IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 22, 2019, accepted October 12, 2019, date of publication October 18, 2019,
date of current version November 11, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2948212

DroidARA: Android Application Automatic Categorization Based on API Relationship Analysis

WENHAO FAN, YE CHEN, YUAN'AN LIU, AND FAN WU
Beijing Key Laboratory of Work Safety Intelligent Monitoring, School of Electronic Engineering, Beijing University of Posts and Telecommunications, Beijing 100876, China

Corresponding author: Wenhao Fan (whfan@bupt.edu.cn)

This work was supported in part by the National Natural Science Foundations of China under Grant 61821001, in part by the Yang Fan Innovative and Entrepreneurial Research Team Project of Guangdong Province, in part by the Fundamental Research Funds for the Central Universities, and in part by the Director Foundation of Beijing Key Laboratory of Work Safety Intelligent Monitoring.

ABSTRACT An application (app) market with well-managed categorization will help users with app search and recommendation. Current categorization methods in app markets mainly rely on manual operation. Existing approaches for automatic Android app categorization suffer from low efficiency and low accuracy due to insufficient analysis of features, or inappropriate choice of features. This will mislead users to download unrelated apps and is not conducive to market stability maintenance. In this paper, we propose DroidARA, an efficient automatic categorization method for Android apps based on API relationships analysis. Considering that the app category can be characterized by API relationships which represent the combinations and links among APIs, we design a complete system to generate API call graphs, extract the API relationships information, transform them into feature vectors and train the classifier. Firstly, the API calls are obtained through static analysis to generate API call graphs that contain the relationships among APIs. A novel matrix structure as well as related algorithm are designed to extract the API relationships information from API call graphs. After that, the matrix is transformed into vector according to two feature selection methods we designed for strengthening the use of effective information in the API relationships. A convolutional neural network (CNN) model is then trained with labeled samples of such feature vectors. To validate the feasibility of DroidARA, we conduct several categorization experiments on 19949 real apps of Google Play. The results demonstrate that DroidARA can achieve an average 88.9% accuracy in categorizing the apps into 24 categories, which outperforms existing methods by 18.5%.

INDEX TERMS Android, categorization, API relationship, static analysis.

I. INTRODUCTION

As the most widely used mobile operating system whose market share has reached 86.7% [1], Android not only attracts a large number of users but also attracts a large number of developers. At present, people are accustomed to using various mobile applications (apps) for daily activities such as communication, trading, photography or working. App markets such as Google Play [2] and Amazon Appstore [3] provide users with app download platforms. The number of android apps in Google play has reached 2,700,000, and there are still nearly 10,000 new apps added every month [4]. This poses a challenge to the management of the market.

A well-managed app market will not only bring convenience to users, but also make developers profitable. App markets often divide the app into many categories. This not only facilitates the quick search of the user, but also allows the market itself to make app recommendations based on categories. Therefore, the accuracy of the categorization greatly affects the stability of the market. However, current categorization systems have certain flaws. The app categories provided by the app markets are often labeled manually by app developers. This will cause two issues. On the one hand, the category is likely to be wrong. The division of the app category is determined by the app market owners, and the developer may have a bias in understanding the defined categories, resulting in a categorization error. And many apps currently have some additional features that can make categorization difficult. In addition, some adware will deliberately choose popular categories such as photography to boost their downloads, which will cause trouble or even economic loss to users. On the other hand, the efficiency of

The associate editor coordinating the review of this manuscript and approving it for publication was Ahmed Farouk.

manual categorization is very low. When the category labels need to be adjusted, it will introduce a huge amount of work, which is not conducive to expansion and maintenance.

Existing research proposes the use of automated categorization techniques to categorize apps. Considering that apps usually use text information to guide users to use their functions, a large part of researches [5]–[11] use natural language processing to categorize apps by mining app description information, user comments, and so on. However, many popular apps do not have adequate or accurate text descriptions while the newly released app does not have sufficient comments. This makes this approach very limited. Besides text descriptions, other studies [12]–[16] also consider the information inside the app. They extract the features by decompiling the android app's installation file named Android app package(APK). One of the important features is the Android API which is written in Java language, acts as interfaces used by Android apps to communicate with Android Framework. But most of them simply put these features together without further analysis, so that the categorization results are based only on the binary values of the features. In fact, there are many different categories of apps that contain the same simple features, which leads to a decrease in accuracy.

In order to improve the efficiency and accuracy of app categorization by mining further characteristic information inside the app, we consider using the relationships among APIs which are the combinations and links between APIs as the core features to model the category of apps. Since the function of an app is implemented by a series of Android APIs, the API relationships are ideal features to reflect the functionality of the app. Based on this opinion, we propose DroidARA, an automatic Android app categorization method through API relationships analysis. Aiming at efficiently extracting the API relationships information and fully using them to perform the categorization, four steps are included in the system: 1. using static analysis technology to extract the API call information of the app and generate API call graphs to represent it. 2. a novel feature matrix for representing API relationships is designed and assigned value by extracting the API relationships from API call graphs. 3. two feature selection algorithm suitable for the obtained features are designed to improve the efficiency and accuracy of the system. 4. training the classifier model with a deep learning algorithm and use the model to categorize the app.

Finally, we use the 19949 sample apps collected from Google Play to perform the experiment. We use the category labels originally defined by Google Play to classify these apps into 24 categories. The experimental results show that our classifier achieves an accuracy of 88.9%. It is 18.5% higher than the reference method.

In summary, we highlight our main contributions made in this paper as follows:

· We propose an automatic Android app categorization method, called DroidARA, which contains a system that fully use the relationships of Android APIs to effectively capture the function of the app. Furthermore, we design a novel feature matrix to well represent API relationships.

· We implement a static analysis algorithm to generate the API call graphs and transform them into our proposed matrix. Besides, two relative feature selection methods are designed respectively to increase the efficiency and remove the redundancy of the system.

· We conduct real-world experiments to validate the feasibility of DroidARA. DroidARA can achieve Android category prediction with an 88.9% accuracy. It demonstrates that DroidARA can be used on real-world app market.

The remainder of this paper is organized as follows: section II shows the existing research works of app categorization; section III introduces the system model of our proposed method in detail; section IV describes the experiments, and compares and evaluates our classifier according to the experiment results; section V concludes our work and discusses the further work.

II. RELATED WORKS

Recently, researches on app automation categorization focus on two kinds of methods: 1. using the app's metadata such as descriptive information, user reviews, and other online resources to categorize. 2. collecting information in the apk files, mainly performed by decompiling the APK files to get information such as APIs, strings and permissions.

A. METHODS OF USING APP'S METADATA

Dae [17] leveraged descriptions and user reviews to find out the important features of apps. They proposed a probabilistic model, named AppLDA, to generate app representations while excluding noise in reviews. Inspired by their work, Bajaj *et al.* [5] jointly modeled app descriptions and user reviews to evaluate their use in predicting other indicators like app categories and ratings. Then they proposed a multitask neural architecture to learn and analyzed the influence of apps' textual data to predict other categorical parameters. Since the text data they obtained from the app market may be not adequate, other studies tend to collect richer textual information from the Internet. Zhu *et al.* [18] enriched the textual information of mobile Apps by exploiting the additional Web knowledge from the Web search engine and combine all the enriched textual information into the Maximum Entropy model to train a mobile app classifier, while [10], [19] leveraged information publicly available from the online stores where the apps are marketed. To complement the existing text-based approach for app categorization, Singla *et al.* [6] presented an app categorization system that uses object detection and recognition in images associated with apps to generate a more accurate categorization. Considering that the original category labels of the market are not fine-grained, Liu *et al.* [7] developed a framework to label the apps with fine-grained categorical information. They discovered the novel inter-class relationships among categories and introduce a customized class hierarchy optimized for app categorization. Similar to them, Al-Subaihi *et al.* [11] clustered

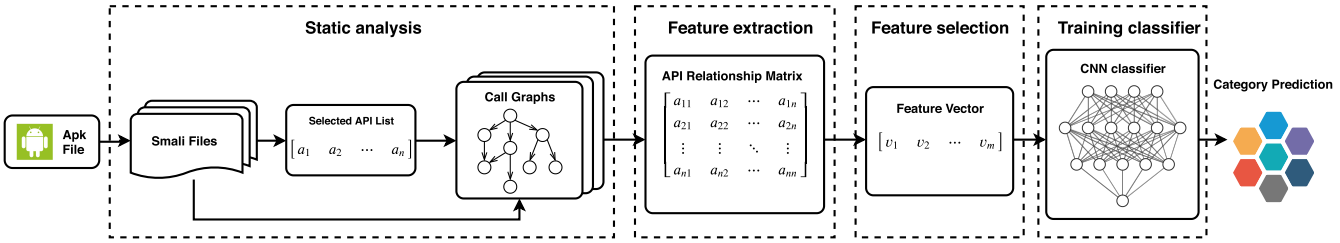


FIGURE 1. System structure.

apps based on their claimed behavior to find the possible sub-categorization of the categorization in the app markets.

These methods mainly used natural language process technology to utilize the textual information associated with apps. However, the textual may not be accurate for the features of the app. And the online resources are always insufficient for newly released apps. What's more, the language of the text may be varied, which will introduce additional noise.

B. METHODS OF COLLECTING INFORMATION IN THE APK FILES

Dong *et al.* [14] chose specific Android APIs as characteristic API to represent the features of the app and generated vectors based on whether the app uses these APIs. Different from them, Yuan *et al.* [16] extracted the used permissions and strings that can reflect the app function to automatically categorize apps based on Bayesian classification. While Hamedani *et al.* [13] combined these features and refined them into single vectors. They applied multi-machine learning algorithm to find the best model for categorization. To further improve the accuracy of the categorization, Wang *et al.* [12] employed the ensemble of multiple classifiers with 11 types of static features to provide a complete solution for the automated categorization of benign apps.

Compared to the metadata, the information in the apk files is sufficient and closed to the nature of the app. However, there is a common issue between these methods. They simply combined the binary value of the features without analyzing the relationship among them. For example, according to the researches [20], [21], there is a mapping relationship between Android API and permission. So collecting both of them is redundant. And nowadays, many apps contain varies functions. The binary value vector may be too coarse-grained to distinguish them. In our work, we further analyze the relationship between the APIs used in apps and design a more precise model to represent them to improve the categorization accuracy.

III. SYSTEM MODEL

A. OVERVIEW

The overall structure of the android app categorization system we propose is depicted in the Fig. 1. The categorization task is divided into four steps: static analysis, feature extraction, feature selection and training classifier.

(1) In the static analysis step, the apps' apk files are decompiled into the corresponding smali files by the Apktool [22]. From the smali files we are able to get the Android APIs that each app contains using the string matching method. After we count the APIs in each app category, the TF-IDF algorithm is applied to assign the weight to each of the Android APIs. Then we select the high-weight APIs as our selected API list. Based on the selected API list, an analysis algorithm is designed to generate API call graphs of an app from its smali files.

(2) In the feature extraction step, the API call graphs of each app are further parsed to collect the relationships between APIs. To use these relationships as functional features of the app, we propose a matrix structure. The value of the matrix is assigned according to our conversion method. Then every app can be represented as a matrix containing features.

(3) In the feature selection step, we first remove the features that are rarely appear based on the statistical results to avoiding over-fitting. After this process, the matrix is transformed into a vector that contains remain features. To further increase the effectiveness of the features, we design a search method to find the redundancy between features. Then the redundant features are removed and the final feature vector is obtained.

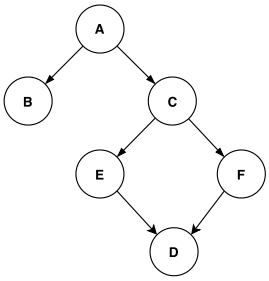
(4) In the training classifier step, using the feature vectors of the apps as input and the original categories of the apps as the labels, we train a classifier. Specifically, the CNN algorithm is applied to obtain the classification model. Finally, we can get the prediction results of the app categories by the trained CNN model.

B. STATIC ANALYSIS

The main purpose of this step is to generate API call graphs and to get a selected API list by analyzing the code structure of the apps. We define the API call graph as follows. It is a directed graph that each node in the graph represents an API called in the app's code. The directed edges in the graph are used to represent the execution order between the APIs. Branches in the same node represent different conditions for program execution. For example, according to our definition, the API call graph of the code depicted in Fig. 2(a) is shown in Fig. 2(b). From this graph, not only the APIs that the app uses is clearly revealed, but also the calling relationships between APIs can be obtained. The APIs recorded in API call

```
public void caller() {
    A();
    if(conditon 1) {
        B();
    } else {
        callee();
    }
}
public void callee() {
    C();
    if(condition 2) {
        E();
    } else {
        F();
    }
    D();
}
```

(a)



(b)

FIGURE 2. Example code and corresponding API call graph.

Landroid/app/Activity;->onCreate(Landroid/os/Bundle;) V
① ② ③

FIGURE 3. Example API invoke statement in smali file.

graphs are all from our selected API list, which contains part of the Android original APIs. The reason we do not use all the Android APIs is that not all APIs are used when designing apps. Besides this, some android APIs are not representative of specific program features, which is not helpful for us to get functional information. Therefore, we need to select an API list to filter out the API with strong functionality.

In order to achieve these goals, we leverage decompile technology to analyze the code in the app. We decompile the apk file by using Apktool [22] which is a tool for reverse engineering 3rd party, closed, binary Android apps. In this process, we can get another form of the code file named smali file, which is a middle code between Java and assembly code. These files contain APIs that are used in the app. In smali files, the syntax of an API call statement is shown in the Fig. 3. The first part represents the class in which the definition of the API is located, the second part represents the name and the parameter list of the API, and the third part is the type of the return value. Through the information in these three parts, we can distinguish between the app custom APIs and Android APIs. For the custom APIs, further search can be performed to figure out the Android APIs in its definition. To select the APIs that are more beneficial to our categorization, we apply TF-IDF(Term FrequencyâĂ"Inverse Document Frequency) algorithm to assign the weight to each of the Android APIs. The code of the app is a special kind of text, and the APIs can be seen as the choice of words in the text. Then the weight calculation formula is:

$$W_{ij} = \frac{S_{ij}}{N_j} \log \frac{L}{M_i} \tag{1}$$

where W_{ij} represents the weight of API i in category j, S_{ij} represents the number of apps using API i in category j, N_j represents the total number of apps of category j, and L represents the total number of apps. M_i represents the number of apps that use API i in all apps. The size of W_{ij} represent the ability of the API i to distinguish category j with other categories. All the value in the formula can be obtained by scanning the smali files. The ideal APIs for us are the APIs with category functionality, so we choose the top weighted APIs in each categories to form our selected API list.

After determining the selected API List, we design an algorithm to generate the API call graphs from the app's smali files. Since the Android app is executed by calling the lifecycle functions of Android components [23] that are registered in the AndroidManifest file, we use the lifecycle functions of each component as entries to generate the corresponding API call graphs. For each lifecycle function, by using string match technology we obtain the statements from the code in it. Follow the statements, we can either record the Android API or find the code of the custom API. By recursively traversing the statements, we could generate the API call graph. The detail of this process is described in **Algorithm 1**.

Algorithm 1 Process of Generating an API Call Graph From an Entry Function e_n

Input: smali files, e_n
Output: A directed graph and its entry node
initialize a node n_0
for each statement s_i in the smali code of e_n **do**
  if s_i is an API invoke statement **then**
    mark the API as a_m
    if a_m is in our selected list **then**
      create a new node n_j contains a_m
    else if a_m is a custom API **then**
      search the code of a_m, recursively generate sub-graph
    end if
  else if s_i is a condition control statement **then**
    search the code under the condition, recursively generate sub-graph
  end if
end for
return n_0

C. FEATURE EXTRACTION

The objective of this step is to transform the API call graphs in the previous step into a specific data structure that represents the features of the app. The feature information we mainly extract is the relationships between APIs. In order to store the features for each app, a matrix data structure is proposed. We define the matrix as:

$$M = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & a_{ij} & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}_{n\times n} \tag{2}$$

where a_{ij} represents the relationship between the API i and the API j in our selected API list. When the app does not use

API i and API j or API j is unreachable for API i in API call graphs, $a_{ij} = 0$. When $a_{ij} > 0$, it means that there is a path from API i to API j in API call graphs of the app and API j will be executed after API i. This also shows a combination relationship between API i and j. We use the numerical magnitude of a_{ij} to indicate the strength of this combination. To evaluate the combination strength, we define a parameter named distance. The distance is the number of edges included in the shortest path between the two nodes. Assume that in the function call graph, the closer the distance between the nodes of the two APIs, the greater the combination strength, and the farther the distance, the smaller the combination strength. Based on this assumption, we define the attenuation function of the combination strength as $H(d)$, where d is the distance. We add up all the combination strength calculated using $H(d)$ to assign value to a_{ij}. Taking $H(d) = \frac{1}{d}$ as an example, the matrix transformed from the API call graph in Fig. 2(b) is shown in the Equ. (3). From the perspective of program code, APIs that implement a function together are often called together, and there may be no functional association between the two APIs with large call intervals. Therefore, elements with larger values in the matrix can reflect the combination of APIs that play a major role in API call graphs. On the other hand, the strength of the API combination also represents the weight of each API combination in the app, which can reflect the major functionality in the app so that the specific category of the app can be predicted.

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1/4 & 1/2 & 1/2 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1/2 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \end{array} \tag{3}$$

To get the matrix of an app, the traversal of API call graphs is needed. The specific conversion process is shown in **Algorithm 2**. Taking the API call graphs as input and the API relationship matrix as output. Starting with the entry node of each graph, we perform breadth-first traversal in the graph. Then recording the traversal depth in the traversal process to represent the distance between nodes to calculate the combined strength between the nodes using the attenuation function, and then assign values to the corresponding positions in the matrix. When the combination is repeated, the value is added, indicating that the combination appears multiple times and should occupy a higher weight in the app. By recursively doing the same process to every node, the traversal of the relationship between the nodes in the whole graph can be completed, that is, the assignment of the matrix is completed.

D. FEATURE SELECTION

This step aims to reduce the number of features in the matrix to improve the stability and efficiency of our system. It is done in two sub-steps. The first sub-step is to remove the invalid features. After the feature extraction, we have the API relationship matrix for each app. Through our observation of these matrices, we find that these matrices are sparse matrices with a large number of elements that have a value of zero. This situation indicates some of the API combinations are never used in any app. These elements have no effect on the app categorization. To further find invalid elements, we perform a statistical analysis to get the frequency of occurrence of each element in the matrix. We sort the frequency of each element and plot histogram diagram shown in Fig. 4. As can be seen from the figure, in addition to the situation we mentioned, there is also a part of the elements has a very low frequency. This low-frequency element means that only a very small number of apps will use such API combination. In order to avoid over-fitting of classifiers and enhance stability, we have to remove the elements in these two cases. In this process, we filtered out a significant portion of the elements and then re-arranged the features into a vector form.

Although the features in the vector may be all valid now, there may be redundancy between them. Based on feature vectors, we conduct the second sub-step to remove

Algorithm 2 Process of Getting API Relationship Matrix From API Call Graphs

Input: API call graphs
Output: API relationship matrix
initialize a matrix M with all elements = 0;
for each API call graph as g_n **do**
  use deep-first traversal to get its nodes
  for each node $node_i$ in call graph g_n **do**
    use breadth-first traversal to get all the subnodes
    for each childnode $subnode_j$ of $node_i$ **do**
      compute the distance d between $subnode_j$ and $node_i$;
      get the index m, n of the two nodes in API list;
      $M[m][n] <= M[m][n] + H(d)$
    end for
  end for
end for
return M

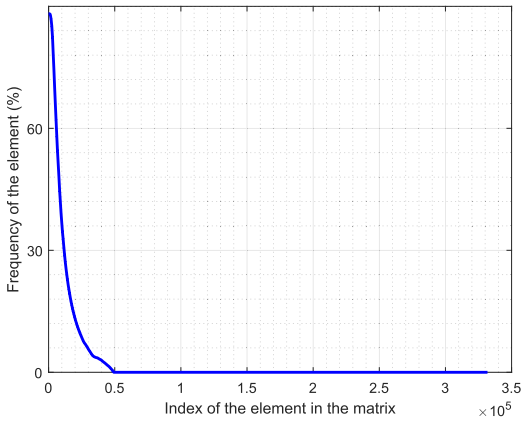


FIGURE 4. Histogram diagram of the matrix elements.

redundant features. The method we proposed to search for redundant elements is described as follows. In all the feature vectors, if there is a mapping relationship between two elements, then there is redundancy between them. This can be written in formula as:

$$\forall v_m, v_n = f(v_m) \tag{4}$$

where v_m, v_n is the m-th and n-th elements in the vector. In this case, we could remove v_n and the final categorization result will not be affected. Regarding the categorization problem as a condition probability problem. The probability of an app with vector V is predicted to be category C_i is:

$$P(C_i|V(v_1 = a_1, \ldots, v_k = a_k)) \tag{5}$$

To simplify, let us consider there are only two elements in the vector. Equ. (5) can be rewrite as:

$$P(C_i|v_m = a, v_n = b) \tag{6}$$

According to the definition of conditional probability,

$$\begin{aligned} P(C_i|v_m = a, v_n = b) &= \frac{P(C_i, v_m = a, v_n = b)}{P(v_m = a, v_n = b)} \\ &= \frac{P(v_n = b|C_i, v_m = a)P(C_i, v_m = a)}{P(v_n = b|v_m = a)P(v_m = a)} \end{aligned} \tag{7}$$

According to Equ. (4), when there is a mapping relationship between v_m and v_n, then

$$\begin{aligned} b &= f(a) \\ P(v_n &= b|v_m = a) = 1 \\ P(v_n &= b|v_m = a, C_i) = 1 \end{aligned}$$

Equ. (7) can be transformed to

$$\frac{P(C_i, v_m = a)}{P(v_m = a)} = P(C_i|v_m = a) \tag{8}$$

So the probability is not affected when v_n is removed from the condition. This proof reveals redundancy between two elements. When the probability is not affected, we should not use so many features. So why there is such kind of redundancy in our vector? A common situation is that app development often uses third-party libraries; especially the same kind of software may use the same library. Using the same library will result in the same part of the call structure in the API call graph, as shown in Fig. 5. When such a structure exists, a fixed API combination and a call relationship occur. That is, the combination of AB and BC always appears at the same time or disappears at the same time. This will result in redundancy in the API call relationship.

The detail of searching the redundancy between two elements is described in **Algorithm 3**. We denote n, m as the two elements' indices in the vector. Then traversing all the vectors and recording the values of these two elements by using a map structure. If the values of the two elements correspond one-to-one, it means that there is a mapping relationship between them. Otherwise, there is no redundancy between them.

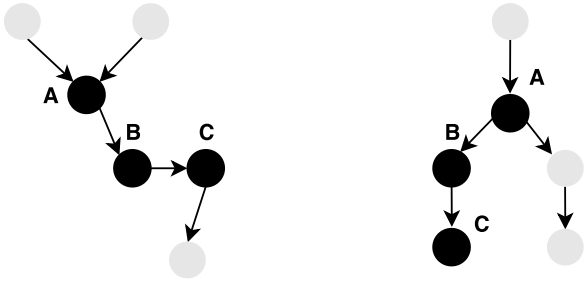


FIGURE 5. Example of same structure in different graphs.

Algorithm 3 Determine Whether There Is a Mapping Between Two Elements

```
Input: Vector index m,n, all feature vectors
Output: Boolean
  initialize a empty map named map
  for each feature vector v_i in all feature vectors do
    if v_i[m] in map then
      if map[v_i[m]] != v_i[n] then
        return FALSE
      end if
    else
      map.put(v_i[m],v_i[n])
    end if
  end for
  return TRUE
```

E. TRAINING CLASSIFIER

During this step, the features extracted from the previous step will be used to train the classifier. In order to explore the mapping relationship between the feature vectors we extracted and the app categories, we use CNN algorithm as the classifier model. Firstly, a CNN model is trained with a sufficient number of app feature vectors and corresponding app category labels. The trained model is then used to categorize unknown apps based on their feature vectors. The CNN model contains a lot of hyperparameters that determine the prediction results, and the parameters are constantly updated based on the results each time we train. In our model, we used the Adam optimizer to update the parameter values. We conducted experiments with different structures and parameters, and the final adjusted model structure is shown in the Fig. 6. It consists of two convolution with maxPool layers and two fully connected layers. The final output is a vector the length of which is the same as the number of categories. Then the softmax function is used to get the probability distribution of each category, and finally the prediction result is obtained. Suppose we denote $p_{i,c}$ as the probability of predicting an app i as category c. The loss function can be written as Equ. (9) using cross-entropy loss [24]. During the training process, as the parameters are updated, the value of the loss function will continue to decrease. When the value is no longer decreasing or fluctuating, the model is trained. That is,

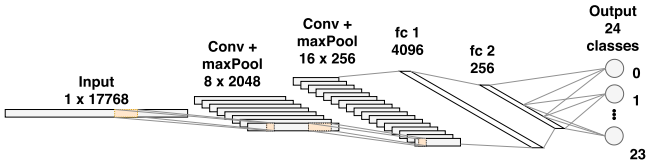


FIGURE 6. The structure of CNN model.

TABLE 1. Category labels.

BOOKS AND REFERENCE	BUSINESS	COMICS
COMMUNICATION	EDUCATION	ENTERTAINMENT
FINANCE	HEALTH AND FITNESS	LIBRARIES AND DEMO
LIFESTYLE	MEDIA AND VIDEO	MEDICAL
MUSIC AND AUDIO	NEWS AND MAGAZINES	PERSONALIZATION
PHOTOGRAPHY	PRODUCTIVITY	SHOPPING
SOCIAL	SPORTS	TOOLS
TRANSPORTATION	TRAVEL AND LOCAL	WEATHER

we can use it to predict the category.

$$Loss = -\frac{1}{N}\sum_{i=0}^{N-1}\log p_{i,c} \tag{9}$$

IV. EVALUATION

A. EXPERIMENT SETUP

To evaluate the feasibility and performance of our proposed method, we conduct several experiments. DroidARA is implemented using Java (J2SE 1.8) with Apktool (v2.4.0) and Python (v3.6.5) with Pytorch (v1.2) which is an open source machine learning framework. It runs in a Dell PowerEdge R720 server with a 2.8 GHz Intel Xeon E5-2680 CPU and 96 GB RAM, which runs Ubuntu 16.04 LTS 64-bit.

The apps we use are all collected from Google Play. In order to make our system easy to maintain the category structure of the existing market, we use the category labels originally defined by Google Play to classify these apps into 24 categories listed in Table 1 and take the label of the app assigned by Google Play as the ground truth. To avoid data imbalance, we obtain the same number of sample apps from each category. After removing invalid data(empty file or cannot be decompiled), we get 19949 apps. Then we choose 20% of the data as our test set and 80% as our train set. Using the pre-mentioned device environment, it takes 319.4s to train the CNN model and 53.2s to test the model. On average, DroidARA takes about 1.2ms for each prediction. The metric of our experiment is the accuracy of the prediction for the app category since it directly reflects performance of the system. we define accuracy as follows. We denote N_T as the number of the predictions that are same with the ground truth, and N_{all} as the number of all the predictions. Then the accuracy can be calculated as:

$$acc = \frac{N_T}{N_{all}}.$$

B. PERFORMANCE OF APP CATEGORIZATION

1) NUMBER OF APIs IN SELECTED API LIST

As mentioned in Section III, in static analysis step, we select top-weighted APIs in each category to form our selected API list. The number of chosen APIs in each category will affect the prediction result. To assess this impact, we select different numbers of APIs for experiments. Fig.7 depicts the results of these experiments. As can be seen from the figure, when the number of selected API is small, as the number increases, the prediction accuracy also increases. When the number reaches a certain value, increasing the number will not improve the performance. It means that while more APIs can introduce more information, there is a ceiling. So the best value to choose is 24.

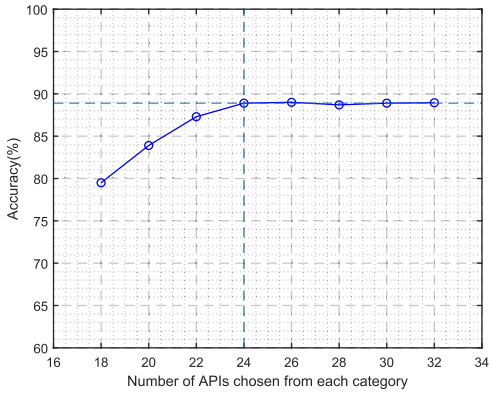


FIGURE 7. Accuracy changes under different API quantities.

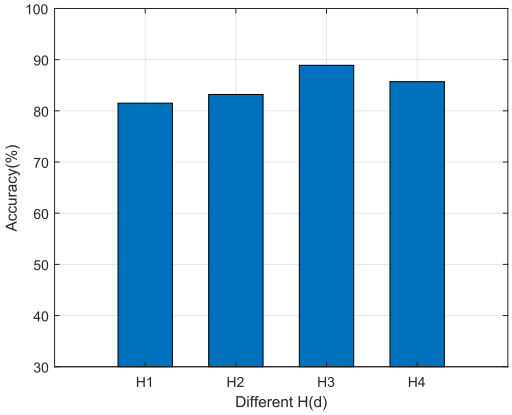


FIGURE 8. Accuracy of different attenuation functions.

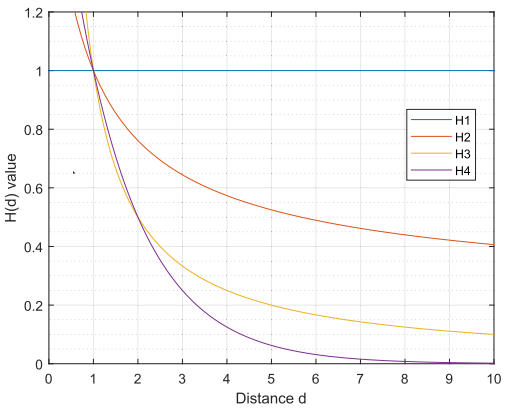


FIGURE 9. Function graphs with different attenuation functions.

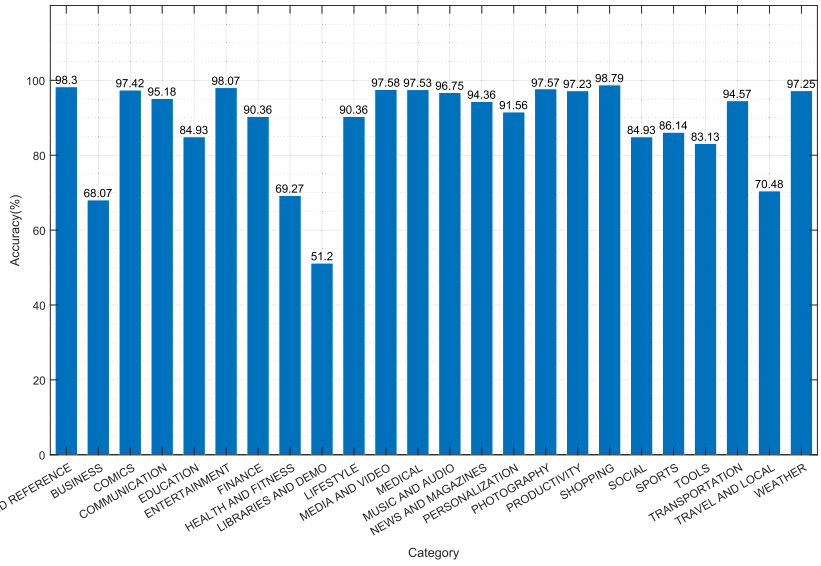


FIGURE 10. Prediction accuracy in each category.

TABLE 2. Expressions of different attenuation functions.

Function	Expression
H1	1
H2	$\frac{1}{ln(d+e-1)}$
H3	$\frac{1}{d}$
H4	$\frac{1}{2^{(d-1)}}$

2) DIFFERENT ATTENUATION FUNCTIONS

In feature extraction step, we define an attenuation function $H(d)$ to calculate the combination strength. Considering that different functions produce different calculation results, we test several different attenuation functions and recorded the corresponding prediction results. In general, the attenuation function should be a monotonically decreasing function. We choose three typical decreasing functions for testing and use a constant function as a reference. The expressions of these functions are shown in Table 2.

As shown in Fig.8, All three decreasing functions perform better than constant function. It indicates that the attenuation calculation we proposed is valid. Among them, H2 performs the best. To further explore the reasons for this, we put together the function graphs of the three functions for comparison in Fig. 9. When $d > 5$, the value of H4 is nearly zero, and the value of H2 is above 0.5. This means that the H4 decreases too fast and H2 decreases too slow. A faster decline will cause some near-combination effects to be over-eliminated while a slower drop will make the effects of attenuation too small. In contrast, H3 is more able to reflect the attenuation of the combination strength.

3) FEATURE SELECTION

In feature selection step, we remove the features that have a very low frequency. The way to determine if it has low frequency is to compare the frequency that it appears to a threshold value. In order to adapt the method to different size data sets, we set the threshold as a percentage of the total amount of data. We test the results of removing features at different scales which is depicted in Fig. 11. In the beginning, when we remove the features the accuracy increases. But if we continue to remove more features, some valid features are removed too, which causes the accuracy to decrease. So in this case, we choose the threshold value that performs best.

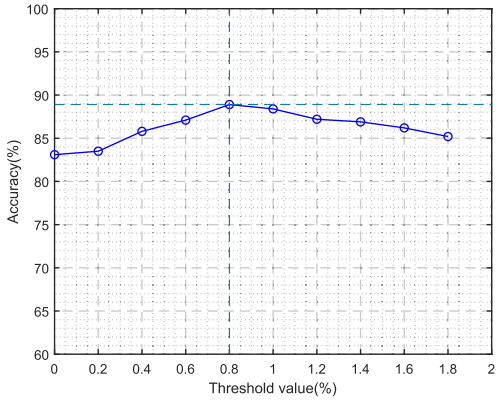


FIGURE 11. Accuracy at different thresholds.

TABLE 3. Features used in different methods.

Method	Features
Proposed	API relationships
Ref 1 [14]	API
Ref 2 [16]	permission, string and description
Ref 3 [13]	API, permission, string, .etc

4) COMPARISON WITH EXISTING METHODS

In order to demonstrate that our proposed method is an improvement of the existing work, we use some of the methods proposed in the reference to conduct some

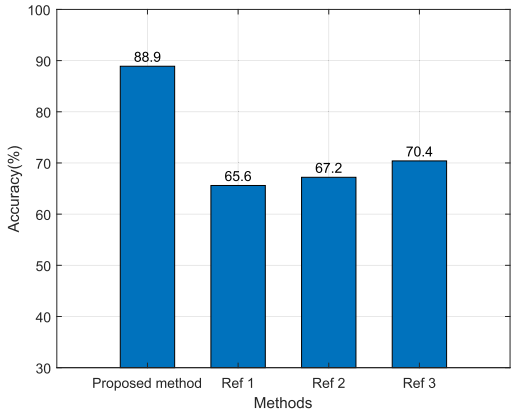


FIGURE 12. **Comparsion of different methods.**

comparative experiments. In specific, we use the features chosen in [13], [14], [16] and the corresponding classification methods to evaluate their performance, respectively. The difference between methods is shown in Table 3. We choose three existing methods that use different types of features, and the accuracy of each method is shown in Fig. 12.

It can be seen from the results that our proposed method has an improvement in performance than the existing methods. As mentioned in Section II, these methods only simply combine the binary value of the features into vectors and do not obtain further information. Since there are many apps containing the same such features, that will result in the loss in accuracy. And although Ref 3 combines many types of features together, they didn't remove the redundant between features so their method has little improvement. In contrast, our proposed method not only mine the relationships between APIs but also assigns values to it in real number range which makes apps more distinguishable.

5) TOTAL PREDICTION RESULTS

The prediction result of each category app is shown in Fig. 10. The highest accuracy is 98.7% while the lowest is 51.2%. And the average accuracy is 88.9%. In our categorization, most categories perform well. The low accuracy of some categories may due to the unclear category definition. For example, the category LIBRARIES AND DEMO performs worst, while it is hard to determine what functions should be included in this category according to the name. So there may be many miscategorized apps. Compared to this, some categories such as WEATHER perform well when their names directly reflect the function.

V. CONCLUSION

In this paper, we introduced DroidARA, an automatic Android app categorization system based on API relationships analysis. To our best knowledge, it is the first approach to categorize apps by analyzing the information of relationships between API calls.

Given a new app, firstly the API call graphs will be constructed using our designed analysis method. Then the relationships between API calls are extracted from these API call graphs and a matrix structure data is generated. Two feature selection methods are designed for the matrix to reinforce the feature elements so that the accuracy and efficiency of the system will be improved. After that, the matrix is transformed into a feature vector as the input of a trained CNN model. Finally, the CNN model will output prediction for the category of this app.

In experiments, we tested 19949 apps collected from Google Play. Using the original category labels, we divided the apps into 24 categories. After the system parameters adjusted according to the experiment, DroidARA achieved an average 88.9% accuracy which outperformed existing methods. This result demonstrated that DroidARA could be used in real-world app market.

In the future, we aim at mining more further information inside the application to make more use of other types of feature such as text. And exploring some ways to improve the system. For example, finding a more suitable attenuation function or extending the selection method to widely use for removing redundancy.

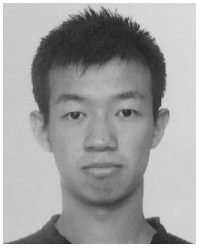
REFERENCES

[1] (2019). *Smartphone os Market Share*. [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp
[2] Google. *Play Store*. Accessed: 2019. [Online]. Available: https://play.google.com/store
[3] Amazon. *Appstore*. Accessed: 2019. [Online]. Available: https://www.amazon.com
[4] AppBrain. (2019). *Android Apps on Google Play*. [Online]. Available: https://www.appbrain.com/stats/number-of-android-apps
[5] A. Bajaj, S. Krishna, H. Tiwari, and V. Vala, "Learning mobile app embeddings using multi-task neural network," in *Proc. Int. Conf. Appl. Natural Lang. Inf. Syst.* Cham, Switzerland: Springer, 2019, pp. 29–40.
[6] K. Singla, N. Mukherjee, and J. Bose, "Multimodal language independent app classification using images and text," in *Proc. Int. Conf. Appl. Natural Lang. Inf. Syst.* Cham, Switzerland: Springer, 2018, pp. 135–142.
[7] X. Liu, H. H. Song, M. Baldi, and P.-N. Tan, "Macro-scale mobile app market analysis using customized hierarchical categorization," in *IEEE 35th Annu. IEEE Int. Conf. Comput. Commun. (INFOCOM)*, Apr. 2016, pp. 1–9.
[8] V. Radosavljevic, M. Grbovic, N. Djuric, N. Bhamidipati, D. Zhang, J. Wang, J. Dang, H. Huang, A. Nagarajan, and P. Chen, "Smartphone app categorization for interest targeting in advertising marketplace," in *Proc. 25th Int. Conf. Companion World Wide Web*, Apr. 2016, pp. 93–94.
[9] D. Surian, S. Seneviratne, A. Seneviratne, and S. Chawla, "App miscategorization detection: A case study on Google play," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 8, pp. 1591–1604, Aug. 2017.
[10] G. Berardi, A. Esuli, T. Fagni, and F. Sebastiani, "Multi-store metadata-based supervised mobile app classification," in *Proc. 30th Annu. ACM Symp. Appl. Comput.*, Apr. 2015, pp. 585–588.
[11] A. A. Al-Subaihin, F. Sarro, S. Black, L. Capra, M. Harman, Y. Jia, and Y. Zhang, "Clustering mobile apps based on mined textual features," in *Proc. 10th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, Sep. 2016, Art. no. 38.
[12] W. Wang, Y. Li, X. Wang, J. Liu, and X. Zhang, "Detecting Android malicious apps and categorizing benign apps with ensemble of classifiers," *Future Gener. Comput. Syst.*, vol. 78, pp. 987–994, Jan. 2018.
[13] M. R. Hamedani, D. Shin, M. Lee, S.-J. Cho, and C. Hwang, "AndroClass: An effective method to classify Android applications by applying deep neural networks to comprehensive features," *Wireless Commun. Mobile Comput.*, vol. 2018, Sep. 2018, Art. no. 1250359.
[14] F. Dong, Y. Guo, C. Li, G. Xu, and F. Wei, "ClassifyDroid: Large scale Android applications classification using semi-supervised Multinomial Naive Bayes," in *Proc. 4th Int. Conf. Cloud Comput. Intell. Syst. (CCIS)*, Aug. 2016, pp. 77–81.

[15] A. Shabtai, Y. Fledel, and Y. Elovici, "Automated static code analysis for classifying Android applications using machine learning," in *Proc. Int. Conf. Comput. Intell. Secur.*, Dec. 2010, pp. 329–333.
[16] C. Yuan, S. Wei, Y. Wang, Y. You, and S. ZiLiang, "Android applications categorization using Bayesian classification," in *Proc. Int. Conf. Cyber-Enabled Distrib. Comput. Knowl. Discovery (CyberC)*, Oct. 2016, pp. 173–176.
[17] D. H. Park, M. Liu, C. Zhai, and H. Wang, "Leveraging user reviews to improve accuracy for mobile app retrieval," in *Proc. 38th Int. ACM SIGIR Conf. Res. Develop. Inf. Retr.*, Aug. 2015, pp. 533–542.
[18] H. Zhu, E. Chen, H. Xiong, H. Cao, and J. Tian, "Mobile app classification with enriched contextual information," *IEEE Trans. Mobile Comput.*, vol. 13, no. 7, pp. 1550–1563, Jul. 2014.
[19] D. L. Ben Lulu and T. Kuflik, "Wise mobile icons organization: Apps taxonomy classification using functionality mining to ease apps finding," *Mobile Inf. Syst.*, vol. 2016, Nov. 2015, Art. no. 3083450.
[20] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android permission specification," in *Proc. ACM Conf. Comput. Commun. Secur.*, Oct. 2012, pp. 217–228.
[21] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, Oct. 2011, pp. 627–638.
[22] C. T. R. Wi Sniewski, *Official Site of Apktool*. Accessed: 2019. [Online]. Available: https://ibotpeaches.github.io/apktool/
[23] Android Developers. *Application Fundamentals*. Accessed: 2019. [Online]. Available: http://developer.android.com/guide/components/fundamentals.html
[24] C. M. Bishop, *Pattern Recognition and Machine Learning* (Information Science and Statistics). New York, NY, USA: Springer, 2006.

WENHAO FAN received the B.E. and Ph.D. degrees from the Beijing University of Posts and Telecommunications (BUPT), Beijing, China, in 2008 and 2013, respectively. He is currently an Associate Professor with the School of Electronic Engineering, BUPT.

His main research interests include network and information security, parallel and distributed computing, and computer networks.



YE CHEN received the B.E. degree from the Beijing University of Posts and Telecommunications (BUPT), Beijing, China, in 2017, where he is currently pursuing the master's degree with the School of Electronic Engineering. His main research interests include network and information security, android software, and computer networks.

YUAN'AN LIU received the B.E., M.Eng., and Ph.D. degrees in electrical engineering from the University of Electronic Science and Technology, Chengdu, China, in 1984, 1989, and 1992, respectively.

He is currently a Professor with the Beijing University of Posts and Telecommunications (BUPT), Beijing, China, where he is also the Dean of the School of Electronic Engineering. His main research interests include network and information security, pervasive computing, wireless communications, and electromagnetic compatibility.

Prof. Liu is a Fellow of the Institution of Engineering and Technology, U.K., the Vice Chairman of the Electromagnetic Environment and Safety of the China Communication Standards Association, the Vice Director of the Wireless and Mobile Communication Committee, Communication Institute of China, and a Senior Member of the Electronic Institute of China.

FAN WU received the B.E. degree from the University of Electronic Science and Technology of China, Chengdu, China, in 2004, and the Ph.D. degree from the Beijing University of Posts and Telecommunications (BUPT), Beijing, China, in 2009. She is currently an Associate Professor with the School of Electronic Engineering, BUPT. Her main research interests include network and information security, and wireless sensor networks.