Computer Science and Information Systems 17(3):907–926 https://doi.org/10.2298/CSIS190908027Y

Graph Embedding Code Prediction Model Integrating Semantic Features

Kang Yang[1], Huiqun Yu[1,2], Guisheng Fan[1,3], and Xingguang Yang[1]

[1] Department of Computer Science and Engineering, ECUST
Shanghai, China
[2] Shanghai Key Laboratory of Computer Software Evaluating and Testing, China
Shanghai, China
yhq@ecust.edu.cn, Corresponding author
[3] Shanghai Engineering Research Center of Smart Energy, Shanghai, China
Shanghai, China
gsfan@ecust.edu.cn, Corresponding author

Abstract. With the advent of Big Code, code prediction has received widespread attention. However, the state-of-the-art code prediction techniques are inadequate in terms of accuracy, interpretability and efficiency. Therefore, in this paper, we propose a graph embedding model that integrates code semantic features. The model extracts the structural paths between the nodes in source code file's Abstract Syntax Tree(AST). Then, we convert paths into training graph and extracted interdependent semantic structural features from the context of AST. Semantic structure features can filter predicted candidate values and effectively solve the problem of Out-of-Word(OoV). The graph embedding model converts the structural features of nodes into vectors which facilitates quantitative calculations. Finally, the vector similarity of the nodes is used to complete the prediction tasks of TYPE and VALUE. Experimental results show that compared with the existing state-of-the-art method, our method has higher prediction accuracy and less time consumption.

Keywords: Big Code, Graph Embedding, Code Prediction.

1. Introduction

With the rapid increase of the code's volume, the use of existing code data for prediction has attracted more and more attention. It makes use of the existing code in the context to suggest the next possible code token, such as method calls or object fields. However, the source code files are written by different programmers and have no fixed structure. For example, Python has a more casual programming style, and feature extraction process is more difficult than Java. These reasons lead to the low accuracy of the prediction model, and we may even get the opposite result. Thus, mining the information between nodes can effectively extract the features in the source code file. Finally, the model can predict the missing nodes through these features. Traditional code prediction methods are based on code contexts and grammatical rules. Early research focused on probability models in this field. The probability model integrates the node information around the predicted node and calculates the probability value of each candidate value in the model. For example, the N-gram[5] model uses the $n-1$ tokens to predict the probability of the nth token. The N-gram model has strong flexibility and high scalability. Yet, N-gram model

only extracts code information in a small range, and cannot deal with the long-distance dependent features of the code. So, the prediction performance of the N-gram model has limitations. Deep learning can effectively extract the features of long-distance node information. These methods have long been used in natural language processing tasks, such as LSTM. Raychev[24] believes that programming languages are also natural languages, such as recurrent neural network (RNN) models can also solve code problems. They uses the RNN model to obtain the order distribution and semantics of the code from the AST of the source code file. Neural language model combined with Attention Mechanism[4] can complete the code prediction task. However, the time consumption of neural network models is very long, and the interpretability of the model is weak. The neural network model uses fixed candidate values, so it is difficult to solve the OoV problem.

Generally, the last component of the neural network model is the softmax classifier. Each output dimension corresponds to a unique word in a predefined vocabulary. Due to the large amount of calculation of the model, the usual approach is to use only the K most frequent words in the corpus to build a vocabulary. The words that are not in the candidate value are defined as Out-of-Vocabulary(OoV) words. In reality, every programmer will artificially define rare node names. The occurrence of these values is very small in the source file, and the probability of appearing in the candidate value table is very low. Therefore, OoV problems are common in code prediction, which also reduces the prediction accuracy of the model. Abstract Syntax Tree(AST) is an abstract representation of the grammatical structure of source code. It shows the grammatical structure of the programming language in the form of a tree, each node on the AST represents a structure in the source code. Each code source file has only one corresponding abstract syntax tree. We extract the rich grammatical structure features in AST and bring them to the downstream prediction model.

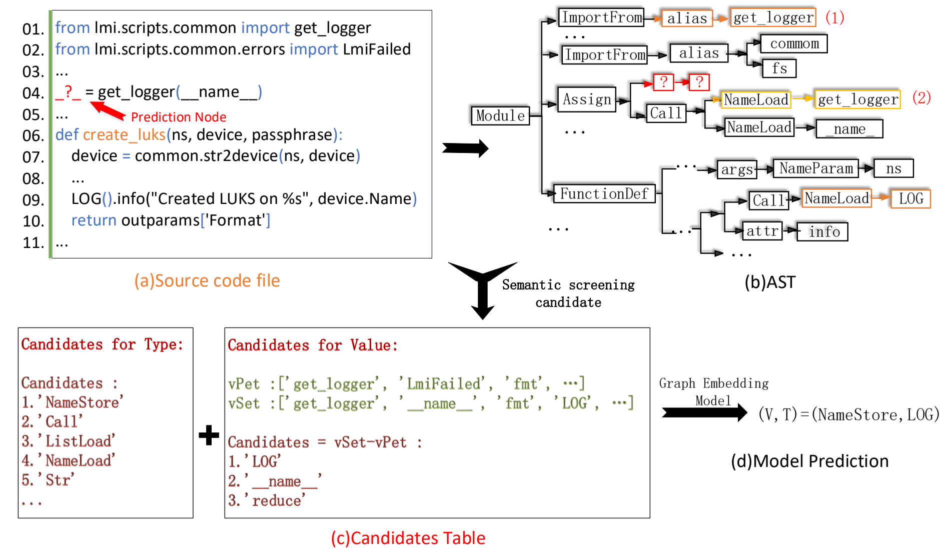

Fig. 1. Examples of Python programs and their corresponding AST

In this paper, we propose a graph embedding prediction model with AST's semantic structure features. We extract semantic structural features from the code context and filter out irrelevant candidate values. This process can narrow the range of candidate values, thereby shortening the prediction time. And the candidate values change according to the context of the prediction files. The dynamic change of candidate values can effectively solve the OoV problem. Besides, we extract the AST's node paths of the source file and convert them into training graphs. Then, we use the Node2vec model to extract the structural features of the graph and embed the structure paths into vector with fixed dimensions. The embedding model reduces the dimensionality of discrete node paths to continuous dimensional vectors, which is useful for quantitative calculations in downstream tasks. The Node2vec model biasly converts the training graph into node sequences. Each node sequence merges different structural information between nodes through parameters (p, q). The greater the overlap degree of node sequences, the higher the similarity of nodes. Therefore, the probability of our predicted value is positively correlated with the similarity of the node's structure. Our model can keep the structural information of the source code, so the model interpretability is higher. Besides, the model consumes less time than the neural network models and has higher prediction accuracy.

As shown in Fig. 1, the source code file in Fig. 1(a) is got from GitHub. The **Prediction Node** in the Fig. 1(a) is the node we want to predict. Fig. 1(b) is an abstract syntax tree for source file conversion. We can use semantic features to filter candidate values. The parameters defined in the source file or the imported package will be called in the following, such as $(1), (2)$. The source code file imported a Python package $get_logging$ at position (1), and $get_logging$ is called at position (2). We extract similar semantic structural features from AST. The defined structure is stored in the collection $vPet$, such as package name or parameters. $vSet$ collection stores the semantic structure of the package or parameter call. Then $vSet - vPet$ is a new candidate value table, which contains the missing nodes. In Fig. 1(c), the candidate value LOG is the OoV word. If the calculated candidate value table is empty, the most frequent K values related to the parent node are selected as candidate values. Finally, the graph embedded model is used to calculate the predicted value, as shown in Fig. 1(d).

The main contribtions of this paper are as follows.

- An effective graph structure is constructed. We extract the path of the terminal node from the AST of the source code and filter out irrelevant paths. These training paths will be converted into training graphs of related nodes. This process can reduce the redundant nodes of the training graphs. The reduced training graphs can speed up the node prediction task.
- The semantic structural features in AST is extracted. We extracted the semantic features can reduce the range of candidate value tables. Since the candidate value is not fixed, it dynamically changes with the prediction file. Thus, the OoV rate can be effectively reduced.
- The efficiency of our model is demonstrated by comparing with the state-of-the-art model. Our model prediction accuracy is improved and the time consumed is shortened. In this model, the graph embedding part can effectively extract the local information of the AST, and the semantic features include the information of global dependence. The combination of these two parts improves the prediction accuracy

of the model. The selection of candidate values by semantic features can shorten the prediction time.

The rest of the paper is organized as follows. The Section 2 introduces related works, and Section 3 explains the basic problem definition. We proposed method in Section 4. Section 5 describes the experiment and the experimental results. Finally, we summarize this paper and outline future work prospects in Section 6.

2. Related Works

Simple and effective probability models are widely used in the field of code prediction. The research content is to improve the prediction accuracy of the N-gram language model. Allamanis et al.[3] expanded the size of the corpus based on the research of Hindle et al.[15]. The experiments show that with the increase of data, there is no performance bottleneck in the N-gram model. Nguyen et al.[21] proposed SLAMC to solve the problem that the N-gram model can only extract a limited range of rules, thereby improving the experimental results. Tu et al.[25] found that the locality rules of the code are not used in the N-gram model, and proved that the source code has local repetitiveness. The local repetition rules can be captured by the local cache and applied to software engineering tasks. Nguyen et al.[20] believed that the N-gram model could not extract structural information from API calls, and proposed a language model GraLan for the API call sequence diagram to predict the next API element. In addition to the N-gram model, machine learning methods can also be applied in this field. Bruch et al.[7] used the KNN algorithm to find the most similar completed fragments in the existing code base and provided candidates for method calls. It also points out that many machine learning algorithms can be introduced into the research of code completion. These are similar to the methods recommended for the code, and there are many in life[11][27]. However, these methods have poor ability to deal with OoV problems, and some of them have not considered this issue.

The prediction method based on the representation method of the neural network model mainly uses the improved RNN model, LSTM[18] and gated network[17]. Jian Li[16] proposed parent pointer hybrid network to predict the OoV words in code completion. Adnan Ul et al.[26] used Bi-LSTM model training to split source code identifiers. The model reduces the number of identifier core libraries, thereby improving the accuracy of code prediction. Bhoopchand et al.[6] proposed a new neural network model (sparse pointer network) to capture long-distance dependencies. Context-aware[13] methods have also been used in the Internet of Things. The real-time coding suggestions provided by programmers are closely related to the stability of the network[12][10].

Graph models are widely used in graph structure node quantization processing. Bryan Perozzi et al.[22] proposed that the DeepWalk model uses a random walk algorithm to process static graphs. However, the extraction of DeepWalk path sequences is random, and no attention is paid to the path between Depth-First-Search(DFS) and Breadth First Search(BFS) between fused nodes. Therefore, Aditya Grover et al. [14] proposed Node2vec, which proposed a biased random walk algorithm based on the edge weight of graph nodes. The Node2vec algorithm can fuse the structural information of the node DFS and BFS. Node2vec can effectively integrate the information around the node, so the extracted features are more effective. The powerful method of GNN[9][8] in modeling

the dependency relationship between graph nodes has made the research field related to graph analysis achieve good results. Allamanis et al.[2] proposed to use graphs to represent the structure and semantic information of source code, pointing out that graph neural networks have better performance in variable completion and variable misuse than convolutional neural networks, and can complete multiple variables. The latest empirical research by Rahman et al.[23] pointed out that the graph has a higher level of repetitive patterns than the N-gram model. It is recommended to further study the statistical code graph model to accurately capture more complex coding models.

3. Problem Definition

In order to facilitate the extraction of the characteristic data in the source code, we converted the source files with different number of lines and volumes into AST. The fixed structure path information of the terminal node is extracted through AST, which is convenient to carry into the downstream model for calculation. All programming languages have clear context-free grammars that can be used to parse source code into AST. Then we extract the path and convert it into a related graph structure (Train-Graph G). Finally, the training graph is brought into the model, and the predicted node combination (Node Combination (T, V)) is calculated.

Definition 1:(Train-Graph G). Train-Graph $G = (F, A, Path)$ is a graph converted from the AST's node path of the source file data. F presents a set of n sounce code files, F = $\{f_1, f_2, f_3, ...f_n\}$. A presents a set of Abstract Syntax Tree (AST) which is transformed by context-free grammer A = $\{a_1, a_2, a_3, ...a_n\}$. These AST files contain a majority of node structural feature information. Each $Path$ contains AST's terminal node Up path and Down path. Finally, all node's paths are converted into a training graph $G = (F, A, Path)$.

Definition 2:(Node Combination (T, V)). (T, V) is calculated by the similarity between the parent node of the prediction node and the candidate value in Train-Graph G. Tc represents s candidate values in TYPE task, Tc = $\{Tc_1, Tc_2, Tc_3, ...Tc_s\}$. Vc represents k candidate values in VALUE task, Vc = $\{Vc_1, Vc_2, Vc_3, ...Vc_k\}$. T is the maximum value calculated by predicting the similarity between the parent node p_node and Tc, $T = S(G, Tc, p_node)$. V is the maximum value calculated by predicting the similarity between the parent node T and V, $V = S(G, Vc, T)$.

For example, as shown in Fig. 1, we predict the missing node=(T, V) in the last line of code. In the corresponding AST, it can be obtained that the p_node is $Assign$ in Fig. 1(b). After embedding the graph node, the model will calculate the vector similarity between the candidate value and the parent node($Assign$) of the predicted node. If the candidate node structure is more similar to the parent node, the more likely the two are connected. So the TYPE prediction task is to calculate $T = S(G, Tc, Assign)$. Determine the type of the prediction node, then we use the type as the new parent node and calculate $V = S(G, Vc, T)$ to predict the VALUE task. This prediction process is transformed into the prediction of the node, as shown in the following Equation (1), (2).

$$\exists i \in \{1, 2, 3, ...s\} : T = S(G, Tc, p_node) \geq S(G, Tc_i, p_node) \tag{1}$$

$$\exists j \in \{1, 2, 3, ...k\} : V = S(G, Vc, p_node) \geq S(G, Vc_j, p_node) \tag{2}$$

Therefore, the solution is to calculate the similarity between each candidate value and the parent node vector of the predicted node. Finally, we obtain the prediction token (T, V).

4. The Proposed Approach

This Section contains four parts: The Framework, Data Processing, Semantic Feature Extraction and Graph Embedding Mode.

4.1. The Framework

Fig. 2 shows the main architecture of our proposed model. The overall framework of this paper is mainly divided into three parts: Data Processing, Semantic Feature Extraction, and Graph Embedding Model.

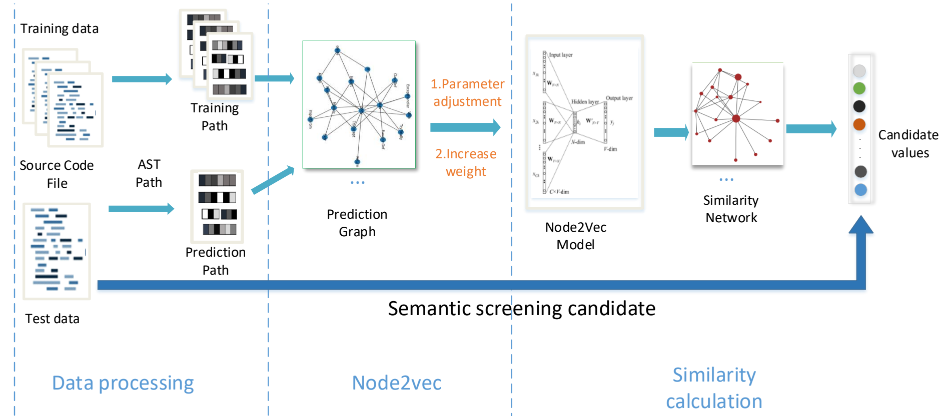

Fig. 2. The model framework

The data processing part uses context-free grammar to transform the source file into an abstract syntax tree. We extract the path of the AST terminal node and convert it into a training graph. Semantic structure feature can extract global information of remote dependencies, and the extracted semantic structure can filter out new candidate values. The graph embedding model uses the Node2vec algorithm to reduce the dimensionality of the node graph. It biasly extracts the relevant node sequences and converts them into vector of fixed dimensions, and the vector will bring into the downstream prediction task for calculation.

4.2. Data Processing

After converting the source code to AST, we extract the path of the terminal node. As shown in Fig. 3(a), take the terminal node $logging$ as an example. We obtain its upward

path $UpPath$ and downward path $DownPath$, and merge the two path to get the $logging$ node's path $NodePath$. $UpPath$ contains the node hierarchy information of AST, which can reflect the hierarchical characteristics of nodes in AST. $DownPath$ mainly extracts local information of terminal nodes and can directly reflect the correlation of neighboring nodes. Each node path contains rich and steric information.

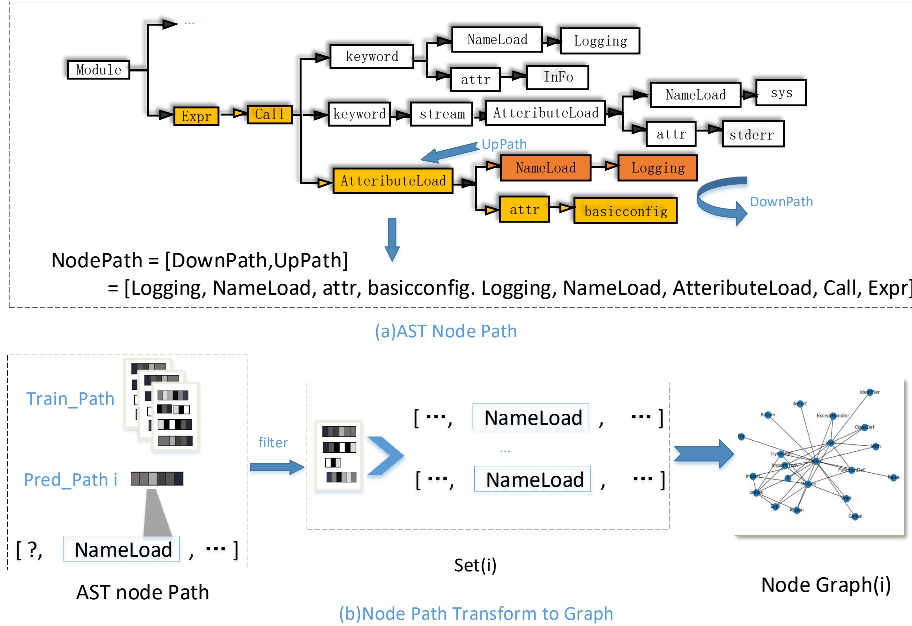

Fig. 3. Data processing

For each prediction path, we make full use of the existing structural information. The parent node of the terminal node can provide information intuitively, so we use the parent node to filter the training data set. In particular, each prediction data will have a corresponding data set and generate a corresponding graph structure. As shown in Fig. 3(b), the known parent node $NameLoad$ of the predicted path is used to filter out the data set $Set(i)$ and convert it into the corresponding graph $G(i)$. For edges extracted from the source data, the model will count the number of occurrences and use this as the weight of the graph $G(i)$.

After data processing, we can remove a lot of redundant information in the source code files. This process not only reduces the node size of the graph, but also retains useful information.

4.3. Semantic Feature Extraction

Programming language is a natural language with obvious repetitive characteristics[1]. We can extract the semantic features of the programming language from the contextual

relevance to help us complete specific tasks. In this paper, we analyze the structure of the AST of the source file and combine the code semantics to find out the rules between the structures. In traditional prediction tasks, the $Top - K$ values with the most occurrences in the training data are established as candidate values (Section 4.1). But there are two conflicting issues in this process:

– The range of the candidate value table is small. Since the number of candidate values is small, there is a high probability that the predicted node value is not in the candidate value table. This situation will not only cause OoV problems, but also reduce the prediction accuracy.

– The candidate value table has a larger range. Since there are too many candidate values, the prediction accuracy will be improved. However, each prediction process must calculate a large number of irrelevant candidate nodes. This leads to a significant increase in the prediction time of the model.

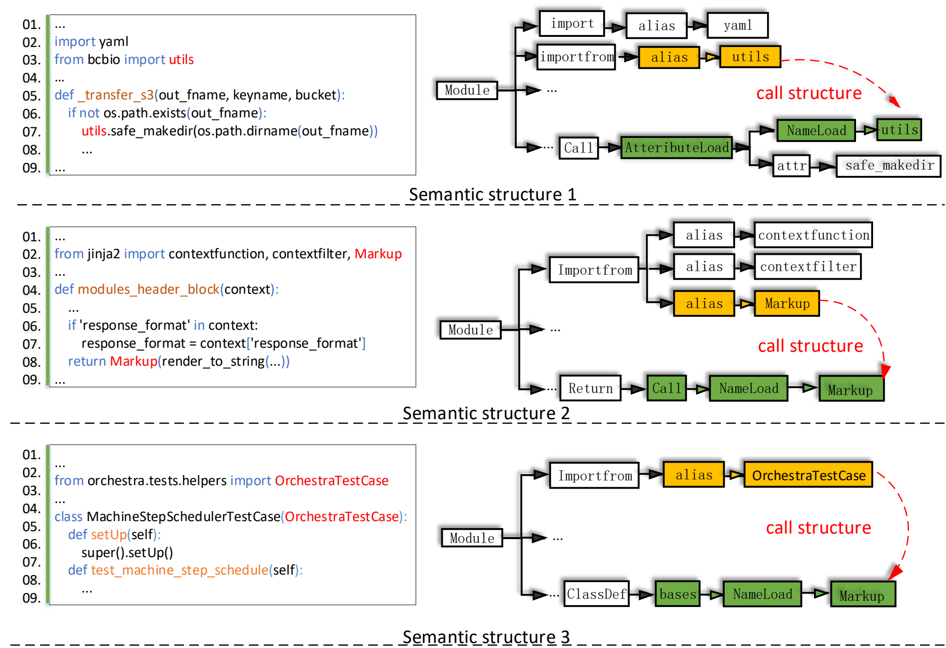

Fig. 4. The semantic structure of calling python packages

Therefore, the core problem of the prediction model is to determine a valid candidate value table. In neural network models, researchers often choose tens of thousands of fixed candidate values. Compared with the traditional model, although the accuracy of the neural network model has been improved, it consumes longer time. In order to solve the problem, this paper dynamically selects candidate value tables by extracting contextual semantic features.

This method is mainly to solve the VALUE prediction task of the missing node. Because there are only about 100 candidate values in the prediction task of TYPE, these words are determined by the programming language. However, most node values are defined by programmers. The randomness of these values is large, which leads to a wide range of candidate values. As shown in the Fig. 4, take the name of the missing package in python as an example:

In the programming language, the packages to be called need to be defined in advance (AST structure: $alias \rightarrow value$). We found that the code called and imported has a fixed semantic structure in AST. The called package has six structures, but the most important are the three semantic structures shown in Fig. 4. We record the known package name in the $vPet$ collection, and record the called package in the $vSet$ collection. $vSet - vPet$ is a candidate value for the called but missing defined package name. Conversely, if the missing node is the code that calls the package, $vPet - vSet$ will be the new candidate value table. When $vSet - vPet$ is empty or the predicted value is not called below, we select the K most frequently occurring values related to the parent node as candidate values. This process can remove the candidate words that are completely impossible to predict. Similarly, we can use the parent node semantics of the prediction node to extract other semantic structural features, such as $NameLoad \rightarrow value$, $NameStore \rightarrow value$, etc. By extracting the context semantic structure of the terminal node, we can reduce thousands of candidate values to tens or even a few. Since the obtained candidate value changes with the prediction file, it can effectively reduce the OoV rate.

The pseudo code for semantic feature extraction algorithm is as shown in:

Algorithm 1: Semantic Feature Extraction Algorithm

```
Input: Prediction File PF, number n;
       Prediction Data TeD = (DownPath, UpPath);
       Candidate value Candidate_Value ;
Output: New Candidate Value: New_Candidate_Value ;
1  p_node ← Each TeD's parent node
2  for i = 1 : n do
3      for j = 1 : len(PF_i) do
4          if Parent_node ==' alias' then
5              vSet.append(Structure(AtteributeLoad → NameLoad → Value));
6              vSet.append(Structure(Call → NameLoad → Value));
7              vSet.append(Structure(bases → NameLoad → Value));
8              vPet.append(Structure(alias → Value));
9          end
10         PF_i New_Candidate_Value = vSet − vPet;
11         Similarly, filter other semantic features by different p_node.
12     end
13     if PF_i New_Candidate_Value == None then
14         PF_i New_Candidate_Value = Candidate_Value;
15     end
16 end
```

4.4. Graph Embedding Model

The sequence of graph nodes in the DeepWalk algorithm is randomly extracted, while the sequence extraction of Node2vec combines DFS and BFS of nodes. Node2vec is a graph embedding method that comprehensively considers the DFS neighborhood and BFS neighborhood information of the graph. It is regarded as an extension algorithm of DeepWalk.

Random Walk : Node2vec obtains the neighbor sequence of vertices in the graph by biased random walk, which is different from DeepWalk. Given the current vertex v, the probability of visiting the next vertex x is:

$$P(c_i = x|c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z} \; if(v,x) \in E \\ 0 \quad otherwise \end{cases} \tag{3}$$

where π_{vx} is the unnormalized transition probability between nodes v and x, and Z is the normalizing constant.

Search Bias α : The simplest method of biased random walk is to sample the next node according to the weight of the edge. However, this method does not allow us to adjust the search process to explore different types of network neighbors. Therefore, the biased random walk should be a fusion of DFS and BFS, rather than mutually exclusive. The model should combine the structural features and content features between the nodes.

The two parameters p and q which guide the random walk. As shown in Fig. 5, we suppose that the current random walk through the edge (t, v) reaches the vertex v, edge labels indicate search biases α. The walk path now needs to decide on the next step. The method will evaluate the transition probabilities π_{vx} on edges (v, x) leading from v. Node2vec set the transition probability to $\pi_{vx} = \alpha_{pq}(t, x) * w_{vx}$, where

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} \; if \quad d_{tx} = 0 \\ 1 \; if \quad d_{tx} = 1 \\ \frac{1}{q} \; if \quad d_{tx} = 2 \end{cases} \tag{4}$$

and w_{vx} is the edge weight between nodes.

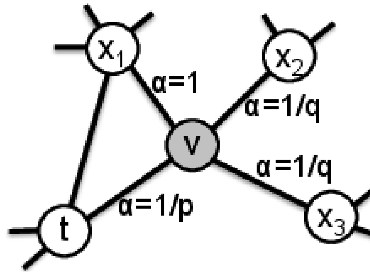

Fig. 5. The next step out of node v

Return parameter, p:

- **If** $p > max(q, 1)$ The probability of sampling to the original node is very small. As shown in Fig. 5, the probability of the next node returning to node t is low.
- **If** $p < max(q, 1)$ The probability of node sampling returning to the previous node is high. This causes some node paths to move around the starting point.

In-out papameter, q:

- **If** $q > 1$ The node sequence will move among the nodes near the starting point, which can reflect the BFS characteristics of the node.
- **If** $q < 1$ The node sequence will become farther from the starting node, and the return probability is small. This reflects the characteristics of DFS.

When $p = 1$ and $q = 1$, the walk mode is equivalent to the random walk in DeepWalk.

Similarity Calculation : The bias random walk algorithm can extract the node structure path in train graph. Then, the model use Word2vec's SkipGram algorithm to convert the node sequence into a vector of fixed dimensions. The Word2vec model can reduce the dimensionality of discrete text data into quantifiable vectors. We use the cosine function to calculate the structural coincidence between nodes. The candidate node with the largest cosine value also contains the most of the same node structure feature.

The calculation function of the similarity between nodes is shown in Equation 5:

$$cos(\theta) = \frac{V_1 \cdot V_2}{\|V_1\| \, \|V_2\|} = \frac{\sum_{i=1}^{n} V_{1i} \times V_{2i}}{\sqrt{\sum_{i=1}^{n} (V_{1i})^2} \times \sqrt{\sum_{i=1}^{n} (V_{2i})^2}} \tag{5}$$

The pseudo code for our entire model: **Algorithm 2**

Algorithm 2: Graph Embedding Code Prediction

Input: Train Data TD=$(DownPath, UpPath)$,number n;
Prediction Data TeD = $(DownPath, UpPath)$;
New Canditite Value $New_Canditite_Value$ and number s ;
Output: Predicted Value $pred_value$;
1 $p_node \leftarrow$ Each $TeD's$ parent node
2 **for** $i = 1 : n$ **do**
3     $Set \leftarrow$ p_node in TD_i
4 **end**
5 $G(i) \leftarrow$ Produce graph by Set's node
6 $G'(i) \leftarrow$ Increase the $G(i)$ weight of edges related to Ted
7 $Embed_model \leftarrow$ Node2vec($G'(i), p, q$), adjusting parameters p, q
8 **for** $j = 1 : s$ **do**
9     $V1 = Embed_model(p_node)$
10     $V2 = Embed_model(Cand_Value_j)$
11     $SimScore \leftarrow Cos(V1, V2)$
12 **end**
13 $pred_value \leftarrow Max(SimScore)$

5. Experiment Set Up

The experiment is mainly divided into four parts. First of all, we introduce the data set. Then, we discuss the prediction tasks of TYPE and VALUE. In the last part, the paper discusses the experimental results. The experimental hardware environment is Intel(R) Xeon(R) CPU E5-2660 v4 @ 2.00GHz; RAM 32.00GB.

5.1. Data Set

In the experiment, we collect Python data set from the Github[4] repository. The data set contains 10,000 training data files and 500 prediction data files. These Python source files have high star mark in github and are public available.

As shown in Table 1, In the TYPE prediction task, there are only 132 types. Such as $NameLoad$, $alias$, $NameParam$, etc. These types are determined by the Python programming language and cannot be set by the programmer, which results in fewer candidate values. In the VALUE prediction task, the source file has 51,000 different node values. There are arbitrary possibilities for encoding the program text. The value can be any program identifier (such as $None$, $format$), literals (such as 0.035, 1075), program operators (such as $/$, $-$, $*$), etc. It is impossible to use all of them for calculation, especially some of these values only appear once, so we need to filter the vocabulary.

Table 1. Dataset statistics

	Category	Size
1	Type Vocabulary	132
2	Value Vocabulary	$5.1*10^5$
3	Training files	10000
4	Test files	500

Table 2. TYPE Nodes type

	Types	Size
1	NameLoad	$1.2*10^6$
2	attr	$1.1*10^6$
3	AttributeLoad	$8.4*10^5$
4	Str	$5.1*10^5$
...	...	...
132	CompareLtELtELtE	1

Table 3. VALUE Nodes type

	Types	Size
1	self	$2.8*10^5$
2	None	$4.0*10^4$
3	0	$3.8*10^4$
4	1	$3.4*10^4$
...	...	...
$5.1*10^5$	Sysbench-read cleanup on %s	1

[4] https://github.com

We extract all the node types in the source code files, a total of 132 different types. As shown in Table 2, we can find that there is a significant difference between the maximum and minimum number of occurrences of node's type. For example, $CompareLtELtELtE$ appears only once in the source data, and it has little effect on other tens of thousands of candidate values during the experiment. Therefore, we choose the type value that appears more than 100 times as the candidate value in the prediction TYPE task.

As shown in Table 3, the number of node values in the training code source file is $5.1 * 10^5$, and most of them are random strings defined by programmers. The number of unique node values in dataset is too large to directly apply neural networks models and the first type $self$ is 2.8*10^5 times different from the last value $Sysbench - read\ cleanup\ on\ \%s$, thus we choose $K = 1000$ most frequent values in all training set to build the global vocabulary of type.

However, using the $Top - K$ values in the training data as fixed candidate values, this method cannot avoid OoV problems. Therefore, we need to dynamically select candidate values with semantic structural features.

5.2. TYPE Prediction

In the TYPE prediction task, the number of candidate values is small, and none are randomly defined by the programmer. Therefore, the model mainly learns the structural features of the nodes in the training graph and completes the prediction of the node types. Secondly, the candidate values of type are fixed and will not change with different prediction data. This is the biggest difference from the prediction task of VALUE.

The selection of the candidate value of TYPE is mainly filtered by the parent node of the terminal node through two methods:

- **Semantic Extraction:** For a fixed code structure, we extract the specific AST path structure. For example, when importing a package, $Import \rightarrow alias$ is a fixed structure, and $Import$ does not appear at other AST node locations.
- **Traverse the Dataset:** Traverse the parent node of each terminal node of all training data, this method is suitable for TYPE prediction tasks. Because the number of TYPEs is limited, the short traversal time can accurately filter out candidate values.

The graph embedding model mainly extracts related node sequences between graph nodes, and uses Word2vec[19] to convert these sequences into fixed-dimensional vectors. Then, we calculate the vector similarity between the candidate value and the parent node of the prediction node. The higher the similarity between nodes, the greater the degree of coincidence of the path structure extracted by the graph model. Therefore, the probability of connection between nodes is higher. In the extraction of graph node sequence, it is mainly affected by the parameters p and q. The parameters p and q affect the search method of the node sequence, so the information contained in the node sequence is different. In order to distinguish different prediction paths with the same parent node, we increase the weight of known nodes in the prediction path. This method can effectively distinguish the paths with the same parent node but different prediction values. After obtaining the training graph related to the predicted data, we train the model to determine the parameter (p, q). By changing the range of the parameters (p, q) with the model accuracy, the approximate range is determined. For example $p > 1, q < 1$, then we fine-tune the model to determine the exact values of the parameters.

Table 4. Candidates for TYPE

Father Node	Candidate values	(p,q) value
ImportFrom	alias	(p=1.8, q=0.9)
Import	alias	(p=2, q=0.8)
args	NameParam, TupleStore	(p=2, q=0.3)
alias	identifier	(p=1.8, q=0.8)
bases	NameLoad, AttributeLoad, Call	(p=2, q=0.8)
AttributeLoad	NameLoad, attr, Str, Num	(p=1.7, q=0.8)
keyword	NameLoad, attr, Str, Num	(p=0.8, q=1.5)
Assign	NameStore, Call, ListLoad, NameLoad, Str, IfExp, GeneratorExp, BinOpMod, SubscriptLoad, AttributeLoad, BoolOpOr, ...	(p=1.6, q=0.5)
Call	NameLoad, AttributeLoad, TupleLoad, keyword, ListLoad, BinOpMod, Str, Dict, Num, ListComp, BinOpMult, ...	(p=1.5, q=0.5)
TupleLoad	NameLoad, AttributeLoad, Str, SubscriptLoad, BinOpAdd, Num, ListLoad, Call, BinOpSub, Dict...	(p=1.5, q=0.9)
ListLoad	NameLoad, Str, Num, Call, AttributeLoad, BinOpAdd, TupleLoad, ListLoad, Dict, BinOpMod,...	(p=2, q=0.8)
...	...	...

In Table 4, there are only a few candidate values for $Import$, $importForm$, and $Assign$ has a maximum of 50 candidate values. So we can see that the range of candidate values has been significantly reduced, and the candidate values of TYPE are fixed after screening. This process can not only increase the accuracy of prediction, but also shorten the prediction time. In model prediction, the model mainly extracts node sequences based on depth-first search$(p > 1, q < 1)$. However, the node sequence of breadth-first search is also included, such as keywords, parameters $(p = 0.8, q = 1.5)$.

5.3. Value Prediction

The prediction task of VALUE is much more difficult than the prediction task of TYPE. First of all, the candidate value of can reach tens of thousands in VALUE prediction task. Secondly, for the artificially defined word names of programmers, it is difficult to obtain effective structural feature information in training data. Especially for nodes of type Str, the range of candidate values is very large and random. But during the experiment, we can extract the candidate values of the semantic structure of the prediction file. We extract the semantic features of the AST according to the parent node of the prediction node and filter out new candidate values. Such dynamic candidate values can effectively solve the OoV problem.

Similar to the TYPE prediction task, we mainly use the graph embedding model to extract related node sequences between graph nodes, and Word2vec algorithm converts these sequences into fixed-dimensional vector.

For the parent nodes with obvious structural semantics, such as $NameLoad$, $alias$, $NameParam$ and $NameStore$, we use the semantic structure features to filter them. As shown in Table 5, the node sequence extraction of the model is based on DFS.

Table 5. Semantic structure for VALUE

Father Node	Related Semantic Structure	(p,q) Value
alias	$identifier/NameStore/alias... \rightarrow value$, $Call/bases/AttributeLoad... \rightarrow NameLoad \rightarrow value$	(p=2, q=0.8)
NameLoad	$alias/NameParam/FunctionDef... \rightarrow value$, $bases/Raise... \rightarrow NameLoad \rightarrow value$	(p=2, q=0.8)
NameStore	$identifier/NameParam/alias... \rightarrow value$, $Call/AttributeLoad... \rightarrow NameLoad \rightarrow value$	(p=1.6, q=0.5)
NameParam	$NameStore/NameParam/alias... \rightarrow value$, $Call/AttributeLoad... \rightarrow NameLoad \rightarrow value$	(p=2, q=0.3)
...	...	...

But for nodes that are randomly defined by the programmer and will not be called below. We traverse all training data sets and select the $Top - K$ words that appear most frequently as related parent nodes as candidate values. Compared with the traditional method of directly extracting the most frequent values. The advantage of this method is that the parent node can select candidate values to remove redundant words. For example, if the parent node is Num, We will extract the numbers in the training data as candidate values. And the most frequently occurring $self$ will not be considered as a candidate. But in the traditional prediction model, $self$ will be brought into the model for calculation.

5.4. Experimental Results

First of all, we introduce prediction accuracy to evaluate the performance of our proposed model, which can be described as Eq.(6):

$$Accuracy = \frac{The\ number\ of\ correct\ prediction\ nodes}{The\ total\ number\ of\ prediction\ nodes} \tag{6}$$

The experimental results compared to the state-of-the-art[16] model in the same data set are shown in the table below:

As can be seen from Table 6, compared with the state-of-the-art model experimental results, our model has better results. Especially in the prediction task of TYPE, its accuracy has improved significantly. The main reason are: 1. We screened the candidate values of TYPE, narrowing the candidate range from hundreds to dozens or even a few. 2. The main prediction of TYPE is the structure of the code, and the path and training map we extract are highly relevant to the code structure. 3. There are few node types in the TYPE task, so the network scope of the composed node graph is small, resulting in improved prediction accuracy. In other words, the model effectively extracts the structural

Table 6. Comparison of final results

	TYPE	VALUE
Attentional LSTM	71.1%	-
Pointer Mixture Network	-	62.2%
Our Model	**77.8%**	**63.8%**

features of the code. In the prediction task of VALUE, the accuracy is improved by 1.6%. Although the accuracy has not improved much, the overall prediction time of the model has been reduced significantly.

Table 7. Comparison of prediction time cost

	Total Time	Type Task Time	Value Task Time
Attentional LSTM **Pointer Mixture Network**	>20h	-	-
Our Model	8.5h	2.3h	6.2h

As shown in Table 7, the running time of the deep learning model method on the same data set exceeds 20 hours. And the interpretability of the deep learning model is low. The total consumption time of our proposed model is 8.5 hours, and the complex VALUE prediction task takes about 70% of the time. This is mainly because the training graph has more nodes than the training graph of the TYPE task, so the embedding of the graph takes longer time.

The model graph structure feature extraction proposed in this paper is more interpretable and more intuitive than deep learning model for prediction tasks. In the overall graph embedded in the model, the value range of the (p, q) parameter is mainly $(p > 1, q < 1)$. The node sequence of the model incorporates more depth-first node information.

We predict the example in Fig. 1 and the result is shown in Fig. 6. We calculated the similarity between the node in the graph structure and the parent node $Assign$. After model calculation, we find that the type of the missing node is $NameStore$. Then we get the candidate value table through the screening of the code semantic structure, and calculate the predicted value of the OoV word LOG as the final value. The missing node is $(T, V) = (NameLoad, LOG)$. Experiments show that the OoV rate in the predicted data drops from 19.6% to 5.7%, and most of them are Str nodes that fail to accurately predict. The overall OoV rate has dropped significantly, and the prediction accuracy has also been significantly improved.

However, nodes of type Str are difficult to predict. First of all, these words may have been created by the programmer themselves, so they will basically not appear in the candidate value table. Secondly, these words appear rarely. they do not even appear in

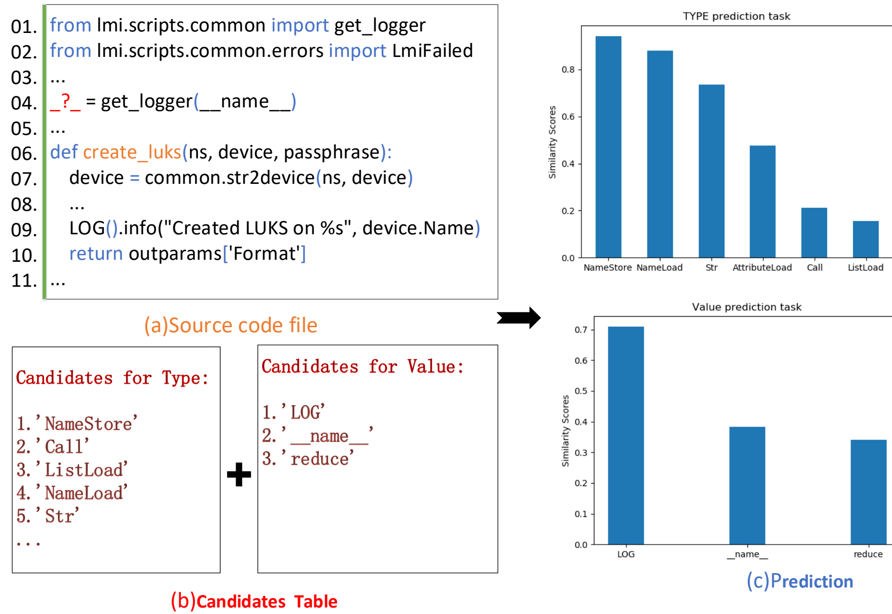

Fig. 6. Code prediction example

the training Graph, which leads to a reduction in the overall prediction accuracy of value. This is also a disadvantage of this model.

6. Conclusion and Future Work

In this paper, we use the node information of the source code AST to construct training graphs, which contain a lot of node structure information. Through the embedding model, we can embed the graph structure node sequence as a fixed-dimensional vector. Then carry it to the downstream task for calculation. In the selection of candidate values, we use semantic structural features to dynamically filter candidate values, which not only reduces the prediction time, but also effectively reduces the OoV situation.

The experimental results show that the model can effectively extract structural features in the prediction task of TYPE, and the prediction accuracy is greatly improved. In the prediction task of VALUE, the screening of candidate values not only improves the accuracy, but also shortens the time.

The accuracy improvement of TYPE shows that the accuracy of the programming language with more strict structure will be more obvious. Therefore, in future work we will add Java data sets to verify our model. And, we will further extract the semantic features of the fusion Control Flow Graph(CFG).

Acknowledgments. This work is partially supported by the NSF of China under grants No.61702334 and No.61772200, the Project Supported by Shanghai Natural Science Foundation No.17ZR 1406900, 17ZR1429700 and Planning project of Shanghai Institute of Higher Education No.GJEL18135.

References

1. Allamanis, M., Barr, E.T., Devanbu, P.T., Sutton, C.A.: A survey of machine learning for big code and naturalness. ACM Comput. Surv. 51(4), 81:1–81:37 (2018)
2. Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to represent programs with graphs. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings (2018)
3. Allamanis, M., Sutton, C.A.: Mining source code repositories at massive scale using language modeling. In: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013. pp. 207–216 (2013)
4. Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. In: 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings (2015)
5. Bahl, L.R., Jelinek, F., Mercer, R.L.: A maximum likelihood approach to continuous speech recognition. IEEE Trans. Pattern Anal. Mach. Intell. 5(2), 179–190 (1983)
6. Bhoopchand, A., Rocktäschel, T., Barr, E.T., Riedel, S.: Learning python code suggestion with a sparse pointer network. CoRR abs/1611.08307 (2016)
7. Bruch, M., Monperrus, M., Mezini, M.: Learning from examples to improve code completion systems. In: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009. pp. 213–222 (2009)
8. Cao, S., Lu, W., Xu, Q.: Deep neural networks for learning graph representations. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA. pp. 1145–1152 (2016)
9. Duvenaud, D., Maclaurin, D., Aguilera-Iparraguirre, J., Gómez-Bombarelli, R., Hirzel, T., Aspuru-Guzik, A., Adams, R.P.: Convolutional networks on graphs for learning molecular fingerprints. In: Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada. pp. 2224–2232 (2015)
10. Gao, H., Huang, W., Duan., Y.: The cloud-edge based dynamic reconfiguration to service workflow for mobile ecommerce environments: A qos prediction perspective. ACM Transactions on Internet Technology (2020)
11. Gao, H., Kuang, L., Yin, Y., Guo, B., Dou., K.: Mining consuming behaviors with temporal evolution for personalized recommendation in mobile marketing apps. ACM/Springer Mobile Networks and Applications (MONET) (2020)

12. Gao, H., Liu, C., Li, Y., Yang., X.: V2vr: Reliable hybrid-network-oriented v2v data transmission and routing considering rsus and connectivity probability. IEEE Transactions on Intelligent Transportation Systems pp. 1–14 (2020)
13. Gao, H., Xu, Y., Yin, Y., Zhang, W., Li, R., Wang., X.: Context-aware qos prediction with neural collaborative filtering for internet-of-things services. IEEE Internet of Things Journal 7(5), 4532–4542 (2020)
14. Grover, A., Leskovec, J.: node2vec: Scalable feature learning for networks. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016. pp. 855–864 (2016)
15. Hindle, A., Barr, E.T., Gabel, M., Su, Z., Devanbu, P.T.: On the naturalness of software. Commun. ACM 59(5), 122–131
16. Li, J., Wang, Y., Lyu, M.R., King, I.: Code completion with neural attention and pointer networks. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden. pp. 4159–4165 (2018)
17. Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R.S.: Gated graph sequence neural networks. In: 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings (2016)
18. Malhotra, P., Vig, L., Shroff, G.M., Agarwal, P.: Long short term memory networks for anomaly detection in time series. In: 23rd European Symposium on Artificial Neural Networks, ESANN 2015, Bruges, Belgium, April 22-24, 2015 (2015)
19. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. In: 1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings (2013)
20. Nguyen, A.T., Hilton, M., Codoban, M., Nguyen, H.A., Mast, L., Rademacher, E., Nguyen, T.N., Dig, D.: API code recommendation using statistical learning from fine-grained changes. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016. pp. 511–522 (2016)
21. Nguyen, T.T., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N.: A statistical semantic language model for source code. In: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013. pp. 532–542 (2013)
22. Perozzi, B., Al-Rfou, R., Skiena, S.: Deepwalk: online learning of social representations. In: The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014. pp. 701–710 (2014)
23. Rahman, M., Palani, D., Rigby, P.C.: Natural software revisited. In: Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019. pp. 37–48 (2019)
24. Raychev, V., Vechev, M.T., Yahav, E.: Code completion with statistical language models. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014. pp. 419–428 (2014)

25. Tu, Z., Su, Z., Devanbu, P.T.: On the localness of software. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014. pp. 269–280 (2014)
26. Ul-Hasan, A., Ahmed, S.B., Rashid, S.F., Shafait, F., Breuel, T.M.: Offline printed urdu nastaleeq script recognition with bidirectional LSTM networks. In: 12th International Conference on Document Analysis and Recognition, ICDAR 2013, Washington, DC, USA, August 25-28, 2013. pp. 1061–1065 (2013)
27. Yang, X., Zhou, S., Cao, M.: An approach to alleviate the sparsity problem of hybrid collaborative filtering based recommendations: The product-attribute perspective from user reviews. MONET 25(2), 376–390 (2020)

Kang Yang received his B.S. degree from Shanghai Ocean University in 2017. He is a Ph.D. student in computer science at East China University of Science and Technology. His research interests include software engineering, natural language processing.

Huiqun Yu received his B.S. degree from Nanjing University in 1989, M.S. degree from East China University of Science and Technology (ECUST) in 1992, and Ph.D. degree from Shanghai Jiaotong University in 1995, all in computer science. He is currently a Professor of computer science with the Department of Computer Science and Engineering at ECUST. From 2001 to 2004, he was a Visiting Researcher in the School of Computer Science at Florida International University. His research interests include software engineering, high confidence computing systems, cloud computing and formal methods. He is a member of the ACM, a senior member of the IEEE, and a senior member of the China Computer Federation.

Guisheng Fan received his B.S. degree from Anhui University of Technology in 2003, M.S.degree from East China University of Science and Technology (ECUST) in 2006, and Ph.D. degree from East China University of Science and Technology in 2009, all in computer science. He is presently a research assistant of the Department of Computer Science and Engineering, East China University of Science and Technology. His research interests include formal methods for complex software system, service oriented computing, and techniques for analysis of software architecture.

Xingguang Yang received his B.S. degree from China University of Mining and Technology in 2016. He is a Ph.D. student in computer science at East China University of Science and Technology. His research interests include software engineering, Software defect prediction.

Received: September 8, 2019; Accepted: July 7, 2020.