Received: 28 February 2021 | Revised: 25 October 2021 | Accepted: 18 November 2021

DOI: 10.1002/smr.2414

RESEARCH ARTICLE - METHODOLOGY

Journal of Software: Evolution and Process WILEY

A graph sequence neural architecture for code completion with semantic structure features

Kang Yang[1] | Huiqun Yu[1,2] | Guisheng Fan[1,3] | Xingguang Yang[1] | Zijie Huang[1]

[1]Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai, China

[2]Shanghai Key Laboratory of Computer Software Evaluating and Testing, Shanghai Computer Software Technique Development Center, Shanghai, China

[3]Shanghai Engineering Research Center of Smart Energy, Shenergy Group Company Limited, Shanghai, China

Correspondence
Huiqun Yu and Guisheng Fan, Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai, China.
Email: yhq@ecust.edu.cn and gsfan@ecust.edu.cn

Funding information
National Natural Science Foundation of China, Grant/Award Number: 61772200; Natural Science Foundation of Shanghai, Grant/Award Number: 21ZR1416300

Abstract

Code completion plays an important role in intelligent software development for accelerating coding efficiency. Recently, the prediction models based on deep learning have achieved good performance in code completion task. However, the existing models cannot avoid three drawbacks: (i) In the existing models, the code representation loses the information (parent–child information between nodes) and lacks many effective features (orientation between nodes). (ii) The known code structure information is not fully utilized, which will cause the model to generate completely irrelevant results. (iii) Simple sequence modeling ignores repeated patterns and structural information. Besides, previous works cannot capture the characteristics of correlation and directionality between nodes. In this paper, we propose a **C**ode **C**ompletion approach named CC-GGNN, which is graph model based on **G**ated **G**raph **N**eural **N**etworks (GGNNs) to address the problems. We introduce a new architecture to obtain the effective code features from code representation. In order to utilize the known information, we propose Classification Mechanism, which classifies the representation of the node using the known parent node and constructs training graph in the model. The experimental results show that our model outperforms the state-of-the-art methods MRR@5 at most 9.2% and ACC at most 11.4% in datasets.

KEYWORDS

code completion, deep learning, program comprehension

1 | INTRODUCTION

As the scale of software continues to increase, developers benefit from code completion to accelerate software development intelligently. Code completion makes use of the existing code in the context to recommend the next code token, such as method calls or object fields. Because the source code has obvious repeatability and predictability,[1] probabilistic statistical model can capture this information well. N-gram[1-3] is the most widely used probabilistic statistical models in code completion. However, the input of the probabilistic statistical model is simple code tokens, which lacks the semantic structure information of the source code. Besides, the source code files are written by different programmers and have no fixed structure. For example, Python has a more casual programming style than Java. These reasons lead to the low accuracy of the prediction model, and we may even get the opposite result. Bielik et al. use the higher-order probabilistic model of probabilistic high-order grammar(PHOG)[4] method to address these problems. PHOG method uses context-free grammar to transform the source code into an Abstract Syntactic Tree (AST) as model input. AST not only reflects the grammatical structure of the code but also contains important lexical information. The AST nodes may represent identifiers and strings in the source code. With the development of deep learning in natural language processing tasks, many researchers believe that code is also a natural language. The method based on the Recurrent Neural Network Model (RNN)[5-7] is the most common neural network technique used for the code completion. Source code is represented as a node sequence or many node paths in these models. The model computes the probability of the candidate node and recommends the AST node with the highest probability.

However, how to effectively combine the syntactic information and structural features of the code to better complete the code is still a problem in the above model. There exist three fundamental problems not well solved in this regard.

The first one is the representation of code. The previous works are mainly divided into three categories for code representation: code tokens, AST sequence, and AST node's path. The code token models[8,9] just use statistical influence of the previous *n* tokens on the predicted value. The AST sequence model[6] obtains the sequence of the AST through the depth-first traversal method. Although the extracted sequence information is rich, the hierarchical information of the node is not represented in the sequence. The same problem also exists in the AST path model,[10-12] which only get the path between the terminal nodes. To address this limitation, Fang et al[13] use the root path and AST sequence to get the hierarchical structural information. However, the sequence models lost many information. The more branches of the AST, the more information lost in the sequence code representation. To preserve information in the code representation, we directly convert the AST of the node into a directed graph. AST graph will completely retain the structural information between nodes, including directions.

The second problem is how to utilize the known structural information in node representation. In previous works,[6,11] researches directly bring the nodes' sequence into the model for calculation. In order to make full use of the existing information, we classify these ASTs by their parent nodes. The predicted node with different parent nodes should consider different candidate values, such as *number* and *string*. This problem was not considered in prior works. Experimental results show that this operation can improve the effect of the prediction, and the prediction for some parent nodes in the type prediction task can reach 100% accuracy (Section 6.3).

Finally, the structural information of the AST provides a strong indication of the dependencies between link nodes that should be considered. However, simple sequential modeling ignores repetitive patterns and structural information, whereas undirected graph modeling ignores sequential and repetitive patterns. In this paper, we combine the correlation and directionality between AST nodes to model together.

To address the above problems, we propose CC-GGNN model for code completion with GGNN by integrating AST into a comprehensive code completion model that can leverage both semantic and syntactical information of code. CC-GGNN model uses classification mechanism to classify ASTs by known parent node and transforms them into different training graphs. Finally, the AST structure used by our model can effectively obtain the format of code repetitive pattern and obvious structural information. The Gate Recurrent Unit (GRU) structure in GGNN can help us solve the problem of long-distance dependence.

To evaluate the performance of our proposed model, we conduct experiments on two real-world datasets, including Python and JavaScript. The data that support the findings of this study are publicly available in previous work.[4,6] We compared CC-GGNN model with the state-of-the-art models: For the node's type prediction, our model achieves the best accuracy of 84.7% and 90.2% on Python and JavaScript datasets, respectively, which improves the state-of-the-art methods from 2.3% to 11.4% in PY50k dataset and improves the state-of-the-art methods from 3.2% to 10.1% in JS50k dataset. For the node's value prediction, our model achieves the accuracy of 73.5% and 78.8% on Python and JavaScript datasets, respectively, which improves the state-of-the-art methods from 1.6% to 10.1% in PY50k dataset and improves the state-of-the-art methods from 1.8% to 9.0% in JS50k dataset. Statistical testing shows that the improvements over the baseline methods are statistically significant.

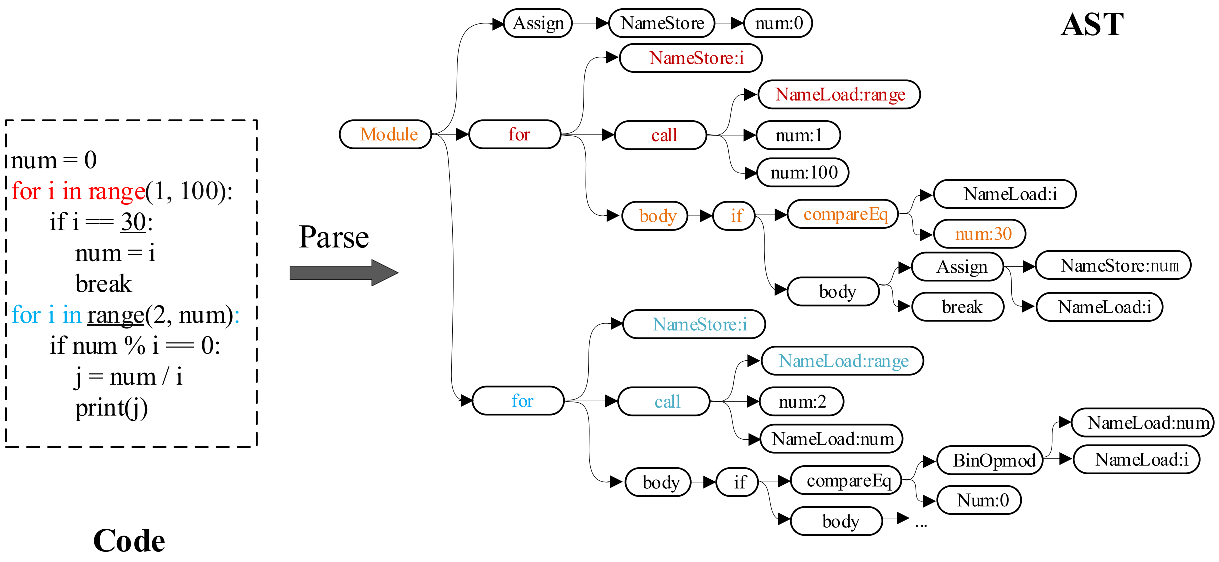
To sum up, the contributions of this work are as follows:

1. We extract the AST representation from the predicted node, which contains the structure of the node in the source code. Then, we construct directed training graph that contains rich structural information. The graph representation can effectively capture the format of code repetitive pattern and obvious structural information.
2. We use the known structure information to divide the training data into different training graphs. The prediction nodes with different parent nodes are considered separately. This classification mechanism can narrow the range of candidate values and improve the accuracy of prediction.
3. We evaluate our proposed model on two real-world datasets. Experimental results show that CC-GGNN achieves the best performance compared with the state-of-the-art models.

The paper is organized as follows. We give a motivating example in Section 2. Section 3 introduces the relevant background. The proposed approach is explained in Section 4. In Section 5, we describe the adopted dataset, data processing, and baselines, respectively. The evaluation and discussion are introduced in Section 6. Section 7 discusses the threats of experiments. Related works are addressed in Section 8. Finally, conclusions and future work are presented in Section 9.

2 | MOTIVATING EXAMPLE

Figure 1 shows a snippet of Python code and the corresponding AST. Each AST consists of terminal nodes and nonterminal nodes. The terminal node format is *type*: *value*, which appears in the AST leaf node. Meanwhile, the nonterminal node format is *type*: *EMPTY*, which is the intermediate



```
num = 0
for i in range(1, 100):
    if i == 30:
        num = i
        break
for i in range(2, num):
    if num % i == 0:
        j = num / i
        print(j)
```

FIGURE 1 An example of Python program and their corresponding Abstract Syntactic Tree (AST)

node of the AST. The special token of *EMPTY* is the same as the previous work[6] (not shown in Figure 1). The traditional sequence model captures the code representation by depth-first traversal of the AST, but this code representation method will cause information loss due to the AST's node bifurcation. For example, the node *for* of the second for loop (blue color) in Figure 1, its parent node should be *Module*, but the information of the previous node becomes *break* in the sequence model. Therefore, the parent node feature information of *for* is lost. The more branches of the AST, the more information lost in the sequence code representation. In this short example, there are 29 node branches. In order to avoid information loss, we input AST directly into the model as graph structure data. Such code representation can make full use of the graph-structured of the AST and can effectively avoid the loss of node information.

Secondly, in order to make full use of the known information, we process the parent node of the predicted node. Through the parent node classification mechanism, we can construct the most relevant training data and candidate values. For example, for the node *30* in code snippet, we know the type of the predicted terminal node: *num*. Then, more attention should be paid to the number in the value prediction, and the candidate value of the type *string* should not be considered. This known information can help us complete the code completion task through the classification mechanism.

Repetitive codes have obvious repetitive structures in the tree structure of AST, and these repetitive structural features are difficult to capture in the sequence model as a sequence code representation. Using AST to construct training graphs with tree structure data as input can clearly obtain repeated structures. Combined with the GGNN model, it has strong fitting performance for nonlinear structure data, and the model is more sensitive to repeated structures. According to the two *for* loop codes in the example, our model can easily predict the node *NameLoad*: *range* of the second *for* loop by using the structure information of the repeated AST.

3 | BACKGROUND

3.1 | Gated Graph Neural Network

The Gated Graph Neural Network (GGNN) is a classical spatial domain message passing model based on GRU. In order to extend the neural network method for processing graph structured data, Scarselli et al[14] proposed the vanilla graph neural network. Li et al[15] further introduced gated recurrent units and propose GGNN in 2015. Afterwards, Beck et al[16] proposed a new variant of the graph model. Formally, the GGNN model update functions are given as follows:

$$h_v^{(1)} = [x_v^T, 0]^T. \tag{1}$$

$$a_v^{(t)} = A_{v:}^T [h_1^{(t-1)T}, \ldots, h_{|v|}^{(t-1)T}]^T + b. \tag{2}$$

$$z_v^t = \sigma(W^z a_v^{(t)} + U^z h_v^{(t-1)}). \tag{3}$$

$$r_v^t = \sigma(W^r a_v^{(t)} + U^r h_v^{(t-1)}). \tag{4}$$

$$\widetilde{h_v^{(t)}} = tanh(W a_v^{(t)} + U(r_v^t \odot h_v^{(t-1)})). \tag{5}$$

$$h_v^{(t)} = (1 - z_v^t) \odot h_v^{(t-1)} + z_v^t \odot \widetilde{h_v^{(t)}}. \tag{6}$$

x_v^T is graph node annotations, which bring into hidden state to calculate the hidden layer vector of the first layer, and pads the rest with zeros in Equation (1). Each node in the graph accepts hidden state information from neighboring nodes and transmits similar information to neighboring nodes too. The model passes message between different nodes of the graph via incoming and outgoing edges, which dependents on the edge type and node direction. The incoming and outgoing connection matrix combined final connection matrix $A_{v:}^T$ corresponding to node v in Equation (2). z_v^t is the update gates, which determines the next step to update the hidden layer information. Equation (4) is reset gates and r_v^t controls reset node's hidden layer information. $a_v^{(t)} \in \mathbb{R}^{2D}$ contains activations from edges in both directions. The remaining are GRU-like updates that incorporate information from the other nodes and from the previous timestep to update each node's hidden state. Equations (3) and (4) are the update gates z and reset gates r, which σ is the sigmoid function $\sigma(x) = 1/(1 + e^{-x})$, and $\odot$ is element-wise multiplication in Equation (5). Finally, we get the final updated node status $h_v^{(t)}$ by Equation (6). Li et al[15] found this GRU-like propagation step is more effective then vanilla RNN-style in experiments.

The GGNN model can not only obtain the structural features of the abstract syntax tree but also use the GRU-like propagation step to increase the long-distance dependency of the node's information. So it is suitable for solving related problems of code semantic structure.

3.2 | AST graph

To learn from sources programs, we need to find a suitable representation that captures semantics features of ASTs. In previous work, a code representation method was to flatten each AST as a sequence of nodes in the in-order depth-first traversal.[6,16] The sequence model's code representation contains all the literal information of the source file. The most relevant node information of the predicted node contains in sequence, such as the parent and brother nodes. However, the code of sequence representation only keeps the order of nodes' features; it lacks the hierarchical information of the node and lost many nodes parent information. Another representation is to decompose the paths between nodes in the AST. An AST path is a path between nodes in the AST, starting from one node, ending in another node, and passing through an intermediate nonterminal in the path, which is a common ancestor of both terminals. However, these methods of code representation have some shortcomings, such as the loss of structure information, insensitivity to repeated structure and insufficient utilization of known information. Thus, we directly use the node's AST to construct the training data graph. Each node can be represented by an AST, which is constructed from previous nodes. We use these nodes' AST to construct the AST graph. Pairs of connected nodes are parent–child relationships that can retain structural characteristics in the AST. Besides, the AST graph will keep the parent node information between nodes. More formally:

> **Definition 1.** (AST graph): An AST is a tree structure consisting of k nodes:[$node_1$, $node_2$ … $node_{k+1}$]. The node $node_{k+1}$'s AST can be represented by the partial AST constructed by the previous k nodes. Through the classification of the $node_k$, all nodes' ASTs are divided into different sets: [set_1, … set_i … , set_t], for $i\in$ [2 … t], each set_i of the nodes' ASTs builds a training graph together. Every two adjacent nodes are parent–child or child–parent connection in each AST. Duplicated nodes in the ASTs are merged into one node in the AST graph.

In this paper, we have the similarities as the code representation in the previous related work:[17] (1) Keep the parent–child connection of each node and de-duplicate the nodes to build a training graph. (2) The weight of node's edge is the frequency of the occurrence of the edge in the corresponding AST.

We also have unique characteristics of code representation: (1) More information: We consider the directions between nodes, and the weights of edges between directions are also different. The direction of the node can effectively preserve the structural characteristics and hierarchical information. For example, an undirected graph can only determine the parent–child connection between two nodes but cannot determine which node is the parent node. (2) Classification mechanism: we combine the characteristics of predicting nodes with different parent nodes and divide the data into different training graphs. At the same time, we will also filter the candidate values of these graphs.

For example, the candidate values of graphs whose parent nodes are *num* and *string* are different. We will introduce the model in the Section 4. Figure 2 shows the process of converting nodes' AST to AST graphs. Figure 2A is the AST set with the parent node n_1. With the use of the connections between nodes and then removing duplicate nodes, we can get the training graph (Figure 2B) with the parent node of the predicted node n_1.

4 | PROPOSED MODEL

In this section, we introduce CC-GGNN model in detail. We formulate the problem at first, then explain how to construct the graph from node's AST. Meanwhile, we describe our approaches of representing learning node embedding on AST graphs, and we show how they are integrated into a embedding vector. As illustrated in Figure 3, CC-GGNN model includes two steps: (1) data processing and (2) graph representation and model prediction.

4.1 | Problem definition

The goal of code completion is to predict the next node in the source code. We get each node's AST to construct the train graph *G*. For prediction AST *P_ast*, we output probabilities $\hat{y}$ for all possible nodes, where an element value of vector $\hat{y}$ is the recommendation score of the corresponding node. The node with *Top-K* types T_i and values V_j in source code will be the candidate nodes for prediction.[6] Next node's type and value prediction process are transformed into the math problem of finding the maximum $\hat{y}$ in candidate nodes, as shown in the following Equations (7) and (8).

$$\exists i \in \{1,2,3,\ldots s\} : argmax_{T_i} \hat{y}(G, T_i, P\_ast). \tag{7}$$

$$\exists j \in \{1,2,3,\ldots k\} : argmax_{V_j} \hat{y}(G, V_j, P\_ast). \tag{8}$$

4.2 | Training graphs

We can know that every node will get a corresponding node's AST *p* in source code from Section 3.2. In the AST *p*, each node represents a code node $v_i \in V$, and these nodes are transformed into connected edges in the graph, that is, adjacent node (v_{i-1}, v_i) means edge $(v_{i-1}, v_i) \in E$ in graph. Then, these AST information will be jointly converted into a training AST graph *G*.

In Figure 3, according to the parent node n_1 of the test AST, we filter (data processing) the training data's AST to construct a training graph. The test data are not involved in the construction process of the training graph. The red nodes in the graph are just the reflection of the test data in the training graph. The operation of converting ASTs classification into multiple graphs can avoid the excessive number of nodes in one train AST graph and can reduce the impact of different AST on the accuracy of the predicted node. After we get the training graph, each node can be

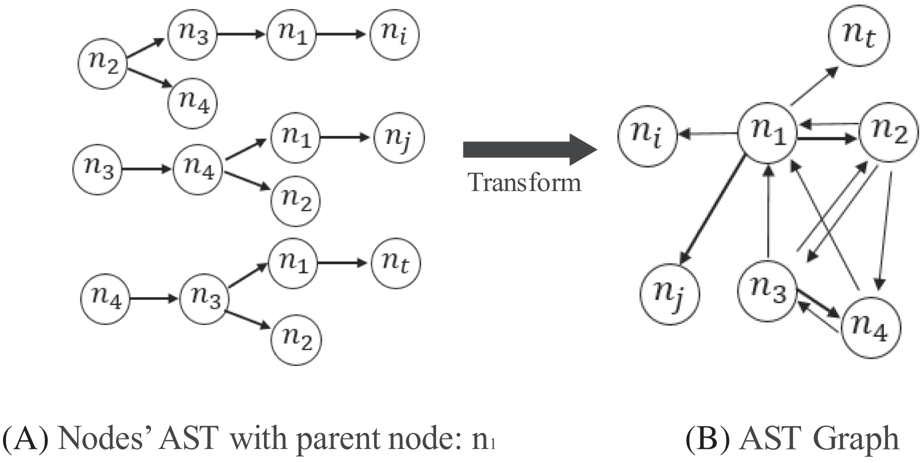


FIGURE 2 An example of Abstract Syntactic Tree (AST) graph

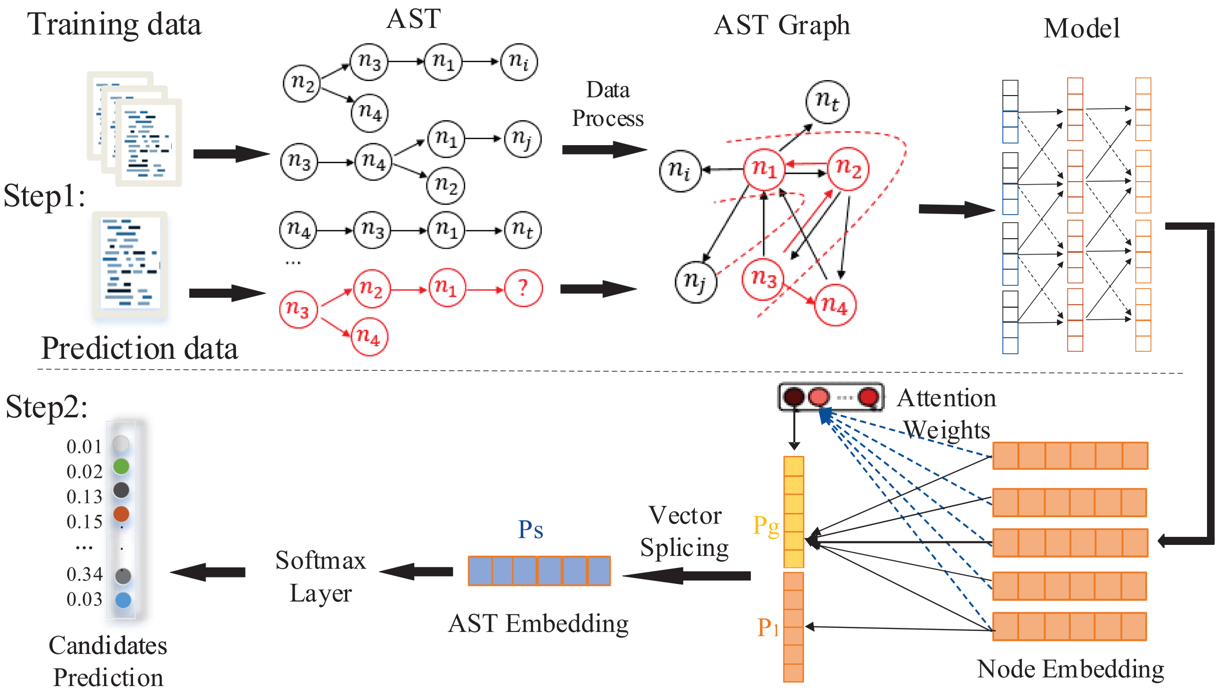
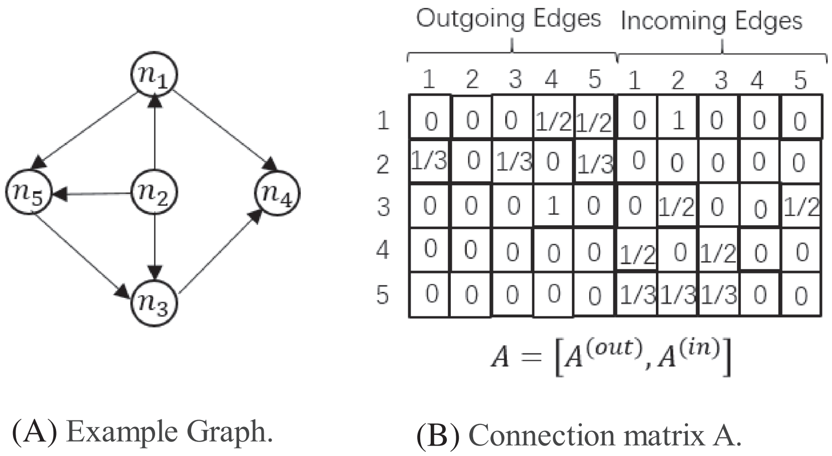


FIGURE 3 The overall architecture of the model



	Outgoing Edges					Incoming Edges				
	1	2	3	4	5	1	2	3	4	5
1	0	0	0	1/2	1/2	0	1	0	0	0
2	1/3	0	1/3	0	1/3	0	0	0	0	0
3	0	0	0	1	0	0	1/2	0	0	1/2
4	0	0	0	0	0	1/2	0	1/2	0	0
5	0	0	0	0	0	1/3	1/3	1/3	0	0

$$A = [A^{(out)}, A^{(in)}]$$

(A) Example Graph.

(B) Connection matrix A.

FIGURE 4 An example graph's connection matrix

embedded as a vector of fixed dimensions by the GGNN model: $v_i \in R^d$, where d is the dimensionality. Each p can be represented a vector based on those node vectors, and we will introduce it in detail in next section.

4.3 | AST node embedding

In this section, we will learn each AST node vector in the constructed graph. For example, the graph G and the matrix A are shown in Figure 4. In directed graph G, the A is defined as the concatenation of two adjacency matrices $A^{(out)}$ and $A^{(in)}$, which represents weighted connections of outgoing and incoming edges in the graph, respectively. The sparsity structure matrices correspond to the edges of the graph, and these matrices can help us calculate the vector of nodes.

For each graph G, the GGNN model gets node's vector. According to the graph, we rewrite Equation (2):

$$a_i^t = A_{i:}[v_1^{t-1},\ldots,v_n^{t-1}]^T N + b. \tag{9}$$

The $[v_1^{t-1},\ldots,v_n^{t-1}]$ is the list of node vectors in train graph, where $N \in R^{d \times 2d}$. In GGNN model, Equation (2) can propagate the information between different nodes by matrix A. Each node in the graph accepts hidden state information from neighboring nodes and transmits similar information to neighboring nodes too. z_v^t is the update gates, which determine hidden layer information update in next step. Equation (4) is reset gates, and r_v^t controls reset node's hidden layer information. Equations (3) and (4) will determine which node's information will be passed on. Using the GRU's update gate, the Equation (6) is the combination of the previous hidden state and the candidate state. After updating all nodes in AST graphs until convergence, we can obtain the final node vectors by Equation (10).

There are several types of outputs that GGNN model produce in different situations. In this task, we get each node's embedding vector by Equation (10).

$$o_v = g(h_v^T, x_v). \tag{10}$$

For each node $v \in V$ output node scores and applying a softmax over node scores. In Equation (10), g is a specific function, using the final state h_v^T and initial state x_v of each node to calculate its output. The final state h_v^T is derived from Equation (6).

4.4 | AST graph embedding and prediction

In our model, the predicted node is represented by its related nodes. It similar to the prior path[10-12] model and sequence-based method,[6] after we transform each node into a different vector, we convert the AST composed of nodes into a vector of fixed dimensions. However, the previous work is only a simple vector addition calculation, and it is hard to obtain effective features. Thus, we plan to apply a new strategy to combine long-term preference and node information.

We obtain the node of vectors in Section 4.3, then generate global vector together. The representation of each graph is an embedding vector $P_s \in R^d$, which can divide in two parts: (1) parent node embedding P_1 and (2) global AST embedding P_g.

In AST, the parent node contains the most abundant information of the predicted node, such as parent–child characteristics. Thus, the parent node embedding is important for the predicted value. For example, in the prediction AST in Figure 3 (red AST structure), the parent node is $n_1, P_1 = v_1$. Then, we aggregate all node vectors to get global AST embedding P_g in AST. The model obtains the priority of node information by adding an attention mechanism, which shows in Equation (11).

$$\alpha_i = W_1^T \sigma(W_2 v_1 + W_3 v_i). \tag{11}$$

And, the global AST embedding vector is calculated by parameter α_i, which is the attention coefficient for each node. The parameters W_1, W_2, and W_3 control the weights of node embedding vectors.

$$P_g = \sum\nolimits_{i=1}^{n} \alpha_i v_i. \tag{12}$$

We compute the embedding P_s by taking vector transformation over the vector splicing of the parent node vector P_1 and global AST embedding vector P_g:

$$P_1 = v_1, \tag{13}$$

$$P_s = W_4\left[v_1; \sum\nolimits_{i=1}^{n} \alpha_i v_i\right] = W_4[P_1; P_g], \tag{14}$$

where matrix W_4 control combined vectors into the latent space R^d. Finally, we get the embedding of each global representation P_s from the model. Then, we compute all prediction score $\hat{y}$ with each candidate node $v_i \in V_x$ for prediction, and the parameter x refers to the number of candidate values:

$$\hat{y} = softmax(P_s^T[v_1,\ldots,v_x]). \tag{15}$$

We get the output vector of the model by softmax function, which is normalization function in output layer of the neural network. The node corresponding to the largest value in the $\hat{y}$ vector is our model predicted value. In train process, the loss function is defined as the cross-entropy of the prediction and the ground truth:

$$Loss(\hat{y}) = -\sum_{i=1}^{x} y_i log(\hat{y}_i) + (1-y_i) log(1-\hat{y}_i). \tag{16}$$

In Equation (16), the y_i represents the predicted value of node v_i at the current stage of the model. In the prediction process, the train graph constructed by the classification mechanism can improve the effectiveness of screening more relevant candidate values.

5 | EXPERIMENT SETUP

This section is mainly divided into four parts. First, we introduce the data availability statement and data information. Then, we discuss the related baselines. Finally, we use MRR@5 and accuracy to evaluate our experiment.

5.1 | Data availability statement

The data that support the findings of this study are publicly available in Machine Learning for Programming at http://plml.ethz.ch. The data[18] consist of two parts: 150k Python Dataset and 150k JavaScript Dataset.

5.2 | Dataset and preprocessing

We evaluate different approaches on benchmarked datasets: Python (PY150) and JavaScript (JS150). PY150 is summerized in Table 1, and JS150 is summarized in Table 2.

As shown in Tables 1 and 2, it is overall dataset statistics. In the type prediction task, there are only 181 types in training data and test data in Table 3, such as *NameLoad*, *alias*, and *NameParam*. These types are determined by the Python programming language and cannot be defined by the programmer, which results in fewer candidate values. In the value prediction task, the source file has $3.4 * 10^6$ different node values in Table 4. 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 /, $-$, $*$). 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. Compared with the PY150 dataset, the structure of the JS150 dataset is simpler. The JS150 only contains 44 types of JavaScript language customization and $2.6 * 10^6$ different value candidates.

The number of unique node values in the datasets is too large to be directly applied to the neural network models, and thus, we only choose K most frequent values in training set to build the global vocabulary, where K is a free parameter. In order to effectively compare with the baseline experiment, we set the value of K to 1k, 10k, and 50k. We further add two special values: *EMPTY* being the value of non-terminal AST nodes and mark all Out-of-Vocabulary (OoV) node values in training set and test set as *UNK*. The number of nodes in the training graph is limited by the parameter K. When $K = 1$k, the number of nodes in the AST graph is the least, and there are only 1000 nodes at most. When $K = 50$k, the number of graph nodes is the largest.

TABLE 1 Dataset statistics for PY

	Category	Size
1	Training files	$1.0 * 10^5$
2	Test files	$5.0 * 10^4$
3	leaf nodes	$1.6 * 10^7$
4	Nonterminal nodes	$1.4 * 10^7$
5	Training queries	$6.2 * 10^7$
6	Test queries	$3.0 * 10^7$
7	Type vocabulary	181
8	Value vocabulary	$3.4 * 10^6$

TABLE 2 Dataset statistics for JS

	Category	Size
1	Training files	$1.0 * 10^5$
2	Test files	$5.0 * 10^4$
3	leaf nodes	$1.6 * 10^7$
4	Nonterminal nodes	$1.4 * 10^7$
5	Training queries	$10.7 * 10^7$
6	Test queries	$5.3 * 10^7$
7	Type vocabulary	44
8	Value vocabulary	$2.6 * 10^6$

TABLE 3 Type nodes type for PY

	Types	Size
1	NameLoad	$1.8 * 10^7$
2	attr	$9.2 * 10^6$
3	AttributeLoad	$8.4 * 10^6$
4	Str	$7.2 * 10^6$
5	Call	$6.9 * 10^6$
6	Assign	$3.9 * 10^6$
…	…	…
181	CompareLtELtELtE	1

TABLE 4 Value nodes type for PY

	Types	Size
1	self	$4.1 * 10^6$
2	0	$6.3 * 10^5$
3	None	$6.1 * 10^5$
4	1	$5.5 * 10^5$
5	True	$4.6 * 10^5$
6	name	$3.5 * 10^5$
…	…	…
$3.4 * 10^6$	http://github.com/lepture/flask-wtf	1

5.2.1 | Classification mechanism

This mechanism is a method of data processing. We obtain the known parent node of the predicted node as a classification node, then we divide the AST of each node into different sets. We convert each set into a training graph and combine the attention mechanism (Section 4.4) to predict the completion result. For example, if we predict the node (n_1 : ?) in Figure 2, the known information is that the parent node is n_1. Model obtains the training graph constructed by n_1 to predict the type.

We transform the most relevant AST into the one graph through the classification mechanism. And, we narrow down the candidate values of each graph by classification mechanism. For example, the candidate value ranges of *num* and *string* as the parent node are different. We traverse the training data to find the candidate value of each parent node and intersect the K candidate values; the remaining values are the new candidates. We can avoid meaningless calculations by exploiting this process.

Through experiments, we found that the types of terminal nodes and nonterminal nodes in the PY150 are divided into 153 graphs, of which there are 141 types of terminal nodes (Section 6.3), and value can be divided into 15 types. The types of terminal nodes and nonterminal nodes in the JS150 are divided into 44 graphs, of which there are 33 types of terminal nodes (Section 6.3), and value can be divided into 16 types.

5.3 | Baselines

As there are already prior investigations conducting code completion on the benchmarked datasets, to validate the effectiveness of our proposed approaches, we need to compare them against the state-of-the-art model. We compare CC-GGNN model with five code completion models from previous works and these baselines can be divided into two categories: (1) probabilistic model and (2) deep learning network.

Probabilistic Model DEEP3:[19] The approach is general and can be used to learn a probabilistic model of any programming language. This model implemented an approach in a system called DEEP3, and the authors evaluated it for the challenging task of learning probabilistic models of JavaScript and Python. The DEEP3 model achieves the highest prediction accuracy of the probabilistic models in code completion task.

VanillaLSTM:[6,20] The model uses LSTM to extract AST node features from the flattened AST sequences. The input of each LSTM unit is the concatenation of the previous hidden state and the type and value embedding of the current AST node. It is a standard LSTM network without any attention or pointer mechanisms.

Pointer Mixture Network:[6] The model is an attention and pointer generator network-based code completion method. This model is inspired by the prevalence of locally repeated terms in program source code and the recently proposed pointer copy mechanism; they propose a pointer mixture network for better predicting OoV words in code completion. Based on the context, the pointer mixture network learns to either generate a within-vocabulary word through an RNN component (Attentional LSTM model) or regenerate an OoV word from local context through a pointer component.

Transformer-XL:[13] The model is an improved transformer architecture[21] with the flattened AST sequence. In order to make effective use of the code structure, the model adds a path from the predicted node to the root, which is biggest difference with Pointer Mixture Network model in data process part. The naive multitask learning is used, and task weights are fixed to be equal.

CCAG:[17] CCAG model flattened sequence of AST as an AST graph. CCAG proposed AST Graph Attention Block to capture different dependencies in the AST graph for representation learning in code completion. The subtasks of code completion are optimized via multitask learning in CCAG, and the task balance is automatically achieved using uncertainty without the need to tune task weights.

For a fair comparison, we set the batch size as 128, hidden unit size of 1500 and train all model for 8 epochs. The initial learning rate is set as 0.001, and we decay it by multiplying 0.6 after every epoch in all methods. These configurations are same as a paper with high citation.[6] For VanillaLSTM and PointerMixtureNet, the gradient norm is clipped to 5, and the size of context windows is set to 50. For transformer based methods, we employ $h = 6$ parallel heads, and the dimension of each head d_{head} is set to 64. We set the segment length to 50 and use a 6-layer layer number Transformer-XL network. We reproduce the baselines: DEEP3, VanillaLSTM, PointerMixtureNet, and Transformer-XL. The experimental results are shown in Table 5. For the CCAG model, our model uses the same parameter configurations for comparison experiments. Embedding size, hidden size, and batch size are all set to 128. The experimental results are shown in the Table 6.

5.4 | Evaluation mertics

We evaluate the quality of code completion using two automatic metrics: Accuracy and MRR@5. In the code completion task, the previous model provides a *Top-1* list of suggestion for each node's type or value in the source code file. Those models use accuracy to evaluate the performance of our model. The calculation of accuracy is shown in Equation (17).

$$Acc = \frac{1}{n}\sum\nolimits_{i=1}^{n} acc_i. \tag{17}$$

For the *i*th predicted node, if *Top-1* prediction is ground truth, $acc_i = 1$, otherwise, $acc_i = 0$. However, in actual code completion tools, models often output more than one reference result. To measure performance on these tasks, we also use mean reciprocal rank (MRR@5). Comparing with the metric (Acc@1), this is closer to the realistic scenario when completion suggestions are presented to developers. The rank is defined as

$$MRR = \frac{1}{n}\sum\nolimits_{i=1}^{n} \frac{1}{rank_i}, \tag{18}$$

where n is the number of predicting locations and $rank_i$ is the rank of the correct label given by the model for the *ith* data point. We present MRR@5 as a percentage, in keeping with prior work.[2223]

6 | EVALUATION AND DISCUSSION

In this part, we will raise the related research questions at first. Second, we compare our model with the baselines on overall performance. And, we add experiment of terminal nodes prediction. Finally, we show each component effect on the model and discuss the effectiveness of the parameter.

6.1 | Research questions

To evaluate our proposed approach, we conduct experiments to investigate the following research questions:
RQ1: How does our proposed method perform in terms of code completion comparing with the state-of-the-art model?
We compared the state-of-the-art methods in probabilistic models and compared models with AST sequences and AST graphs as input in deep learning methods.
RQ2: For terminal nodes that are more complex and difficult to predict, how effective is our proposed model?
In previous work[6] for the completeness of prediction, the model artificially added value: *EMPTY* to nonterminal nodes in the process of data processing (Section 5.2). And the nonterminal node has $1.4 * 10^7$ (about 43%) in the dataset. In other words, about half of the value prediction tasks are *EMPTY*. Therefore, in order to further show the prediction effect of the model, we separately increase the terminal nodes' value prediction experiments.
RQ3: What is the effectiveness of code representation and classification mechanism for our proposed model?
In this part, we calculate the impact of each component for our proposed model. We experimentally study the impact of the semantic structure information of AST on the model.
RQ4: What is the effectiveness of the hidden size parameter and OoV problem on the experiment?
We analyze the experimental results of the hidden size values of 128, 256, 512, 1024 and 1500. Then, we discuss the influence of the OoV problem in our proposed method.

6.2 | Overall performance

For **RQ1**, Tables 5 and 6 illustrate the overall performance of our combined model compared with baselines. Our combined model obviously outperforms all baselines on accuracy in six datasets. Comparing with the purely AST sequence-based baselines, that is, Pointer Mixture Network and VanillaLSTM, our model achieves obvious improvements, indicating that it is much more capable for modeling ASTs. Compared with the Transformer-XL model, the improvement brought by CC-GGNN is more bigger in type prediction task. Our model is also better than CCAG model, which also takes graph data as input.

We first discuss the application of our model for predicting types of AST nodes, it is essentially a prediction of the code structure. As can be seen from the Table 5, for the node's type prediction, our model achieves the accuracy of 84.7% on the PY50k datasets, respectively, which

TABLE 5 Accuracy of various *types* and *values* for six datasets

	JS1k		JS10k		JS50k		PY1k		PY10k		PY50k	
	Value	Type	Value	Type	Value	Type	Value	Type	Value	Type	Value	Type
VanillaLSTM	62.6%	74.7%	66.3%	78.4%	69.8%	80.3%	59.2%	68.6%	61.3%	72.9%	63.4%	77.5%
PointerMixtureNet	69.8%	78.5%	73.4%	83.2%	75.3%	85.9%	64.7%	76.4%	68.3%	77.4%	69.2%	78.1%
Transformer-XL	**73.9%**	**84.1%**	**75.3%**	**85.7%**	**77.2%**	**87.0%**	**68.1%**	**78.9%**	**70.6%**	**80.7%**	**71.9%**	**82.4%**
CC-GGNN	**75.3%**	**87.1%**	**77.1%**	**88.4%**	**78.8%**	**90.2%**	**69.9%**	**80.3%**	**71.8%**	**83.2%**	**73.5%**	**84.7%**

TABLE 6 Accuracy of various *types* and *values* for six datasets with CCAG model

	JS1k		JS10k		JS50k		PY1k		PY10k		PY50k	
	Value	Type	Value	Type	Value	Type	Value	Type	Value	Type	Value	Type
CCAG	62.8%	75.7%	66.7%	78.6%	68.2%	80.1%	61.9%	76.7%	63.2%	80.9%	64.2%	75.3%
CC-GGNN	**64.1%**	**78.8%**	**67.9%**	**81.1%**	**70.5%**	**83.3%**	**63.3%**	**78.4%**	**65.7%**	**81.3%**	**66.9%**	**81.6%**

improves probabilistic model VanillaLSTM by 7.2%, Meanwhile, it also improves Pointer Mixture Network by 6.6% and improves Transformer-XL 2.3%. Transformer-XL model can obtain the hierarchical structure information of sequence data, and the experimental results are better than the Pointer Mixture Network model.

Compared with the CCAG model in type task, our model narrows the range of the candidate values and considers the directionality of the nodes. Therefore, the CC-GGNN model can capture more AST structure information, and the effect of the model is improved by 3.6% in the JS50 K dataset.

For the node's value prediction, our model achieves the accuracy of 73.5% in PY50k dataset. Compared with the Pointer Mixture Network, CCAG, and Transformer-XL models, our model has increased the performance by 4.3%, 2.7%, and 1.6% respectively. Through the classification mechanism of the model, we can effectively filter out the most relevant candidate values, which can effectively help us improve the accuracy of prediction. In the value prediction, the predictions of the *UNK* targets are treated as wrong predictions. When the parameter $K = 50\text{k}$, the *UNK* rates for Python is 11%.

6.3 | Terminal node prediction

In the Python dataset, there are $1.4 * 10^7$ nonterminal nodes form the intermediate structure of ASTs and $1.6 * 10^7$ terminal node form the leaf node. Nonterminal nodes have no value type in the original AST, so the most predicted value in the overall experiment is *EMPTY*, which is artificial definition. For **RQ2**, we add experiments to predict 16,003,628 leaf nodes in PY50k and predict 83,118,887 leaf nodes in JS50k. The difficulty of leaf nodes prediction mainly lies in the terminal Node's value prediction, and the difficulty can be divided into several points:

1. The leaf node has a wide range of candidate values. In our training data, the type of value is as much as $3.4 * 10^6$ in Python dataset, which is much larger than the range of type. However, most of them are values that have appeared several times, so we limit the candidates to calculate by the parameter *K* value. Values that are not within the *K* value range are marked as *UNK*.
2. The value of the leaf node appears random. Even if the range of candidates is selected by the parameter *K* value, the value of the terminal node still appears random. The programmers use diverse node values in leaf nodes to support the functionality of programs. If the relevant dataset is small, the prediction results are very poor.
3. The value of the leaf node has weak relationship with the structure of the AST. The same parent node *Nameload* has tens of thousands of candidate values, and the AST structure contains little valuable information for prediction. The prediction of value requires an overall understanding of the entire snippet code, and the dependent nodes may have a long distance from the predicted node.

Overall, Table 7 shows that our model achieves better scores than the probability model DEEP3 in PY50k and JS50k. Besides, our model achieves better scores than deep neural network models: PointerMixtureNet and Transformer-XL on both datasets. Our best performing model is 4.5% higher than the MRR@5 of PointerMixtureNet (from 44.8% to 49.3%) and is 2.9% higher than the MRR@5 indicator of the Transformer-XL model (from 46.4% to 49.3%) in the task of predicting the value of the leaf nodes..

Our model significantly improves the performance of type prediction. For PY50k dataset, the experimental result of MRR@5 is 91.8%. For JS50k dataset, the experimental result of MRR@5 is 93.6%. We further show the experimental results in the Tables 8 and 9.

For the type prediction of leaf nodes, our model divides all ASTs into 141 different training graphs by the known parent nodes of the leaf nodes. Each training graph contains different number of AST. The training graph constructed with *AttributeLoad* has 5003371 nodes' AST, which contains the largest number of nodes in all training graphs. Our approach has achieved high accuracy and MRR@5 scores. There are also training graphs with only one AST, such as *CompareNotEqNotEqNotEq* and *CompareIsIsIsIs*. The prediction accuracy and MRR@5 value of the model decline since the number of samples of training data is too small.

The details of experimental results of the model are shown in Tables 8 and 9. In the PY50 K dataset, the experimental results show that the model can achieve 100% prediction accuracy for 28 training graphs such as *ImportForm*. These nodes account for approximately 8.8% of the total,

TABLE 7 MRR@5 of various *types* and *values* of leaf token predictions for PY50k and JS50k

	PY50k		JS50k	
	Value	Type	Value	Type
DEEP3	42.1%	82.6%	49.7	86.5%
PointerMixtureNet	44.8%	85.2%	56.3	89.4%
Transformer-XL	46.4%	87.3%	58.6	90.3%
CC-GGNN	**49.3%**	**91.8%**	**59.1**	**93.6%**

TABLE 8 The *type* prediction result of the leaf node

Type	ACC	MRR@5	Type	ACC	MRR@5	Type	ACC	MRR@5
Module	81.04%	90.22%	CompareLtE	69.41%	83.21%	CompareLtGt	33.33%	66.67%
ImportFrom	100.00%	100.00%	CompareIn	80.06%	89.20%	AugAssignMod	54.22%	63.72%
alias	100.00%	100.00%	name	100.00%	100.00%	CompareGtGt	39.65%	60.13%
bases	99.99%	99.99%	CompareNotIn	80.31%	89.79%	CompareEqEqEqEq	54.15%	71.55%
AttributeLoad	99.36%	99.68%	CompareIsNot	98.14%	99.11%	CompareIs	90.12%	91.33%
body	94.71%	96.82%	finalbody	100.00%	100.00%	CompareLtLtEEqGtEGtNotEq	100.00%	100.00%
Assign	90.19%	95.30%	GeneratorExp	92.38%	94.54%	CompareLtLtEEqLtLtLt	100.00%	100.00%
Call	68.45%	73.31%	comprehension	99.88%	99.94%	CompareLtELtLtE	75.00%	87.50%
arguments	80.01%	87.62%	BinOpDiv	61.32%	72.65%	CompareEqIs	70.11%	73.49%
args	100.00%	100.00%	UnaryOpUSub	99.87%	99.87%	AugAssignBitXor	50.97%	67.33%
defaults	71.31%	79.05%	IfExp	72.26%	81.33%	CompareLtGtEqGtELtENotEqInNotInIsIsNot	0.00%	0.00%
Return	81.13%	86.62%	BinOpPow	72.62%	84.55%	CompareGtGtGt	100.00%	100.00%
decorator_list	100.00%	100.00%	ListComp	96.21%	97.41%	CompareGtGtE	58.31%	66.75%
TupleLoad	85.36%	93.18%	Print	74.16%	84.24%	AugAssignPow	33.33%	67.63%
Expr	99.90%	99.95%	Lambda	80.15%	89.73%	CompareGtEGtE	50.00%	75.00%
Import	100.00%	100.00%	Assert	61.21%	78.43%	AugAssignRShift	72.65%	84.33%
If	99.75%	99.87%	Set	77.32%	83.22%	CompareGtELt	0.00%	0.00%
SubscriptLoad	99.96%	99.98%	CompareLtELtE	63.18%	64.68%	AugAssignLShift	54.31%	72.13%
Slice	77.36%	83.15%	AugAssignMult	54.21%	67.14%	CompareLtEGtE	100.00%	100.00%
keyword	53.24%	71.45%	AugAssignBitAnd	71.91%	79.94%	CompareLtELtELtE	50.18%	63.54%
With	99.25%	99.62%	SetComp	100.00%	100.00%	CompareEqGt	50.00%	75.00%
Dict	89.25%	93.09%	AugAssignBitOr	72.08%	81.88%	CompareLtEq	100.00%	100.00%
Index	64.27%	78.88%	DictComp	93.21%	95.36%	CompareIsIsIs	100.00%	100.00%
For	99.93%	99.96%	Yield	79.10%	86.25%	CompareNotEqEq	100.00%	100.00%
TupleStore	100.00%	100.00%	Delete	100.00%	100.00%	CompareEqNotEqEq	80.00%	90.00%
UnaryOpNot	92.13%	94.90%	SubscriptDel	100.00%	100.00%	CompareGtEGt	64.37%	76.19%
Raise	89.35%	96.44%	CompareEqEq	43.82%	65.16%	CompareEqNotEq	33.33%	50.00%
compareEq	74.55%	85.85%	Global	100.00%	100.00%	TupleDel	100.00%	100.00%
AttributeStore	99.99%	99.99%	BinOpBitAnd	69.14%	78.31%	CompareEqEqEqEqEq	50.00%	75.00%
SubscriptStore	100.00%	100.00%	BinOpBitOr	84.31%	92.62%	CompareGtLt	52.13%	73.81%
orelse	50.42%	68.67%	ListStore	100.00%	100.00%	CompareGtEqGt	0.00%	0.00%
BoolOpAnd	90.22%	95.12%	Exec	70.34%	83.33%	CompareGtELtE	0.00%	0.00%
BinOpAdd	68.47%	80.62%	CompareLtLt	59.53%	73.36%	CompareLtEGt	25.00%	25.00%
BinOpMult	57.12%	78.31%	UnaryOpInvert	86.42%	89.34%	CompareLtEGtGt	0.00%	0.00%

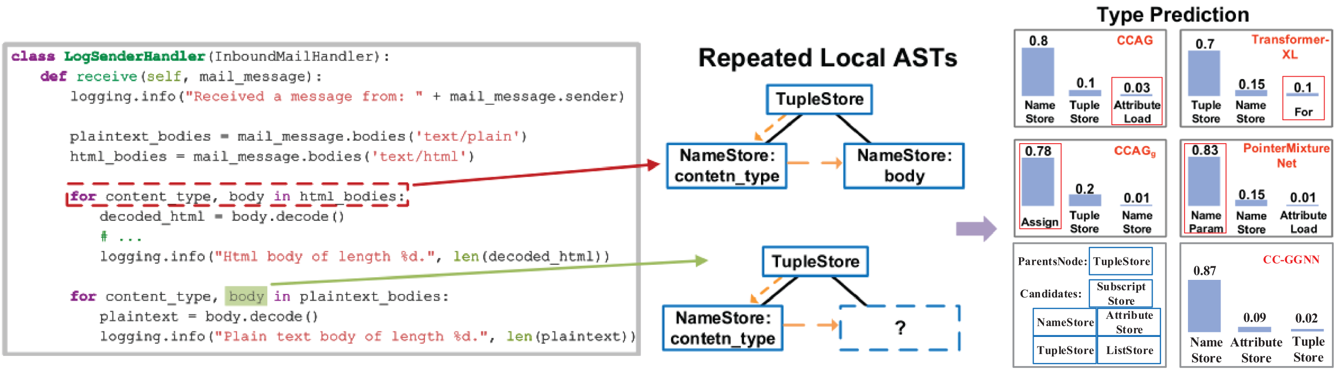
(Continues)

TABLE 8 (Continued)

Type	ACC	MRR@5	Type	ACC	MRR@5	Type	ACC	MRR@5
ListLoad	85.00%	87.43%	BinOpFloorDiv	64.38%	82.17%	CompareInIn	0.00%	0.00%
BinOpMod	91.07%	95.43%	AugAssignDiv	65.95%	80.10%	CompareInIs	0.00%	0.00%
CompareGt	81.77%	89.41%	BinOpRShift	60.58%	70.43%	CompareLtGtE	0.00%	0.00%
type	100.00%	100.00%	UnaryOpUAdd	66.42%	73.56%	CompareNotEqNotEq	100.00%	100.00%
While	82.55%	84.42%	Repr	100.00%	100.00%	CompareNotEqNotEqNotEqNotEq	100.00%	100.00%
CompareNotEq	63.02%	76.19%	BinOpLShift	69.63%	81.49%	CompareIsIsIsIs	100.00%	100.00%
AugAssignAdd	82.45%	90.74%	AugAssignFloorDiv	48.14%	70.11%	CompareLtLtLtLt	0.00%	0.00%
BoolOpOr	79.35%	83.46%	AttributeDel	89.31%	90.13%	CompareNotEqNotEqNotEq	0.00%	50.00%
BinOpSub	70.34%	78.55%	CompareLtLtE	64.52%	76.98%	CompareEqGtEq	100.00%	100.00%
CompareGtE	76.12%	87.35%	BinOpBitXor	80.31%	87.66%	CompareGtGtGtGt	0.00%	0.00%
CompareLt	73.01%	76.95%	CompareEqEqEqEqEqEq	50.00%	50.00%	CompareLtLtLt	0.00%	0.00%
AugAssignSub	70.49%	83.48%	CompareEqEqEq	75.00%	85.55%	CompareIsNotIsNot	0.00%	0.00%
CompareIs	99.39%	99.67%	CompareLtELt	39.67%	58.73%	CompareInNotIn	0.00%	0.00%

TABLE 9 The *type* prediction result of the leaf node

Type	ACC	MRR@5	Size	Type	ACC	MRR@5	Size
VariableDeclarator	72.85%	85.74%	$9.14 * 10^5$	CallExpression	85.64%	91.85%	$1.21 * 10^7$
FunctionDeclaration	99.99%	99.99%	$1.10 * 10^6$	AssignmentExpression	89.07%	93.01%	$4.56 * 10^6$
ArrayExpression	93.48%	95.73%	$4.64 * 10^6$	ExpressionStatement	95.16%	96.86%	$6.28 * 10^4$
MemberExpression	94.35%	97.34%	$3.53 * 10^7$	FunctionExpression	98.36%	99.65%	$3.06 * 10^6$
ArrayAccess	82.36%	87.64%	$3.96 * 10^6$	UnaryExpression	92.28%	93.62%	$8.24 * 10^5$
NewExpression	89.72%	92.45%	$7.52 * 10^5$	BinaryExpression	77.82%	82.43%	$7.04 * 10^6$
Property	76.55%	79.40%	$3.58 * 10^6$	IfStatement	86.16%	88.47%	$4.30 * 10^5$
ConditionalExpression	81.01%	86.18%	$7.09 * 10^5$	LogicalExpression	67.87%	72.73%	$7.74 * 10^5$
ReturnStatement	78.65%	83.84%	$7.80 * 10^5$	VariableDeclaration	100.00%	100.00%	$1.01 * 10^6$
BlockStatement	82.37%	87.55%	$3.94 * 10^5$	UpdateExpression	100.00%	100.00%	$4.08 * 10^5$
ForInStatement	99.78%	99.81%	$1.18 * 10^5$	ForStatement	91.14%	93.32%	$7.46 * 10^4$
WhileStatement	76.25%	81.41%	$1.56 * 10^4$	CatchClause	95.89%	97.10%	$7.87 * 10^4$
TryStatement	99.39%	99.60%	$6.09 * 10^4$	SwitchStatement	98.64%	99.17%	$2.00 * 10^4$
SwitchCase	84.89%	88.10%	$3.59 * 10^5$	ThrowStatement	95.02%	96.68%	$1.98 * 10^4$
DoWhileStatement	65.64%	69.41%	2914	Program	99.52%	99.56%	7784
SequenceExpression	89.02%	92.47%	$3.44 * 10^4$	LabeledStatement	50.00%	75.00%	8
AssignmentPattern	44.44%	74.07%	9				



```
class LogSenderHandler(InboundMailHandler):
    def receive(self, mail_message):
        logging.info("Received a message from: " + mail_message.sender)

        plaintext_bodies = mail_message.bodies('text/plain')
        html_bodies = mail_message.bodies('text/html')

        for content_type, body in html_bodies:
            decoded_html = body.decode()
            # ...
            logging.info("Html body of length %d.", len(decoded_html))

        for content_type, body in plaintext_bodies:
            plaintext = body.decode()
            logging.info("Plain text body of length %d.", len(plaintext))
```

FIGURE 5 A code completion example

and they are marked as gray in the table. However, there are also 13 training graphs constructed by parent nodes whose prediction accuracy is zero. We find that these training data are less than a hundred ASTs through analysis. Most of them appeared only once. In other words, the training data rarely leads to a low prediction accuracy.

Compared with the PY50k dataset, JS50 K is relatively simple. Although JS50 K has more leaf nodes, there are fewer types and value categories. There are only 33 types of parent nodes of leaf nodes, which greatly reduces the difficulty of prediction. It can be seen from Table 9 that the *VariableDeclaration* and *UpdateExpression* training graphs of the parent node marked in gray can achieve a prediction accuracy of 100%, accounting for 1.71% of all leaf nodes.

6.3.1 | Case study

Figure 5 depicts a case in CCAG, which is a test file in Python dataset. The code segment is used to illustrate how the incoming email is handled in Google Cloud. *NameStore:body* highlighted with green is the node to predict.

First, it is easily find that the parent node of the prediction node is *TupleStore*, after we convert the code file into AST. Combining our classification mechanism and traversing the training dataset, we find that there are only five candidates whose parent node is *TupleStore*. The candidates are *[NameStore, AttributeStore, TupleStore, ListStore, SubscriptStore]*.

However, all results from Figure 5 contain values that should not be considered, which are marked by the red boxes. CCAG$_g$ and PointerMixtureNet models even predict that the maximum probability value is a noncandidate value. Therefore, a large and invalid candidate value table not only wastes computing power and time but also reduces the accuracy of the model. Transformer-XL mainly gives high probability to the node type on the path from the predicted node to the root in the AST. Similarly, our model successfully captures the repetitive pattern (the repeated local ASTs) and has a higher probability than the CCAG model. The two major reasons of our model that lead to better performance are as follows: (1) The GGNN model has a strong nonlinear fitting ability to graph structure data. Besides, the CC-GGNN model can learn the inherent characteristics of the directions and edges between nodes in the graph structure, which is helpful for accurate prediction of repeated structure data. (2) In the GGNN model, the method of information transmission between nodes is that each node spreads to surrounding nodes, and the information transmitted by the same graph structure data is the same. These advantages led to the success of CC-GGNN captures the repetitive pattern. Another point is that for the MRR@5 metric, our model can always display the predicted value correctly.

6.4 | Each component effect on the model

For **RQ3**, we conducted an ablation study to examine the impact of the two proposed components used in our model: classification mechanism and different AST representations. We conduct experiments without classification mechanism and use the AST representation of the CCAG and Transformer-XL models, respectively.

For type prediction task, the results are shown in Figure 6. The blue line shows the results of our full model. The yellow line shows the code representation of our model combined with Transformer-XL model's code representation. The green line shows our model with CCAG model's undirected graph representation settings. Moreover, the red line removes the classification mechanism from the full model. It is obvious that full model is better than other three models. It can be seen that the effect of using the graph model in combination with other AST representations is weaker than the full model. Besides, the model effect of removing the classification mechanism decreases more. The variation in performance demonstrates that both the classification mechanism and our AST representations are necessary to improve the performance, and classification mechanism contributes more to the improvements.

For value prediction task, the results are shown in Figure 7. The blue line shows the results of our full model. The yellow and green lines present the results of our model compared with other AST representation, and the red line removes the classification mechanism from the full model. Similar to the experimental results of type prediction, these models have worse results with no classification mechanism and other AST representation.

For **RQ3**, we draw conclusions from the above experimental results. Both classification mechanism and structure characteristics of AST graph representation can effectively improve the predictive ability of the model. The classification mechanism has the most obvious effect on the overall model, and it is effective to increase the attention mechanism to the parent node of the predicted node.

6.5 | Hidden size and OoV problem

For **RQ4**, we take the PY50k dataset as an example. First, we analyze impact on the model by adjusting the range of hidden size. Then, we discussed the impact of the OoV problem on the value prediction task.

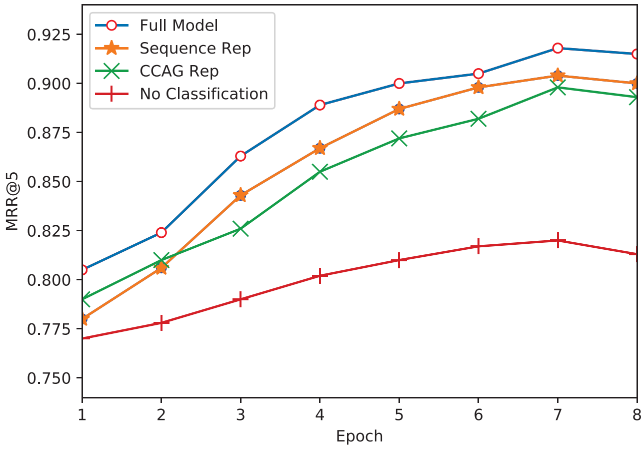


FIGURE 6 Effectiveness of each component in *type* prediction task

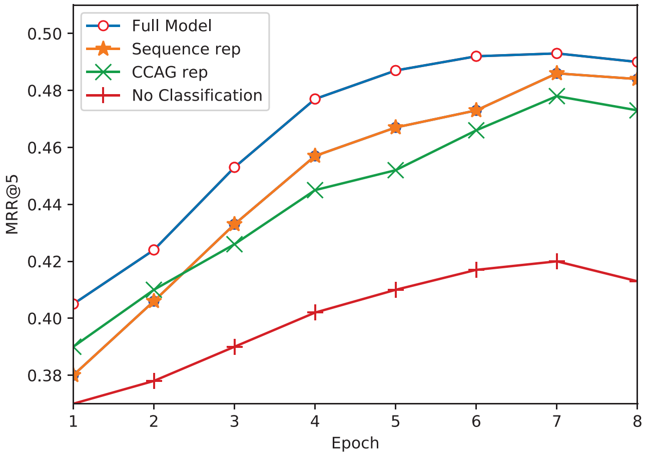


FIGURE 7 Effectiveness of each component in *value* prediction task

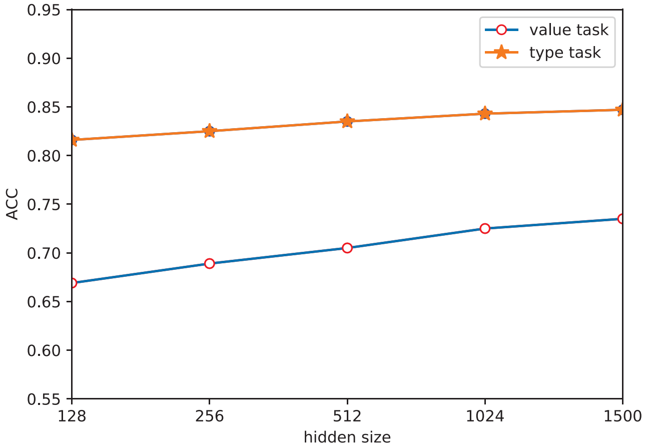


FIGURE 8 The impact of the hidden sizes in our model

As shown in Figure 8, with the increase of the hidden size parameter, the accuracy of the value and type prediction tasks are improved. The orange line shows that the change of hidden size has no obvious effect on the prediction accuracy of type. Because in the CC-GGNN model, there may be only dozens of candidate values in the type prediction task, and the change of hidden size has little effect on the type in the model. However, the trend of value prediction task changes is more significant from the blue line. The reason is that the number of value is more than the type in the datasets, and thus, it is more sensitive to hidden size.

The influence of the OoV problem are as follows: (1) The OoV problem will affect the upper limit of the model's prediction accuracy. For the six datasets we use, the OoV rates of JS1k, JS10k, JS50k, PY1k, PY10k, and PY50k are 20%, 11%, 7%, 24%, 16%, and 11%. This problem leads to the upper limit of the prediction accuracy of our CC-GGNN model of 80%, 89%, 93%, 76%, 84%, and 89%. (2) Since we define OoV words as *UNK* participating in the training of the model, even if we successfully predict *UNK* node, we think it is wrong prediction. Therefore, some nodes are predicted to be *UNK*, which will reduce the accuracy of model prediction.

6.6 | Training cost analysis

In order to compare the training time of the model, we selected the pointer network and the Transformer-XL model as comparators. With the help of the Transformer-XL model, the representation of each input of each segment depends on the self-attention layer calculation, and repetition only occurs between segments. Therefore, it allows more parallelization and requires less time to train. In Pointer Mixture Network, they adopt LSTM as the language model, where most of the recurrent computations are performed during the hidden states' updating process. Similar to the pointer network, our model also spend a lot of time in the vector transfer of the hidden layer, so the training time is longer than the Transformer-XL model, but our classification mechanism reduces the range of candidate values and reduces the training time. The experimental results are shown in Table 10, Transformer-XL model spends 78% of the time compared to our model. Compared with the pointer network model, our training time is shortened by about 10%.

7 | THREATS TO VALIDITY

Threats to external validity relate to the quality of the datasets we use. Take the Python dataset as an example. All the data come from more than 3000 projects. There are 150,000 training data and test data, and thus, there are multiple data sources for one project. Moreover, the code styles of the programmers of this project have a greater impact on the training of the model. Python and JavaScript are two benchmark datasets that have been used in previous code completion work.[6,13] All programs in the dataset are from GitHub repositories. Therefore, further research is needed to verify our findings and extend them to other programming languages. For example, the Java dataset.[13]

Threats to internal validity includes the influence of classification mechanism on the number of training graph. The performance of our model would be affected by the different graph. For example, in Section 6.3, node has an accuracy of 0% because of there are too few relevant samples to construct an effective training graph.

Threats to construct validity relate to the suitability of our evaluation measure. We use accuracy and MRR@5 as the metrics that evaluate the proportion of correctly predicted next token. Those metrics are classical evaluation measures for code completion, and they are used in almost all the previous code completion work. Recently, the MRR metric begin to be used in code completion work.[2223]

8 | RELATED WORK

8.1 | Code completion

The n-gram language models[24-27] are the early machine learning models for code completion. These probabilistic models can predict the next code token effectively. An n-gram language model computes the probability of the next token given previous *n* code tokens as model input. These models only use the most primitive tokens representation of the code and do not consider effective semantic structure features. Therefore, in subsequent research, the researchers used the abstract syntax trees representation code and bring its into the probability model to improve the accuracy of the prediction. These models include probabilistic context-free grammars[28] and probabilistic higher-order grammars.[4] This type of model uses context-free grammar to convert the source code into an AST, and it predicts next token based on the information selectively collected across paths in AST. The AST path contains rich structural and semantic features, which is the key to improve the prediction accuracy of the next node in probabilistic grammar models. Bruch et al[29] 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. Subsequently, Alon et al[11] used the classic model of decision tree model combined with the path information of AST to achieve the highest prediction accuracy of this line of works. Their model learns a decision tree model that uses this information essentially as features. Our work is significantly better than DEEP3.

With the development of deep learning in natural language processing tasks, many researchers believe that code is also a natural language. Deep learning is able to capture longer dependency within a sequence or node's path. RNN is the most common neural technique for code prediction. Moreover, their variants feed input code as a linear AST sequence. In this way, the RNN essentially learns a language model over the training corpus, and it is generally believed to do so better than n-grams[22] model. Indeed, several papers in the literature, including work done in industrial contexts,[20] have used RNNs. Liu et al[30] attempt has been made on feeding RNN with serialized ASTs. The accuracy in their further research is improved by using more AST guided architectures.[6] In addition to the above models, there are other different types of code completion models. For example, use the data after the prediction node to make predictions.[31-34]

Basides, many pretrained language models are proposed for modeling programs. CodeBERT[35] captures general-purpose code representations that support downstream programming language and natural language applications such as codesearch and code documentation generation tasks. However, CodeBERT model did not consider the inherent structure feature of code. Therefore, Daya et al[36] added the data flow in the pretraining

TABLE 10 Training cost analysis in the Python dataset

Model	# of Parameters	Training time (h/epoch)
PointerMixtureNet	162.6 M	9.4
Transformer-XL	98.9 M	6.7
CC-GGNN	147.5 M	8.5

stage, which is a semantic-level structure of code that encodes the relation of variables. The semantic-level structure feature significantly improves the prediction accuracy in multiple code tasks, that is, code generation and code completion. Wasi Uddin et al[37] introduce PLBART, which is able to perform a wide range of programs and language understanding tasks. PLBART model had pretrained a large number of code functions and related natural language texts. Different from the above pretrained models, the CodeT5[38] model can better utilize the code semantics conveyed by the identifier assigned from the developer. And, CodeT5 uses a unified framework to seamlessly support code understanding tasks and allows multitask learning. Although the existing pretrained model can be fine-tuned for the downstream tasks of BERT with less labeled data and training budget, it still has two problems: (1) The model has not yet tried to obtain high-quality contextual embedding of the source code. (2) It cannot be evaluated on multiple program understanding tasks simultaneously. To solve these problems, Aditya et al[39] create a massive, deduplicated corpus of 7.4 M Python files to train the model. Then, these comprise multiple code tasks with the previous works. Inspired by these pretrained models, Liu et al[40] proposed a multitask learning-based pretrained language model for code completion with a transformer-based neural architecture. In addition, the IntelliCode[41] has transformed the model into an actual code completion tool. The IntelliCode model can use the tree-based cache on the client side, the efficient parallel implementation of the beam search decoder, and the optimization of the calculation graph to meet the requirements of VS Code IDE.

8.2 | Graph embedding

Graph models are widely used in graph structure node quantization processing. The graph embedding obtains the link conversion sequence of the nodes and edges in the structure and then embeds the sequence into a fixed-dimensional vector through the graph model. Perozzi et al[42] proposed DeepWalk model to process graph problem, which used a random walk algorithm. However, the random walk algorithm extracts random path sequences, and no attention is paid to the depth-first search (DFS) and breadth-first search (BFS) paths between fusion nodes. To solve this problem, Grover et al[43] proposed Node2vec model, which used 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 in modeling the dependency relationship between graph nodes has made the research field related to graph analysis achieve good results. Graph models can also be combined with depth models to predict tasks. Structural deep network embedding (SDNE)[44] recommends using a deep autoencoder to maintain the proximity of the first-order and second-order networks. It achieves this goal by jointly optimizing these two approximations. The graph convolutional network (GCN)[45-47] model aggregates the neighborhood embedding representations of nodes through iteration, and it uses the previous iteration embedding combined with the embedding function to obtain a new embedding. Only the aggregation embedding of the local neighborhood makes it scalable, and iteratively learns to embed a node to describe the global neighborhood. SR-GNN[48] is graph embedding for session-based recommendation, which proposed a graph embedding method that combines global embedding and local embedding. GGNN[14,15] is a classical spatial domain message passing model based on GRU,[49,50] which can solve the problem of gradient in sequence long-term memory and back propagation. The CCGGNN model graph embedding part is inspired by the above models.

Allamanis et al[31] use graph model to represent the structure and semantic structure information of the source code, and they found that graph neural networks have better performance than convolutional neural networks in variable completion and variable misuse tasks. The research of Rahman et al[51] pointed out that the graph model has a better predictive effect than the n-gram model. However, there lacks a work that compares the performance of the existing DNN models, and it is recommended to further study the statistical code graph model to accurately capture more complex coding models. Fernandes et al[52] design a hybrid sequence-graph model to embed code nodes, and they develop a framework to extend a graph component that find long-distance relationships in weakly structured data such as text.

9 | CONCLUSION AND FUTURE WORKS

In this work, we propose a model to solve the problems of the code completion. To better represent AST nodes, we classify node's ASTs by known parent node and transform them into training directed graph. Finally, we combine powerful GGNN model with the ability to integrate structural information to predict the next node. Compared with previous work, the experimental results show that our model can effectively improve the results of code completion. Notably, for the *type* prediction of leaf nodes, our model has a significant improvement effect.

In future work, we will apply our model to more programming languages, such as Java. Because Python grammar has more casual style, the Python dataset is one of the most challenging prediction tasks.

Secondly, our model cannot solve the OoV problem. It is a great challenge for predicting the rare words artificially defined by programmers in source code. In previous work, researchers used the Pointer Mixture Network and copying mechanisms[6] to solve OoV problem. Besides, BPE algorithm[22] is widely used in recent research for addressing the OoV problem. We will try these strategies in future works to improve our model.

Finally, we will try to capture more code information, such as using Control Flow Graph (CFG).[53,54] CFG contains a lot of call relationships between nodes, which may help us improve the prediction accuracy of value prediction task.

ACKNOWLEDGMENTS

This work is partially supported by the National Natural Science Foundation of China under grants No. 61772200 and the project supported by Natural Science Foundation of Shanghai No. 21ZR1416300.

CONFLICT OF INTEREST

The authors declare that they have no conflicts of interest.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in Machine Learning for Programming at http://plml.ethz.ch. The data consists of two parts: 150k Python Dataset and 150k JavaScript Dataset.

ORCID

Kang Yang https://orcid.org/0000-0002-5863-1003
Huiqun Yu https://orcid.org/0000-0002-1899-1135
Guisheng Fan https://orcid.org/0000-0002-2702-0242
Xingguang Yang https://orcid.org/0000-0001-7489-2265
Zijie Huang https://orcid.org/0000-0002-8911-9889

REFERENCES

1. Hindle A, Barr ET, Gabel M, Su Z, Devanbu PT. On the naturalness of software. 34th International Conference on Software Engineering, Zurich, Switzerland; 2016:122-131.
2. Hellendoorn VJ & Devanbu PT Are deep neural networks the best choice for modeling source code? In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering; 2017:763-773.
3. Tu Z, Su Z, Devanbu PT. On the localness of software. In: Proceedings of the 22nd International Symposium on Foundations of Software Engineering; 2014:269-280.
4. Bielik P, Raychev V, Vechev MT. PHOG: Probabilistic model for code. In: Proceedings of the 33nd International Conference on Machine Learning; 2016:2933-2942.
5. Bhoopchand A, Rocktäschel T, Barr ET, Riedel S. Learning Python code suggestion with a sparse pointer network. In: CoRR; 2016. http://arxiv.org/abs/1611.08307
6. Li J, Wang Y, Lyu MR, King I. Code completion with neural attention and pointer networks. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence; 2018:4159-4165.
7. Malhotra P, Vig L, Shroff G, Agarwal P. Long short term memory networks for anomaly detection in time series. In: 23rd European Symposium on Artificial Neural Networks; 2015.
8. Allamanis M, Sutton C. Mining source code repositories at massive scale using language modeling. In: Proceedings of the 10th Working Conference on Mining Software Repositories; 2013:207-216.
9. Nguyen TT, Nguyen AT, Nguyen HA, Nguyen TN. A statistical semantic language model for source code. In: oint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering; 2013:532-542.
10. Alon U, Zilberstein M, Levy O, Yahav E. A general path-based representation for predicting program properties. In: Proceedings of the 39th Conference on Programming Language Design and Implementation; 2018:404-419.
11. Alon U, Brody S, Levy O, Yahav E. Code2seq: generating sequences from structured representations of code. In: 7th International Conference on Learning Representations; 2019.
12. Alon U, Zilberstein M, Levy O, Yahav E. Code2vec: learning distributed representations of code. In: Proc. ACM Program. Lang, Vol. 3(POPL); 2019:1-40:29.
13. Fang L, Ge Li, Bolin W, Xin X, Zhiyi F, Zhi J. A self-attentional neural architecture for code completion with multi-task learning. In: Proceedings of the 28th International Conference on Program Comprehension; 2020:37-47.
14. Scarselli F, Gori M, Tsoi AC, Hagenbuchner M, Monfardini G. The graph neural network model. *IEEE Trans Neural Netw*. 2009;20(1):61-80.
15. Li Y, Tarlow D, Brockschmidt M, Zemel RS. Gated graph sequence neural networks. In: 4th International Conference on Learning Representations; 2016.
16. Beck D, Haffari G, Cohn T. Graph-to-sequence learning using gated graph neural networks. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics; 2018:273-283.
17. Yanlin W, Hui L. Code completion by modeling flattened abstract syntax trees as graphs. In: Thirty-Fifth AAAI Conference on Artificial Intelligence, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, The Eleventh Symposium on Educational Advances in Artificial Intelligence; 2021:14,015-14,023.
18. [dataset]Raychev Veselin, Vechev Martin, Krause Andreas. 150k Python Dataset and 150k JavaScript Dataset; http://plml.ethz.ch; https://doi.org/10.1145/2676726.2677009; 2015.
19. Raychev V, Bielik P, Vechev MT. Probabilistic model for code with decision trees. In: Proceedings of the 2016 ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications; 2016:731-747.
20. Svyatkovskiy A, Zhao Y, Fu S, Sundaresan N. Pythia: AI-assisted code completion system. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining; 2019:2727-2735.

21. Zihang D, Zhilin Y, Yiming Y, Jaime G, Le QV, Salakhutdinov R. Transformer-XL: attentive language models beyond a fixed-length context. In: Proceedings of the 57th Conference of the Association for Computational Linguistics; 2019:2978-2988.
22. Karampatsis R, Babii H, Robbes R, Sutton C, Janes A. Big code != big vocabulary: open-vocabulary models for source code. In: Proceedings of the 42nd International Conference on Software Engineering; 2020:1073-1085.
23. Seohyun K, Jinman Z, Yuchi T, Satish C. Code prediction by feeding trees to transformers. In: Proceedings of the 43rd International Conference on Software Engineering; 2021:150-162.
24. Nguyen AT, Hilton M, Codoban M, et al. API code recommendation using statistical learning from fine-grained changes. In: Proceedings of the 24th International Symposium on Foundations of Software Engineering; 2016:511-522.
25. Roos P. Fast and Precise Statistical Code Completion. In: Proceedings of the 37th IEEE/ACM International Conference on Software Engineering; 2015:757-759.
26. Franks C, Tu Z, Devanbu PT, Hellendoorn V. CACHECA: a cache language model based code suggestion tool. In: Proceedings of the 37th IEEE/ACM International Conference on Software Engineering, 2015, Volume 2IEEE Computer Society; 2015:705-708.
27. Raychev V, Vechev MT, Y ahav E. Code completion with statistical language models. In: ACM SIGPLAN Conference on Programming Language Design and Implementation; 2014:419-428.
28. Allamanis M, Sutton C. Mining idioms from source code. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering; 2014:472-483.
29. 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: 213-222.
30. Liu C, Wang X, Shin R & Gonzalez JE aDS. Neural code completion. https://openreview.net/forum?id=rJbPBt9lg; 2016.
31. Allamanis M, Brockschmidt M, Khademi M. Learning to represent programs with graphs. In: Proceedings of the 6th International Conference on Learning Representations; 2018.
32. Alon U, Sadaka R, Levy O, Yahav E. Structural Language Models for Any-Code Generation. http://arxiv.org/abs/1910.00577
33. Brockschmidt M, Allamanis M, Gaunt AL, Polozov O. Generative Code Modeling with Graphs. In: Proceedings of the 7th International Conference on Learning Representations; 2019.
34. Nguyen AT, Nguyen TN. Graph-based statistical language model for code. In: Proceedings of the 37th IEEE/ACM International Conference on Software Engineering; 2015:858-868.
35. Zhangyin F, Daya G, Duyu T. CodeBERT: a pre-trained model for programming and natural languages. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing:1536-1547.
36. Daya G, Shuo R, Shuai L, Zhangyin F. GraphCodeBERT: pre-training code representations with data flow. In: 9th International Conference on Learning Representations; 2021:1536-1547.
37. Wasi Uddin A, Saikat C, Baishakhi R, KaiWei C. Unified pre-training for program understanding and generation. In: Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies; 2021:2021.
38. Yue W, Weishi W Shafiq RJ, Steven CHH. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing; 2021.
39. Aditya K, Petros M, Gogul B, Kensen S. Learning and evaluating contextual embedding of source code. In: Proceedings of the 37th International Conference on Machine Learning; 2020:5110-5121.
40. Liu F, Li G, Zhao Y, Jin Z. Multi-task learning based pre-trained language model for code completion. In: 35th International Conference on Automated Software Engineering; 2020:473-485.
41. Alexey S, Shao Kun D, Shengyu F, Neel S. IntelliCode compose: code generation using transformer. In: 28th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering; 2020:1433-1443.
42. Perozzi B, Al-Rfou R, Skiena S. DeepWalk: online learning of social representations. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining; 2014:701-710.
43. 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; 2016:855-864.
44. 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; 2016:1225-1234.
45. Bruna J, Zaremba W, Szlam A, LeCun Y. Spectral networks and locally connected networks on graphs. In: Proceedings of the 2nd International Conference on Learning Representations; 2014.
46. Defferrard M, Bresson X, Vandergheynst P. Convolutional neural networks on graphs with fast localized spectral filtering. In: Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016; 2016:3837-3845.
47. Kipf TN, Welling M. Semi-supervised classification with graph convolutional networks. In: Proceedings of the 5th International Conference on Learning Representations; 2017.
48. Yamaguchi F, Golde N, Arp D, Rieck K. Session-based recommendation with graph neural networks. In: The Thirty-Third Conference on Artificial Intelligence, 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence; 2019:346-353.
49. Chung J, Gülçehre C, Cho K, Bengio Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. In: CoRR; 2014. http://arxiv.org/abs/1412.3555
50. Cho K, Merrienboer vB, Gülçehre C, Bahdanau D, Bougares F, Schwenk H, Bengio Y. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing; 2014:1724-1734.
51. Rahman M, Palani D, Rigby PC. Natural software revisited. In: Proceedings of the 41st International Conference on Software Engineering; 2019: 37-48.
52. Fernandes P, Allamanis M, Brockschmidt M. Structured neural summarization. In: Proceedings of the 7th International Conference on Learning Representations; 2019.

53. Zhou Y, Liu S, Siow JK, Du X, Liu Y. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019; 2019:10,197-10,207.
54. Yamaguchi F, Golde N, Arp D, Rieck K. Modeling and discovering vulnerabilities with code property graphs. In: IEEE Computer Society; 2014: 590-604.

How to cite this article: Yang K, Yu H, Fan G, Yang X, Huang Z. A graph sequence neural architecture for code completion with semantic structure features. *J Softw Evol Proc*. 2022;34(1):e2414. doi:10.1002/smr.2414