Learning API Suggestion via Single LSTM Network with Deterministic Negative Sampling

Jinpei Yan, Yong Qi, Qifan Rao, Hui He
School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an, China
Emails: yjp2013@stu.xjtu.edu.cn, qiy@xjtu.edu.cn, asd5510@stu.xjtu.edu.cn, huihe@xjtu.edu.cn

***Abstract*—Modern programming relies on a large number of fundamental APIs, but programmers often take great effort to remember names and the usage of APIs when coding, and repeatedly search the related API documents or Q&A websites. To improve the programming efficiency, we present a Java API suggestion approach called APIHelper which learns API sequence pattern via the Long Short-Term Memory (LSTM) network, then provides API suggestion based on the program context. Previous related works use statistical methods based on Hidden Markov Model (HMM), which require establishing one specific model for each class. We propose Determininstic Negative Sampling (DNS) to make API suggestion for a large number of Java classes by one single end-to-end LSTM. To verify this approach, we make API suggestion for 50,000 Java classes and evaluate it with top-K accuracy. Results show that APIHelper outperforms other prior works both on accuracy and computation efficiency.**

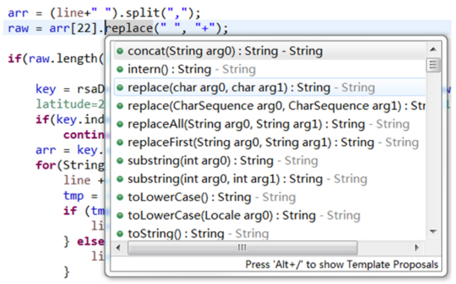
***Keywords*—API suggestion; Long short-term memory; Negative sampling**

I. Introduction

Modern programming languages continue to evolve and introduce more and more high level APIs/methods/functions, while increasing third-party libraries are available for programmers to use. The constantly optimized and rich APIs allow programmers to achieve demo codes easily for different kinds of requirements, but give programmers a great challenge to memorize all these APIs. Programmers often use multiple programming languages, and each programming language involves a large number of grammar rules and APIs. Some programming languages, like Java, have a large number of very complex and long API names, which may be very similar. A simple example, Java offers a lot of classes for I/O operations, such as *FileInputStream()*, *ByteArrayInputStream()*, *CharArrayReader()*, *InputStream()*, *StringReader()*, *StringWriter()*, *PrintStream()*, *PrintWriter()*, *BufferedInputStream()*, *BufferedOutputStream()*, *BufferedReader()*, *BufferedWriter()*.

For this problem, some current IDEs are integrated with API tip tools, such as the one in Eclipse (shown in Fig.1). But these kinds of tools have a common problem that only shows all API methods in the alphabetical order containing the current Java class. Since a Java class often has a lot of methods, programmers still need to look over the list to find the right one in a bunch of similar method names.

To this end, we put forward a new approach of API suggestion called APIHelper to complete APIs pattern learning for full Java classes through the Long Short-Term Memory (LSTM) network. Our approach has two advantages over



```
arr = (line+" ").split(",");
raw = arr[22].replace(" ", "+");

if(raw.length(

    key = rsaD
    latitude=2
    if(key.ind
        contin
    arr = key.
    for(String
        line +
        tmp =
        if (tm
            li
        } else
            li
        }
```

concat(String arg0) : String - String
intern() : String - String
replace(char arg0, char arg1) : String - String
replace(CharSequence arg0, CharSequence arg1) : Str
replaceAll(String arg0, String arg1) : String - String
replaceFirst(String arg0, String arg1) : String - String
substring(int arg0) : String - String
substring(int arg0, int arg1) : String - String
toLowerCase() : String - String
toLowerCase(Locale arg0) : String - String
toString() : String - String
Press 'Alt+/' to show Template Proposals

Fig. 1. Eclipse IDE tools integrated with API tips.

the statistics-based approach. First, LSTM can capture more long-term contextual patterns or relationships, resulting in a higher prediction accuracy. Second, the mainstream statistical approaches [1], [2] are based on Hidden Markov Model (HMM), which require building separate HMMs for each single Java class. Since the existing Java classes are huge and keep increasing, the scalability of HMM-based method is seriously challenged. Instead, we propose a method called Deterministic Negative Sampling (DNS) to model all Java classes and APIs in one single LSTM network, which is more flexible and scalable.

Our main contributions are summarized as follows:

- We propose a new API usage pattern learning method based on LSTM, which learns context features from Java API sequences to make API suggestion.
- We build an API suggestion approach called APIHelper. It makes use of one single LSTM network combining with a proposed DNS method for all Java classes&APIs learning and predicting.
- We collect 18,000 Java project codes from Github for the API suggestion experiment. Results show that APIHelper has a better classification accuracy and computational efficiency compared with HMM and N-gram.

II. Related work

API Mining/Usage Learning. These prior works are closely associated with API suggestion. Xie et al. [3] first proposed the method to mine API usage patterns from source code called MAPO. UP-Miner [4] was proposed to improve the efficiency on API sequences mining compared to MAPO. They used a probabilistic graph to describe API sequences and introduce a set of N-gram features of API call sequences for clustering distance metric calculation. Recently, Fowkes et al. [5] proposed a tool named PAM which is a near parameter-free probabilistic algorithm to output a list of API call patterns.

DOI reference number: 10.18293/SEKE2018-193

PAM significantly outperforms both MAPO and UPMiner. Similar work was proposed by Nguyen et al. [6] which also used the graph to represent API sequences. They built a tool called GROUM, a vector-based approximation approach, to dig out the object usage pattern by finding the isomorphic subgraph for anomaly detection.

Other works try to use NLP techniques to represent and mine API usage patterns. For example, Nguyen et al. [7] proposed API2VEC which represents APIs with a dense vector. They came up with a series of rules to extract API sequences from Java source codes and used API2VEC to do the Java to C# code translation work for evaluation. Gu et al. [8] used natural language to represent APIs and extracted them from Javadoc annotations for code blocks to collect *<API seq,annotation>* pairs as the dataset and mine the correspondence.

API Suggestion/Recommendation. Currently, these are some strategies for API suggestion/recommendation. The first strategy is focused on sequential pattern mining. Nguyen et al. [1] designed an API suggestion tool specifically for Android mobile development. They used GROUM to extract the API call graph from the Android source code, then simply traversed the graph to generate the API sequences and process it using a special HMM model for each Java class. On this basis, further improvements were made by Pham et al. [2], who analyzed directly the Android applications to find the usage for APIs. They first decompiled the Android application to get bytecodes, then transferred them into Control Flow Graph (CFG), finally built HMMs to mine API usage patterns. Raychev et al. [9] implemented a tool called SLANG for API prediction. They tried N-gram, Recurrent Neural Network (RNN) and the combination of both methods to extract the API call sequence, and used N-gram or RNN for training.

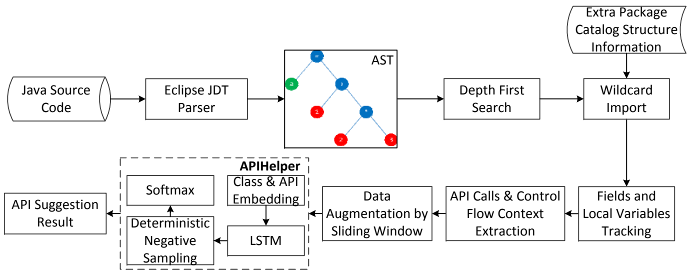
The second strategy is focused on frequent subgraph or itemset mining. Zhong et al. [10] used code search engines and code snippets to extract API call sequences. These extracted APIs are clustered according to the distance metric which reflects the similarity between class names and API names. They mined the most frequent API calls using SPAM for each cluster. Nguyen et al. [11] presented a tool named LIBSYNC to guide the developer to update APIs so as to fit the third-party library update. They used several graph-based techniques to describe changes in the APIs.

Besides, Niu et al. [12] mined API usage patterns without relying on frequent-pattern mining, but automatically extracted usage patterns by clustering the data based on the co-existence relations between object usages. Tetsuo Yamamoto [13] instead took a simply and lightweight method, the author designed a specific algorithm based on rules to give a method call suggestion according to the context.

III. Our Methodology

A. Overview

The main idea is to use the LSTM network to learn the semantic and context information from Java source codes, the embedding representation of Java classes&APIs (methods) and the context features from API call sequences, respectively. We treat API suggestion as a multi-class classification problem. Given an API call sequence as an input, LSTM uses a softmax output layer to predict next API call and outputs probabilities of all candidate API calls. The core flowchart of our approach is shown in Fig.2. We first extract API sequences from Java source codes. Then APIHelper embeds Java classes and APIs into dense vector representations. As a result, it transfers the API sequence into a vector data stream as LSTM's input for training. Here API suggestion can be regarded as a "predict next word" task. Given an API call sequence as one input, LSTM tries to predict next API call.



Extra Package Catalog Structure Information
Java Source Code
Eclipse JDT Parser
AST
Depth First Search
Wildcard Import
APIHelper
Class & API Embedding
Softmax
Deterministic Negative Sampling
LSTM
API Suggestion Result
Data Augmentation by Sliding Window
API Calls & Control Flow Context Extraction
Fields and Local Variables Tracking

Fig. 2. The core flowchart of proposed method.

B. API Sequence Extraction

We mainly extract the API call information from source codes. For this information, we extract Java method invocations and Java class instance creations. Concretely, we first obtain the corresponding Abstract Syntax Tree (AST) with Eclipse JDT parser tool from Java source codes. Then we traverse the AST tree through Depth First Search (DFS) to extract nodes for Java classes and API calls, and track all fields and local variables' method calls. Thus we can resolve the fully qualified name of an API call according to the field or local variable. In this phase, we also need to parse the import statements to get fully qualified class names to identify all Java classes explicitly imported. Note that in some codes, wildcards can be used to load all classes in a high level package path, which will cause the fully qualified name of some classes cannot be resolved. Under this situation, we can obtain the package catalog structure information to get a fully qualified path through the relevant Java documents. After that, some method names cannot be fully resolved, so we filter them in the data preprocessing phase. It should be noted that the superclass method, method override, constructor call, and class conversion expression do not take into account for simplification in our scenario.

C. Learning API Usage Pattern with LSTM

Programming languages have some similarities to natural languages, so here we use language models for API suggestion. For a Java API sequence, since Java is an object-oriented language and all methods are encapsulated in an object belonging to one specific class, we can define an API sequence as $S = \{c_1, a_1, c_2, a_2, ..., c_T, a_T\}$, where $c_t, t \in (1, T)$ represents the class to which the tth object belongs, and a_t represents the corresponding API called by the object. Then we can use the statistical language model to define the occurrence probability of this API sequence. According to the Bayesian algorithm,

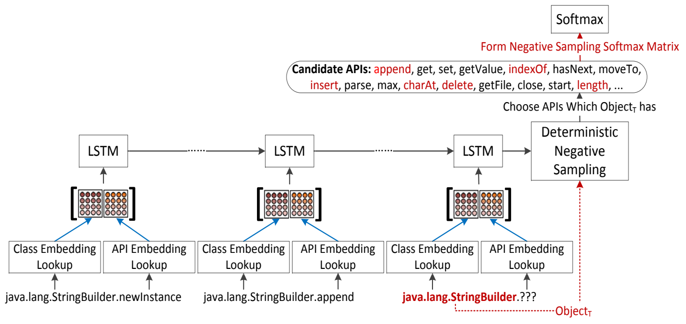

Softmax
Form Negative Sampling Softmax Matrix
Candidate APIs: append, get, set, getValue, indexOf, hasNext, moveTo, insert, parse, max, charAt, delete, getFile, close, start, length, ...
Choose APIs Which Object_T has
Deterministic Negative Sampling
LSTM
LSTM
LSTM
Class Embedding Lookup
API Embedding Lookup
Class Embedding Lookup
API Embedding Lookup
Class Embedding Lookup
API Embedding Lookup
java.lang.StringBuilder.newInstance
java.lang.StringBuilder.append
java.lang.StringBuilder.???
Object_T

Fig. 3. The neural language model for API suggestion by the LSTM network.

the probability that occurrence probability of a sequence S equals to the combination of the probability that each "word" appears, so $P_w(S)$ can be expanded to:

$$P_w(S) = P_w(a_1) * \prod_{t=2}^{T} P_w(a_t|c_t, a_{t-1}, c_{t-1}, ..., a_1), \quad (1)$$

where w is a parameter learned by the statistical language model and updated through training model. For the API sequences in the training dataset, we maximize the occurrence probability:

$$argmax_w P_w(c_1, a_1, c_2, a_2, ..., c_t, a_t). \quad (2)$$

For the API suggestion task, we choose a_t to maximize the occurrence probability of current API sequence, and calculate the maximum likelihood probability as the objective function in the training phase which gets the highest probability when a_t equals to the true value y:

$$argmax_w P_w(a_t = y|c_1, a_1, c_2, a_2, ..., c_t). \quad (3)$$

However, the actual Java API sequence often has a long-term context, like follows:

```
FileWriter writer = new FileWriter("XX.csv");
{...a bunch of codes...};
writer.close();
```

The bunch of codes in the middle part may be very long. Therefore, we need a way to learn longer context information to suggest API *close()*. The neural language model is one good solution which is widely used in NLP tasks in recent years. In this paper, we use LSTM to build a neural language model. It is a powerful deep neural network for temporal data mining and learning. As a variant of RNN, LSTM takes a bunch of previous API sequences as context to predict next API call, which actually gives a prediction category from all candidate API calls as classification. The overall model part that APIHelper uses for API suggestion is shown in Fig.3.

Specifically, the whole model contains three parts: embedding layer, LSTM layer, and classification layer. First, the embedding layer will embed these input API sequences into dense vectors that actually learn some semantic and logic information from Java classes and API calls. These dense vectors will be used as the input for several LSTM units to learn the context features. Then, LSTM sequentially takes the elements in the API sequence as an input. Each step, LSTM will update Constant Error Carousel (CEC) and "gates" to selectively store information from previous inputs and calculate them with the input of current time node. In this way, after receiving the complete API sequence input, LSTM can learn the context information in the sequence to generate a dense vector for the final prediction. Then this dense vector is regarded as the highly abstract feature information to be added to the softmax output layer, which outputs the prediction result. The formula is as follows:

$$P_w(a_t = \widehat{y}|c_1, a_1, c_2, a_2, ..., c_t) = \frac{exp(W_{\widehat{y}}^T h_t + b_{\widehat{y}})}{\sum_{k'=1}^{K} exp(W_{k'}^T h_t + b_{k'})}, \quad (4)$$

where $W_{\widehat{y}}^T$ represents the part of weight matrix W corresponding to the prediction category $\widehat{y}$, and h_t is the final dense vector calculated by LSTM forward propagation along the time series. The softmax layer uses a nonlinear transformation to calculate the conditional probability for outputting the final prediction result. Meanwhile, it uses maximum likelihood probability as the objective function to train model parameters.

It should be noted that, unlike dealing with natural language, we have conducted some specialized training strategies for API sequence learning. As mentioned earlier, each of Java classes follows a specific API call. A straightforward approach to digitalize these elements is directly mapping them using one same dense embedding matrix. But this approach is not good for several reasons. First of all, these two elements are not the same type. Using one embedding matrix would mix them up and lose this boundary between two elements. Second, Java language contains a large number of classes and APIs, and needs a very large embedding matrix to represent them, which would greatly slow down the training speed. And most importantly, LSTM is not effective due to the gradient vanishing when dealing with a very long API sequence. Therefore, we propose a new approach using two embedding matrixes W_c^e and W_a^e for representing Java classes and APIs,

$$e_t^c = W_c^e[c_t], e_t^a = W_a^e[a_t]. \quad (5)$$

Considering that each Java class must be followed by an API call, we come up with a strategy called Embedding Concatenation (EC), which uses $[e_t^c, e_t^a]$ to concatenate the Java class embedding in the current timestep and the corresponding API call embedding as the input for LSTM, that is $x_t = [e_t^c, e_t^a]$.

Since one embedding vector can represent one Java class and one API call, and LSTM can deal with a *Class&API* unit in one timestep, the length of the API sequence that LSTM needs to process is actually reduced by nearly half. It allows LSTM to handle and learn more long-term context information and eases the gradient vanishing problem.

The last API call a_T of an API sequence is to be predicted, so the last timestep input is received by LSTM which contains only the Java class information c_T. Here we use an identifier e^{pred} to remind LSTM that it is the section to make a prediction, that is $x_T = [e_T^c, e^{pred}]$.

D. *Deterministic Negative Sampling*

The concept of Negative Sampling (NS) comes from a training strategy for word vectors in the NLP area. Some famous models using negative sampling are Skip-gram with

Negative Sampling (SGNS) and CBOW with Negative Sampling (CBOW-NS). The main purpose of negative sampling is to improve computational efficiency. For a multi-class classification problem, machine learning models usually use the softmax layer to output multi-class prediction probabilities:

$$P_w(a_t = \widehat{y}|context) = \frac{exp(W_{\widehat{y}}^T h_t + b_{\widehat{y}})}{\sum_{k'=1}^{K} exp(W_{k'}^T h_t + b_{k'})}. \quad (6)$$

As we can see the softmax layer outputs probabilities by a normalization operation, which is usually called the Cross Entropy (CE) error function in machine learning. The computation cost increases linearly with the number of categories. So the cost of CE is $|V| + 1$, where V represents the number of vocabularies to be predicted, and the cost of NS is $|K| + 1$. The speed up ratio is K/V (NS is much faster). Since in each NLP word classification task, there is only one positive category (the word to be predicted) and all other words are negative. So instead of updating weights through whole negative vocabularies, NS only samples some negative words for weight updating.

APIHelper tries to build one single LSTM network for all API suggestions. However, the biggest challenge is that the number of candidate APIs to be predicted is so huge (for common Java classes, there are more than 30,000 candidate APIs in total). From above we know the softmax is inefficient when dealing with such a big multi-class classification. Therefore we propose DNS to solve this. The main difference between DNS and NS is that DNS is a deterministic sampling strategy rather than a random sampling. The deterministic sampling is from the specific constraints for API calls that every API call must come from one specific Java class. Considering the API we need to predict must belong to the current Java class, we can use the last Java class c_T to limit the range of candidate APIs to be predicted. There are around 5 to 108 APIs for a common Java class, so the LSTM network does not need to do a prediction in the entire candidate APIs set. Instead, it first picks out APIs subset for current Java classes, and then makes a prediction based on that. Meanwhile, the LSTM network computes softmax errors and does weight updating only for negative API samples in this subset.

For the specific implementation of DNS, current mainstream deep learning frameworks (such as Tensorflow, Caffe, Theano, etc.) only provide a random negative sampling function. For example, Tensorflow simply provides a random negative sampling error function: $tf.nn.sampled\_softmax\_loss()$, which computes the cross entropy over the subset of candidate classes. Since it cannot achieve the function of DNS, we devise a special LSTM training module to implement DNS in Tensorflow. Specifically, in each iteration, we introduce an additional Tensorflow place holder DNS to reduce the prediction probability manually for APIs not belonging to current Java classes. Then LSTM calculates the cross entropy error for this fixed output probability:

$$dns_{1*N} = [b_{API_1}, b_{API_2}, b_{API_3}, ..., b_{API_N}], where$$

$$b_{API_n} = \begin{cases} \tau & API_n \in current \quad Java \quad class \quad C_T \\ 0 & otherwise \end{cases} \quad (7)$$

$$logtis_{1*N} = [p_{API_1}, p_{API_2}, p_{API_3}, ..., p_{API_N}], \quad (8)$$

where $logtis$ represents the logistics probability of the original LSTM prediction. $logtis$ adds dns as bias, that is $logtis = logtis + dns$, and calculates the loss for back propagation, shown as: $tf.nn.softmax\_cross\_entropy\_with\_logtis(logtis{=}logtis, labels{=}\ ...)$. It can be seen as passing the prior knowledge to the LSTM network by dns. By manually reducing the output probabilities of some unrelated candidate APIs, their cross entropy error is very low. This in turn lets the LSTM network focus on learning to classify or distinguish APIs belonging to current Java classes during error back propagation.

IV. Experiment and Results

A. Dataset and Data Preprocessing

We crawled 18,000 Java projects from Github and extracted corresponding Java source codes for a total of 16GB data. To ensure a high quality of Java codes dataset, we only collect Java projects with over 100 stars. More stars means more popular project and higher programming quality.

For example, APIHelper learns the most popular Java APIs, as follows: *java.**, Java foundation classes; *javax.**, Java foundation classes extension; *android.**, Android related Java classes; *org.apache.**, all top level Java project from Apache software foundation; *com.google.**, all Java project provider by Google; *org.springframework.**, one of the most popular structure for Java web development.

Above Java classes and APIs cover daily function needs for programmers. However, these Java libraries also contain some rarely used Java classes and APIs. Thus we do the frequency counts for APIs and use UNK to represent Java classes that occur less than 20 times and Java APIs less than 10 times in our dataset. This greatly reduces the size of embedding matrix for LSTM. Only 50,000 Java classes and 60,000 Java APIs are reserved, so the hyperparameter num_{emb} is reduced from original 540,000 to 110,000.

The total number of candidate APIs to be predicted is 20,000, but the frequency of these APIs varies a lot. The most frequent API is *newInstance* which occurs more than 5,000,000 times in our dataset, while the least popular API only appears 50 times. Hence we use data augmentation strategy to rebalance the distribution, which achieves the maximum augmentation ratio up to 40 times.

B. Model Setup and Training Details

In experiment, APIHelper is trained by GPU. We use NVIDIA's GTX980 GPU and related software repositories including CUDA 7.5, cuDNN V4, Python 2.7.6, Numpy 1.8.2, Scipy 0.13.3, and Tensorflow 0.9.0. APIHelper uses a two-layer LSTM network, and each layer contains 128 neuron units. For the design of LSTM structure, we use ReLU as activation function and use Dropout with dropping probability 0.5 to ease overfitting problem. To ensure the LSTM network convergence, we use orthogonal weight initialization with a range of ± 0.04, add batch normalization layers, and set gradient regularization factor to 10 (it is used to control gradient expansion). APIHelper adopts an optimizer based on the Adam optimization algorithm, sets the initial learning rate of $\eta = 5e-03$, and takes sequence length of 60 as the input and batch size of 228. Also, we set the maximum number of training epoches (=150,000). Since we use early stop strategy, usually the training phase stops after 5 complete rounds.

Each element of an input API sequence consists of two parts: Java class embedding with vector length of 150 and API embedding with vector length of 200. We propose EC to concatenate two-part embedding vectors together along the horizontal axis, $x_t = [e_t^c, e_t^a]$. So the input API sequence $[x_1, x_2, x_3, ..., x_T]$ can be represented as $\{[e_1^c, e_1^a], [e_2^c, e_2^a], [e_3^c, e_3^a], ..., [e_T^c, e^{pred}]\}$, where e^{pred} is the API to be predicted. The embedding lookup matrix is initialized randomly, and it uses $0.1 * \eta$ to adjust learning rate (otherwise model will get seriously overfitting during training).

C. Results and Evaluation

We use 18,000 Java projects and 10-fold cross validation to evaluate the performance of LSTM network used in APIHelper, as well as the effect of using EC and DNS to verify whether they are valid. The results are shown in Table I.

TABLE I
THE PERFORMANCE FOR DIFFERENT LSTM NETWORKS

Methods	Accuracy(%)	top-5 Accuracy(%)
Standard LSTM	40	79
LSTM + EC	43	81
LSTM + EC + DNS (**APIHelper**)	**53**	**90**

We use accuracy and top-5 accuracy as evaluation indicators. Here top-5 accuracy represents the prediction accuracy when the model can provide top-5 predicted APIs. As we can see from Table I, both EC and DNS give an improvement of prediction accuracy. Specifically, the standard LSTM model serves as the baseline method which regards each Java class or API as one input, while EC merges and concatenates them into one long vector as another input. EC enables LSTM to learn API sequence in a more efficient way and capture more long-term contextual information. So top-5 accuracy raises from 79% to 81% when using EC.

Apart from this, DNS has the most benefit to LSTM performance, raising top-5 accuracy from 81% to 90%. For each input API sequence, DNS builds one corresponding softmax layer which narrows down the candidate APIs to predict according to the last input Java class. Hence LSTM only needs to choose one prediction API belonging to the last input Java class instead of all Java APIs. This greatly reduces the search space of LSTM network, making it more efficiently during training and easier to converge. Most importantly, this makes it possible to predict massive Java APIs (give a API suggestion for massive Java classes) by using one single LSTM network, because it simplifies the original multi-class classification problem with hundreds of thousands of categories into hundreds of categories classification.

In addition, we compare APIHelper with the current two mainstream methods, HAPI [2] and N-gram. Original task of HAPI is for Android API learning. It first extracts Android bytecodes from *.apk* file through dex parser tool (e.g. *baksmali*) and converts them to CFG, then traverses CFG to generate API sequences for training HMM (HMM takes the sequence as the input). For each Java class, HAPI trains one specific HMM model and takes two experiments for evaluation, which are called "predict next API" and "fill API hole". The difference is the latter task not only takes the context before the API to be predicted, but also the context after it. The experiment takes a 63GB dataset for 2,700 Java classes with 17,000,000 API sequences. Each Java class has around 6,000 API sequence samples and the average length of an API sequence is 4. HAPI trains 1,200 HMM models on 2,700 Java classes, with averaging eight hidden states per HMM.

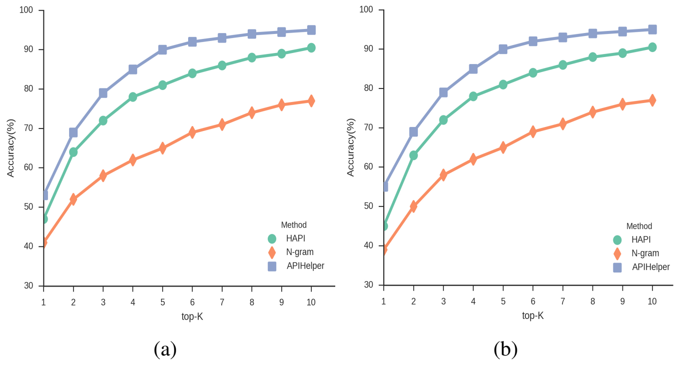


Fig. 4. (a) The results of API suggestion for predicting next API. (b) The results of API suggestion for filling API hole.

The goal of HAPI is to solve Android API learning, which is a subtask compared with Java API suggestion, but APIHelper needs to predict APIs for 18,000 Java project codes which contain all Android related APIs and many other Java APIs. Thus, in order to better evaluate and compare APIHelper with HAPI, we reproduce the HAPI method and conduct a comparative experiment with the same task on our dataset.

Also, we make the experiment of 3-gram for API suggestion or recommendation as a baseline. We first filter out low frequency 3-gram combinations, then use the Random Forest (RF) model for feature importance analysis, from which a 43,000-dimensional feature vector is extracted. Finally, we use these features with softmax regression to achieve multi-class classification and get the API suggestion result. As we can see from the results shown in Fig.4(a) and Fig.4(b), APIHelper outperforms other two methods. The performance of HAPI is somehow lower than the original results in their paper. One reason is that our task is to make an API suggestion or recommendation at any randomly chosen location rather than at the end of a well-extracted API sequence. Thus sometimes the context is relatively insufficient.

D. Computation Efficiency

We analyze the computational efficiency and computation resource consumption of all methods, and separately calculate the storage consumption, training time and API suggestion time-consuming for comparison. The evaluation results are shown in Table II. As we can see, the training time consumption for standard LSTM model takes 2.4 hours, which is obviously slower than HAPI. Besides, the slowest method

TABLE II
THE EXPERIMENT RESULTS FOR DIFFERENT METHODS

Methods	Accuracy(%)	top-5 Accuracy(%)	Training Time(h)	Disk Space Consumption(MB)	Suggestion Time(ms)
Standard LSTM	40	79	2.4	328	23.3
LSTM + EC	43	81	1.5	263	13.5
3-gram	43	68	1.4	182	172
HAPI [2]	48	81	1.2	**18.5**	14.4
APIHelper	**53**	**90**	**0.5**	263	**13.2**

is 3-gram model, mainly because 3-gram takes a lot of time for feature extraction. As mentioned before, 3-gram counts all occurrences of 1-gram, 2-gram and 3-gram in the dataset, which means doing millions of counts as features. After that the N-gram statistical method uses a simple classifier which costs a little time for training.

However, after using EC and DNS, APIHelper only needs half an hour to complete the training process. The main reasons are as following: first EC almost reduces the input sequence length by half for LSTM. EC concatenates class and API embeddings as $[e_t^c, e_t^a]$, so LSTM timestep takes more information for each one timestep. Then DNS further speeds up the training phase because the back propagation calculation only needs to be done on some softmax nodes instead of all of them. This greatly accelerates the weight updating speed for LSTM network. At last, coupled with GPU-accelerated deep neural network computation, APIHelper takes the shortest time for training. For HAPI [2], thought training a single HMM model for a Java class takes less time, HAPI needs to train thousands of HMMs for all Java classes. Overall, it is less efficient than APIHelper. Since the model training phase is often offline, the more critical indicator is the time-consuming in API suggestion phase, which reflects the response speed of API suggestion system. In the prediction phase, the LSTM network is the fastest (13.2ms) which only needs to do one forward propagation to get the prediction result.

Finally, we compare the memory and disk consumption. Since data storage is critical for all methods, here the main storage consumption we evaluate is the model storage for storing model parameters. It can implicitly indicate the memory consumption because usually a system loads all model parameters into memory for fast computation. The evaluation results show APIHelper and N-gram consume more storage than HAPI. APIHelper needs more storage because LSTM network is complex and contains mass parameters. However, all three methods consume less than 500MB of storage space, which we think is not a bottleneck for current computer resources in a server with GPU computation.

V. CONCLUSIONS

In this paper, we explore a new LSTM-based API suggestion approach to improve programming efficiency. To this end, we construct a prototype implementation called APIHelper, which uses what we called deterministic negative sampling to build one single end-to-end LSTM network to make API suggestion for tens of thousands of Java APIs. While experiments show that APIHelper can effectively provide API suggestions according to the API contexts and has better performance in terms of suggestion accuracy, computational efficiency and scalability compared with previous works.

For the limitation of APIHelper, it has to rely on class object information for API suggestion, so currently it can only be applied to strongly typed languages. In the future work, we will plan to explore APIHelper making API suggestion for weakly typed languages like Python, which is a more challenging task.

ACKNOWLEDGMENT

This work is partially supported by the National Natural Science Foundation of China under Grant No. 61672421 and the Shaanxi Provincial Natural Science Foundation under Grant No. 2017JM6109.

REFERENCES

[1] T. T. Nguyen, H. V. Pham, P. M. Vu, et al. Recommending API usages for mobile apps with hidden markov model. In Proc. of 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 795-800, 2015.
[2] H. V. Pham, P. M. Vu, T. T. Nguyen. Learning API usages from bytecode: A statistical approach. In Proc. of the 38th International Conference on Software Engineering (ICSE), 416-427, 2016.
[3] T. Xie and J. Pei. MAPO: mining API usages from open source repositories. In Proc. of the International Workshop on Mining Software Repositories (MSR), 54-57, 2006.
[4] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie and D. Zhang. Mining succinct and high-coverage API usage patterns from source code. In Proc. of the 10th Working Conference on Mining Software Repositories (MSR), 319-328, 2013.
[5] J. Fowkes, C. Sutton. Parameter-free probabilistic API mining across GitHub. In Proc. of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 254-265, 2016.
[6] T. T. Nguyen, H. A. Nguyen, N. H. Pham, et al. Graph-based mining of multiple object usage patterns. In Proc. of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE), 383-392, 2009.
[7] T. D. Nguyen, A. T. Nguyen, H. D. Phan, et al. Exploring API embedding for API usages and applications. In Proc. of the 39th IEEE/ACM International Conference on Software Engineering (ICSE), 438-449, 2017.
[8] X. Gu, H. Zhang, D. Zhang, et al. Deep API learning. In Proc. of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 631-642, 2016.
[9] V. Raychev, M. Vechev, E. Yahav. Code completion with statistical language models. In Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 419-428, 2014.
[10] H. Zhong, T. Xie, L. Zhang, J. Pei and H. Mei. MAPO: Mining and recommending API usage patterns. In European Conference on Object-Oriented Programming (ECOOP), 318-343, 2009.
[11] H. A. Nguyen, T. T. Nguyen, Jr. G. Wilson, et al. A graph-based approach to API usage adaptation. In Proc. of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 302-321, 2010.
[12] H. Niu, I. Keivanloo, Y. Zou. API usage pattern recommendation for software development. The Journal of Systems and Software, 129: 127-139, 2017.
[13] T. Yamamoto. Code suggestion of method call statements using a source code corpus. In Proc. of the 24th Asia-Pacific Software Engineering Conference (APSEC), 2017.