Code Prediction Based on Graph Embedding Model

Kang Yang[1], Huiqun Yu[1,2(✉)], Guisheng Fan[1,3(✉)], Xingguang Yang[1], and Liqiong Chen[4]

[1] Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai, China
{yhq,gsfan}@ecust.edu.cn
[2] Shanghai Key Laboratory of Computer Software Evaluating and Testing, China, Shanghai, China
[3] Shanghai Engineering Research Center of Smart Energy, Shanghai, China
[4] Department of Computer Science and Information Engineering, Shanghai Institute of Technology, Shanghai 200235, China

Abstract. Code prediction aims to accelerate the efficiency of programmer development. However, its prediction accuracy is still a great challenge. To facilitate the interpretability of the code prediction model and improve the accuracy of prediction. In this paper, the source code's Abstract Syntax Tree (AST) is used to extract relevant structural paths between nodes and convert them into training graphs. The embedded model can convert the feature of the node sequence in the training graph into a vector that is convenient for quantization. We calculate the similarity between the candidate value and the parent node vector of the predicted path to obtain the predicted value. Experiments show that by using prediction data to increase the weight of related nodes in the graph, the model can extract more useful structural features, especially in Value prediction tasks. Adjusting the parameters embedded in the graph can improve the accuracy of the model.

Keywords: Big code · Graph embedding · Code prediction

1 Introduction

Recent years, with the development of technology, code prediction has attracted more and more attention in the field of Big Code. The emergence of code prediction technology can effectively improve the efficiency of programmers. Because the code has obvious repetitive characteristics [1], the analysis of the source code file can successfully extract the characteristic information of the code. Then, we bring them into the probability model or deep neural model to predict the missing code.

Traditionally, for probability models such as n-gram, the model uses the node's $n-1$ tokens to predict the probability of the nth token. The probability

H. Gao et al. (Eds.): CollaborateCom 2020, LNICST 350, pp. 419–430, 2021.
https://doi.org/10.1007/978-3-030-67540-0_26

prediction model is simple and effective, but the accuracy is poor. These technologies [2] appeared in early research in this field. Subsequently, in order to improve the accuracy of prediction, Raychev et al. [3] improved the code prediction model by combining technologies in the NLP field. Since the AST of the programming language is also a natural language with a rich structure, NLP technology can be used to extract more characteristic information. The RNN, Bi-RNN, Attention mechanism [4] and other NLP technologies are fused into the code prediction model by researchers. These models can effectively complete the prediction task, but the interpretability of the model is poor, and the training process takes a long time.

Node2vec is a simple neural network algorithm that can extract graph structural features. This model can embed the structural path between graph nodes as a fixed-length vector which contains feature information between graph structure nodes, and is beneficial to the quantitative calculation of the model. Compared with the DeepWalk [5] algorithm, the random walk method of Node2vec is biased. Adjust the model parameters (p, q), the structure of node extraction can be inclined to Depth-First-Search(DFS) or Breadth First Search(BFS). Due to the rich structural characteristics of the code, we can convert the AST node path of the source code into training graph. The Node2vec model can convert the training graph into node sequences, and convert and embed them into vectors.

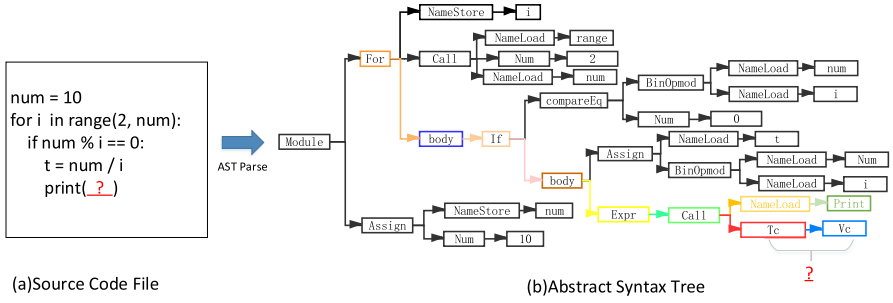

Fig. 1. Examples of Python programs and their corresponding AST

As can be seen in Fig. 1, a source code for finding prime numbers is transformed into an AST, where each node may have one attribute: type or value. Only terminal nodes contain type and value. The code lacks the output of the last line, and our task is to predict the node's type and value. We extract a large amount of AST's paths and convert them into a node graph to facilitate the extraction of related structural information. Finally, we embed the node path to quantify the prediction task and find the most suitable prediction value t.

In this paper, to facilitate effective code prediction. The main contribtions of this paper are as follows:

– Extract effective training graph structure. We extract the path of the AST terminal node from the source code and filter out irrelevant paths. Each set of

paths will be converted into a related training graph. This process effectively reduces the node range of the training graph and can speed up the prediction task.

– The task prediction accuracy has been improved. Compared with the state-of-the-art experiment results, the experiment shows that increasing the weight of the edges associated with the prediction data and adjusting the parameters can effectively improve the prediction accuracy.

This paper continues as follows. The Sect. 2 introduces some related works, the basic problem definition and the proposed method are explained in Sect. 3. Section 4 describes the experiment and the experimental results. Section 5 summarizes this paper and outlines future work prospects.

2 Related Works

Probabilistic models [6] are widely used in the field of code prediction because of the characteristics of simple and effective learning. The n-gram probabilistic model [7] derived from statistical natural language processing, it has been proven to successfully capture source code repeatability and predictable regularity. Besides, the model can perform code profiling, transplantation and design auxiliary coding equipment. Vincent J [8] compared the effects of n-gram model and deep learning model in code prediction tasks, and the results show that deep learning is not optimal. DeFreez et al. [9] uses the method of control-flow-graph. Traditional machine learning algorithms, such as decision trees, conditional random fields [10]are used in code prediction.

In addition, the neural network model [11]is also an effective prediction method in code prediction model. Veselin et al. [12] uses neural networks to automatically learn code from large dynamic JavaScript codes. The paper shows that the neural network not only uses token-level information, but also uses code structure information. Adnan Ul et al. [4] used the Bi-LSTM model training to split the source code identifiers, reducing the number of identifier core libraries, thereby improving the prediction accuracy of the code. Pointer networks [4] are widely used in code prediction tasks. The LSTM-based Attention Hybrid Pointer Network, which proposes a soft attention [13] or memory mechanism to alleviate the gradient disappearance problem in standard RNN [14]. Jian Li [15] proposed parent pointer hybrid network to better predict the OoV words in code completion.

In the past ten years, many graphics embedding methods have been proposed, including DeepWalk [5], Node2vec [16], and SDNE [17]. These methods have no obvious difference in the conversion method of embedded word vectors. However, the way of extracting paths is different. The Node2vec algorithm proposes an bias random walk algorithm based on the weights of graph node edges, which can fuse the structural information of node's DFS and BFS. The powerful function of GNN [18] in modeling the dependency relationship between graph nodes, which makes graph analysis related research fields achieve better results.

3 The Proposed Approach

3.1 Problem Definition

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. F represents a set of n source code files, $F = \{f_1, f_2, f_3, ...f_n\}$. A represents 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. The $Path$ contains AST's terminal node Up path and Down path. Finally, all node's paths are converted into a training graph G.

Definition 2 (Node Combination (T, V)). (T, V) is the predicted combination of missing nodes. It is calculated by the similarity between the parent node of the prediction node and the candidate value in Train-Graph G. T represents s candidate values in TYPE task, $T\_cad = \{T_1, T_2, T_3, ...T_s\}$. V represents k candidate values in VALUE task, $V\_cad = \{V_1, V_2, V_3, ...V_k\}$. T is the maximum value calculated by predicting the similarity between the parent node P_node and T, $T = S(G, T_i, P\_node)$. V is the maximum value calculated by predicting the similarity between the parent node and V, $V = S(G, V_i, P\_node)$.

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 parent node is $Call$ in Fig. 1(b).

After embedding graph nodes, the model calculates the similarity between the candidate value and the $Call$ vector of the parent node of the prediction node. Because the structural characteristics of the node are more similar to the parent node $Call$, the more likely the two are connected in AST. So the TYPE prediction task is to calculate $T = S(G, T_i, Call)$. Determine the type of the prediction node, then we use the type as the new parent node and calculate $V = S(G, V_j, T)$ to predict the VALUE task. This prediction process is transformed into the prediction of the node, as shown in the following Eq. (1), (2).

$$\exists i \in \{1, 2, 3, ...s\} : \arg\max_{T_i} S(G, T_i, P\_node) \tag{1}$$

$$\exists j \in \{1, 2, 3, ...k\} : \arg\max_{V_j} S(G, V_j, P\_node) \tag{2}$$

Therefore, the solution is to calculate the vector similarity of each candidate value with parent node of the prediction node. Finally the prediction token (T, V) is obtained.

3.2 Model Framework

In this Section, we will introduce the framework of our model in detail. Figure 2 shows the main architecture of our proposed model.

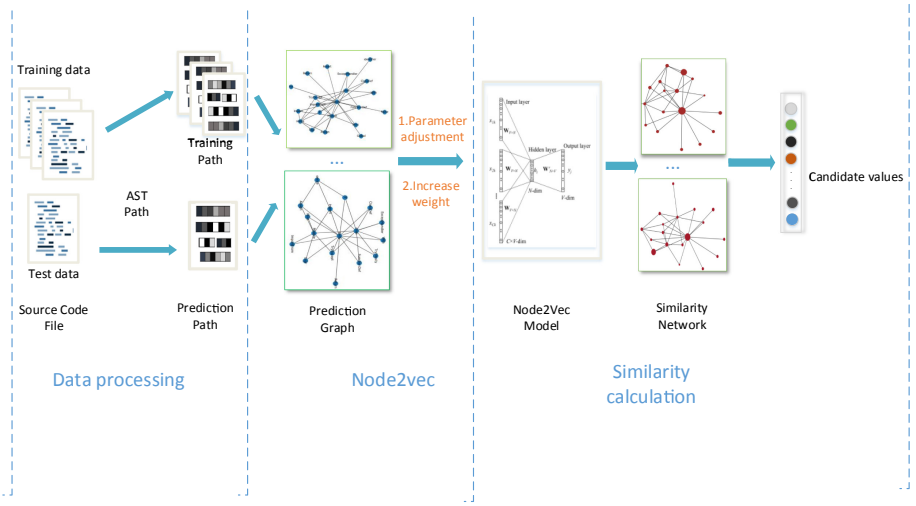

Fig. 2. Model framework

3.3 Data Processing

We parse source file to AST node's path and use the parent node of the prediction data to filter all extracted paths. In other words, each prediction data gets a corresponding set. Data processing can effectively use the information of existing prediction nodes to filter nodes that are not related to the prediction data.

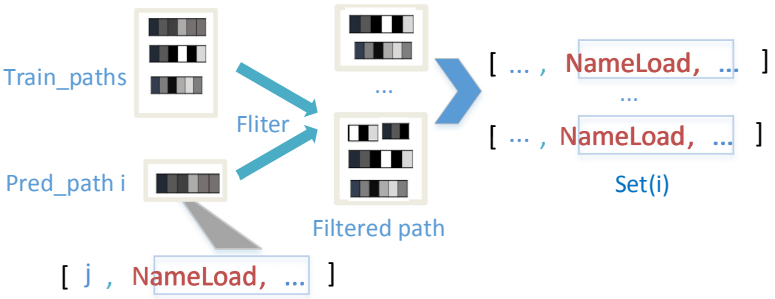

Fig. 3. Data processing

As shown in Fig. 3, the parent node *Nameload* of the prediction node is used to filter the training data. Finally, we convert these node paths into training graphs, and each prediction path *Set* has a corresponding training graph.

3.4 Graph Embedding Model

Node2vec is a graph embedding method that comprehensively considers the DFS and BFS node of the graph. It is regarded as an extended 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 $c_i = x$ 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 does not allow us to guide our search procedure to explore different types of network neighborhoods. Therefore, the biased random walk should be the fusion of DFS and BFS, rather than the mutual exclusion of the two method, combining the structural characteristics and content characteristics between nodes.

The two parameters p and q which guide the random walk. As shown in Fig. 4, 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 evaluates 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

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

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

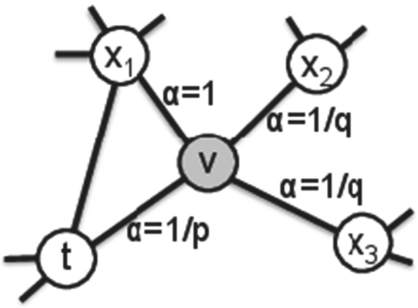

Fig. 4. The next step out of node v

The range of parameters p and q can control the path structure of node extraction. When $p = 1$ and $q = 1$, the walk mode is equivalent to the random walk in DeepWalk.

Similarity Calculation: We use Word2vec's SkipGram algorithm to convert the node path into a vector of fixed dimensions. 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 Eq. 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** 1

Algorithm 1: Graph Embedding Code Prediction

Input: Train Data TD=($DownPath, UpPath$),number n;
Prediction Data TeD = ($DownPath, UpPath$);
Canditite value $Cand_Value$ and number s ;
Output: Predicted Value $pred\_value$;

```
Parent_node ← Each TeD's parent node
for i = 1 : n do
    Set ← parent_node in TD_i
end
G(i) ← Produce graph by Set's node
G'(i) ← Increase the G(i) weight of edges related to Ted
Embed_model ← Node2vec(G'(i), p, q), adjusting parameters p, q
for j = 1 : s do
    V1 = Embed_model(parent_node)
    V2 = Embed_model(Cand_Value_j)
    SimScore ← Cos(V1, V2)
end
pred_value ← Max(SimScore)
```

4 Experiment SetUp

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

4.1 DataSet

In the experiment, we collect Python data sets from the Github 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.

4.2 TYPE Prediction

In Table 1, we extract the node types in all source files, a total of 132 types. However, there is a big difference between the maximum and minimum numbers. For example, *CompareLtELtELtE* appears only once in the source data, which has little effect on other candidate values that appear tens of thousands of times.

Table 1. 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 2. 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

For the prediction task of TYPE, we mainly learn the structural characteristics of the nodes in the training graph. Besides, finding the relative path of the parent node in the prediction data. We increase the weight of these path edges in the training graph, which can effectively affect the structural feature extraction. The adjustment of parameters is mainly related to the parent node of the prediction node.

4.3 Value Prediction

As shown in Table 2, 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 the data set is too large and there is a large gap in the number, which cannot be fully applied to the neural language model. Therefore, we choose the most frequent value of $K = 1000$ in each training set to build a type of global vocabulary. The training graph filtered by each parent node has related K candidate values. For example, $self$ appears the most frequently in source file, it will not be a candidate for num.

The prediction task of VALUE is much more difficult than the prediction task of TYPE. First, the prediction candidate value of the VALUE task reaches 1,000 more. Secondly, for the artificially defined word names of programmers, it is difficult to train effective structural feature information in the training data. Especially the node whose type is Str. However, during the experiment, we found that it is not necessary to bring every candidate value into the model for calculation. Because the value of the parent node is the type of prediction node will help us filter candidate values.

4.4 Experimental Results

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

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

The experimental results compared to the-state-of-art [16] in the same data set are shown in the table below: Nw,Np: The model does not increase the weight of related nodes, nor adjust the parameters. Nw,Wp: The model does not increase the weight of related nodes, but has adjustment parameters. Ww,Wp: The model increases the weight of related nodes and adjusts the parameters.

Table 3. Comparison of final results

	TYPE	VALUE
Attentional LSTM	71.1%	–
Pointer mixture network	–	62.2%
Nw, Np	47.8%	38.6%
Nw, Wp	70.2%	47.8%
Ww, Wp	**75.4**%	**63.6**%

Compared with the state-of-art model experimental results, our model prediction results are better. As can be seen from the Table 3, the adjustment of parameters (p, q) can effectively integrate the DFS and BFS structural information of the prediction nodes of the training graph. Secondly, artificially increasing the weight of the relevant edges of the predicted node path can affect the node selection during the random walk of the path. This process can effectively distinguish codes with similar structures.

Table 4. The effect of path length on accuracy

Length path	3	4	5	6	7
Type Acc	67.1%	71.8%	**75.4**%	74.3%	72.8%
Value Acc	52.5%	58.4%	61.6%	**63.6%**	62.3%

Extracting different path lengths has an impact on the prediction accuracy of the model. In Table 4, it can be seen that the longer the path length, the accuracy will not always increase. Because the longer the extracted path, the more overlap between the paths. The prediction accuracy of the model will decrease.

It can be seen from the Table 3 that for the TYPE prediction task, by adjusting the parameters (p, q), the structural feature method of fusing DFS and BFS

is more effective than adjusting the weight of the relevant edges of the node path. Therefore, it can be seen that for TYPE prediction tasks, the structural information between nodes is more important than the characteristics of some edge nodes. However, in the prediction task of VALUE, because the relevant edge nodes can help the model effectively reduce the prediction candidate value, increasing the weight of the relevant path is greater than the accuracy of adjusting the parameters.

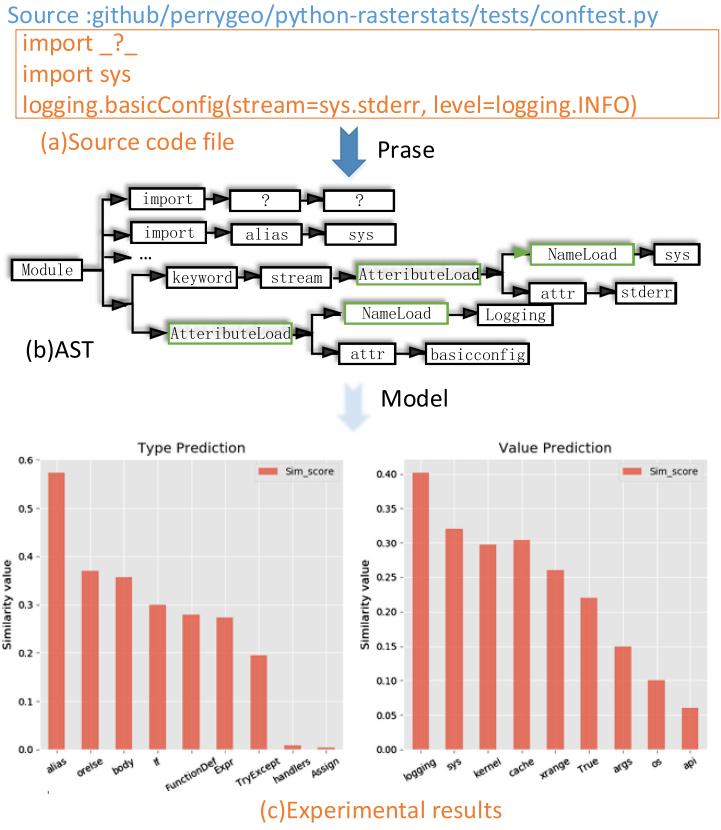

Fig. 5. A code prediction example

As shown in Fig. 5, an actual prediction example, this is the most common configuration test code in Python, which lacks the name of an imported package. In the training graph, the structure of *alias* and *import* are very related and they are appear in pairs. So the type can be accurately predicted to be *alias*. After completing the prediction of the node type, we can narrow the prediction range of the value by type *alias*. At the same time, we will increase the weight of the relevant node edges to facilitate the algorithm to extract the relevant

path. As shown in the above example, the missing source code is the name of the calling package. Obviously the name will be called later in the next code line, so we should pay attention on the structure path of the package call of $AttributeLoad-nameLoad-CadWord$. And the set $CadWord = \{logging, sys\}$ in the source code of this example. This method can accelerate the calculation of the entire model, and the accuracy is high.

5 Conclusion and Future Work

In this paper, we use the node information of the source code AST to construct a training graph that contains a large amount of node structure information.

The code prediction task is divided into two parts: one is node TYPE prediction, and the other is node VALUE prediction. Due to the small number of candidate values, the prediction task of TYPE is relatively simple, and the node structure can be effectively extracted in the training graph. For the prediction task of VALUE prediction, although the candidate value table is very large, we can filter the candidate values when the type of the node is known. The appropriate parameters (p, q) can effectively integrate the DFS and BFS information between nodes. On the other hand, increasing the edge weights of related nodes can also improve the accuracy of prediction. However, for VALUE prediction of type Str, the accuracy of model prediction is very low.

In future work, we will consider the OoV problem of the value prediction task. Besides, we will test the model on other data sets and compare the results with other algorithms.

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. 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, pp. 532–542 (2013)
3. Raychev, V., Vechev, M.T., Yahav, E.: Code completion with statistical language models. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, pp. 419–428 (2014)
4. 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, pp. 1061–1065 (2013)

5. 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, pp. 701–710 (2014)
6. Allamanis, M., Barr, E.T., Bird, C., Sutton, C.A.: Suggesting accurate method and class names. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 38–49 (2015)
7. Maddison, C.J., Tarlow, D.: Structured generative models of natural source code. In: Proceedings of the 31th International Conference on Machine Learning, pp. 649–657 (2014)
8. Vincent J, Devanbu, P.T.: Are deep neural networks the best choice for modeling source code? In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 763–773 (2017)
9. DeFreez, D., Thakur, A.V., Rubio-Gonzalez, C.: Path-based function embedding and its application to error-handling specification mining. In: Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 423–433 (2018)
10. Raychev, V., Vechev, M.T., Krause, A.: Predicting program properties from "big code". In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 111–124 (2015)
11. White, M., Vendome, C., Vasquez, M.L., Poshyvanyk, D.: Toward deep learning software repositories. In: 12th IEEE/ACM Working Conference on Mining Software Repositories, pp. 334–345 (2015)
12. Veselin, R., Martin, T.V., Andreas, K.: Predicting program properties from "big code". Commun. ACM **62**(3), 99–107 (2019)
13. Vinyals, O., Fortunato, M., Jaitly, N.: Pointer networks. In: Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, pp. 2692–2700 (2015)
14. Xu, K., et al.: Show, attend and tell: Neural image caption generation with visual attention. In: Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, 6–11 France July 2015, pp. 2048–2057 (2015)
15. Graves, A.: Supervised Sequence Labelling with Recurrent Neural Networks, Studies in Computational Intelligence, vol. 385. Springer (2012)
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, pp. 4159–4165 (2018)
17. 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, pp. 855–864 (2016)
18. Wang, D., Cui, P., Zhu, W.: Structural deep network embedding. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1225–1234 (2016)