, some of the features such as ratings, number of ratings, and the sizes of the apps are *not* suitable for app classification since they do *not* represent the app's functionalities. The ratings and number of ratings reflect the *user satisfaction* (i.e., the *quality* of the provided functionalities by the app) and the number of users who have rated the app, respectively; they do not represent the app's functionalities. This method extracts permissions from *both* AndroidManifest.xml files and the app stores then combine them as a unique feature to be used in classification. However, the permissions obtained from the app store are *not* enough precise to represent the actual requested permissions by an app [19]. This method may have *problem* for classifying *unreleased* or *newly* released apps since there is not any rating or number of rating information about these types of apps in the app stores. LACTA [28] extracts strings by decompiling DEX files as the feature where an app is represented as a document of terms. Then, by applying Latent Dirichlet Allocation (LDA), two matrices as term-topic and topic-software are constructed; the similar topics based on the cosine similarity value are combined. Finally, according to the topic-software matrix, app are divided into the different categories. The size of the employed dataset is very *small* (i.e., only 49 app and 8 categories). This method simply considers an app as a *bag of words* such as a web page or a text document, thereby easily *neglecting* those features that represent the app's functionalities such as the requested permissions and API calls. Furthermore, the TF weighting scheme is utilized, which *cannot* accurately show the importance of a term in an app in comparison to TF-IDF weighting scheme [38]. In [36], the permissions requested by the apps are utilized as the feature and the apps are categorized by applying neural networks (NN). However, the proposed method is actually applied to malware detection where NN provides a probability value as the possibility of classifying the app into a category. If this probability value is larger than a threshold, the app is benign; otherwise it is malware. The employed dataset is very *small* (i.e., 50 apps and 34 categories). In addition, real malware data are *not* utilized; instead, the permissions in some apps are *randomly* changed *without* changing their categories to generate the malicious apps. This method also neglects useful features such as the API calls and strings in app classification.

Reference [29] utilizes web search engines and user smartphones to extract required features for classification. The app name is submitted to a web search engine and the content of the first page showing the results is obtained. Also, the historical context data and the app usage records such as time stamp, time range, profile, battery level, and location are obtained from the device logs. However, analyzing device logs on smartphones is *expensive*, *time-consuming*, and *not* always feasible and may pose issues to the *user privacy* as well. In addition, this method *cannot* be used for classifying unreleased or newly released apps since there is *not* any information about these types of apps in web search engines or users' smartphones. The method also considers the app classification as the problem of classifying simple *text documents* where many useful features such as API calls, permissions, and strings (minded from the apps) are neglected. Also, the utilized features are not *stable* since different search engines may provide us different textual information about an app. Reference [27] proposes a method for app classification where different features such as permissions, hardware components, and API calls are extracted to characterize the behaviors of apps; the best features for classification are selected by applying support vector machine (SVM). Actually, an app is identified as game or nongame one; then, it is classified into the predicted category by utilizing

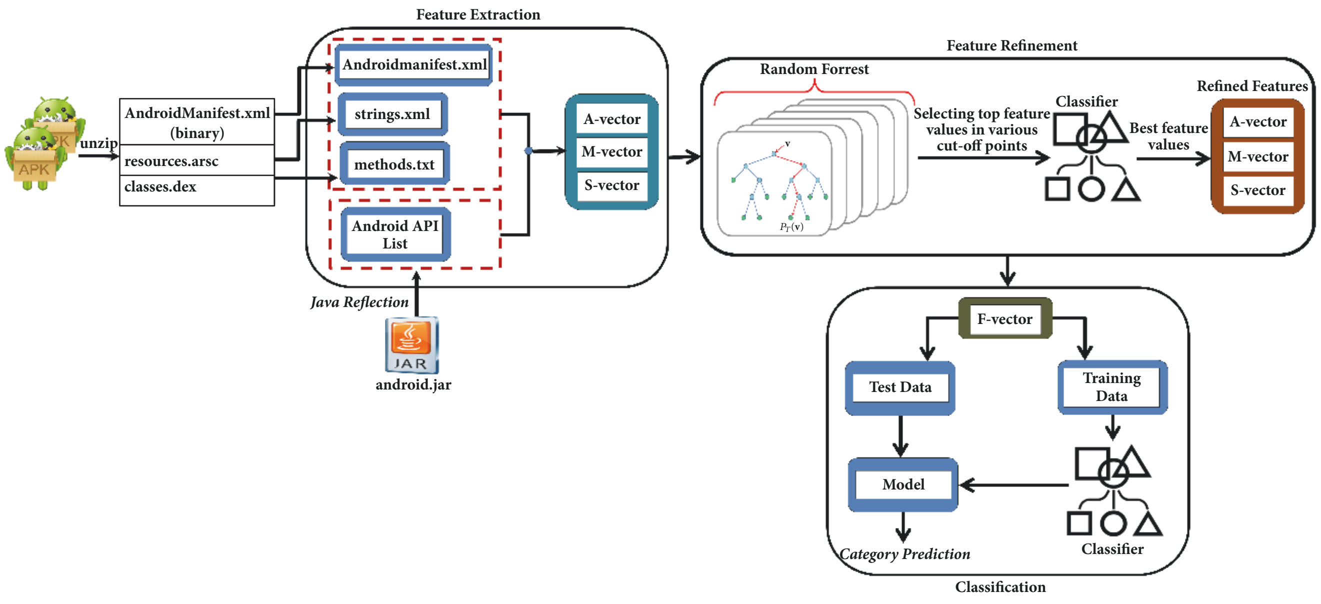


Figure 1: An overview of AndroClass.

different classification algorithms. Finally, the ensemble of multiple classifiers with majority voting is used for the final classification. In this method, SVM is applied to refine the features based on only *two* classes as malicious and benign apps. However, this kind of feature refinement may *not* be suitable for classifying apps based on their functionalities since the features are actually refined for a *different purpose* (i.e., malware detection) *not* for app classification; also, the features are refined based on only *two* classes (i.e., malicious and benign apps) not based on *multiple* classes. In addition, the ensemble of multiple classifiers is *time-consuming* since we have to execute multiple classification algorithms and then perform voting on their predictions. Although the dataset size is large (i.e., 107,327), the number of categories is very *small* (i.e., 24) and categorization is *not* fine-grained that may *bias* the classification effectiveness as explained in Section 4.2.3. ClassifyDroid [26] uses only the method names of API calls as the feature and utilizes semisupervised multinomial Naive Bayes (SMNB) for classification. Although the method names of APIs (i.e., in the case of "android.content.AsyncQueryHandler.removeMessages(int what)" API, the method name is "removeMessages") could represent the behavior of an app, they have some *difficulties* to clarify the actual functionalities of the app. For example, as explained in Section 4.2.2, the aforementioned API and "android.content.AsyncQueryHandler.removeMessages(int what, Object obj)" perform *different* tasks; however, they have an *identical* method name. Furthermore, although the dataset size is large, the number of categories is *small* (i.e., 10 categories) and app classification is considered as classifying simple text documents where other useful features such as permissions and strings are neglected.

3. Proposed Method

This section presents our proposed method, AndroClass, in detail. Section 3.1 starts with an overview of the method. Section 3.2 explains the feature extraction mechanism. Section 3.3 describes the refinement process of features. Section 3.4 explains the classification of apps. Section 3.5 discusses the specifications of AndroClass and the existing methods.

3.1. Overview. As shown in Figure 1, AndroClass performs three steps of feature extraction, feature refinement, and classification. First, the apps are unzipped by an archiving utility [39]. In the feature extraction step, three separate informative files as *AndroidManifest.xml*, *strings.xml*, and *methods.txt* are obtained *directly* from an unzipped APK file (i.e., the app). In addition, *Android API List* is obtained *directly* from the Android platform (i.e., the "android.jar" file provided by the Android SDK) as well. All this information is mined for extracting the API calls, manifest information, and strings as the feature types of an app. Consequently, the app is represented by the three separate vectors as *A-vector*, *M-vector*, and *S-vector* containing the above feature types, respectively.

In the feature refinement step, we refine each of our feature types *separately* as follows. A weight is assigned to each feature value indicating its *importance* by applying the Random Forest algorithm [35]. The effectiveness of the feature type in classification is evaluated with its *top* feature values in *various* cut-off points by utilizing different classification algorithms. Finally, the cut-off point showing the *best* effectiveness (i.e., in terms of accuracy, precision, recall, and F-score) is selected to refine the feature type. In the classification step, the refined features in the previous step are combined into a *single* feature; each app is represented by a single feature vector, *F-vector*. Then, various classification algorithms as KNN [40], NB [41], SVM [42], and DNN [43] are applied to obtain classification models. The appropriate category of an app can be predicted by these models.

3.2. Feature Extraction Step. For feature extraction, we do not use the existing third-party tools such as dedexer, baksmali disassembler, and dex2jar used in [4], [23], and [28], respectively. Instead, we developed a *unified tool suite* for obtaining information from the APK files (i.e., AndroidManifest.xml, classes.dex, and resources.arsc files) and the Android platform and also mining this information in order to extract different features. The reason is that the third-party tools are *not* flexible enough to cover our *entire requirements* (i.e., obtaining the informative files, extracting features, and constructing the feature vectors) for feature extraction.

3.2.1. API Calls. The Android platform provides a collection of APIs, which are utilized by apps to *interact* with the underlying Android system and the smartphones [1, 9, 24]; for example, an app can send SMS by calling "android.telephony.SmsManager.sendTextMessage(L,L,L,L,L) (for simplicity, we use Dalvik symbols to represent variables where the letter *L* indicates a class; for example, the first *L* in this parameter list represents java.lang.String)". More specifically, API calls can *clearly* capture the app's *behaviors* and *functionalities* [1, 22, 24, 33]. We consider *four* possible candidates as *API-package*, *API-class*, *API-method*, and *API-full-method* for API calls feature type where we utilize the API packages (e.g., "android.telephony" in the above example), the API classes (e.g., "android.telephony. SMSManager"), the API methods (e.g., "android.telephony.SmsManager .sendTextMessage"), and the API-full-method signatures (e.g., "android.telephony.SmsManager.sendTextMessage(L, L,L,L,L)"), respectively. In Section 4, by conducting extensive experiments, we evaluate which candidate is *more* beneficial to app classification and select it as the *representative* of the API calls feature type. In order to extract the four possible candidates of the API calls feature type, we utilize the information obtained from *both* the APK file and the Android platform.

First, we obtain the required information from the APK file as follows. The DEX file contains different sections such as a *header*, *method_ids*, *string_ids*, *type_ids*, *proto_ids*, and *data*. The method_ids section contains identifiers for all the methods (i.e., APIs and user-defined methods) referred to the app and does not contain any duplicate entries. The string_ids section contains identifiers for all the strings (e.g., classes, methods, parameters, variables, error messages, and dialogues) used by the app, which is sorted by string contents and does not contain any duplicate entries. The type_ids section contains identifiers for all the types (i.e., classes, arrays, or primitive types) in the app and does not contain any duplicate entries. The proto_ids section contains identifiers for all the prototypes (i.e., parameter list and return type for all methods) and does not contain any duplicate entries. The data section contains all the supported data for the other sections. In the header section, the *offset* and the *size* of each of the aforementioned sections are indicated; therefore, accessing to each section is easily possible [44, 45].

To construct the methods.txt file (a list of all the available methods) of an app, we refer to the app's method_ids section via its starting address in header and read *all* entries in the section until reaching its end. We note that the ending address of the section is easily calculated by adding its starting address to the size of the section, which is indicated in header as well. Each entry in the method_ids section is a data structure that contains various kinds of information about a method including class_idx, name_idx, and proto_idx as indices to offsets in the type_ids section, string_ids section, and proto_ids section, respectively. We extract the method's owner class through class_idx since its pointed offset in the type_section also contains an index to another offset in the string_ids section. Also, we extract the name of the method itself and its parameter list through name_idx and proto_idx, respectively. We concatenate the owner class to the method name and the related parameter list to make the full-method signature and store it in the methods.txt file. Second, we construct Android API List by applying the Java reflection to the "android.jar" file and obtain the descriptions (i.e., the class path, method name, and prototype) of Android APIs.

Now, after obtaining all the required information, we can extract the feature for an app. First, we filter out every app's methods.txt file and neglect any entries that is not an API by using Android API List. Then, *all* the constructed methods.txt files (i.e., each app has its own methods.txt file) are combined together as a *method set* containing *all* the observed methods in the dataset. We note that A-vector of an app representing *any* of the four possible candidates as API-package, API-class, API-method, and API-full-method can be *easily* constructed by comparing the app's methods.txt file with the method set and applying some string manipulation techniques on entries in app's methods.txt file and the method set. For the API calls feature type (i.e., *any* of the four possible candidates), app a 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. More specifically, A-vector of app a is represented as *A-vector(a)*=$< v_0, v_1, \ldots, v_{l-1} >$ with dimensionality l; l is the number of feature values extracted from *all* the apps in the dataset *before* dividing them into training and test instances; $v_i = 1$ $(0 \leq i \leq l-1)$ if a contains the feature value v_i, otherwise $v_i = 0$ [38].

3.2.2. Manifest Information. The AndroidManifest.xml file holds manifest information about the app and provides data that supports *both* the installation and execution of the app as follows [1, 9, 16].

Requested Permissions. All the permissions that the app requires to perform critical tasks should be declared; for example, in order to send an SMS or access the smartphone's information such as IMEI, the app requires to request "SEND_SMS" or "READ_PHONE_STATE" permission, respectively.

Hardware and Software Components. The hardware (e.g., camera) and software (e.g., VoIP) components that the app requires to access should be declared. The component could be *either* an essential one that the app cannot function without it or an optional one that the app prefers to have it but can function without it as well.

App Components. There are four different types of components in an app as *activity*, *service*, *broadcast receiver*, and *content provider*. The activity component implements the UI (user interface) of the apps, which can also have a return value. The service component implements a task without UI that is running as a background service. The broadcast receiver component enables the app to receive events broadcast by the Android system or other apps even when other components of the app are not running. The content provider component supplies an access interface to the data required by the app.

Intent Filters. An intent facilitates communication between the app's components and also between different apps such as starting an activity, starting a service, and delivering a broadcast. The intent filter, which contains different data such as the *action* and *category*, specifies the types of intents that an activity, service, and broadcast receiver can respond to it; for example, a service component is only invoked when it receives the system intent with a specific action.

The aforementioned information in the AnfroidManifest.xml file *can* also capture the app's behaviors and functionalities [1, 9, 22] as API calls do; thus, we extract the manifest information to understand what operations an app executes. Since the AnfroidManifest.xml file is in the binary format, we decompile apps by using Apktool (https://ibotpeaches.github.io/Apktool/) to access the file in the nonbinary format [36]. Then, *all* the extracted manifest information (i.e., each app has its own manifest information) is combined together as a *manifest information set* containing all the observed manifest information in the dataset. We consider *two* possible candidates for the manifest information feature type as *manifest-permission* and *manifest-complete*. In the case of manifest-permission, we utilize *only* the permissions requested by the app. In the case of manifest-complete, we utilize *all* the available information as permissions, hardware and software components, activities, services, broadcast receivers, content providers, actions, and categories. In Section 4, by conducting extensive experiments, we evaluate which candidate is *more* beneficial to classification and select it as the representative of the manifest information feature type. For the manifest information feature type, an app is represented as a binary vector, *M-vector*, where each dimension corresponds to a feature value and the content of a dimension indicates the presence or absence of its corresponding feature value in the app, as in A-vector, which is described in Section 3.2.1. M-vector of an app representing *any* of the two possible candidates can be easily constructed by comparing the app's manifest information with the manifest information set. Note that the dimensionality of each M-vector is identical to the number of feature values extracted from all the apps in the dataset before dividing them into training and test instances.

3.2.3. Strings. The strings contained in an app normally represent its semantic information and describe the app's main functionalities [4, 46] as well as the API calls and manifest information. Therefore, the strings could be regarded as another useful feature for app classification. The strings.xml file is a single reference for various strings with optional text styling and formatting appeared in an app where each string has a *name attribute* as its *unique* identifier; Figure 2 represents a part of this file for the "Weather Forecast" app. We extract both the string and its name attribute since, as we can see in Figure 3, the name attribute also can represent some semantic information about the app. The strings.xml file is located in "/res/values/" folder, which is in the binary format; to access this file, we decompile apps by using Apktool. Then, we remove nonalphabetical characters, split the strings, remove stop words including the Java reserved keywords as well, and perform stemming on the remaining strings. Finally, *all* the extracted strings (i.e., each app has its own strings) are combined together as a *strings set* containing *all* the observed strings in the dataset. For the strings feature type, an app is represented as a binary vector, *S-vector*, which is similar to A-vector and M-vector; S-vector of an app can be easily constructed by comparing the entries in app's strings and the strings set. Note that the dimensionality of S-vector is identical to the number of feature values extracted from all the apps in the dataset before dividing them into training and test instances. To the best of our knowledge, among all the existing studies in both app classification and malware detection topics, AndroClass is the *first* method that takes advantage of the information in the strings.xml file.

3.3. Feature Refinement Step. In real-world applications such as app classification, there are lots of feature values, which are *irrelevant* for classification. Therefore, we have to perform the feature refinement to reduce the dimensionality and select those feature values that are *mostly* relevant for classification, thereby leading to a *better* effectiveness and avoiding the *overfitting problem* [27, 34]. To clarify the issue, consider the two following samples from our Google dataset. The "android.os.Message.sendToTarget()" API that is used by an app to send a message to a specific handler has been called in more than 90% of apps and the "INTERNET" permission that allows an app to open the network sockets and access to Internet has been requested in more than 95% of apps. Therefore, it is *not* effective in considering all the extracted API calls and manifest information in app classification. In the case of strings, we face the same issue as well.

In the feature refinement step, we employ embedded models [34] by utilizing Random Forest (RF) [35]. RF is an ensemble classifier suitable for multiclass classification containing multiple decision trees where each decision tree is independently trained with a randomly selected data. RF internally uses information gain [34] to assign a weight to each feature value as its importance in classification. We apply RF to refine each of API calls, manifest information, and strings feature types, *separately*. As an example, in order to refine the API calls feature type, the apps are considered by their appropriate A-vectors (i.e., only the API calls feature is regarded) where A-vectors can represent *any* of the four candidates API-package, API-class, API-method, or API-full-method. We apply RF to assign the *weight* to each of the existing feature values and select *top* feature values in various *cut-off points* (e.g., 2%, 4%, 6%, 8%, and 10%). Since AndroClass is equipped with different classification algorithms as KNN, NB, SVM, and DNN, we perform feature refinement with *each of* these classification algorithms; the

```xml
<string name="add">Add</string>
<string name="cancel">Cancel</string>
<string name="sunrise">Sunrise:</string>
<string name="sunset">Sunset:</string>
<string name="weather_clear">Clear</string>
<string name="weather_sunny">Sunny</string>
<string name="weather_PartlyCloudy">Partly cloudy</string>
<string name="weather_Cloudy">Cloudy</string>
<string name="weather_Overcast">Overcast</string>
<string name="weather_Mist">Mist</string>
<string name="weather_Patchyrain">Patchy rain</string>
<string name="weather_Lightsnow">Light snow</string>
<string name="weather_sleet">Sleet</string>
<string name="weather_freezingrain">Freezing rain</string>
<string name="weather_thundershower">Thunder shower</string>
<string name="weather_Blizzard">Blizzard</string>
<string name="weather_shower">Shower</string>
<string name="weather_Moderaterain">Moderate rain</string>
<string name="weahter_Heavyrain">Heavy rain</string>
<string name="weather_Moderatesnow">Moderate snow</string>
<string name="weather_Heavysnow">Heavy snow</string>
<string name="weather_snowshowers">Snow showers</string>
<string name="weather_thunderysnow">Thundery snow</string>
<string name="public_time">Public:</string>
<string name="public_time_unknow">Unsynchronized</string>
```

Figure 2: A part of a strings.xml file.

effectiveness of applying the API calls feature type to classification is evaluated in considered cut-off points by utilizing the classification algorithm, and the one showing the *best* effectiveness is selected to refine the feature. More specifically, we select *four* best cut-off points with each of KNN, NB, SVM, and DNN where API-package, API-class, API-method, and API-full-method have their own cut-off point. The same process is applied to refine the manifest information and string feature types.

3.4. Classification Step. In the classification step, first, the features refined in the previous step are combined into a *single* feature. Consequently, an app is represented as a single binary feature vector, *F-vector*, which will be utilized in app classification. Finally, we apply four well-known classification algorithms to classify the apps.

K-Nearest Neighbor (KNN) [40]. KNN is a nonparametric algorithm suitable for both classification and regression. KNN does not make any assumptions on the underlying data distribution, and there is not any explicit training phase. The training data is utilized during the testing phase where the distances between training instances and a test instance are measured; by using majority voting among the k-nearest training instances, the class of the test instance is predicted.

Naive Bayes (NB) [41]. NB is a probabilistic classification algorithm based on Bayes theorem. To estimate probabilities of the appearance of various classes under the condition of the appearance of a test instance by Bayes theorem, joint probabilities of feature values under condition of a class have to be calculated. NB simplifies this calculation with an assumption that feature values are independent under the condition of each class.

Support Vector Machine (SVM) [42]. SVM is an algorithm suitable for both classification and regression. Although SVM is originally designed for binary classification, it is also applicable to multiclass classification via multistage SVM where a maximum-margin optimization problem is solved by finding multiple hyperplanes that provide an ideal separation between training instances belonging to different classes. The margin is defined by the farthest distance between the instances of a class and all the instances belonging to other classes computed based on the distances between the closest instances from opposite sides, which are called supporting vectors. In testing phase, the instances are mapped into the same space to predict the appropriate category of an instance.

Deep Neural Networks (DNNs) [43]. DNNs are learning methods with multiple levels of representation, obtained by composing simple but nonlinear modules where all or most of them are subject to learning, and each transforms the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level. With the composition of enough such transformations, very complex functions can be learned. DNNs discover intricate structure in large and high-dimensional features by using the

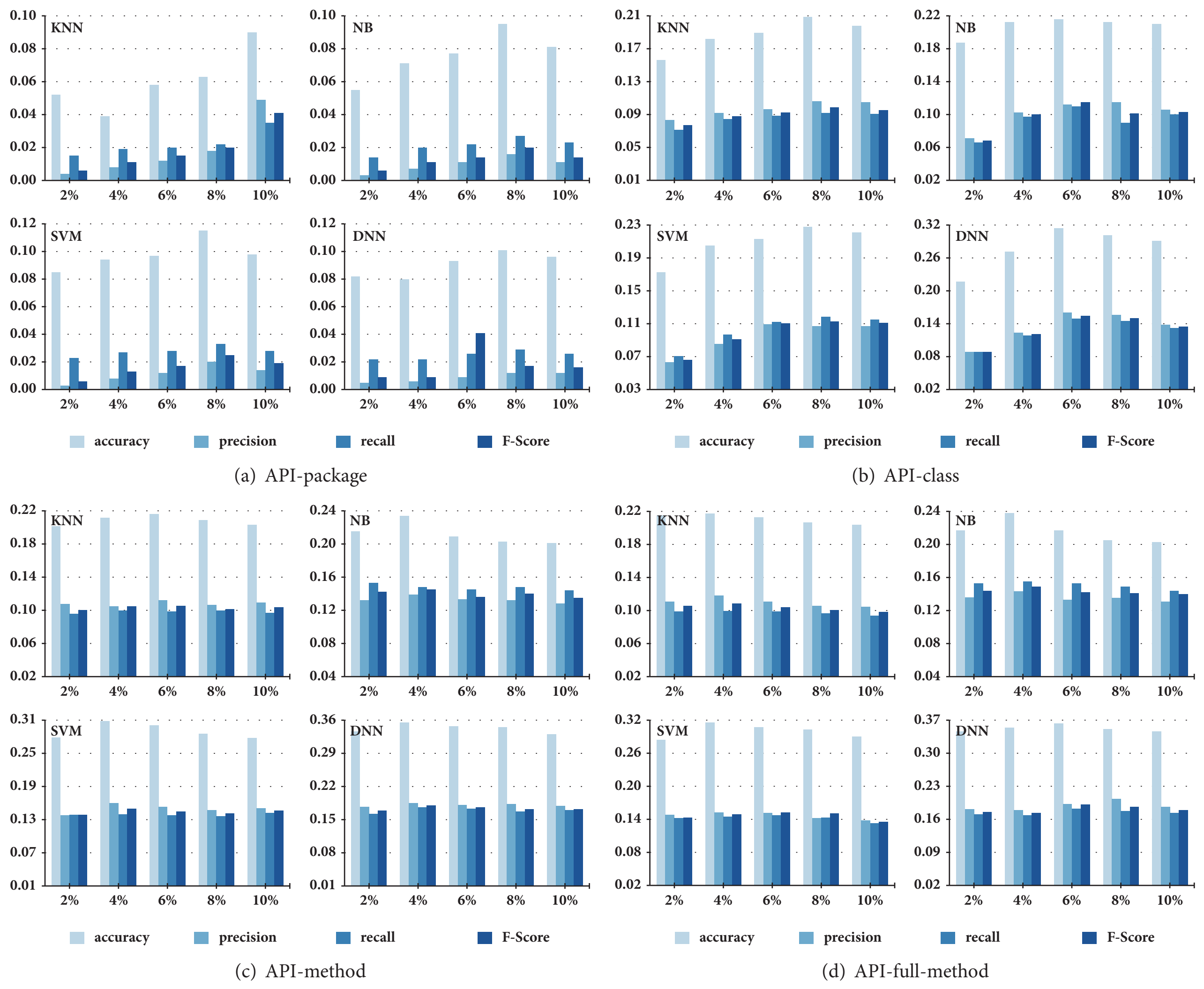


Figure 3: Result of feature refinement for API calls feature type with Google dataset.

backpropagation algorithm to indicate how a machine should change its internal parameters that are used to compute the representation in each layer from the representation in the previous layer. DNNs have dramatically improved the state-of-the-art methods in speech recognition, visual object recognition, object detection, drug discovery, and genomics. To the best of our knowledge, AndroClass is the *first* method which applies DNN to app classification. Although DNN has already been utilized for malware detection and classification such as the proposed methods in [47, 48], these studies applied DNN to *only* malware detection (i.e., classifying apps into *two* categories as malicious or benign) *not* for multiclass classifications (i.e., classifying apps based on their functionalities into the *multiple* categories).

3.5. Discussion. AndroClass is an effective method to automatically classify Android apps based on their real functionalities, which performs three orthogonal steps of feature extraction, feature refinement, and classification. AndroClass has the following advantages over the existing methods explained in Section 2. First, AndroClass utilizes *more* useful features (i.e., 14 features) in app classification than the existing methods do; the number of utilized features by the proposed methods in [4], [28], [36], [29], [27], and [26] are 5, 1, 1, 6, 11, and 1, respectively. Second, there are three main approaches for feature refinement as filter models, wrapper models, and embedded models. The embedded models embed feature selection *with* classifier construction and have the advantages of *both* wrapper models and filter models [34]. The existing methods proposed in [26, 28, 36] do *not* perform any feature refinement process, which is an essential technique to neglect irrelevant feature values and also avoid overfitting problem [27, 34]. The existing methods proposed in [4, 29] utilize filter models for feature refinement by applying information gain and maximum entropy models, respectively. Among existing methods, the proposed method in [27] is the only one utilizes embedded models for feature refinement by applying SVM based on *only* two classes as malicious and benign apps; however, in this method, the feature refinement may *not* be suitable for classifying apps based on their functionalities as explained in Section 2. AndroClass also utilizes embedded models for feature refinement by applying RF; however, on

Table 1: A summary on specifications of AndroClass and existing methods.

	Snaz.	LACTA	Ghorb.	Zhu.	Wang.	ClassifyDroid	AndroClass
Number of features	5	1	1	6	11	1	14
Number of employed datasets	1	1	1	1	1	1	3
Utilizing stable features	√	√	√	✘	√	√	√
Features representing actual fn. of apps	✘	√	√	✘	√	√	√
Posing issues to user privacy	✘	✘	✘	√	✘	✘	✘
Classifying un/newly released apps	✘	√	√	✘	√	√	√

the contrary to the proposed method in [27], in AndroClass, the features are refined for classifying apps into *multiple* classes based on their functionalities.

Third, the strings extracted by decompiling DEX files, which are utilized by proposed methods in [4, 28], may *not* be a suitable feature for app classification since this feature has *difficulties* to overcome the *language barriers problem* among apps. If two apps provide very *similar* functionalities but target users who speak in *different* languages, they contain strings in the target language. For example, both "*ft*fb" and "asos" are online shopping apps in Google Play Store; however, not only their titles but also the strings in these apps are written in Korean and English languages, respectively. On the contrary, in the strings.xml file that is utilized by AndroClass for extracting the strings feature, name attributes are *always* written in English even if the strings themselves are written in the other language (i.e., when the app targets non-English speaking users). More specifically, AndroClass can overcome the language barrier problem. Forth, in the proposed method in [29], the utilized features are *not* stable. In the proposed methods in [4, 29], the utilized features *cannot* clearly represent the actual functionalities of apps. Although utilized features by the proposed methods in [26–28, 36] represent the actual functionalities of apps, AndroClass considers *more* useful features (i.e., 14 features) in app classification than they do. Fifth, all the existing methods utilize only *one* dataset for evaluating the effectiveness of their classification methods; however, we conducted extensive experiments with *three* datasets (i.e., two real-world datasets and one manual dataset) to evaluate the effectiveness of AndroClass.

In Table 1, we summarize the specifications of the exiting methods in comparison with AndroClass by considering the provided explanations in this section and Section 2. In this table, Snaz., Ghorb., Zhu., and Wang. are used for referring to the proposed methods in [4], [36], [29], and [27], respectively, since these methods do not have any indicated titles.

4. Experimental Evaluation

In this section, we carefully evaluate the effectiveness of AndroClass in app classification by employing two real-world datasets of Android apps and a dataset constructed by human experts. Section 4.1 describes our experimental setup. Section 4.2 presents the results and analyses of our experiments.

4.1. Experimental Setup. We employ *three* different datasets for our evaluations; *Google* and *APKPure* are two *real-world* datasets constructed from the data obtained by crawling Google Play Store and APKPure, respectively. We have implemented a crawler to extract the apps, their category names, and their titles from the aforementioned online app stores. Furthermore, since it is *difficult* to evaluate the effectiveness of an app classification method without performing user studies, we constructed a *manual* dataset by selecting a few number of apps from the above real-world datasets and carefully dividing them in various categories based on their *functionalities* where each app assigned only to a *single* category. We did *not* consider the apps' features (i.e., API calls, manifest information, and strings features) for categorizing them. Instead, we installed apps on smartphones and investigated their provided functionalities; then, we assigned apps to the categories based on their functionalities.

In the case of our real-world datasets, we *cannot* perform user studies since it is quite expensive and time-consuming with large datasets. Instead, in order to conduct accurate evaluations, we consider a *fine-grained* categorization in the Google and APKPure datasets. For example, in the Google dataset, there is an original category named "Tools", which contains various subcategories such as "Alarm", "Flashlight", "Calculator", "Input", and "Wi-Fi"; instead of considering a single category as "Tools", we consider *multiple* separate categories as "Tools_Alarm", "Tools_Flashlight", "Tools_Calculator", "Tools_Input", and "Tools_Wi-Fi", respectively, each of which contains its own apps. We note that our employed real-world datasets contain apps that belong to more than one category (i.e., *duplicate* apps). Table 2 shows the statistics of our datasets where notation "#" denotes the number of instances. The #app column denotes the summation of the number of apps in all categories of the dataset.

Two possible techniques to avoid the overfitting problem in classification are feature refinement and cross validation [34]; in addition to feature refinement, we apply the 10-fold cross validation in our experiments as well. Also, we consider 90% of the apps in a dataset as the training instances and 10% of them as the test instances. In order to evaluate the effectiveness, we utilize accuracy, precision, recall, and F-score [37] as our evaluation measures, which are widely used to evaluate the classification methods. Before explaining these measures, we need to define some required terminologies as follows; *positive (P)* instances belong to the main class of interest, *negative (N)* instances belong to other classes, *True Positive (TP)* refers to the correctly labeled positive instances, *True Negative (TN)* refers to the correctly labeled negative instances, *False Positive (FP)* refers to negative instances that are mislabeled as positive, and *False Negative (FN)* refers to

Table 2: Statistics of our datasets.

	#app	#duplicate apps	#categories
Google	8902	505	64
APKPure	11068	371	40
Manual	364	0	12

positive instances that are mislabeled as negative. Now, we can calculate accuracy by the following formulas:

$$accuracy = \frac{TP + TN}{P + N} \quad (1)$$

where $P + N$ is identical to the size of the dataset.

In order to calculate precision, recall, and F-score, we compute these values per each category as follows:

$$precision = \frac{TP}{TP + FP} \quad (2)$$

$$recall = \frac{TP}{P} \quad (3)$$

$$F\text{-}score = \frac{2 \times precision \times recall}{precision + recall} \quad (4)$$

Finally, we take the *average* values of precision, recall, and F-score over *all* categories as the final measures.

We employ scikit-learn [49], which is a free software machine learning library written in Python to utilize RF, KNN, NB, and SVM in our experiments. In the case of DNN, we employ TensorFlow [50], which is an open source software library for numerical computation using data-flow graphs. We apply the following settings to the classification algorithms based on the effectiveness evaluation through several experiments. In the case of RF, the number of trees in the forest is set as 500 for the manual dataset and 1,000 for the Google and APKPure datasets; the number of bootstrapped feature values is set as $P^{0.5}$ where P is the number of feature values; and information gains are used to choose the best split. In the case of KNN, we set the number of neighbors for majority voting as 10 with the manual dataset and 30 with the Google and APKPure datasets (in the case that half of the nearest neighbors to an instance are positive and half of them are negative, we *randomly* select a category among the ones with highest probability). We use NB for multinomial target variables. In the case of SVM, we utilize the linear kernel where the penalty parameter is set as 0.3.

We employ *two* structures for the network in DNN as follows. In the first structure, there are an input layer, first hidden layer, second hidden layer, third hidden layer, and output layer where the number of nodes in these layers is set as P, $P/2$, $P/4$, $P/8$, and C, respectively; C is the number of categories. In the second structure, we have four hidden layers where the number of nodes in input and output layers are identical with the first structure, while the number of nodes in the hidden layer one to the hidden layer four is $P/4$, $P/8$, $P/16$, and $P/16$, respectively. With the manual dataset, we utilize the first structure since the number of values for each available feature is less than 20,000 as shown in Table 3. With Google and APKPure datasets, we utilize the first structure when the number of values for the feature is less than 20,000 (e.g., API-package and API-class); otherwise, we use the second structure. In order to train the hidden layers in both structures, we use stacked autoencoder algorithm [47]. As the activation function between the input layer and the first hidden layer and also between hidden layers, we use hyperbolic tangent function (tanh) [51] and between the last hidden layer and the output layer, we use normalized exponential function (softmax) [51]. The information gain is used as the cost function where we train the models until the cost is less than 0.005 and 0.01 with the first and second structure, respectively. The learning rate is set as 0.001 and 0.0005; also the size of minibatch way is set as 10 and 30 with the first and second structure, respectively.

4.2. Results and Analyses.

4.2. Results and Analyses. In this section, we perform the feature refinement, select the best candidate for API calls and manifest information feature types as described in Section 3.2, and analyze the effectiveness of AndroClass in app classification.

4.2.1. Feature Refinement. Table 3 represents our features and the *original* number of their values (i.e., *before* performing feature refinement) in each dataset (we consider those feature values that are used by at least one app in the dataset.). In the feature refinement step, for *every* dataset, we apply RF to refine each of the four possible candidates of the API calls feature type (i.e., API-package that represents API-package feature, API-class that represents API-class feature, API-method that represents API-method feature, and API-full-method that represents API-full-method signature feature in Table 3), each of the two possible candidates of the manifest information feature type (i.e., manifest-permission that represents permission feature and manifest-complete that represents permission, hardware and software component, activity, service, broadcast receiver, content provider, action, and category features in Table 3), and also the strings feature type (i.e., representing string and name attribute features in Table 3) with KNN, NB, SVM, and DNN, *separately*.

In order to refine the API calls feature type with the Google dataset, the apps are considered by their appropriate A-vectors, which can represent *any* of the four candidates API-package, API-class, API-method, or API-full-method. We apply RF to assign a weight to each of the feature values, which indicates the importance of the feature in classification. The feature values are sorted in the *descent* order based on their weights and the *top* feature values in various cut-off points regarded from 2% to 10% in steps of 2% are selected. Then, we apply each of KNN, NB, SVM, and DNN to classify

Table 3: A summary of original observed features in each dataset.

	Feature	Google	APKPure	Manual
1	API package	167	170	90
2	API class	2,705	3,019	1,408
3	API method	31,264	34,606	15,175
4	API full method signature	35,522	39,310	16,772
5	permission	988	1,011	271
6	hardware and software components	111	124	41
7	activity	15,953	15,131	1,442
8	service	3,092	3,129	425
9	broadcast receiver	2,494	2,588	348
10	content provider	988	1,125	101
11	action	3,358	3,758	682
12	category	241	248	33
13	string	81,008	91,031	12,010
14	name attribute	77,760	80,704	11,810

the apps based on their refined A-vectors in considered cut-off points. For each classification algorithm, the cut-off point that shows the *best* effectiveness is selected to refine the feature. Figure 3 illustrates the result of feature refinement for all the four possible candidates of the API calls feature type with the Google dataset. As an example, KNN shows its best effectiveness in terms of accuracy, precision, recall, and F-score when the cut-off point is set as 10% (since the best cut-off point is 10%, we considered two more cut-off points as 12% and 14%. However, the effectiveness of KNN in these two cut-off points are less than that in cut-off point 10%) in Figure 3(a) and 8% in Figure 3(b). Therefore, with the Google dataset, to refine API-package and API-class for KNN, the cut-off point is set as 10% and 8%, respectively.

The API calls feature type is refined with the APKPure and manual datasets in the same way as with the Google dataset; however, since the number of feature values in the manual dataset are *less* than those in Google and APKPure datasets as shown in Table 3, we set the cut-off points with the manual dataset from 5% to 25% in steps of 5%. Table 4 summarizes the best cut-off points for the API calls feature type with all the three datasets.

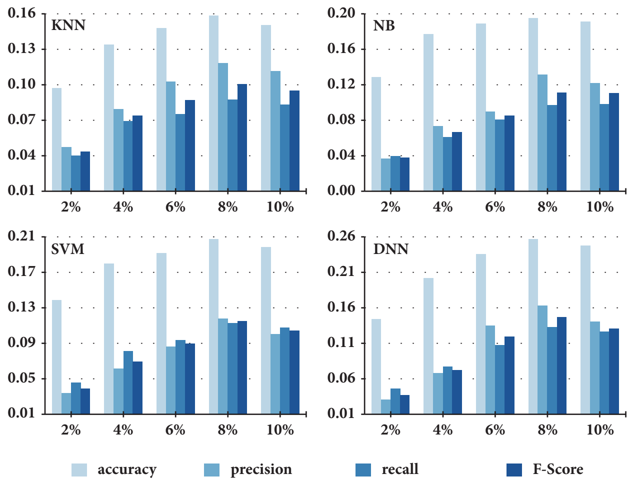
In order to refine the manifest feature type with the Google dataset, the apps are considered by their appropriate M-vectors, which can represent *any* of the two candidates manifest-permission or manifest-complete. We apply RF to assign a weight to each of the feature values, sort them in the descent order based on their weights, and select the top feature values in various cut-off points. Then, we apply each of KNN, NB, SVM, and DNN to classify the apps based on their refined M-vectors in considered cut-off points. For each classification algorithm, the cut-off point that shows the best effectiveness is selected to refine the feature. Figure 4 illustrates the result of feature refinement for the two possible candidates of the manifest feature type with the Google dataset.

The manifest feature type is refined with the APKPure and manual datasets in the same way as with the Google dataset; we note that the cut-off points with the manual dataset are considered from 5% to 25%. Table 5 summarizes the best cut-off points for the manifest information feature type with all the three datasets.

In order to refine the strings feature type with each dataset, the apps are considered by their appropriate S-vectors. We apply RF to assign a weight to each of the feature values and select the top feature values in various cut-off points. Then, we apply each of KNN, NB, SVM, and DNN to classify the apps based on their refined S-vectors in considered cut-off points. Figure 5 illustrates the result of feature refinement for the strings feature type with our three dataset and Table 6 summarizes the best cut-off points.

4.2.2. Best Candidate Selection. Now, after refining the features, we select the best candidate for each of the API calls and manifest information feature types with every dataset for each classification algorithm. In order to select the best candidate for the API calls feature type between API-package, API-class, API-method, and API-full-method, we compare their effectiveness in app classification by applying each classification algorithm to our datasets. In this comparison, for each candidate, we employ its best detected cut-off point; for example, in the case of DNN with the Google dataset, we set the cut-off points as 8%, 6%, 4%, and 6% from Table 4 for API-package, API-class, API-method, and API-full-method, respectively. Figure 6 shows the result of this comparison with the three datasets; in this figure, for simplicity, each of the aforementioned candidates is represented as package, class, method, and full-method, respectively.

As observed in Figure 6, API-full-method shows *better* effectiveness than API-package, API-class, and API-method in terms of accuracy, precision, recall, and F-score *regardless* of the classification algorithm with *all* datasets. The reason is that API-full-method provides *more* accurate information about the apps functionalities than the other three candidates. As an example, consider the two following API-full-method signatures created by method overloading technique as "android.content.AsyncQueryHandler.removeMessages(int


(a) Manifest-permission

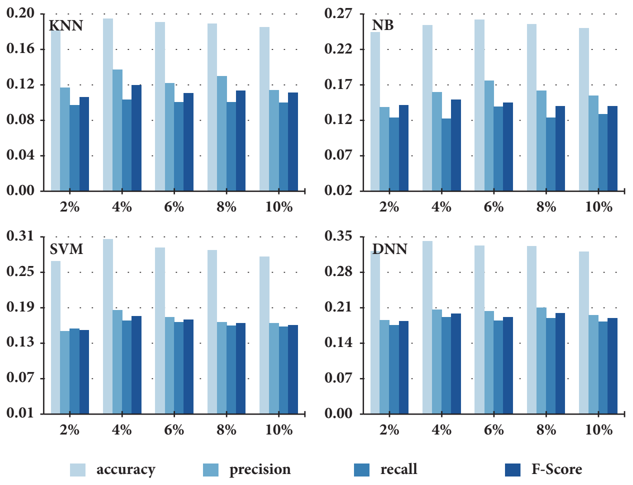
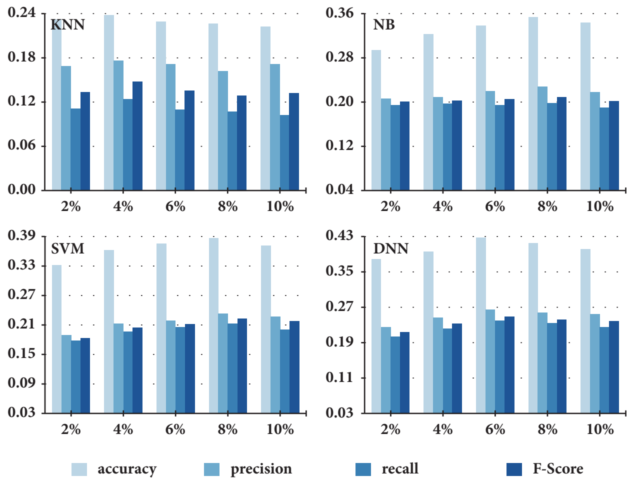
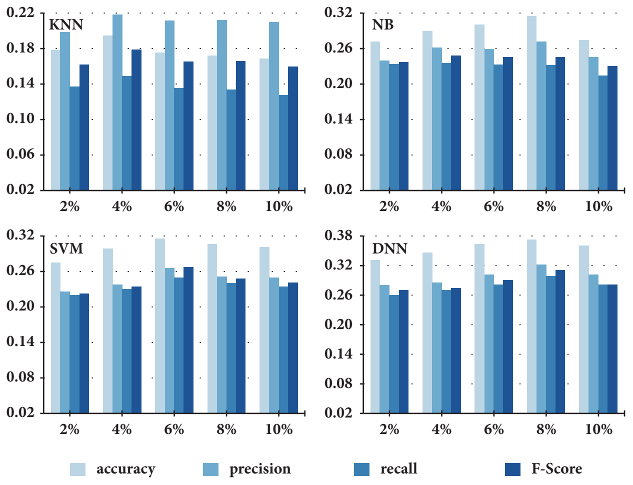

(b) Manifest-complete

Figure 4: Result of feature refinement for manifest information feature type with Google dataset.


(a) Google


(b) APKPure

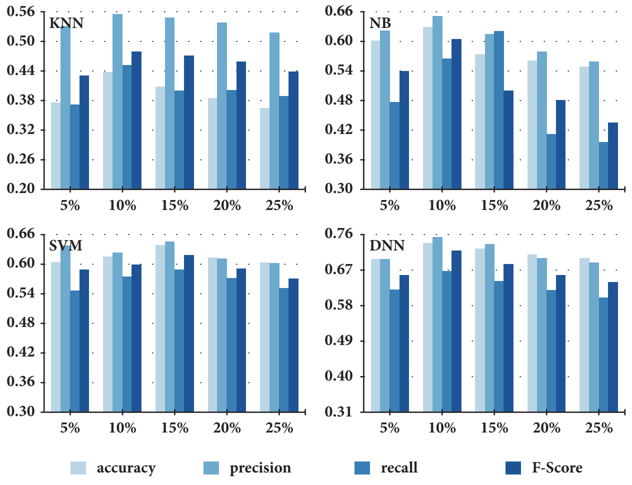

(c) Manual

Figure 5: Result of feature refinement for strings feature type with all datasets.

Table 4: Best cut-off points (%) for the API calls feature type.

	Google				APKPure				Manual			
	KNN	NB	SVM	DNN	KNN	NB	SVM	DNN	KNN	NB	SVM	DNN
API-package	10	8	8	8	8	8	8	8	20	20	20	15
API-class	8	6	8	6	8	6	6	6	15	20	15	15
API-method	6	4	4	4	4	6	4	4	15	10	10	15
API-full-method	4	4	4	6	4	4	4	4	10	10	15	10

Table 5: Best cut-off point s(%) for the manifest information feature type.

	Google				APKPure				Manual			
	KNN	NB	SVM	DNN	KNN	NB	SVM	DNN	KNN	NB	SVM	DNN
manifest-permission	8	8	8	8	8	8	8	6	20	15	20	20
manifest-complete	4	6	4	4	4	4	4	4	10	15	10	10

Table 6: Best cut-off points (%) for the strings feature type.

Google				APKPure				Manual			
KNN	NB	SVM	DNN	KNN	NB	SVM	DNN	KNN	NB	SVM	DNN
4	8	8	6	4	8	6	8	10	10	15	10

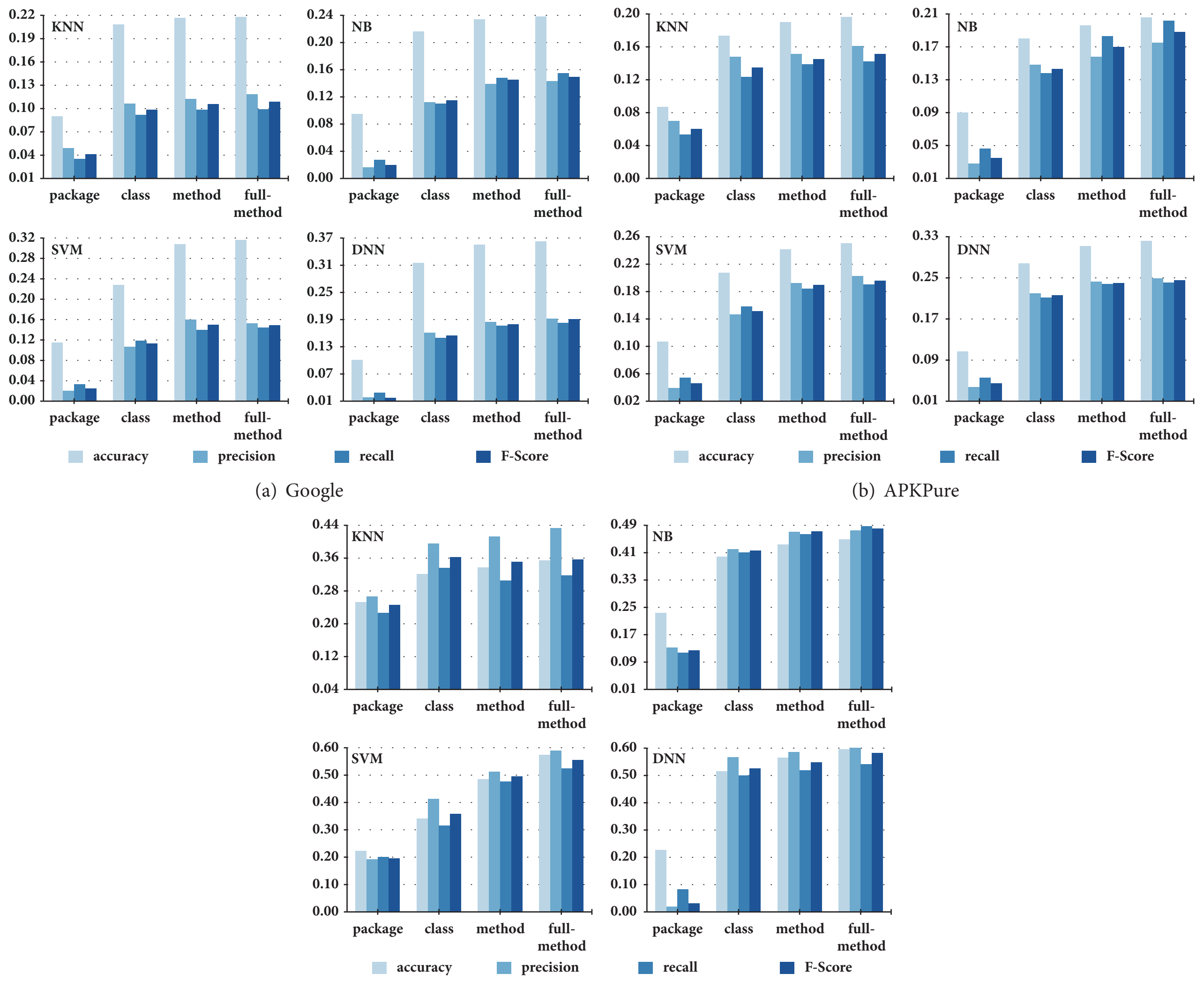
what)" and "android.content.AsyncQueryHandler.removeMessages(int what, Object obj)"; the first API removes any pending posts of messages with code "what" that are in the message queue, while the second API removes any pending posts of messages with code "what" and whose object is *identical* with "obj" that are in the message queue. Regarding API-package, API-class, and API-method candidates, these API calls are considered identical where both represented as "android.content", "android.content.AsyncQueryHandler", and "android.content.AsyncQueryHandler.removeMessages", respectively; API-full-method is the *only* candidate that does *not* consider these two separate API calls identical. Therefore, we select API-full-method as the representative of the API calls feature type, which means hereafter that *an app's A-vector represents an app in terms of API-full-method in the best cut-off point*. By considering the aforementioned example, it is obvious that why API-package shows the worse effectiveness, and why the effectiveness of API-class is less than that of API-method regardless of the classification algorithm with all datasets.

In order to select the best candidate for the manifest feature type between manifest-permission and manifest-complete, we compare their effectiveness in app classification by applying each classification algorithm to our datasets. In this comparison, for each candidate, we employ its best detected cut-off point; for example, in the case of NB with the Google dataset, we set the cut-off points as 8% and 6% from Table 5 for manifest-permission and manifest-complete, respectively. Figure 7 shows the result of this comparison with the three datasets. As observed in the figure, manifest-complete *significantly* outperforms manifest-permission in terms of accuracy, precision, recall, and F-score *regardless* of the classification algorithm with *all* datasets. The reason is that manifest-complete takes advantage of *all* the available manifest information as permissions, hardware and software components, activities, services, broadcast receivers, content providers, actions, and categories; however, manifest-permission takes advantage of *only* permissions requested by the apps. More specifically, manifest-complete provides *more* accurate information regarding the apps functionalities than manifest-permission. Therefore, we select manifest-complete as the representative of the manifest information feature type; hereafter, *an app's M-vector represents an app in terms of manifest-complete in the best cut-off point*.

We did not define any candidates for strings feature type; therefore, hereafter, *an app's S-vector represents an app in terms of strings and name attributes features in the best cut-off point*. For example, in the case of SVM with the Google dataset, we set the cut-off points as 8% from Table 6.

4.2.3. Classification Evaluation. Now, we evaluate the effectiveness of AndroClass in app classification. Furthermore, we suggest a setting of AndroClass, which may leads us to obtain high effectiveness in app classification with other datasets as well.

First, for each of KNN, NB, SVM, and DNN with our datasets, we combine A-vector, M-vector, and S-vector of every app into a single feature vector, F-vector. We note that, based on our experimental results explained in Sections 4.2.1 and 4.2.2, each of the A-vector, M-vector, and S-vector contains the appropriate features on their best cut-off points. More specifically, in Section 4.2.1, we detected the best cut-off points for feature refinement; also, in Section 4.2.2, we selected API-full-method and manifest-complete as the final representative for the API calls and manifest information feature types regardless of the classification algorithm with our three datasets. For example, in the case of DNN with the Google dataset, for every app, we combine A-vector containing API-full-method on cut-off point 6% (i.e., from Table 4), M-vector containing manifest-complete on cut-off



(a) Google

(b) APKPure

(c) Manual

Figure 6: Accuracy comparison of different candidates for API calls feature type with all datasets.

point 4% (i.e., from Table 5), and S-vector on cut-off point 6% (i.e., from Table 6) into a single F-vector.

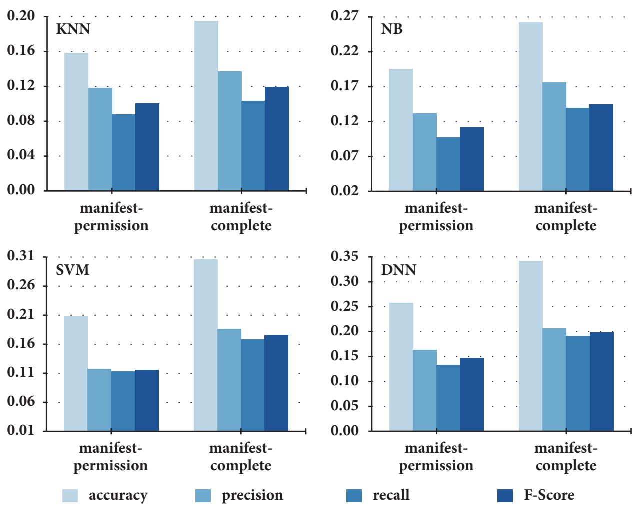
After constructing the appropriate F-vectors of all apps for each classification algorithm and dataset, we apply AndroClass to classify the apps. Figure 8 illustrates the effectiveness of AndroClass in app classification with our three datasets. As we already mentioned in Section 4.1, the Google and APKPure datasets contain apps belonging to more than one category; however, this figure shows the results when AndroClass predicts only a single category for each app.

As observed in Figure 8, AndroClass shows the *worse* effectiveness when it is equipped with KNN, while it shows the *best* effectiveness when it is equipped with DNN in terms of accuracy, precision, recall, and F-score with *all* datasets. The reason is that KNN does *not* perform any learning process and predicts the category for a test instance only based on its distances from training instances. On the contrary, DNN performs an effective learning process by utilizing nonlinear learning functions in multiple layers, thereby learning complex underlying relations in data. Table 7 represents the *percentage* of improvements in classification effectiveness when AndroClass is equipped with DNN over other classification algorithms.

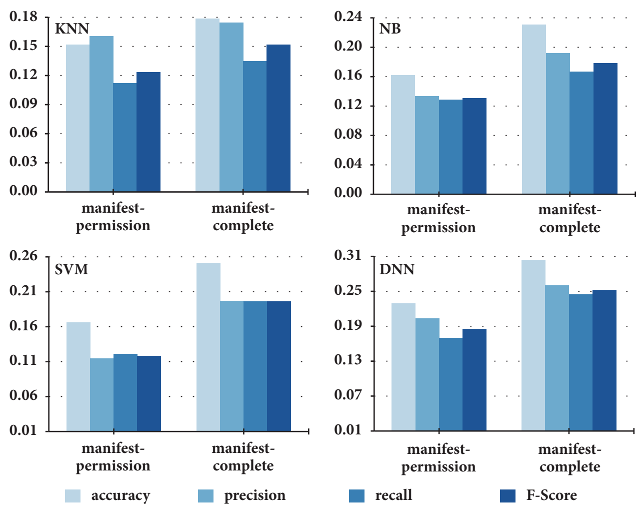
As an example, in the case of the Google dataset, AndroClass equipped with DNN shows 77%, 70%, 114%, and 93% improvements in classification in terms of accuracy, precision, recall, and F-score over AndroClass equipped with KNN, respectively.

As observed in Figure 8, although the APKPure dataset has a large number of apps and small number of categories than the Google dataset (i.e., the APKPure dataset is not well fine-grained categorized as the Google dataset), AndroClass shows *lower* classification accuracy with the APKPure dataset than that with the Google dataset *regardless* of the classification algorithms, which indicates that the misclassification problem in the APKPure dataset is *much* worse than that in the Google dataset.

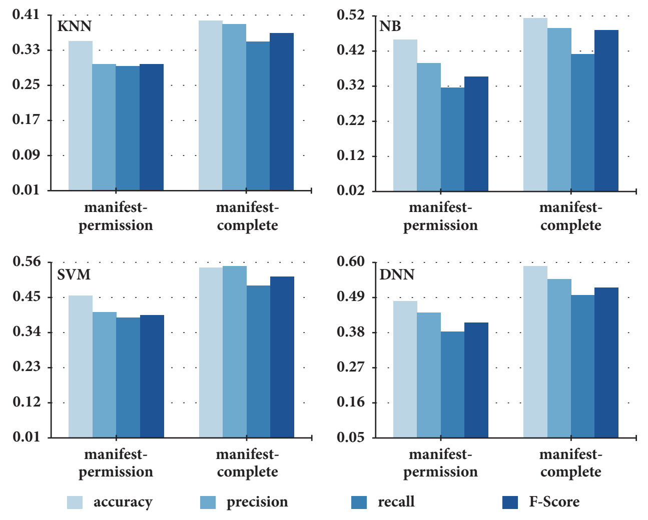
As an interesting result, AndroClass shows a *higher* effectiveness with the manual dataset than that with the Google and APKPure datasets in terms of accuracy, precision,



(a) Google



(b) APKPure



(c) Manual

FIGURE 7: Accuracy comparison of different candidates for manifest information feature type with all datasets.

TABLE 7: Improvement in classification effectiveness (%) by AndroClass when equipped with DNN.

	Google			APKPure			Manual		
	KNN	NB	SVM	KNN	NB	SVM	KNN	NB	SVM
accuracy	77	26	18	88	27	23	50	18	9
precision	70	45	25	47	16	25	39	17	10
recall	114	50	26	90	45	28	51	13	12
F-score	93	48	26	69	32	27	45	15	11

recall, and F-score *regardless* of the classification algorithms as observed in Figure 8. Table 8 represents the *percentage* of improvements in classification effectiveness obtained by AndroClass with the manual dataset over real-world datasets. For example, when AndroClass is equipped with NB, the values of accuracy, precision, recall, and F- score with the Google dataset are 0.414, 0.247, 0.216, and 0.231 while these values with the manual dataset are 0.706, 0.748, 0.688, and 0.717, respectively. The improvements in classification effectiveness obtained with the manual dataset over the Google dataset in terms of accuracy, precision, recall, and F-score are 70.5%, 203.1%, 218.5%, and 211.8%, respectively. These results show that AndroClass takes advantage of rich and comprehensive features representing the *actual* behaviors and functionalities of the apps, thereby leading AndroClass to classify apps significantly closed to the one performed by human experts (i.e., the authors) since the authors classified different apps (i.e., selected from the Google and APKPure datasets) into the

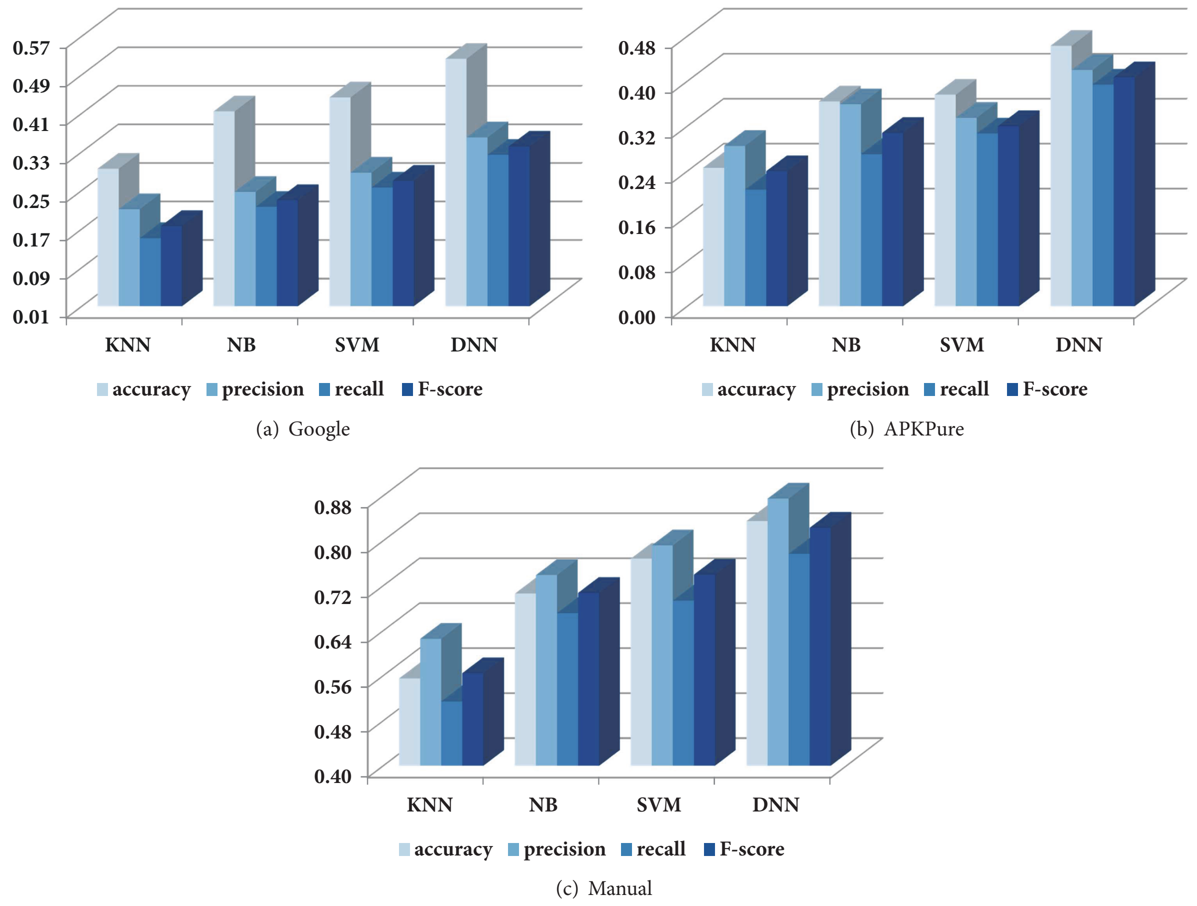


FIGURE 8: Classification effectiveness of AndroClass with all datasets.

manual dataset based on their functionalities. Furthermore, it shows that the misclassification problem is *serious* in our real-world datasets and consequently in online app stores (Google Play Store and APKPure) which are used to construct our real-world datasets.

With the manual dataset, AndroClass shows accuracy, precision, recall, and F-score as 83.5%, 87.4%, 77.6%, and 82.3%, respectively, when equipped with DNN. Table 9 represents the accuracy of AndroClass equipped with each classification algorithm for all categories of the manual dataset where #APP column contains the number of apps in the category. We note that when a very *large* number of apps are assigned to a very *small* number of categories, the classification accuracy will be *biased* as observed with the employed dataset in [27]. In this case, the classification accuracy for the few categories containing the large number of apps are *very* high, while the classification accuracy for the other categories containing the small number of apps are very much *lower* than that for the former categories. However, since the classification accuracy with the former categories is high, the total classification accuracy will be biased to a high value. On the contrary, with our manual dataset, the difference between classification accuracy for various categories obtained by AndroClass is *not* high *regardless* of the classification algorithm as shown in Table 9. For example, in the case of AndroClass equipped with DNN, the best accuracy (i.e., 91.5%) is observed for the Tools category, 7 categories have accuracy values more than 70%, and 4 categories have accuracy values more than 60%; the standard deviation of accuracy values for different categories is 11.06.

Also, Table 10 represents the confusion matrices of the classification results with the manual dataset where AndroClass is equipped with KNN, NB, SVM, and DNN. In this table, the category names (i.e., defined in Table 9) are written *short* to save the space.

As already explained, Figure 8 illustrates the classification effectiveness when only a single category is predicted by AndroClass for each app. However, in the case of the Google and APKPure datasets, we did not perform any user studies and these datasets contain duplicate apps as shown in Table 2; as an example, the "Skype" app is assigned to *three* separate categories as "Communication_SNS", "Communication_Message", and "Communication_VideoChat" in the Google dataset. Therefore, instead of predicting a single category for each app, we assign top N (i.e., N=1, 2, 3) (note that we can set N to a larger value than three; however, in app classification, assigning a single app to a large number of categories is not meaningful and may bias the classification effectiveness) categories with the highest probability to each apps; Table 11 shows the accuracy of AndroClass equipped

Table 8: Improvement in classification effectiveness (%) with manual dataset over real-world datasets.

	Google				APKPure			
	accuracy	precision	recall	F-score	accuracy	precision	recall	F-score
KNN	88.1	196.4	240.4	220.6	125.6	119.5	148.3	135.1
NB	70.5	203.1	218.5	211.8	93.9	108.6	154.8	132.8
SVM	73.3	175.8	170.8	173.9	104.2	136.3	125.8	131.1
DNN	59.6	143.0	139.7	141.3	80.3	108.3	97.1	102.2

Table 9: Classification accuracy (%) of AndroClass for all categories with manual dataset.

			accuracy			
	category	#APP	KNN	NB	SVM	DNN
1	Weather	15	53.3	60.0	53.3	66.6
2	Food & Drink	21	66.7	71.4	76.1	85.7
3	Web Browser	7	42.8	57.1	71.4	71.4
4	Office & Business	31	48.3	67.7	70.9	67.7
5	Keyboard	10	40.0	60.0	60.0	60.0
6	Music & Video	51	49.0	80.3	88.2	90.1
7	Tools	83	61.4	71.0	81.9	91.5
8	Photo & Art	42	57.1	64.2	76.1	78.5
9	Theme & Wallpaper	37	59.4	67.5	75.6	89.1
10	Finance	15	46.6	73.3	66.7	80.0
11	Virtual Reality	8	37.5	62.5	62.5	62.5
12	SNS & Communication	44	54.5	70.4	77.2	88.6
	total accuracy		55.5	70.6	76.8	**83.5**

with different classification algorithms when top N categories are considered. When we consider top N categories, AndroClass shows the *worse* effectiveness when it is equipped with KNN, while it shows the *best* effectiveness when it is equipped with DNN in terms of accuracy, precision, recall, and F-score with *both* datasets *regardless* of the value of N. By increasing the value of N, the classification accuracy of AndroClass is improved as well regardless of the classification algorithm; AndroClass shows the *significant* classification accuracy as 71.2% and 67.6% with the real-world datasets Google and APKPure, respectively, when equipped with DNN and the top three categories are regarded.

Although we conducted extensive experiments for feature refinement, best candidate selection, and analyzing the classification effectiveness of AndroClass, we can *suggest* the following setting that may lead us to obtain high effectiveness in classification when applying AndroClass to any other datasets. As the classification algorithm, we suggest DNN since AndroClass shows its *best* effectiveness in classification with our three datasets when it is equipped with DNN as observed in Figure 8. As the features, we suggest utilizing the combination of API-full-method (i.e., API-full-method signatures), manifest-complete (i.e., permissions, hardware and software components, activities, services, broadcast receivers, content providers, actions, and categories), and strings (i.e., strings, and name attributes) on their best cut-off points; note that API-full-method and manifest-complete show the best accuracy among possible candidates for API calls and manifest information feature types as shown in Section 4.2.2 regardless of the classification algorithms with all datasets. The values of the cut-off points are highly dependent on the size of the dataset; however, they can be *easily* detected by applying DNN on the aforementioned features as explained in Section 3.3.

5. Conclusion

In this paper, we proposed an effective method, AndroClass, to automatically classify Android apps based on their real functionalities, which performs three orthogonal steps of feature extraction, feature refinement, and classification. In the feature extraction step, 14 rich and comprehensive features representing the actual functionalities of the apps are extracted and summarized into the three types as API calls (i.e., API packages, API classes, API methods, and API-full-method signatures), manifest information (i.e., permissions, hardware and software components, activities, services, broadcast receivers, content providers, actions, and categories), and strings (strings and name attributes). We developed a unified tool suite to mine APK files and the Android platform in order to obtain the required information for extracting the features. An app is represented as three binary vectors, A-vector, M-vector, and S-vector, containing the aforementioned feature types, respectively. For the API calls feature type, we considered four possible candidates as API-package, API-class, API-method, and API-full-method where the API-full-method was selected as the best candidate. For the manifest information feature type, we considered two possible candidates as manifest-permission and

Table 10: Confusion matrices of classification results with manual dataset.

	Weather	Food	Browser	Office	Keyboard	Music	Tools	Photo	Theme	Finance	VR	SNS
KNN												
Weather	8	2	0	0	0	2	1	1	0	1	0	0
Food	0	14	0	4	0	0	1	1	0	0	0	1
Browser	0	0	3	1	0	0	0	1	0	1	0	1
Office	2	4	0	15	0	1	6	2	0	1	0	0
Keyboard	0	0	0	1	4	0	2	1	1	0	0	1
Music	1	8	0	7	0	25	4	4	0	0	2	0
Tools	1	12	0	2	0	2	51	8	0	2	3	2
Photo	1	3	0	4	0	0	9	24	0	0	0	1
Theme	0	4	0	1	0	0	8	1	22	0	0	1
Finance	0	1	1	2	0	2	2	0	0	7	0	0
VR	0	0	0	1	0	1	2	1	0	0	3	0
SNS	0	10	0	1	0	0	7	2	0	0	0	24
NB												
Weather	9	2	0	2	0	0	1	0	0	0	0	1
Food	0	15	0	3	0	1	0	1	0	0	0	1
Browser	1	0	4	0	0	0	0	0	0	1	0	1
Office	1	4	0	21	1	1	0	0	0	2	0	1
Keyboard	0	0	0	0	6	2	0	1	0	0	0	1
Music	0	1	0	4	0	41	0	1	0	1	1	2
Tools	1	2	0	8	0	4	59	1	1	1	1	5
Photo	0	2	0	7	0	2	1	27	0	0	0	3
Theme	1	2	0	2	0	1	2	2	25	0	0	2
Finance	0	1	0	2	0	0	0	0	0	11	0	1
VR	0	1	0	0	0	1	0	0	0	0	5	1
SNS	0	6	0	3	0	1	1	0	1	1	0	31
SVM												
Weather	8	1	1	0	0	0	2	1	2	0	0	0
Food	0	16	0	0	0	1	0	1	0	0	0	3
Browser	0	0	5	0	0	0	0	0	1	0	0	1
Office	0	0	0	22	0	1	3	1	0	0	1	3
Keyboard	0	0	0	0	6	0	2	1	0	0	0	1
Music	0	0	0	0	0	45	3	1	0	0	2	0
Tools	0	0	0	6	0	1	68	2	2	0	0	4
Photo	0	3	0	1	1	0	3	32	1	0	0	1
Theme	1	2	1	0	0	0	4	1	28	0	0	0
Finance	0	1	0	0	0	0	3	0	0	10	0	1
VR	0	0	0	0	0	3	0	0	0	0	5	0
SNS	0	1	0	3	0	1	3	1	1	0	0	34
DNN												
Weather	10	0	0	0	0	1	2	1	1	0	0	0
Food	0	18	0	0	0	0	1	1	0	0	0	1
Browser	0	0	5	0	0	0	1	0	0	0	0	1
Office	0	2	0	21	0	2	3	1	0	0	0	2
Keyboard	0	0	0	0	6	1	2	1	0	0	0	0
Music	0	0	0	2	0	46	2	1	0	0	0	0
Tools	0	1	0	0	0	1	76	0	1	0	0	4
Photo	0	2	0	2	1	0	5	33	0	0	0	1
Theme	1	0	0	0	0	1	1	1	33	0	0	0
Finance	0	0	0	0	0	0	2	1	0	12	0	0
VR	0	0	0	0	0	3	0	0	0	0	5	0
SNS	0	0	0	2	0	0	2	0	1	0	0	39

Table 11: Accuracy (%) of AndroClass by considering top N categories.

	Google				APKPure			
	KNN	NB	SVM	DNN	KNN	NB	SVM	DNN
Top 1 category	29.5	41.4	44.3	52.3	24.6	36.4	37.6	46.3
Top 2 categories	41.0	54.7	53.7	63.5	36.1	50.8	49.0	61.0
Top 3 categories	49.4	62.5	59.0	71.2	45.2	59.2	57.8	67.6

manifest-complete where manifest-complete was selected as the best candidate. In the feature refinement step, we applied RF to refine each of API calls, manifest information, and strings feature types, separately. In the classification step, we combine refined features into a single one and an app is represented as a single binary feature vector, F-vector. Then, we applied AndroClass (i.e., equipped with KNN, NB, SVM, and DNN) to classify apps into their appropriate categories. To the best of our knowledge, AndroClass is the first method applies DNN to app classification and mines the string.xml file to extract features for classification.

In order to carefully evaluate the effectiveness of AndroClass, we employed two real-world datasets of apps, Google and APKPure, with a large number of fine-grained categories and also we constructed a manual dataset where apps are categorized by the human experts (i.e., authors) in 12 categories based on their real functionalities. Our extensive experimental results demonstrated that (1) AndroClass provides its best effectiveness regardless of the dataset when it is equipped with DNN; (2) AndroClass shows a significant accuracy as 83.5% with the manual dataset, which means the classification performed by AndroClass is meaningful and very close to human intuition since the manual dataset is constructed by human experts; (3) although the Google and APKPure datasets contain very large number of fine-grained categories and suffer from misclassification, AndroClass shows accuracy as 71.2% and 67.6% with these datasets when top 3 categories is assigned to apps.

AndroClass only extracts the required features from the information obtained by mining the APK files and the Android platform; therefore, it has different advantages over the existing methods for app classification as (1) AndroClass does not pose any issues to the user privacy since it does not access any parts of users' smartphones for feature extraction; (2) all the utilized features in AndroClass are stable and clearly represent the actual functionalities of the app; (3) it can be applied to classify unreleased or newly released apps.

We figured out interesting directions for our future work as follows. Since AndroClass utilizes the features representing the actual functionalities and underlying behaviors of apps, we can apply it to malware detection [15, 21] and malware classification (i.e., detecting the malware family) [52, 53] as well. However, we need to make some minor changes in AndroClass to make it adapted to the aforementioned topics. For example, we should differentiate between the security sensitive features (i.e., API calls and permissions) and the nonsecurity sensitive ones (i.e., strings and name attributes) when training the classification algorithms and constructing the models. Furthermore, we plane to analyze the *efficiency* of AndroClass when it is equipped with different classification algorithms. As an example, one possible solution to improve the classification efficiency could be employing distributed computing techniques in AndroClass.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (no. NRF-2015R1D1A1A02061946).

References

[1] A. Feizollah, N. B. Anuar, R. Salleh, and A. W. A. Wahab, "A review on feature selection in mobile malware detection," *Digital Investigation*, vol. 13, pp. 22–37, 2015.

[2] S. Lee, J. Dolby, and S. Ryu, "HybriDroid: Static analysis framework for android hybrid applications," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16)*, pp. 250–261, September 2016.

[3] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, "A survey of app store analysis for software engineering," *IEEE Transactions on Software Engineering*, vol. 43, no. 9, pp. 817–847, 2017.

[4] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, and P. G. Bringas, "On the automatic categorisation of android applications," in *Proceedings of the 9th Annual IEEE Consumer Communications and Networking Conference-Security and Content Protection (CCNC '12)*, pp. 149–153, January 2012.

[5] J. Crussell, C. Gibler, and H. Chen, "AnDarwin: scalable detection of android application clones based on semantics," *IEEE Transactions on Mobile Computing*, vol. 14, no. 10, pp. 2007–2019, 2015.

[6] P. Faruki, V. Laxmi, A. Bharmal, M. Gaur, and V. Ganmoor, "Androsimilar: robust signature for detecting cariants of android malware," *Information Security and Applications*, vol. 22, pp. 66–80, 2015.

[7] J.-W. Jang, H. Kang, J. Woo, A. Mohaisen, and H. K. Kim, "Andro-AutoPsy: anti-malware system based on similarity matching of malware and malware creator-centric information," *Digital Investigation*, vol. 14, pp. 17–35, 2015.

[8] T.-E. Wei, H.-R. Tyan, A. B. Jeng, H.-M. Lee, H.-Y. M. Liao, and J.-C. Wang, "DroidExec: root exploit malware recognition against wide variability via folding redundant function-relation graph," in *Proceedings of the 17th IEEE International Conference on Advanced Communications Technology (ICACT '15)*, pp. 161–169, July 2015.

[9] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: effective and explainable detection of android malware in your pocket," in *Proceedings of the 14st International Conference on Network and Distributed System Security Symposium*, pp. 1–12, February 2014.

[10] N. Chen, S. C. Hoi, S. Li, and X. Xiao, "Simapp: a framework for detecting similar mobile applications by online kernel learning," in *Proceedings of the 8th ACM International Conference on Web Search and Data Mining*, pp. 305–314, Shanghai, China, Feburary 2015.

[11] P. Yin, P. Luo, W.-C. Lee, and M. Wang, "App recommendation: a contest between satisfaction and temptation," in *Proceedings of the 6th ACM International Conference on Web Search and Data Mining (WSDM '13)*, pp. 395–404, February 2013.

[12] "Android developers site," https://developer.android.com/studio/build/multidex.

[13] Q. Do, B. Martini, and K.-K. R. Choo, "Exfiltrating data from Android devices," *Computers & Security*, vol. 48, pp. 74–91, 2015.

[14] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *Proceedings of the International Conference on Security and Privacy in Communication Systems*, pp. 86–103, 2013.

[15] A. Demontis, M. Melis, B. Biggio et al., "Yes, machine learning can be more secure! A case study on android malware detection," *IEEE Transactions on Dependable and Secure Computing*, 2017.

[16] P. Faruki, A. Bharmal, V. Laxmi et al., "Android security: a survey of issues, malware penetration, and defenses," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.

[17] F. A. Narudin, A. Feizollah, N. B. Anuar, and A. Gani, "Evaluation of machine learning classifiers for mobile malware detection," *Soft Computing*, vol. 20, no. 1, pp. 343–357, 2016.

[18] V. Rastogi, Y. Chen, and X. Jiang, "Catch me if you can: Evaluating android anti-malware against transformation attacks," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 1, pp. 99–108, 2014.

[19] B. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: a perspective combining risks and benefits," in *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, pp. 13–22, ACM, June 2012.

[20] H. Shahriar and V. Clincy, "Anomalous android application detection with latent semantic indexing," in *Proceedings of the IEEE 40th Annual Computer Software and Applications Conference (COMPSAC '16)*, pp. 624-625, June 2016.

[21] K. Sokolova, C. Perez, and M. Lemercier, "Android application classification and anomaly detection with graph-based permission patterns," *Decision Support Systems*, vol. 93, pp. 62–76, 2017.

[22] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "DroidMat: android malware detection through manifest and API calls tracing," in *Proceedings of the 7th Asia Joint Conference on Information Security (AsiaJCIS '12)*, pp. 62–69, August 2012.

[23] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik, "New android malware detection approach using Bayesian classification," in *Proceedings of the 27th IEEE International Conference on Advanced Information Networking and Applications (AINA '13)*, pp. 121–128, March 2013.

[24] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware Android malware classification using weighted contextual API dependency graphs," in *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS '14)*, pp. 1105–1116, ACM, Scottsdale, AZ, USA, November 2014.

[25] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of piggybacked mobile applications," in *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY '13)*, pp. 185–195, February 2013.

[26] F. Dong, Y. Guo, C. Li, G. Xu, and F. Wei, "ClassifyDroid: large scale android applications classification using semi-supervised multinomial naive bayes," in *Proceedings of the 4th IEEE International Conference on Cloud Computing and Intelligence Systems (CCIS '16)*, pp. 77–81, August 2016.

[27] W. Wang, Y. Li, W. Xing, J. Liu, and Z. Xiangliang, "Detecting android malicious apps and categorizing benign apps with ensemble of classifiers," *Future Generation Computer Systems*, vol. 78, pp. 987–994, 2018.

[28] C. Z. Yang and M. H. Tu, "Lacta: an enhanced automatic software categorization on the native code of android applications," in *Proceedings of the International MultiConference of Engineers and Computer Scientists*, pp. 1–5, 2012.

[29] H. Zhu, E. Chen, H. Xiong, H. Cao, and J. Tian, "Mobile app classification with enriched contextual information," *IEEE Transactions on Mobile Computing*, vol. 13, no. 7, pp. 1550–1563, 2014.

[30] Y. Liao, J. Li, B. Li, G. Zhu, Y. Yin, and R. Cai, "Automated detection and classification for packed android applications," in *Proceedings of the IEEE 5th International Conference on Mobile Services (MS '16)*, pp. 200–203, July 2016.

[31] J. Escobar-Avila, M. Linares-Vásquez, and S. Haiduc, "Unsupervised software categorization using bytecode," in *Proceedings of the 23rd IEEE International Conference on Program Comprehension (ICPC '15)*, pp. 229–239, May 2015.

[32] C. McMillan, M. Linares-Vásquez, D. Poshyvanyk, and M. Grechanik, "Categorizing software applications for maintenance," in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM '11)*, pp. 343–352, September 2011.

[33] M. Zheng, M. Sun, and J. C. S. Lui, "Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware," in *Proceedings of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom '13)*, pp. 163–171, July 2013.

[34] J. Tang, S. Alelyani, and H. Liu, "Feature selection for classification: a review," in *Data Classification: Algorithms and Applications*, 2014.

[35] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[36] M. Ghorbanzadeh, Y. Chen, Z. Ma, T. C. Clancy, and R. McGwier, "A neural network approach to category validation of Android applications," in *Proceedings of the International Conference on Computing, Networking and Communications (ICNC '13)*, pp. 740–744, January 2013.

[37] J. Han, M. Kamber, and J. Pei, *Data Mining Concepts and Techniques*, Elsevier, 2012.

[38] C. Manning, P. Raghavan, and H. Schutze, *Introduction to Information Retrieval*, Cambridge University Press, 2008.

[39] M. Yang and Q. Wen, "Detecting android malware by applying classification techniques on images patterns," in *Proceedings of*

the 2nd IEEE International Conference on Cloud Computing and Big Data Analysis (ICCCBDA '17), pp. 344–347, April 2017.

[40] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[41] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, 2003.

[42] V. N. Vapnik, *The Nature of Statistical Learning Theory*, Springer, 1995.

[43] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[44] "Dalvik executable format," https://source.android.com/devices/tech/dalvik/dex-format.

[45] J. Levin, *Android Internals - A Confectioner's Cookbook*, vol. I of *The Power User's View*, Cambridge, MA, USA, 2015.

[46] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: detecting cloned applications on android markets," in *Proceedings of the European Symposium on Research in Computer Security*, pp. 37–54, 2012.

[47] O. E. David and N. S. Netanyahu, "DeepSign: deep learning for automatic malware signature generation and classification," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN '15)*, pp. 1–8, July 2015.

[48] W. Hardy, L. Chen, S. Hou, Y. Ye, and X. Li, "Dl4md: a deep learning framework for intelligent malware detection," in *Proceedings of the International Conference on Data Mining*, pp. 61–67, 2016.

[49] F. Pedregosa, G. Varoquaux, and A. Gramfort, "Scikit-learn: machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[50] M. Abadi, P. Barham, J. Chen, and Z. Chen, "Tensorflow: a system for large-scale machine learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pp. 265–283, 2016.

[51] C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, New York, NY, USA, 2006.

[52] M. Sun, X. Li, J. C. S. Lui, R. T. B. Ma, and Z. Liang, "Monet: a user-oriented behavior-based malware variants detection system for android," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 5, pp. 1103–1112, 2017.

[53] T. Chakraborty, F. Pierazzi, and V. S. Subrahmanian, "EC2: ensemble clustering and classification for predicting android malware families," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–16, 2017.