

# Deep Learning Based Code Completion Models for Programming Codes

Shuai Wang<sup>i</sup>

Academy for Advanced  
Interdisciplinary Studies  
Peking University  
Beijing, China

wangshuai\_2016@pku.edu.cn

Jinyang Liu<sup>i</sup>

Academy for Advanced  
Interdisciplinary Studies  
Peking University  
Beijing, China

jyliu.aais@pku.edu.cn

Ye Qiu

School of Electronics Engineering &  
Computer Science  
Peking University  
Beijing, China

qiuye2014@pku.edu.cn

Zhiyi Ma<sup>\*</sup>

Computer Department, Key  
Laboratory of High Confidence  
Software Technologies  
Peking University  
Beijing, China

mazhiyi@pku.edu.cn

Junfei Liu

School of Software and  
Microelectronics  
Peking University  
Beijing,  
China

liujunfei@pku.edu.cn

Zhonghai Wu

School of Software and  
Microelectronics  
Peking University  
Beijing,  
China

wuzh@ss.pku.edu.cn

## ABSTRACT

With the fast development of Information Technology, program software and mobile applications have been widely used in the world, and are playing important roles in human's daily life. Thus, writing programming codes has been important work in many fields. However, it is a hard and time-cost task which presents a great amount of workload to programmers. To make programmers' work easier, intelligent code completion models have been a popular research topic in recent years. This paper designs Deep Learning based models to automatically complete programming codes, which are LSTM-based neural networks, and are combined with several techniques such as Word Embedding models in NLP (Natural Language Processing), and Multihead Attention Mechanism. Moreover, in the models, this paper raises a new algorithm of generating input sequences from partial AST (Abstract Syntax Tree) that have most relevance with nodes to be predicted which is named as RZT (Reverse Zig-zag Traverse) Algorithm, and is the first work of applying Multihead Attention Block into this task. This paper makes insight into codes of several different programming languages, and the models this paper presents show good performances in accuracy comparing with the state-of-art models.

## CCS Concepts

•Computing Methodologies→Machine Learning→Machine Learning Algorithms→Feature Selection

## Keywords

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [Permissions@acm.org](mailto:Permissions@acm.org).

ISCSIC 2019, September 25–27, 2019, Amsterdam, Netherlands

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7661-7/19/09...\$15.00

<https://doi.org/10.1145/3386164.3389083>

Software Engineering; Programming Language; Deep Learning; Artificial Intelligence

## 1. INTRODUCTION

With the development of human society and Information Technology, computer software is playing an irreplaceable role in the world, and has been applied into almost every aspect in people's lives. However, to design and develop computer software is not an easy task, as there are lots of work and procedures in software engineering for develop a mature software product, which are accompanied with great time cost and workload. To make the software development become easier work, this paper takes insight into the core part of software development: the coding work.

Coding is the main part and the greatest time-consumer of software development. When programmers are writing codes, they often face with the following situations:

- 1.They are writing code fragments that are similar or even identical again and again.
- 2.They are spending time checking for the usage of the external APIs or methods they just have used before.
- 3.They are making careless mistakes, which can be totally unaware of themselves for long time.

These kinds of situations in coding work can obviously and greatly lower the efficiency of programming. Therefore, the research of helping programmers in programming work with specific techniques and algorithms is of great value. One kind of these techniques has been used in some big IDEs (Integrated Development Environments), which is to recommend code elements for programmers which have been used before in their codes, or members and methods of the classes they have written. However, these recommendations are just based on the code files of one user, using string matches and ranked with alphabetical order.

Since a few years before, with the fast development in machine learning theory and proceedings in Natural Language Processing, researchers have started to get a new view of these tasks: Since

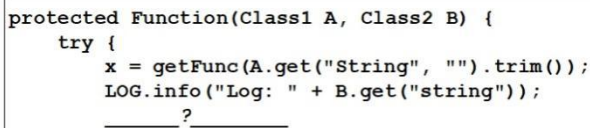
programming languages are also human-used language with constricted grammars and readability, can we process the programming language data with similar methods in Natural Language Processing? The work in this paper is derived from this idea as well. Using a considerable size of learning data of programming codes, this paper builds Deep Learning based models with newly introduced model structures of code element prediction, which are using the AST (Abstract Syntax Tree) information of codes, and are for the following task shown in figure 1: given a partly completed code, predict its subsequent content. It will be useful in helping programmers in proceeding their coding work. This paper presents models that combine NLP (Natural Language Processing) techniques and code analysis methods, and the main contributions of this paper are as following:

1. This paper provides new structures of Deep Learning based Neural Network models for code prediction task, which is the first work of applying Multihead Attention Mechanism in this field as far as the authors know, and the models have outperformed conventional Deep Learning models and recent works in the same field.

2. In this paper, a new traverse method on tree data structure is raised for predict nodes in tree-structure based data, which is called RZT (Reverse Zig-zag Traverse) Algorithm. It can not only be used in programming code domain, but is also useful in all kinds of data in tree structure.

3. This paper carries out experiments and verifies the effectiveness of models on datasets of codes in different programming languages (Android codes, Python codes, and JavaScript codes).

The rest part of this paper is organized as following: Part 2 introduces the related work in this field, Part 3 describes the data this paper is dealing with and the models designed for the task, Part 4 shows the experiments and its results in terms of the effectiveness of models in this paper, and Part 5 is the conclusion of this paper's work and achievement.



```
protected Function(Class1 A, Class2 B) {
    try {
        x = getFunc(A.get("String", "").trim());
        LOG.info("Log: " + B.get("string"));
        _____?_____
    }
}
```

**Figure 1. Introduction of the task this paper deals with. With a partly completed code fragment, the model gives predictions/recommendations of the probable subsequent content/tokens for this code fragment.**

## 2. RELATED WORKS

The researches of computer-based programming codes generation or completion tasks started from early time, and early works are mostly based on the grammar structure and fundamental statistical information of codes[14,31]. From the work of Hindle et al. [10], which shows the way of making analysis of programming codes by statistic models, the aspects of view of the researchers have begun to turn to build statistical language models for program languages. Like the successful works in Natural Language Processing, Allamanis et al. [3] collects a great number of Java codes from GitHub, and maintains a database called GitHub Java Corpus, then it proves the effectiveness of simple N-Gram models on the task of designing language model

for code data. Works like [2,26] use graph to represent codes, and introduce models such as CRF(Conditional Random Fields) to calculate the appearance probability of codes. [4,7,17] are examples of works on the Syntax Tree structure of codes, and make improvements on PCFG(Probabilistic Context Free Grammar) to make it fit for programming language. It is worth mentioning that there is a branch of works that attempt to design vector representation models for code elements, such as the Word Embedding models in Natural Language Processing. [9,20,23,24] are some examples.

On the specific task of code element prediction and completion, there have also been lots of works. The work of Nguyen et al. [21] uses frequent pattern mining techniques for code completion, and Proksch et al. [25] applies Bayesian Network to this problem. With the trend of fast development of Deep Learning, many works utilize Neural Network models for this problem[6,13,15,27]. Works like [6,13,15] make use of AST(Abstract Syntax tree) information of codes, and design LSTM[11] based Neural Network model combining with additional functional model blocks such as Attention Mechanism and pointer network. As API usages are useful and important parts of programming (especially for Android programming), some works focus on predicting the API usages of codes, e.g., [12,22,28].

For the survey and summary of research of natural language models used in programming codes, Allamanis et al. in [1] presents a good survey and summarization. White et al. [30] gives an introduction of Deep Learning techniques in software engineering.

## 3. DATA AND MODELS

### 3.1 Data Process

To learn the features of codes, this paper uses the AST (Abstract Syntax Tree) form of codes, as it contains rich grammar and structural information of codes and can be easily converted from the code texts. Using a segment of Java code as example, this paper first parses the code file into AST as figure 2 shows, then uses the tree structure and node information to build prediction models. E.g., for the Android codes in Java language, the work of this paper designs a program script using AST parsing classes from org.eclipse.core<sup>1</sup> package, and generates parsed ASTs.

From the AST in figure 2, it can be seen that each element of the code is mapped to a node in the Abstract Syntax Tree. Each node is consisted of 2 kinds of information: a node type (In Java AST, the type is consisted of 2 parts: node class on the first line of node in figure 2, and node label on the middle line of node in figure 2, this paper uses their combination as the type of nodes. In AST of other languages this paper does experiments on, there is only one field for type information in each node), and a node text (the bottom line of each node in figure 2). All nodes have non-empty types, and only part of the nodes with specific types have non-empty texts, which represents the texts, names, numbers and so on in the code, and they are all leaf nodes of the AST. When using as input, this paper uses a combined representation *Type Text* for each code (this paper also calls it ‘info’), and transforms them into numerical tensors. For prediction targets, in other words, the output of models, this paper uses 2 different representation of them corresponding to the function of the models: Type prediction and Name prediction, which will be

<sup>1</sup> <http://help.eclipse.org/kepler/index.jsp>

introduced next. When this paper builds train data for models with complete codes, each single node in the AST can be used as target output, and the corresponding input is the nodes before it in

the preorder DFS traverse sequence of the AST, as the preorder DFS traverse sequence of the AST is the natural written order of code elements, which is also illustrated in figure 2.

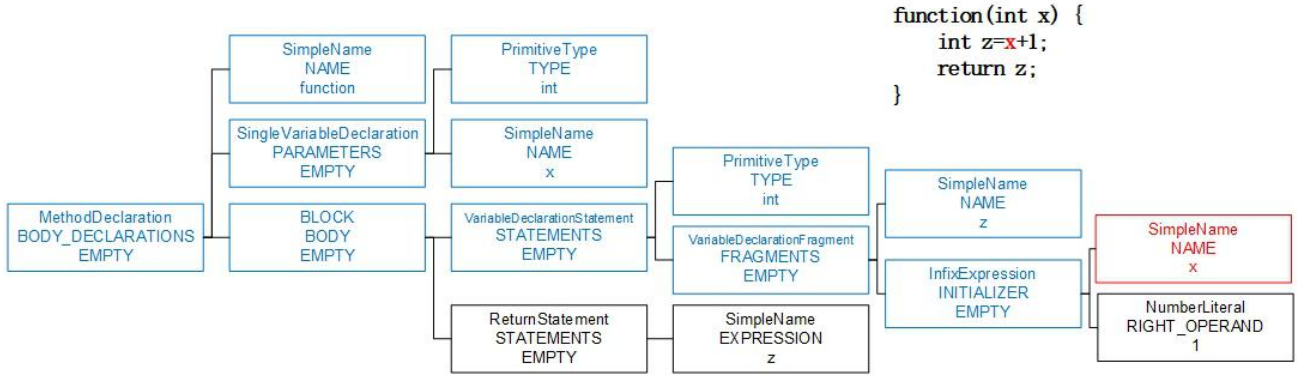


Figure 2. An example of AST (Abstract Syntax Tree) from a Java code fragment. When the red node is to be predicted, the partial tree with nodes prior to it in preorder DFS sequence (the blue nodes) can be used.

### 3.2 Type Prediction Model

The first task this paper deals with is type prediction task: giving an incomplete code, predict the type of the next code element that will appear. This task is important for learning and understanding the structure of programming language. Works like [13,15] use flattened DFS sequence as text-like data for their models to process, but this paper would like to go further. As mentioned before, the input data is generated from part of AST nodes in a full tree, which can form a **partial tree**. It means that there is abundant structural information to be exploited in the input data. In these prediction tasks, as [13,15] did, this paper makes the same assumption that the position (in other words, the parent) of the node to be predicted has been known. It is a natural assumption because there are usually few positions for a partial AST to add nodes, and in practical usage a system embedded with the prediction model has approaches to getting the position information of the nodes to be predicted, for example, from the position of the cursor in a code editor.

The structure of the type prediction model this paper presents is in figure 3. The core part of the model is an LSTM network, which has been proved to be an effective neural network type for processing sequence data such as ordered element sequences, then together with a special sequence parsing algorithm on partial AST and a Multihead Attention Block, the model is built up. The partial AST whose next node is to be predicted is first parsed into a special element sequence with the new algorithm called RZT (Reverse Zig-zag Traverse) raised in this paper, and its embedded vectors by a pre-trained embedding model is fed into the LSTM network. Like common Attention Mechanism, the Multihead Attention Block learns a probability weight distribution on the LSTM outputs of each time step, then the weighted attention output is concatenated with the original LSTM output on the last time step, and at last go to the Softmax classification layer. In the following parts, this paper will introduce the model part by part in detail.

#### 3.2.1 Sequence Parsing on Partial AST

As LSTM accepts sequences as input, the partial AST cannot be directly input into the model. Commonly, the preorder DFS traverse algorithm is used to generate a sequence from AST that it can be input into the LSTM model. However, the preorder DFS traverse algorithm only output the contents of nodes in AST in

order, so it ignores the important structural information in the code. Thus, to solve this problem, this paper designs a new algorithm which generates a sequence from a partial tree and the position of the next node. The central idea of this algorithm is that the nodes which has more relevance with the node to be predicted appears in the last positions of the sequence (because the last positions will have a greater impact on the results of LSTM output and final prediction).

This paper proposes a traverse algorithm called Reverse Zig-zag Traverse(RZT) to transfer the partial AST of code snippet to the sequence of nodes. To extract the structural information from codes better, the idea is that for the target node P, its parent node and sibling nodes have the strongest relevance with it. Therefore, the RZT algorithm puts the parent node of P at the end of the sequence, and put the sibling nodes at corresponding positions in terms of the distance, as in the network of LSTM the last nodes have most influences on the last output. By this way, the generated sequence represents the structural relationship of codes. The algorithm traverses the AST in reverse order, layer by layer and add subtree information to the sequence recursively.

In order to make the sequence extraction more reasonable, there are 2 rules in the algorithm:

1. All nodes deeper than the target node P will not appear in the sequence.
2. There is a parameter D, to control the max depth of whether the algorithm will acquire descendants of a node. In the practical use of this paper, the D is set to 1.

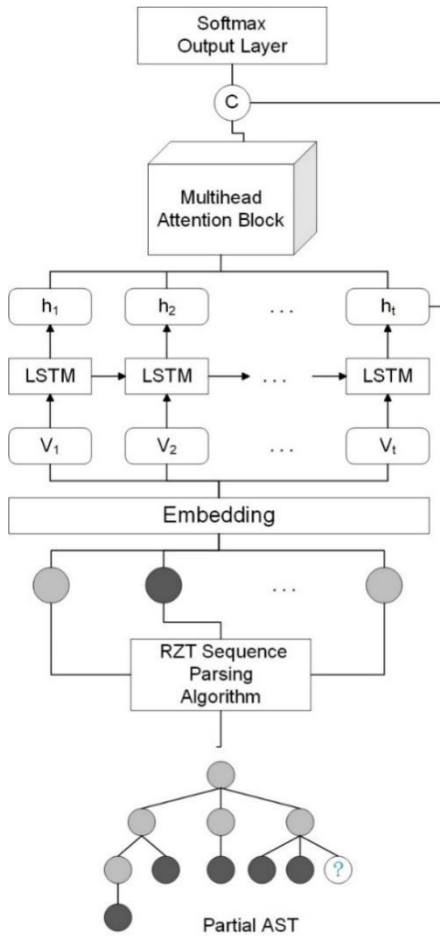
Algorithm 1 and algorithm 2 shows the full description of the algorithm, in which algorithm 1 is a part of algorithm which returns a pre-order DFS traverse sequence of a tree with reversed children order, and algorithm 2 is the RZT algorithm. Figure 4 is an example of the RZT algorithm, and figure 5 is the comparison of RZT algorithm and conventional DFS traverse, which shows that RZT algorithm alleviates the problem that neighbor nodes on DFS traverse sequence are not always have the most relevance, as the parent and the siblings are always nearest to the node to be predicted.

#### 3.2.2 Embedding Model

The concept of mapping code elements to Euclidean vector space

with a pre-trained sub-model comes from Word Embedding, the broadly used techniques in Natural Language Processing. By using embedding models, not only the fixed low dimension representation of data is obtained, but also features with respect to the relevance between elements are acquired. Therefore, the embedding model will undoubtedly improve the performance of the whole model.

Benefitted from the similarity between natural language data, and programming codes data, the same model can be transferred from Natural Language Processing to this task. This paper uses 2 different kind of embedding models to train embedding vectors, and the flattened AST (the pre-order DFS traverse sequence of AST) is utilized as texts to train the embedding model. One of the word embedding model is Skip-gram[18,19] in Word2Vec, and the other is FastText[8], a derivation and improved model of Skip-gram, which takes the subword information of words into account by training embedded vectors for n-grams (word fragments) of words.



**Figure 3.** The full structure of type prediction model. The partial AST is first parsed into a sequence of nodes by the algorithm this paper designs, and its embedded vectors are input into LSTM. The original output at the list time step of LSTM is concatenated with the Multihead Attention Block output.

---

**Algorithm 1** child\_reverse\_preorderDFS
 

---

**Input:** node N.

**Output:** sequence S

```

1: L, S ← empty stack
2: if N == null:
3:   return
4: L.push(N)
5: while not N.isEmpty():
6:   N ← L.pop()
7:   S.push(N)
8:   while not N.child.isEmpty():
9:     L.push(N.child.pop())
10: return S
  
```

---

**Algorithm 1.** the child\_reverse\_preorder DFS traverse Algotirhm.

---

**Algorithm 2** Reverse Zig-zag Traverse(RZT)
 

---

**Input:** A partial AST Tree and a node N on it.

**Parameter:** max depth D for acquiring its descendants.

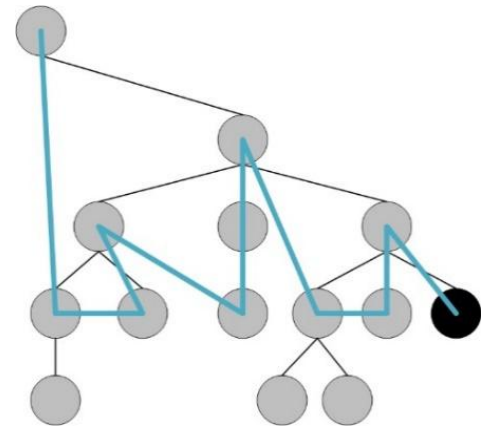
**Output:** A Reverse Zig-zag Traverse sequence S.

```

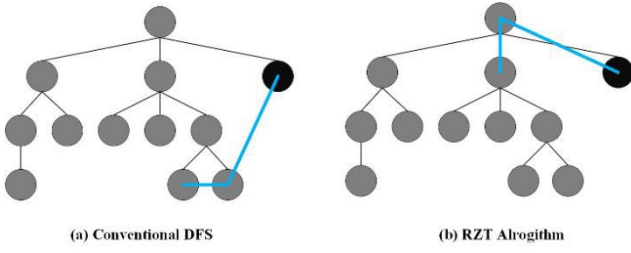
1: set d ← 0, P ← N, S ← empty list
2: delete all nodes deeper than N in Tree.
3: while 1:
4:   if P is the root of Tree:
5:     Break
6:   S.append(P.Parent)
7:   for sibling in reversed(P.eldersiblings):
8:     if d ≤ D:
9:       S.append(child_reverse_preorderDFS(sibling))
10:    else:
11:      S.append(sibling)
12:    d ← d + 1
13:   P ← P.Parent
14: return Sreverse()
  
```

---

**Algorithm 2.** The Reverse Zig-zag Traverse (RZT) Algorithm.



**Figure 4.** An example of the RZT (Reverse Zig-zag Traverse) traverse algorithm. Start from the black node, the blue trajectory shows the sequence generated by the algorithm.



**Figure 5. The comparison between conventional DFS traverse sequence and the sequence by the new algorithm. In DFS traverse sequence (a), the 2 neighbor nodes of the target black nodes have long distances to it, but in the new traverse algorithm, the neighbors are its parent and the nearest elder sibling.**

### 3.2.3 Multihead Attention

A vanilla LSTM network will provide the hidden state at the last time step as output, and it represents the features of the sequence in which contains more information of the last time steps. As the task is a sequence prediction problem, the output at the last time step will play an important role in prediction, however, to make use of outputs in other time steps, Attention Mechanism[5,16] has been raised for generate context vector from LSTM outputs of time steps besides the last one, and Li et al. [13] combines it with LSTM network for the code prediction task. In the work of this paper, other than the original Additive[5] or Multiplicative[16] Attention, a Multihead Attention block is added to the model. It is raised by [29], and the main idea of it is to stack several attention layers in the model, which use the same input but different weights and biases trained separately. The formula of attention calculation is as following, where  $V$  are the LSTM outputs of each time steps, and  $W_i^1, W_i^2, W_i^3$  are weights of the attention layer  $i$  in the attention block,  $d_k$  is the dimension of vectors. The only difference of this formula with the original formula in [29] is that the model in this paper uses self-attention, so there is only one value Matrix  $V$  instead of the original triplet  $(Q, K, V)$ . The attention calculation formula is similar to the Multiplicative Attention[16], except that the dot products  $VV^T$  is scaled (divided) by the square root of the dimension  $d_k$  of embedding vectors, as [29] finds that the dot product grows extremely large with the increase of embedding dimension, and the Softmax function will fall into regions where it has extremely small gradients (only few dimensions with have outputs that are not nearly zero).

$$Attention(V_1, V_2, V_3) = \text{Softmax}\left(\frac{V_1 V_2^T}{\sqrt{d_k}}\right) V_3$$

$$Multihead(V) = \text{Concat}(head_1, \dots, head_h)$$

$$\text{Where } head_i = Attention(VW_i^1, VW_i^2, VW_i^3)$$

## 3.3 Name Prediction Model

Since most nodes in AST are non-leaf nodes with empty text, the type prediction can deal with the prediction problem at most circumstances. Nevertheless, the names information of variables, methods and so on in the codes is also of great importance for programming, which is contained in the text part of leaf nodes in AST. As a supplement to type prediction model, the name prediction model focuses on the nodes representing names in codes, and learn to give predictions of them. The structure of this model is in figure 6. To make accurate prediction of the names in

the nodes, 3 kinds of information (context) are extracted from the partial AST. Besides of the conventional DFS sequence before the node, the sequence from the RZT algorithm is used as structural context, and the sequence of names appear before is used as name context. The 3 parts of information are embedded with embedding model, and come to the LSTM and Multihead Attention network, then the LSTM and attention outputs are averaged to provide 1 vector for each input. 3 parts of output vectors are concatenated for final Softmax probability calculation. In the following, this paper will describe the 3 parts of input in detail.

### 3.3.1 Embedded RZT Sequence

This is identical with the sequence parsed by RZT algorithm in the type prediction model. It is the ‘structural context’ of the name to be predicted.

### 3.3.2 Embedded DFS Sequence

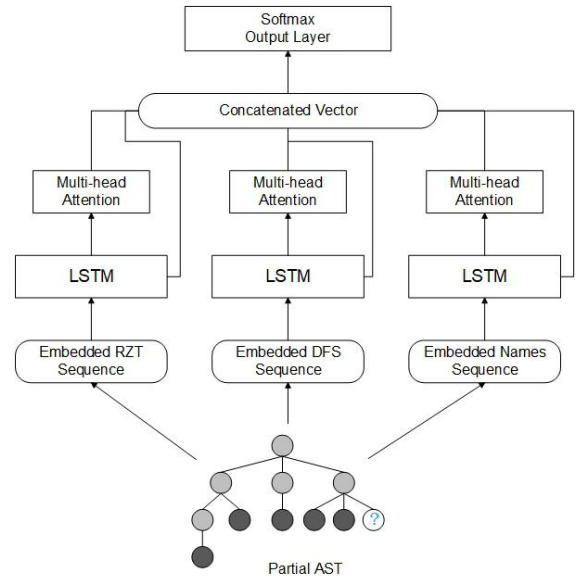
The ordinary preorder DFS Traverse sequence of the partial tree before the token to be predicted. It is the ‘textual context’ of the name to be predicted.

### 3.3.3 Embedded Names List

The names that have appeared in the same code file before have great relevance with the names to be used after, it is because that many names of variables, classes or methods are used repeatedly in one code, and some names have obvious co-occurrence features. By recording names list, the model can get a long-range information of names used before.

### 3.3.4 Network Component

The LSTM network and Multihead Attention Block that name prediction model uses have the same structure and principles as the ones in type prediction model. To prevent the size explosion of parameters number, the attention output and original LSTM output are first averaged, then the 3 averaged vectors are concatenated for Softmax layer. The dimension of the output vector of Softmax Layer is the number of names to be predicted.



**Figure 6. The structure of name prediction model. 3 kinds of information (context) are extracted from the partial AST to make prediction.**



## 4. EXPERIMENTS

### 4.1 Dataset

This paper makes experiments on 3 different datasets.

#### 4.1.1 Android Dataset

The first dataset this paper uses and trains models on consists of Android projects and code files from GitHub Java Corpus<sup>2</sup>, and some other Android projects manually collected from GitHub, then it is split into train set and test set. In the whole dataset of 3584 Android projects, this paper randomly selects 10% of them as test set, and the rest part is used as training set. It is worth to be noted that the work of this paper splits the training and test set by projects rather than code files, because there are always more similarities between codes inside one certain project, but the model is trying to obtain features from programming language level other than the specificity of projects. The statistics of the database are in table 1. There are  $2.4 \times 10^5$  code files in total in the dataset, with 491 type of nodes and  $4.8 \times 10^6$  different patterns of full info of nodes. Nearly all of the node types and tokens(info) appear in the training set. The huge number of info mostly comes from the great number of various names of classes, variables and methods used in the codes. In the data preprocessing, the work of this paper has transferred all the digits/numbers and strings in the codes to fixed tags, as it is obvious that to predict what number or string one will write in the codes is nearly impossible.

**Table 1. the number of projects, code files and AST elements in Android dataset, and the counts of type and full info of nodes in Android dataset.**

Set	Project	File	Element
Train	3225	$2.3 \times 10^5$	$1.4 \times 10^8$
Test	359	12740	$9.2 \times 10^6$
Overall	3584	$2.4 \times 10^5$	$1.5 \times 10^8$

Set	Type	Info
Train	489	$4.6 \times 10^6$
Test	407	$4.5 \times 10^5$
Overall	491	$4.8 \times 10^6$

#### 4.1.2 Python and JavaScript Dataset

This paper also uses 2 other code file datasets, which are collected by ETHZ SRI Lab. The 2 datasets are 150k Python Dataset<sup>3</sup> (Python codes) and 150k JavaScript Dataset<sup>4</sup> (JavaScript codes). Each of the datasets has 150k code files and their corresponding ASTs, and is split into trainset of 100k size and testset of 50k size. More detailed statistical information is in table 2 and table 3. For convenience, this paper calls them py150 and js150 datasets. In conclusion, all of the datasets has abundant codes for the model to learn with. In the rest parts of this chapter, this paper will introduce the experimental details and results.

**Table 2. the statistics of Py150 Dataset.**

Set	File	Nodes	Type	Info
Train	$1 \times 10^5$	$6.22 \times 10^7$	168	$3.66 \times 10^6$
Test	$5 \times 10^4$	$3.04 \times 10^7$	167	$2.07 \times 10^6$
Overall	$1.5 \times 10^5$	$9.26 \times 10^7$	181	$4.94 \times 10^6$

**Table 3. the statistics of Js150 Dataset.**

Set	File	Nodes	Type	Info
Train	$1 \times 10^5$	$8.47 \times 10^7$	44	$2.32 \times 10^6$
Test	$5 \times 10^4$	$4.18 \times 10^7$	44	$1.40 \times 10^6$
Overall	$1.5 \times 10^5$	$1.27 \times 10^8$	44	$3.06 \times 10^6$

### 4.2 Model Configuration and Train Details

For type prediction model, this paper uses LSTM with 512 hidden units. For name prediction model, all LSTMs used in the model have 1024 hidden units. The number of heads (parallel layers of Multihead Attention block) is set to 32 when Multihead Attention block is used. The lengths of input sequences are padded to fixed number due to their types: the conventional DFS sequence: 50, RZT algorithm sequence: 25, names list: 50. In the task of type prediction, 10% of the nodes in train set is used for output, as using all the nodes for output has a great time cost, and in fact lowers the efficiency of training. In the task of name prediction, this paper chooses the frequently appeared names in the top K tokens of all nodes for prediction, where  $K=10000$ . All models are trained over 10 epochs, using Adam optimizer with 0.001 initial learning rate, and the learning rate is multiplied by 0.7 after each epoch. In the learning, the norms of weights are clipped with threshold 5. For embedding model, in the embedding module, a 200-dimension Skip-gram embedding model with window size of 10 is used. The embedding models are chosen with a grid search on parameters and model kinds.

### 4.3 Experimental Results

The tasks this paper carries out experiments on to validate the effectiveness of models are type prediction task and name prediction task which are described before, and the results are as following. As introduced before, this paper carries out experiments on 3 programming code datasets in 3 different programming languages: Android(Java), Python and JavaScript.

#### 4.3.1 Type Prediction Model

For comparison with the new model in this paper, several other models are also trained and tested on the dataset, which include a vanilla LSTM network with conventional DFS sequence input, vanilla LSTM with the new traverse sequence, and LSTM with normal Additive or Multiplicative Attention layers using the new traverse sequence. The comparison models are from recent state-of-art works [6,13,15], and the accuracies (top-1 precision of classification) is in table 4, 5, and 6: table 4 for the Android dataset, table 5 for the Py150 dataset, and table 6 for the Js150 dataset.

**Table 4. Type prediction model experimental results on Android dataset.**

Model	Accuracy(%)
Vanilla LSTM + DFS Input[24]	70.42
Vanilla LSTM + RZT Seq Input	80.99
LSTM + RZT + Attention (Additive)[20]	80.78
LSTM + RZT + Attention (Multiplicative)	80.96
LSTM + RZT + Attention (Multihead)	<b>81.45</b>

**Table 5. Type prediction model experimental results on Py150 (Python) dataset.**

Model	Accuracy(%)
Vanilla LSTM + DFS Seq Input [24]	61.56
Vanilla LSTM + RZT Seq Input	75.17
LSTM + RZT + Attention (Additive)[20]	74.93
LSTM + RZT + Attention (Multiplicative)	75.08
LSTM + RZT + Attention (Multihead)	<b>75.41</b>

<sup>2</sup> <http://groups.inf.ed.ac.uk/cup/javaGithub/>

<sup>3</sup> <https://www.sri.inf.ethz.ch/py150>

<sup>4</sup> <https://www.sri.inf.ethz.ch/js150>

**Table 6. Type prediction model experimental results on Js150 (JavaScript) dataset.**

Model	Accuracy(%)
Vanilla LSTM+DFS Seq Input[24]	63.97
Vanilla LSTM + RZT Seq Input	72.92
LSTM+RZT+Attention(Additive)[20]	71.92
LSTM+RZT+Attention(Multiplicative)	72.10
LSTM+RZT+Attention(Multihead)	<b>73.30</b>

In the experiment of this paper, the vanilla LSTM network with DFS sequence input is the baseline for type prediction experiment. When replacing DFS with RZT algorithm proposed in this paper, the accuracy of prediction on each dataset gets a great leap (10.55% on Android dataset, 13.61% on Python dataset, and 8.95% on JavaScript dataset). This paper argues that such a large improvement is derive from RZT Algorithm’s ability to extract and retain structure information from code snippet, and it can be explained with the knowledge of the code organization structure in the software engineering domain. Codes have lay-by-layer nest structure in natural, and codes in the same action scope, rather than simple close together, have a stronger correlation in most situations. RZT Algorithm grasps the parent-child and sibling relationship of AST and the scope information of code, which are ignored in traditional DFS Algorithm.

Another important improvement of the model this paper designs is the Multihead Attention Block. It can be seen that, when classical Additive Attention and Multiplicative Attention are used for further information extraction in the sequence, the accuracies are not improved in comparison with the model without attention, and are even lowered. However, Multihead Attention Block shows its effectiveness, and make contribution to the improvement of the prediction accuracies. Firstly, by using Multihead Attention, output vectors are added up using different weights. Several self-attention processes are done through the Multihead Attention Block. So the information from output features are extracted using different weights by different ways in multiple times. Richer information from the output features is obtained. As a result, as more information is integrated into the output feature vector, model performance increases to some extent. Secondly, the Multihead Attention Block uses a refined attention calculation formula from the original Multiplicative Attention by adding a scaling factor of dimension. The experiments prove that Multihead Attention Block can learn a more robust weight distribution of the input vectors.

By analyzing the experimental results, the framework proposed in this work gets leading results on codes of different programming languages. The framework in this paper has a certain generalization ability for prediction on code snippets in different programming languages.

#### 4.3.2 Name Prediction Model

The comparison models this paper uses are vanilla LSTM and LSTM with Multihead Attention Block using several combinations of input contexts, and the details are in Table 7, 8 and 9, corresponding to experimental results on Android, Python and JavaScript datasets. The model names with “MH Attention” are models using Multihead Attentions after LSTM. In this part of experiments, this work tries to examine the joint effect of different model frameworks and different input information extracted from codes. As a result, lots of combinatorial experiments are done. Top 1/3/5 accuracies are used for evaluating the models.

**Table 7. Name prediction model experimental results on Android dataset.**

Models	Accuracy(%)		
	top1	top3	top5
DFS	67.43	81.29	85.13
RZT&Names	68.14	82.21	86.08
DFS&Names	67.95	81.80	85.68
RZT&DFS&Names	68.64	82.56	86.33
DFS&MH Attention	67.82	81.76	85.54
RZT&Names&MH Attention	62.85	78.74	83.26
DFS&Names&MH Attention	68.42	82.54	86.34
RZT&DFS&Names&MH Attention	70.93	83.97	87.48

**Table 8. Name prediction model experimental results on Python dataset(py150).**

Models	Accuracy(%)		
	top1	top3	top5
DFS	65.94	80.02	83.83
RZT&Names	64.49	80.06	84.33
DFS&Names	66.14	81.05	85.12
RZT&DFS&Names	67.14	81.44	85.26
DFS&MH Attention	66.08	80.98	84.87
RZT&Names&MH Attention	64.36	80.35	84.67
DFS&Names&MH Attention	67.00	81.86	85.85
RZT&DFS&Names&MH Attention	<b>67.51</b>	<b>82.32</b>	<b>86.21</b>

**Table 9. Name prediction model experimental results on JavaScript dataset(js150).**

Models	Accuracy(%)		
	top1	top3	top5
DFS	63.52	76.62	80.93
RZT&Names	63.87	77.01	81.35
DFS&Names	63.91	77.86	81.90
RZT&DFS&Names	64.79	78.56	82.59
DFS&MH Attention	63.76	77.17	81.66
RZT&Names&MH Attention	63.24	76.93	81.47
DFS&Names&MH Attention	64.61	78.49	82.91
RZT&DFS&Names&MH Attention	<b>66.05</b>	<b>79.67</b>	<b>83.74</b>

First, a name list extracted from context of code snippet is added for supporting prediction. By adding a name list, the search area of this classification problem has been greatly reduced. As the search range is reduced, the experimental results are improved naturally. Second, this paper finds out that DFS and RZT are almost neck and neck in name prediction problem, and in some cases, DFS works very well. After analysis, this paper believes that the result is due to that name prediction is a more detailed task. Hence, in some samples, sequential information is more important than structure information (e.g. there is always a class name after ‘class’) Furthermore, DFS has the ability to retain sequential information naturally. So it’s not surprising that the DFS gets such a good result. However, the structure information is also very influential. And the strong ability to acquire structural information of RZT help it perform well in name prediction task, too. In general, although RZT do not has a dominant act compared to the task of Type prediction, RZT still has an advantage in most cases. In addition, Multihead Attention shows its effectiveness in improving the accuracies of models in this task as well, and among all of the experimental results on 3

datasets, the full framework in this paper (LSTM+RZT+names+Multihead Attention, Figure 6 in section 3.3) shows best prediction accuracies in top1/3/5 prediction tasks on each dataset.

#### 4.4 Discussion

Through experiments, this paper achieves convincing results. By using an LSTM framework, a code prediction problem can also be settled well. In addition, Reverse Zig-zag Traverse (RZT), the traverse method proposed in this paper, extracts the relation information of abstract syntax trees and structural information of codes. As a result, the LSTM framework combined with RZT makes models in this paper very competitive in performance.

#### 5. CONCLUSION

This paper gives a new solution for programming codes prediction models for different programming languages. By using RZT algorithm, structural information of programming codes is captured and obtained. By using Multihead Attention, more accurate prediction of types and names of target nodes can be achieved. This paper integrates methods above with LSTM network framework and achieved state-of-art performance compared with modern researches of computer-algorithm based programming codes completion tasks. The good performances of the models on datasets of various programming languages proves that the framework proposed in this paper is not only effective but also has generalization ability in codes of different programming languages. This paper is interested in applying the algorithms proposed in this paper to a wider range of problems and integrate the methods into an existing IDE in the future.

#### ACKNOWLEDGEMENTS

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

#### REFERENCES

- [1] Allamanis, Miltiadis, Barr, Earl T, Devanbu, Premkumar, et al. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4), 81.
- [2] Allamanis, Miltiadis, Brockschmidt, Marc, and Khademi, Mahmoud. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*.
- [3] Allamanis, Miltiadis, and Sutton, Charles. 2013. *Mining source code repositories at massive scale using language modeling*. In Proceedings of the 10th Working Conference on Mining Software Repositories. 207-216.
- [4] Amodio, Matthew, Chaudhuri, Swarat, and Reps, Thomas. 2017. Neural Attribute Machines for Program Generation. *arXiv preprint arXiv:1705.09231*.
- [5] Bahdanau, Dzmitry, Cho, Kyunghyun, and Bengio, Yoshua. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.
- [6] Bhoopchand, Avishkar, Rocktäschel, Tim, Barr, Earl, et al. 2016. Learning Python code suggestion with a sparse pointer network. *arXiv preprint arXiv:1611.08307*.
- [7] Bielik, Pavol, Raychev, Veselin, and Vechev, Martin. 2016. *PHOG: probabilistic model for code*. In International Conference on Machine Learning. 2933-2942.
- [8] Bojanowski, Piotr, Grave, Edouard, Joulin, Armand, et al. 2016. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*.
- [9] Corley, Christopher S, Damevski, Kostadin, and Kraft, Nicholas A. 2015. *Exploring the use of deep learning for feature location*. In 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). 556-560.
- [10] Hindle, Abram, Barr, Earl T, Su, Zhendong, et al. 2012. *On the naturalness of software*. In Software Engineering (ICSE), 2012 34th International Conference on. 837-847.
- [11] Hochreiter, Sepp, and Schmidhuber, Jürgen. 1997. Long short-term memory. *Neural computation*, 9(8), 1735-1780.
- [12] Hou, Daqing, and Pletcher, David M. 2011. *An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion*. In Software Maintenance (ICSM), 2011 27th IEEE International Conference on. 233-242.
- [13] Li, Jian, Wang, Yue, King, Irwin, et al. 2017. Code Completion with Neural Attention and Pointer Networks. *arXiv preprint arXiv:1711.09573*.
- [14] Li, Zhenmin, and Zhou, Yuanyuan. 2005. *PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code*. In ACM SIGSOFT Software Engineering Notes. 306-315.
- [15] Liu, Chang, Wang, Xin, Shin, Richard, et al. 2016. Neural Code Completion.
- [16] Luong, Minh Thang, Pham, Hieu, and Manning, Christopher D. 2015. Effective Approaches to Attention-based Neural Machine Translation. *Computer Science*.
- [17] Maddison, Chris, and Tarlow, Daniel. 2014. *Structured generative models of natural source code*. In International Conference on Machine Learning. 649-657.
- [18] Mikolov, Tomas, Chen, Kai, Corrado, Greg, et al. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.
- [19] Mikolov, Tomas, Sutskever, Ilya, Chen, Kai, et al. 2013. *Distributed representations of words and phrases and their compositionality*. In Advances in neural information processing systems. 3111-3119.
- [20] Mou, Lili, Li, Ge, Liu, Yuxuan, et al. 2014. Building program vector representations for deep learning. *arXiv preprint arXiv:1409.3358*.
- [21] Nguyen, Anh Tuan, Nguyen, Tung Thanh, Nguyen, Hoan Anh, et al. 2012. *Graph-based pattern-oriented, context-sensitive source code completion*. In Software Engineering (ICSE), 2012 34th International Conference on. 69-79.
- [22] Nguyen, Tam The, Pham, Hung Viet, Vu, Phong Minh, et al. 2015. *Recommending API usages for mobile apps with hidden markov model*. In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). 795-800.
- [23] Nguyen, Trong Duc, Nguyen, Anh Tuan, and Nguyen, Tien N. 2016. *Mapping API elements for code migration with vector representations*. In Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on. 756-758.



- [24] Nguyen, Trong Duc, Nguyen, Anh Tuan, Phan, Hung Dang, et al. 2017. *Exploring API embedding for API usages and applications*. In Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on. 438-449.
- [25] Proksch, Sebastian, Lerch, Johannes, and Mezini, Mira. 2015. Intelligent code completion with Bayesian networks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1), 3.
- [26] Raychev, Veselin, Vechev, Martin, and Krause, Andreas. 2015. *Predicting program properties from big code*. In ACM SIGPLAN Notices. 111-124.
- [27] Raychev, Veselin, Vechev, Martin, and Yahav, Eran. 2014. *Code completion with statistical language models*. In Acm Sigplan Notices. 419-428.
- [28] Thung, Ferdian, Wang, Shaowei, Lo, David, et al. 2013. *Automatic recommendation of API methods from feature requests*. In Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering. 290-300.
- [29] Vaswani, Ashish, Shazeer, Noam, Parmar, Niki, et al. 2017. *Attention is all you need*. In Advances in Neural Information Processing Systems. 5998-6008.
- [30] White, Martin, Vendome, Christopher, Linares-Vásquez, Mario, et al. 2015. *Toward deep learning software repositories*. In Proceedings of the 12th Working Conference on Mining Software Repositories. 334-345.
- [31] Xie, Tao, and Pei, Jian. 2006. *MAPO: Mining API usages from open source repositories*. In Proceedings of the 2006 international workshop on Mining software repositories. 54-57.

---

<sup>i</sup> Both authors contribute equally to this work

\* corresponding author