Design of an API recommendation system in android programming

Jinyang Liu1* and Zhiyi Ma2,3,4
1Academy for Advanced Interdisciplinary Studies, Peking University, Beijing, China
2School of Electronics Engineering & Computer Science, Peking University, Beijing, China
3Key laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China
4Author to whom any correspondence should be addressed mazhiyi@pku.edu.cn
*jyliu.aais@pku.edu.cn

Abstract. Android has been the most popular operation system in the smartphone field, and Android programming is an import branch in programming work. When dealing with programming tasks, there are many APIs that are often used by programmers, and it costs much time for them to choose and check the usages of APIs repeatedly. This paper raises a design of API recommendation system in android programming, which uses AST tools in Java to extract key features of codes, and uses tf-idf, LDA and API2Vec models on the data to generate recommendation results. The experiment results—the maximum 80% overall recall rate and nearly 50% top-10 recall rate prove the efficiency of the system.

1. Introduction
Over the past decades, smart phones have been a most popular goods all around the world, and the systems and applications running on the phone platforms also have become a research hotspot of computer science. As the main, or in a sense, “the only” open source operation system of smart phone platform, Android system has developed a lot during these years. According to statistic data from Kantar[1], 77.4% smart phones in China and 59.1% of them in US is in Android system at March 2018. As a huge amount of people are working on Android, to design and develop new applications, there is a strong need of them which is not resolved, that is, a systematic and effective algorithm or tool which can help them in coding. As we all know, to write a program file even just a code sequence is a hard mission that needs knowledge of programming language, the comprehension of functions, and logical reasoning. So, nowadays computer cannot fully complete a programming task without any human’s participation (expect for some easy and patterning tasks). However, many researches have been made on the topic of computer doing programming work as an assistant of human. In many popular IDEs (Integrated Development Environment), the software could point out grammar mistakes of the codes, and list out the existing variable and function names to save programmers’ time for typing them again and again. Early stage of the research focuses on grammar and structural attributes of the code, and when the generation of data arrived, recent researchers do more work and make more progresses on the content and semantics of codes, to let computers comprehend more about the function and objective of them. With trained model on big data, new algorithms may have the ability of complete code sequences based on context, and recommend coding elements for programmer in the deduction of the algorithm the programmers may use after the existing code fragment.
This paper provides a model for Android-programming that can generate recommendations for programmers of API usage, which is a content-based method and designers have tested several data-processing models on it. The model contains three parts: one is a module of transferring codes into a new data form for computer to process with, the second is a model to learn and develop with the data, and the last is a tool to generate coding sequences or recommendations based on the results of the model. The rest part of this paper is organized as follows: Part 2 introduces existing works of others in the area, part 3 describes the main parts of the model—— preprocessing for code files and learning model of processed data. Part 4 is a display and analysis of the experimental result, and part 5 is a conclusion.

2. Related Work

Many researches have been made in recent years, and this paper could only introduce some representative ones. Back to 2000, Ye et al. [2][3] designed a programming assisting system called CodeBreaker, which can recommend code sequences for users. It is based on a searching technology, and Bajracharya et al.[4] at 2010 did improved work on the same topic. In 2017, Nyugen[5] use Word2Vec on APIs in the codes to made a new vector representation of codes in searching or other methods of recommendation. In 2015, Lu et al.[6] raised a new searching technics based on WordNet, which has a good experiment result. Focusing on API use recommendation, Stylos [7] designed a tool of searching API class, methods and use examples called Mica. Chan et al.[8] give an API searching method based on a graph model representing the relationship between APIs and their invocations. On the strategy of API recommending, Bruch et al.[9] raised 3 common strategies, and Proksch et al.[10] designed a model of Bayesian Network. Nguyen[11] developed a tool of API recommendation based on Hidden Markov Model(HMM), which is called DroidAssist. Also, he gave an API embedding method called API2Vec, and this paper will discuss about it later.

3. The recommendation model

To help readers understand the work and model of this paper better, figure 1 is a complete sketch of our system. Firstly, the paper will introduce the code data preprocessing unit in our model.

Figure 1. The whole structure of our recommendation system.

3.1. Data preprocessing unit

Android programs are written in Java Language, and figure 2 shows a typical Android code fragment. It is easy to find out that a code contains plenty of information and features, and when code data is used in code recommendation system, designers need to extract some key attributes of code itself, as there are redundant information in the code that is irrelevant with the content and subject of the code, for example, names of local variables, structural statements (such as loop and determine statements),
and so on. In the model this paper provides, it extracts 2 kinds of information in the code: The imported packages and the invocation methods. This two kinds of information contains main semantic features of a code. To generate a new object of an existed class, can be regarded as to invoke a constructor of a class, so it is also in the information field the model takes account into. Other elements in the code, such as a definition of a local class, will be only used in at most one program set, so this model does not extract this kind of information. Figure 3 and table 1 shows the original Java code, and the corresponding data extracted from it in this method. After the extraction step, a code is transferred into a list of “imports” and “invocations” for models to process with, which can be called an I-I list. The elements of the list is sorted by the visit order of the grammar tree of the code, so it also includes some of the information of the order in which the element appears in the code (though a code is a tree structure other than a 1-dimension list, so the list could not describe the order of execution completely).

```
package packageA
import A.B.C
import D.E.F

public class CLASSA extends B implements C {
    private TYPEA a;
    private TYPEB b;
    private CLASSTYPE ca = new CLASSTYPE();

    public Foo(TYPEA a) {
        F1(a);
    }

    public Foo2(TYPEA a, TYPEB b) {
        F2(a, b);
        F3(acts);
    }
}
```

**Figure 2.** part of an Android code file and the instruction of some import elements.

Next, this paper introduces the method the model used in extraction step in detail. To extract the information needed, the model fulfills the task by generating, visiting and parsing the AST (Abstract Syntax Tree). The data structure AST represents codes in a tree form, and each node contains an element in the code, which is distributed in logical order and easy to be parsed. In Java Language, Eclipse provides useful classes in JDT for programmer to view and extract the information of grammar/syntax tree in the org.eclipse.jdt package[12], and with the ASTNode classes and the ASTVisitor Class, a parser tool is designed that can parse the syntax tree of any Java codes, and generate the lists of imports and invocations from them. With looking into the AST information deeply, our tools can distinguish the methods that have the same name but come from different fields. The classes and methods that is defined in a single android project is abandoned from our data extraction, for example, an imported class whose name is started with ‘com.’. Figure 4 is an example of extraction.
Table 1. Code elements extracted from the code in figure 3. The red lines are the elements used in building the I-I list.

| Type                  | Name                                      | Field          |
|-----------------------|--------------------------------------------|----------------|
| Import                | com.sun.istack.internal.Nullable           |                |
| Import                | java.lang.Override                        |                |
| Class                 | User                                       |                |
| Field                 | String                                     | foo            |
| Field                 | Foo                                        | a              |
| MethodDeclaration     | UserMethod                                 | User           |
| Class                 | Foo                                        |                |
| Field                 | String                                     | fooString      |
| MethodDeclaration     | FooMethod                                  | Foo            |
| MethodDeclaration     | TestMethod                                 | User           |
| MethodInvocation      | FooMethod                                  | Foo            |

Figure 3. The original code.

```java
package com.remarkable.javaparser;
import com.sun.istack.internal.Nullable;
import java.lang.Override;

public class User {
    @Nullable
    private String foo = "123123";
    private Foo a;
    public void UserMethod() {}

    static class Foo {
        private String fooString = "123123";
        public Foo FooMethod() {
            Foo g;
            return g;
        }
    }
    public void TestMethod() {
        Foo f;
        f.FooMethod();
    }
}
```

Figure 4. An example of the extraction process, the code is on the left and the extracted features are on the right. Reader can find that some features of this code have been abandoned.

```
import org.eclipse.jdt.core.dom.CompilationUnit;
import com.sun.istack.internal.Nullable;
import java.lang.Override;

public class DemoVisitorTest {
    public DemoVisitorTest(String path, String despath) throws FileNotFoundException {
        CompilationUnit comp = JvmUtil.getCompilationUnit(path);
        DemoVisitor visitor = new DemoVisitor[];
        PrintStream p = new PrintStream(new FileOutputStream(despath));
        comp.accept(visitor);
    }
    public static void main(String[] args) throws FileNotFoundException {
        String desFilePrefix = "";
        new DemoVisitorTest(desFilePrefix);
    }
```

After that, the model needs to do filtering work on the lists: just like many NLP (Natural Language Processing) tasks, elements that appear below certain times would be erased from data. To make our experiment more convincing, several thresholds of appearing times are chosen to compare the results.

The database this paper used is collected from GitHub, partly from the GitHub Java Corpus dataset and partly is collected by the designers. It has 95,505 code files in all. After the extraction, some statistics data is listed in figure 5. From the statistics data, it can be found out that there are a huge number of APIs and their usages/invocations in the dataset, and the times of each element is used are
subjected to a typical long-tail distribution. Comparing with the average length of the I-I lists of codes, the data to be used is really sparse, which is a great challenge.

3.2. Data processing and training models
Part 3 talks about the preprocessing step in figure 1, and in this part this paper will introduce the key part in our model: the model used to learn on the data fetched from codes. As is mentioned before, the goal of this paper is to build an API recommending method based on the content other than grammar rules. Thus, 3 semantic models in NLP are tested on the model to get a vectorized representation of the codes for recommendation. In the recommending step, a code is vectorized with the model, the most similar codes in database are found as the recommendation. The models this paper use are tf-idf model, LDA model and API2Vec model. These models provide varied vector representations of codes, which are frequently used in similar tasks.

3.2.1. tf-idf Model. As one item may appear in an I-I list for many times, to use a normal one-hot vector representation for an I-I list is not adequate for our task. To take the appearing times and total appear frequency of one term into account, tf-idf model can be used to make a vector representation of an I-I list. If one uses the traditional aspect of recommending system, to regard one code as a user and one term as an item, the tf-idf vector can give the “interest” of one code on its terms, with the weights of each term as its components.

The weight of each term in a document (in the task, the “document” means the I-I list) in tf-idf model is calculated as following [13]:

$$\text{tfidf}(t,d,D) = \text{tf}(t,d) \times \text{idf}(t,d)$$

(1)

where \( tf \) is called term frequency and \( idf \) is inverse document frequency. \( \text{tfidf}(t,d,D) \) is the tf-idf weight of term \( t \) in document \( d \) out from corpus \( D \).
tf is the frequency of one term in the document:

\[ tf(t, d) = \frac{f(t, d)}{\sum_{t' \in D} f(t', d)} \quad (2) \]

\( f(t, d) \) is the count/frequency that term \( t \) appears in document \( d \).

\( idf \) is a modifying factor that evaluating the popularity of one term in all the documents:

\[ idf(t, d) = \log \frac{N}{|\{d \in D : t \in d\}|} \quad (3) \]

\( N \) is the total amount of documents, and the denominator of \( \log \) is the count of documents that \( t \) appears in it. The more popular a term is, the smaller corresponding \( idf \) factor is. \( idf \) factor is used to balance the weights of terms between popular ones and unpopular ones. For example, if \( A \) is a very popular term that appears in many codes, \( B \) is a term that is rarely used, and they both appear in a code at the same time, the “interest” of this code on \( B \) should be much more than \( A \), which would be reflected on the \( tf-idf \) weights. This feature is very important for the task, as there are really popular package and methods in our database, and much more rarely-used terms.

After \( tf-idf \) vectorization, an I-I list is transferred into an \( N \)-dimension vector, where \( N \) is the total amount of terms, and each component is the weight/interest of one term in the code. The model will use these vectors to rank the recommendation results of single terms, and can calculate the similarities between the codes, where other methods will also be adopted.

3.2.2. LDA. To calculate the similarities between codes, except for directly using terms in the I-I lists, the model can also extract some semantic features from the code, for example, the “topics”. There have been many topic models, such as LSA (Latent Semantic Analysis), p-LSA (probabilistic Latent Semantic Analysis), and LDA (Latent Dirichlet Allocation). The last one is the topic model the model like to adopt. LDA is raised by David Blei et al. [14], which is in some extent an improvement of p-LSA model. It regards documents as mixed-model of topics, and each word is corresponding to a distribution vector of topics, that is, each word has a contribution factor to each topic. Actually, LDA is a directed graph model with the theory of Bayesian Estimation. After training the LDA model on the corpus, the model could generate a topic distribution for each code, and use the topic vector (the dimension is number of topics) to calculate similarities.

3.2.3. API2Vec. API2Vec, as the implication of its name, is a method derived from Word2Vec, the widely-used method in the vectorization of natural language data/vector embedding or words. Word2Vec[15] uses a neural network model called CBOW model to train a vector representation for words. Nguyen et al. in [16] applied the Word2Vec on Java codes data, then examined and proved the availability and practicability of API2Vec that use API sets as input data. Figure 6 shows the CBOW model in Word2Vec and the application of this model into API learning. In the task from this paper, API2Vec model cam also be used to learn a vector for each term, then a matrix representation of code files can be formed for similarity calculation. For each code file, the normalized average vector of terms in it is used for vector representation of the whole file.

3.3. Generate recommendation results
The model of this paper used the following method for API-use recommendation: For an incomplete code, it uses one of the former methods to calculate the similarities between it and the codes already in the data corpus. The model chooses the \( K \) most similar codes of the incomplete one, and provides the APIs in them as recommendations. The APIs is ranked by the accumulated \( tf-idf \) weights in the \( K \) codes. In the next part, this paper will show the detailed experiment results in the database.
Figure 6. (a) CBOW Model and its application on API data. This figure shows a CBOW model with window length \((2n+1)\). (b) How API2Vec works. The model is trained on code corpus, and generate matrix representation on any given code.

4. Experimental Results

The database this paper use has 95,505 code files, and the paper randomly select about 1,000 from them as test data, and train our models on the rest. When testing the model, on each test file, one term is randomly removed from its I-I list, and system will check if the recommendation result contains the lost one. The overall recall rates are listed in Table 2 with varied thresholds of how to remove barely-used terms. This paper use 10 as the value of factor K. For LDA and API2Vec models, the paper tested several parameter groups, and chose out the best ones to provide results. As for LDA, number of topics is chosen in 30, 50, 70 and 100. For API2Vec, the dimension of vectors is chosen from 100, 200 and 300, and the length of window is chosen between 5 and 10. The parameters are elected for the range in which earlier primal experiment show good performance.

| Threshold | tf-idf | LDA | API2Vec |
|-----------|--------|-----|---------|
| 3         | 79.84  | 72.94 | **81.03** |
As table 2 shows, tf-idf and API2Vec methods outperform the LDA model, and these two have close evaluation scores. To evaluate and compare tf-idf and API2Vec deeply, table 3 provides the top-N recall rates, that is, only return N terms that have most accumulated tf-idf weights in recommendation results.

| Threshold | tf-idf | API2Vec | tf-idf | API2Vec | tf-idf | API2Vec | tf-idf | API2Vec |
|-----------|--------|---------|--------|---------|--------|---------|--------|---------|
| 5         | 81.05  | 71.19   | 81.86  |         |        |         |        |         |
| 10        | 80.23  | 73.12   | 82.23  |         |        |         |        |         |
| 20        | 79.92  | 73.02   | 83.01  |         |        |         |        |         |
| 50        | 78.05  | 71.77   | 83.07  |         |        |         |        |         |
| 100       | 79.02  | 72.72   | 84.21  |         |        |         |        |         |

Table 3. Top-N recall rates

On top-5 recall rates, API2Vec does not perform so well, but it has obviously better performance on other top-N recommendation results. In practical scene, a weakness of API2Vec is that it costs more time than tf-idf to calculate code similarities. Whatever, these two methods have proved themselves to be efficient in API recommendation task of Android programming.

From the experimental results, it is concluded that using API2Vec representation for code files in similarity calculation, and use tf-idf weights in term ranking performs best. The shortage of LDA model is that, when “topics” are extracted from data, the representation loses some features, where the topics may not describe the code fully. Also, the lengths of I-I lists in our task are varied, some of which are great, and the most of which are small. On short text data, conventional LDA model cannot perform as well as that on long text data.

As the thresholds of API used times increases, the precision of recommendation also increases naturally. Thus, if users focus on APIs in a certain domain, they can have good feedback from the system. With the expansion of database and optimization of parameters, API2Vec will show significant growth of effects. Nevertheless, due to huge amount of APIs in data, and the sparsity of distribution of them, there would be many redundant APIs on the recommendation rank list, which lowers the top-N recall rate. In future work, designers will deal with the problem by applying other technics, e.g., using grammar rules to erase some candidates, to improve the top-N recall rates.

5. Conclusion

This paper discusses about the API recommendation task in the Android programming. The paper builds a recommendation system, and evaluate several model units on it. A mixed model of API2Vec and tf-idf has a relatively good performance, but still has considerable improvement space in top-N recommendation recall rate. As mentioned before, our experiment and the real application scene both must face with a data source that is much sparser than the data in stereotype recommendation systems, which contributes to the great difficulty in this task. As for recommendation algorithms, some researchers have tried to use deep learning models independently or mix the network into Collaborative Filtering. However, deep learning model cannot deal with our task well as the number of terms need a network model with great number of parameters to be built, which challenges the
computing environment strongly, and questions the availability of its self. Also, deep learning methods, such as LSTM and AutoEncoder have not shown its considerable advantage over traditional algorithms. With the improvement of database size and computing environment, the result of the model and system in this paper will show more competitive power.

References
[1] Kantar Worldpanel, https://www.kantarworldpanel.com/
[2] Ye, Y., & Fischer, G. (2002). Supporting reuse by delivering task-relevant and personalized information. International Conference on Software Engineering (pp.513-523). IEEE.
[3] Ye, Y., Reeves, B., & Reeves, B. (2000). Integrating active information delivery and reuse repository systems. Century Applications (Vol.25, pp.60-68). ACM.
[4] Bajracharya, S. K., Ossher, J., & Lopes, C. V. (2010). Leveraging usage similarity for effective retrieval of examples in code repositories.
[5] Nguyen, T. N., Nguyen, T. N., Nguyen, T. N., Nguyen, T. N., & Nguyen, T. N. (2017). Combining Word2Vec with revised vector space model for better code retrieval. Ieee/acm, International Conference on Software Engineering Companion (pp.183-185). IEEE.
[6] Lu, M., Sun, X., Wang, S., Lo, D., & Duan, Y. (2015). Query expansion via WordNet for effective code search. IEEE, International Conference on Software Analysis, Evolution and Reengineering (pp.545-549). IEEE.
[7] Stylos, J., & Myers, B. A. (2006). Mica: A Web-Search Tool for Finding API Components and Examples. Visual Languages and Human-Centric Computing (pp.195-202). IEEE Computer Society.
[8] Chan, W. K., Cheng, H., & Lo, D. (2012). Searching connected API subgraph via text phrases. FSE (pp.1-11).
[9] Bruch, M., Monperrus, M., & Mezini, M. (2009). Learning from examples to improve code completion systems. The, Joint Meeting of the European Software Engineering Conference and the ACM Sigsoft Symposium on the Foundations of Software Engineering (pp.213-222). ACM.
[10] Proksch, S., Lerch, J., & Mezini, M. (2015). Intelligent Code Completion with Bayesian Networks. ACM.
[11] Nguyen, T. T., Pham, H. V., Vu, P. M., & Nguyen, T. T. (2016). Recommending API Usages for Mobile Apps with Hidden Markov Model. Ieee/acm International Conference on Automated Software Engineering(pp.795-800). IEEE.
[12] Eclipse documentation - Archived Release, http://help.eclipse.org/kepler/
[13] tf-idf, Wikipedia, https://en.wikipedia.org/wiki/Tf%E2%80%93idf
[14] Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent dirichlet allocation. J Machine Learning Research Archive, 3, 993-1022.
[15] Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. Computer Science.
[16] Nguyen, T. D., Nguyen, A. T., Phan, H. D., & Nguyen, T. N. (2017). Exploring API Embedding for API Usages and Applications. International Conference on Software Engineering (pp.438-449). IEEE Press.

Acknowledgements
This work is supported by the National Natural Science Foundation of China (No. 61672046).