Deep Learning based Code Completion with ASTToken2Vec

AUTHOR: Li Dongfang 17M38058
Supervisor: Masuhara Hidehiko

Tokyo Institute of Technology
Mathematical and Computing Science

July 2019

Abstract

Software development has been a quite complicated job in recent years due to the complexity of modern software and program systems. In order to work efficiently, programmers depend on libraries and frameworks to develop software so that the existing excellent source code can be reused to develop new programs. However, As the software grows huger and includes more source code, it is difficult to remember all Application Programming Interfaces (APIs) and every variable name in a software project. To support programmers writing code more efficient, this thesis focuses on the code completion system which is one of the most useful features of modern integrated development environments (IDEs) and able to improve software development efficiency, reduce the possibility of writing errors. When a programmer types code, a good code completion system provides a list of completion suggestions with high predicting accuracy and inserts suggestion code automatically to enable the software development becoming faster.

Most of traditional code completion systems are based on statistical techniques which have a not good performance and fails to predict tokens in the long term program context. In this thesis, we attempt to increase the prediction performance of deep learning-based code completion systems by introducing a new embedding method(a vector representation method) for AST nodes. This new embedding model is called ASTToken2Vec which is inspired by Word2Vec embedding. This model trains a neural network by using the context information in ASTs to generate semantic-based representation vectors which contain more semantic knowledge for AST nodes and valuable for code completion. We name our model as AT2V-LSTM integration model which means we integrate our embedding method ASTToken2Vec with an LSTM model to predict both program text(next non-terminal and terminal) and structural components of non-terminal nodes. Our model is able to extract more AST structural information by ASTToken2Vec embedding and take advantage of the sequence of program tokens by LSTM model.

We evaluate the prediction performance of our integration model on a JavaScript AST dataset generated from open-source programs which contains a total of 150,000 JavaScript files. Our model achieves a good performance of the next token prediction with the help of the semantic-based representation vectors generation by ASTToken2Vec model.

Contents

1 INTRODUCTION **5**
- 1.1 Motivation 5
- 1.2 Research Problems 6
- 1.3 Deep Learning Methods 7
- 1.4 Our Work 8

2 BACKGROUND **11**

3 PRELIMINARIES **16**
- 3.1 Data Format 16
  - 3.1.1 Source Code Text 17
  - 3.1.2 Sequence of Tokens 18
  - 3.1.3 Abstract Syntax Tree 18
- 3.2 Data Processing for the LSTM Model 22
  - 3.2.1 AST to Sequence Conversion 22
  - 3.2.2 Conclusion 25
- 3.3 AST Reconstruction 27
- 3.4 Identifiers Renaming 30

4 ASTToken2Vec EMBEDDING **33**
- 4.1 Data Represention 33
  - 4.1.1 One-hot Encoding 34
  - 4.1.2 Embedding Representation 34
  - 4.1.3 Word2Vec Embedding 35
- 4.2 AST Nodes Embedding 36
  - 4.2.1 Contexts of Non-terminal Tokens 38
  - 4.2.2 Contexts of Terminal Tokens 40
- 4.3 Data Processing 41

4.4 Model Structure 42
4.5 Joint Loss Function 44
4.6 Conclusion 46

5 LSTM INTEGRATION MODEL 47
5.1 Recurrent Neural Network 47
5.1.1 Long Short Term Memory(LSTM) 48
5.2 LSTM for Code Completion 49
5.2.1 Model Structure 50
5.2.2 Conclusion 53
5.3 Integration 54
5.4 Conclusion 54

6 EXPERIMENT 56
6.1 Dataset and Data Processing 56
6.1.1 Dataset Details 56
6.1.2 Data Processing 58
6.2 Experiment of ASTToken2Vec 59
6.2.1 Training details 59
6.2.2 Visualization 60
6.2.3 Similarity Calculation 62
6.2.4 Conculsion 63
6.3 Experiment of AT2V-LSTM Model 63
6.3.1 Training details 64
6.3.2 Next Token Prediction 64
6.3.3 Prediction Analysis 67
6.3.4 Conclusion 69

7 CONCLUSION 70
7.1 Summary 70
7.2 Future Research Directions 72

List of Figures

1.1 Code completion in Eclipse 6
1.2 Overview of ASTToken2Vec LSTM integration model 10

3.1 A JavaScript 'loop statement' and its AST format 20
3.2 One AST sample of a JavaScript code snippet 21
3.3 Training samples generated from a toy non-terminal binary tree 25
3.4 The complete binary tree of JavaScript 'loop printing' code snippets 27
3.5 (1) is a toy complete binary tree and (2) is the sequence of samples generated in terms of AST processing regulations . . . 29
3.6 AST reconstruction from a sequence 30
3.7 Identifier renaming 31

4.1 Non-terminal and terminal context for nodes. (1) is the context for non-terminal nodes, (2) is the context for terminal nodes 40
4.2 model structure for ASTToken2Vec. (1) is the structure of NT2V, (2) is the structure of TT2V 44

5.1 The structure of NTI2P model 51

6.1 The visualization for the 2-D terminal representation vectors generated by ASTToken2Vec 62
6.2 Validation accuracy for non-terminal prediction during the training phase 65
6.3 Validation accuracy for terminal prediction during the training phase 66
6.4 Code snippets for prediction result analysis 68

List of Tables

3.1 Training samples generated from the AST representation complete binary tree in the figure3.4 26

4.1 Training samples for NT2V model 42
4.2 Training samples for TT2V Model 43

6.1 Dataset 57
6.2 Non-terminal Evaluation Accuracy 66
6.3 Terminal Evaluation Accuracy 66
6.4 Node Information Evaluation Accuracy 66

Chapter 1

INTRODUCTION

1.1 Motivation

With the rapid development of the software industry, modern software has become much more complex and huger in recent years. The development of software depends on the existing excellent source code like frameworks and libraries. For many programmers, it is very difficult to remember every Application Programming Interfaces(APIs) to call the existing functions and every variable which has been specified. Software development has become hard work due to this complexity. In order to overcome this problem, more and more programs begin to write source code with modern Integrated Development Environments (IDEs). A good IDE is able to integrate interpreters, compilers, servers and other tools together so that it can help programmers writing code and developing software more efficient with a low possibility of bugs and errors. Code completion systems play a significant role in modern IDEs which are used by most programmers. These systems can help programmers writing code faster and more correct. Code completion has become an indispensable tool in IDEs. An effective code completion system can predict what kind of code will be written and generate a code suggestion list for programmers when they are writing some code. It can predict the next method names, APIs and even next variables programmers want to write in real-time. Figure1.1 shows a code completion system in Eclipse for Java. In this example, code completion system is going to predict what kind of code the programmer wants to write according to the existing information of the program.

```
public class Main {
    public static void main(String[] argv) {
        String string1 = "hello";
        String string2 = "world";
        string1.
    }
}
```

```
charAt(int index) : char - String
chars() : IntStream - String
codePointAt(int index) : int - String
codePointBefore(int index) : int - String
codePointCount(int beginIndex, int endIndex)
codePoints() : IntStream - String
compareTo(String anotherString) : int - String
compareToIgnoreCase(String str) : int - String
concat(String str) : String - String
contains(CharSequence s) : boolean - String
Press '⌥/' to show Template Proposals
```

Figure 1.1: Code completion in Eclipse

1.2 Research Problems

Traditional code completion systems pay attention to the aforementioned context of the source code, they are able to scan all libraries and classes which are included and imported by current programs and analyze the static and syntactic methods and variables to do prediction and suggestion. Programmers do not need to write down all code because some code can be predicted and inserted by code completion systems. This is the reason why code completion can substantially accelerate software development and improve programmers' efficiency. But this traditional code completion system has some problems that it is not intelligent, the code it suggests only depends on the statistical information generated from libraries. This statistical method is based on simple term frequency statistics which often relatively and brittle have a higher error rate which means some crucial information such as common structural idioms and the names of currently scoped variables would be ignored and cannot be used to predict code even it is a very significant influence on the completion. Traditional systems can not analyze the existing code but only can gather statistical information so that the prediction is hard to depend on the context of the program in real-time.

Another problem of the traditional code completions is that most of the systems rely on strong typing information (like Visual Studio for C++, Eclipse for Java) to give a strict restriction to complete code for a high pre-

diction accuracy, however, the dependency on typing information limits their applicability to widely used in dynamically typed languages like Python and JavaScript programming languages. Basing on statistics and the reliance on typing information, both of these two problems limit the usage of the code completion. In order to solve these problems and increasing the usability and accuracy of code completion systems, people try to find a better way to predict next code programmers want to type and increase the efficiency of code completions especially the prediction performance of accuracy.

1.3 Deep Learning Methods

Deep learning techniques, also known as part of machine learning methods based on hierarchical artificial neural networks, develop rapidly in recent years. Deep neural networks are inspired by the discovery of information processing in biological systems especially the working mechanism of human brains. Deep learning has made great achievements in many fields and becomes more and more potential for handling existing tremendous data. The representative deep learning models like convolutional neural networks(CNNs) are much more powerful handling images data than any other existing methods to solve image-related tasks like image recognition and picture segmentation. Rather than the strength of image comprehension abilities, recurrent neural networks(RNNs) achieved an excellent performance in the natural language processing (NLP) field like sentences prediction or speech recognition. One of the typical RNNs, the long-short-term-memory(LSTM) model are even better than human beings of the abilities to understand natural languages.

The LSTM model[1], which is widely used in sequence data like natural language sentences, leverages memory structure and gate operations to understand the crucial knowledge hidden in the given sequence data. With 'cell' memory structure, LSTM models are able to store information over a very long time which plays a significant influence on the following prediction. By defining three operation gates: forget gate, update gate, output gate, they not only can forget some useless information intelligently of prior input data but also are able to record more meaningful information inputted in the present moment. With the help of this 'cell' structure and three operation gates, LSTM models make a great achievement on capturing the useful information of the input of sentences sequence and predict the next words

by the storing information in natural languages processing tasks.

Due to the success of the usage of LSTM models in natural languages sentences, people consider applying these models to the research of programming environment like code completion systems because of the similarity between natural languages and programming languages. Concretely, a programming language is able to be considered as a special language just with more semantic and syntax restrictions. Programmers write source code in a code file, just like people write natural sentences in a text. In this case, a programming language is a special natural language, program expressions are similar to natural sentences and code files can be considered as natural language text files. Due to this similarity, it is evident that recurrent neural networks are also available to programming languages tasks like code completion and may have an exciting performance on the prediction of code. With the help of applying the RNNs to programming languages, we can gain insight into code structure, locality, and latent information and extract the deeper semantic and idiomatic meaning of the code easily which are unavailable to traditional statistic-based code completions systems.

1.4 Our Work

There has been some research about applying deep learning to code completion. These deep learning-based code completion models give a better performance of predicting next tokens. These works show that the usage of deep learning to complete code is available and worth to have a try. They also indicate that it is valuable to digging into deep learning-based code completion systems deeply to improve the prediction performance of next tokens. However, most of existing deep learning used code completion leverages a basic LSTM model to predict code rather than a totally special new proposed method for code completion. For example, the representation vectors for tokens used in LSTM models are randomly initialized comparing with the embedding methods for natural languages like Word2Vec[2] which are able to generate semantic-based representation vectors. Embedding methods, like the name of them, are used to represent discrete variables as continuous vectors. This technique is practically applied in many areas especially word embeddings for natural languages processing. Word2Vec, one of the most widely used word embedding methods, trains a neural network with a single hidden layer to generate the representation vectors of words which is more

meaningful and semantic-based than random initialization and improves the performance of word-based tasks.

In this work, we explore the embedding method for nodes in abstract syntax trees(ASTs) to improve the prediction performance of code completion. We propose an embedding method called ASTToken2Vec which is inspired by the idea of Word2Vec. the ASTToken2Vec model is just similar to Word2Vec, trains a neural network with the context information of a node in an AST and able to generate the embedding representation vectors for AST nodes. These representation vectors are semantic and syntax-based and contain more knowledge hidden behind ASTs. We consider this ASTToken2Vec model as a pre-trained model and use the AST nodes representation vectors to initialize the vectors fed in LSTM models. We integrate this ASTToken2Vec model and an LSTM model for the prediction of the next token. We evaluate the performance of our integration model with a JavaScript AST dataset[3] collected from open-source programs containing a total of 150,000 JavaScript files. In the experiment, we convert the ASTs in the dataset to sequences of training sample and feed these samples to our integrated model to predict next tokens in the given sequence. The way to convert ASTs in the previous work proposed by Chang Liu et al.[4] is, first, an AST is transformed to a left-child-right-sibling(LCRS) binary tree, and then, a deep first in-order traversal is applied in this binary tree to generate a visiting sequence of training samples and each training samples contains two tokens: *non-terminal, terminal*. We extend the AST transformation in the previous work, the way to process ASTs we use is similar to the previous work but a little bit different. We build our LCRS tree as a complete binary tree by padding special non-terminal token and our training samples contain four elements: *non-terminal, terminal, nonleaf-or-leaf, right-or-left*. The extended two bits of information is used to reconstruct the AST from the prediction sequence easily. The first bit is whether the non-terminal token is a leaf node or a nonleaf node in the LCRS complete binary tree, and the second bit is whether the non-terminal token is the right child of its parent node or left child. The overview of our model illustrates in Figure1.2

In the experiment, we evaluate both ASTToken2Vec embedding method by representation vector visualization and ASTToken2Vec LSTM integrated model for the prediction of next tokens with the same JavaScript AST dataset[3]. We also analyze in which case, our integrated LSTM model is more possible to give the correct prediction of the next terminal token. From the result of the evaluation, we draw a conclusion that the represen-

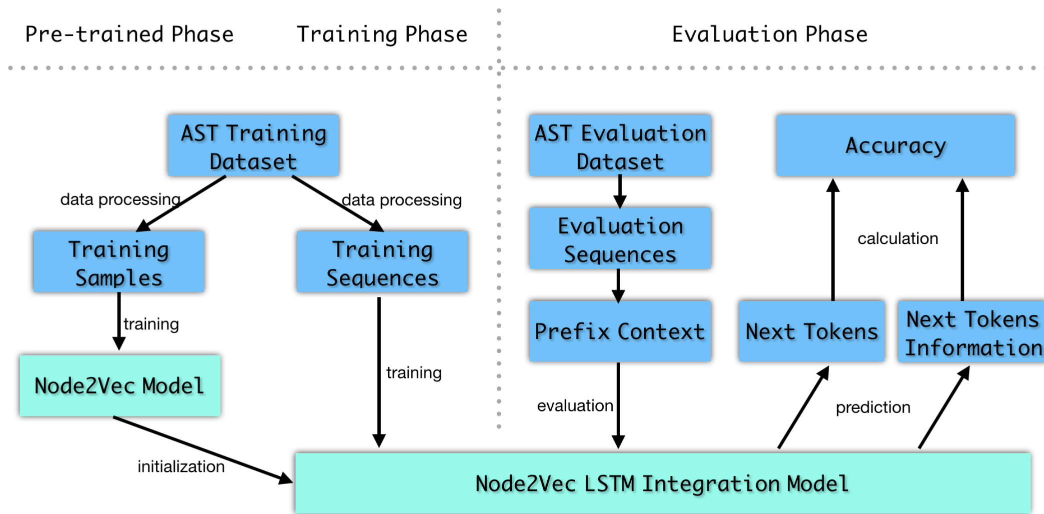


Figure 1.2: Overview of ASTToken2Vec LSTM integration model

tation vectors generated by ASTToken2Vec model are semantic-based and ASTToken2Vec could be used to improve the performance of the next token prediction as a pre-trained model for ASTToken2Vec LSTM integrated model.

Chapter 2

BACKGROUND

Due to the obvious advantages of code completion systems, there is more and more research about code completion to improve its performance in recent years. Hindle et al.[5] explore how to use a widely adopted *n-gram* statistical language model for code completion and provide empirical evidence which is able to prove that programming language code is even more repetitive than natural languages. Their work improves the complete capability of exciting completion engine for Java in Eclipse. Nguyen et al.[6] propose generative models of natural source code with hierarchical structure and a distributed representation of source code element. They also leverage compiler logic and abstractions to improve their generative models. Tung et al.[7] extends the state-of-the-art *n-gram* approach by incorporating semantic information into code tokens. Their model is called SLAMC, a novel statistical semantic language model for source code which is able to model the regularities of programming languages. These three works are based on the *n-gram* model which enable to show that the *n-gram* model have a great predicting performance of code suggestion and completion.

Comparing with *n-gram* model, a probabilistic programming language grammar-based method[8] is proposed to extract code idioms from the existing written code files in software projects in 2014 by Allamanis et al. They present a statistical nonparametric Bayesian probabilistic tree-based system for mining code idioms. With the semantically meaningful resulting-idioms of this system, programmers can write idiomatic code more efficient or use libraries which programmers may not be familiar with. In 2015, Allamanis et al.[9] extends their model[8] with a similarity embedding method which can extract more semantic information about the tokens. Their extension work

is more about method naming problem and the suggesting name for classes. Liang et al.[10] focus on learning programs for multiple related tasks with a few training samples for each learning task. They propose a nonparametric hierarchical Bayesian model which is able to share the statistical information across multiple tasks for code completion.

Bielik et al.[11] introduce a generative model for code. This model is called probabilistic higher-order grammar (PHOG) which is able to capture the rich context information between tokens by allowing conditioning of a production rule. Raychev et al.[12] explore how to apply the decision tree to learn a general probabilistic model of code. They create a domain-specific language(DSL) over abstract syntax trees(ASTs) called TGEN which is able to encode an AST to a specific language context and be fed into the decision tree. They also propose a special decision tree called DEEP3 which can make predictions about new programs leveraging the dynamically computed context in the ASTs encoded by the TGEN model.

In the recent year, deep learning-based models have a great achievement in both industries and research area, especially RNN models[13] for natural language processing tasks. However, due to the problem of gradient vanishing and long-term dependencies, traditional RNN models do not work well in some cases like the current prediction depend on the far previous information. Hochreiter et al[1] proposed a special variant of RNN models which is called the long short term memory model(LSTM) with a special cell hidden state and three operating gates. The LSTM model enables RNN models to work better and two main problems of RNN model are solved well by it. This model is widely used in sequence-structure data like natural languages, speeches, and videos and achieves a great performance in many tasks. Due to the success of LSTM model and programming languages can be considered as peculiar kinds of languages just with more syntax constraints comparing with natural languages, source code files are same with languages texts in many ways. Tokens in a code expression can be regarded as words in a natural sentence. Applying deep learning models to handle the source code and predict the next tokens as suggestion attracts increasing interest recently.

Raychev et al.[14] explore how to apply RNN models to facilitate the task of code completion. They address the synthesizing code completions problem with APIs. and compare several statistical language models like *n-gram*. However, their work focus on training an RNN model with token sequence data and applying this trained model to predict next tokens based on the abstract object input of all prior context. Although with token sequences

data, it is much more direct and easier to apply RNN models to predict for code completion, the token sequences contain less structural information of source code file which is very useful and necessary for the prediction of tokens. Structural information of a source code file which is expressed as an abstract syntax tree contains significant knowledge about language syntax and language semantics. If an RNN model leverages this information in ASTs rather than learning from sequences of token directly, it can be trained better by the structural information in the AST training dataset and may achieve a better performance than a token sequence-based dataset. White et al.[15] propose two particular deep learning models and show the effectiveness of these two models on a corpus of Java projects. Nevertheless, same with research by Raychev[14], these two exploratory research works only consider the usage of sequences of tokens as the dataset but hard to leverage the structural information contained in ASTs directly.

Chang et al.[4] propose several LSTM-based models for code completion with an AST dataset. They leverage the ASTs by transforming the original AST to a sequence of training samples. They convert ASTs to a left-child-right-sibling tree which is a binary tree, and the in-order deep first traversal is applied on this binary tree so that a traversal visiting sequence can be generated. Each element in this sequence is a training pair which contains two tokens: a non-terminal token and a terminal token. Training pairs in the sequences are fed into several variants of LSTM models, and trained LSTM models are able to predict the next non-terminal and terminal tokens. Their work gives us inspiration about how to convert an AST to a sequence and how to train an LSTM model with AST dataset. However, The representation vectors of tokens are initialized randomly rather than embedding representation methods are applied which are widely used in natural language processing tasks as a pre-trained model like Word2Vec[2]. This is what we want to extend in our work, an embedding method which is able to generate syntax-based representation vectors for both non-terminal and terminal tokens.

In recent years, program generation with deep learning models has attracted great attention. The target of program generation tasks is to generate a meaningful sequence of code just like "there is a machine writing programs intelligently and automatically". Program generation tasks are similar to code completion in the theory, both of these two research try to leverage the previous contexts and complex structure of source code. Knowing research about program generation may inspire our code completion.

Matthev et al.[16] propose a deep learning-based model called neural attribute machines(NAMs) which is a logical machine for program generation in 2017. This model is able to learn the rules and constraints of the programming languages' grammar. With the specification of context-free grammar extended by attaching attributes to the non-terminal tokens' and terminal tokens' symbols, NAMs model can extract the constraints of the underlying grammar and integrate these constraints with RNN models to generate source code with a lower grammar error. This idea is quite instructive to our code completion tasks. By specifying the attributes grammar (AG) and integrating its language's constraints with RNN model, the code suggestions would have a lower possibility of grammar error. However, the grammar constraints machine limits the application scope of our code completion. A basic deep learning-based code completion system is able to be applied in all programming languages. If grammar constraints of a special programming language (e.g. Java, C++, etc.) is employed in code completion models, these models can only work for this special programming language. It loses its extensive application.

Due to the similarity between programming languages and natural languages, the research about using deep learning for natural language processing tasks also needs the people who research learning model-based code completion to pay attention to. For example, Tomas Mikolov et al.[2] propose an embedding model which is called Word2Vec to generate the semantic-based embedding representation vectors for words in natural sentences. Word2Vec model is widely used in most natural language tasks and has a good achievement of performance improvement. It also gives people many ideas to explore how to represent the original data in a better way. Our embedding model for AST nodes is also inspired by this pre-trained model.

Another research direction of natural language processing which may give us inspiration is the extension of basic LSTM models. The traditional LSTM models are chain-structured models so that they can be applied in a sequence-like dataset which is the most frequent data format of natural language processing tasks. The input of traditional LSTM models is a sequence and the output of them is also the elements in the feeding sequence. The exploration of LSTM models' structure gives us the idea about how to invent variants of LSTM models to expand a more wider scope of their applications. For example, KaiSheng Tai, et. al.[17] propose a tree-structured LSTM model to improve the semantic representations of natural languages. In their paper, they expand the basic chain-structure LSTM model to a tree-structure

LSTM. In their model, each unit at every moment would have more than one input rather than only single input of traditional models, these inputs are the calculation results of its children units. In this way, A tree-structured LSTM model is created and it can make use of the syntax tree of natural language directly. From their experiment, this tree-based model can discover more implicit semantic knowledge hidden behind the natural language sentences. Due to the improvement of the input structure of LSTM models, this tree-structured model can be applied in many tree-based datasets not only syntax trees of natural languages and is able to increase the performance of effectiveness. Due to the programs written by programming languages can also be parsed to an abstract syntax tree, it is worth to explore whether it is possible to apply tree-structured LSTM models to code completion system with an ASTs dataset.

Chapter 3

PRELIMINARIES

In this chapter, we first discuss the advantages and disadvantages of each data format we use as the dataset in our code completion models especially abstract syntax tree(AST), the main data format in our dataset. Second, we introduce how to convert an AST to a sequence of training samples which is able to be fed into an LSTM model. The transformation we employ is inspired by the previous work proposed by Chang et al.[4]. We extend their work about AST transformation so that our transformation is much friendly for the AST reconstruction from the predicting sequence. The training samples in the sequence is also different from training pairs in the previous work by adding two more bits of information about the non-terminal token. Then, we explain how to reconstruct an AST from the predicting sequence with two bits of information in each training samples. We use an example to illustrate the AST format of a program, how does its left-child-right-sibling binary tree look like and all the training samples generated from this program. Finally, we propose a trick named identifier renaming to improve the performance of the next identifier prediction.

3.1 Data Format

Using the different format of programs as the dataset has a significant influence on the performance of deep learning-based code completion system, choosing a favorable type of data can make the code completion system work more efficient or have a better accuracy performance. There are several kinds of the code format like source code text, sequences of tokens and AST. As

for code completion tasks, all of these three formats are available and worthy to have a try. In this section, we will introduce several kinds of code format and discuss the shortcoming of each format. We will also explain the reason why we choose AST as the main data format in our dataset.

3.1.1 Source Code Text

Source code texts are the most common and straightforward format of programs, they can be used as a training dataset for learning based code completion directly. The benefit of using source code texts is that they are the most basic format of code files which means the source code text-based dataset can be collected from online repositories easily, and there are no need of lexers, parsers or other interpreter tools to create a dataset for learning models. Another good point of source code texts usage is that it is very easy to feed texts to an LSTM model because of their sequence structure. There is no other necessary processing of data and models. After the model is trained well, during the use phase, it can be used to predict next code programmers want to write in a real-time directly. Because of the simple sequence structure of source code texts, Using them as the data format is the easiest way to complete code with deep learning models. Nevertheless, it is plain work to find the obvious disadvantage of source code texts usage. Because of its simplicity, code texts do not contain enough information about program files which, however, has a significant influence on the performance of code completion. For example, a source code text is just a sequence of token programmers type, there is no program analysis(lexers or parsers) applied on this original text to attach some information about each token like the type of tokens (identifier, literal-string or other types) or the structure information of code snippets which can be obtained from the AST generated by parsers. But both of these two information plays a significant role in predicting the next codes.

Due to the lack of type information of token and structural information of programs, most of deep learning-based code completion systems do not use source code texts as the data format of the dataset even they are the easiest way to implement. In our model, we also discard the idea of using source code texts.

3.1.2 Sequence of Tokens

For the most modern compilers, there are several steps to process code after programmers write down the source code as a text so that the program can be checked and executed by computers. Lexical analysis is the first step to analyze code, it is a process of converting a source code text into a sequence of tokens. The tool that performs the lexical analysis is called a lexer. The output of a lexer is a sequence of tokens and the elements in it are not simple tokens but contain the type information of tokens themselves. Lexers can analyze what kind of type a token belongs to and enclose it to each token to create a type-labeled token in a sequence automatically. This type of information contains more semantic knowledge of code which is serviceable to enable code completion systems to predict the next code more precisely. Choosing the sequence of tokens as the data format of the dataset for deep learning models is a better idea than source code texts usage due to the lexical token contains type information which we can be sure that it would help deep learning models achieve a better performance of the code completion task. Another advantage of lexical token sequences is that it is still sequence-structure which means they are easy to be applied to the LSTM based model without too much change of the model structure.

In order to use the sequence of tokens as the dataset, we first need to covert the source code texts to sequences of tokens to train models. After models predict the next tokens in the input sequence, this sequence is converted back to a source code text to give an intuitive suggestion result for programmers. The shortcoming of lexical token sequences is similar to source code texts, sequences of tokens only have type information analyzed by lexers but still do not contain structural information of code snippets. This information covers some more available knowledge like programming syntax restriction and language grammars which is more useful for deep learning-based code completion systems. Due to this shortcoming and the goal of code completion performance improvement, we do not use the sequence of tokens as our data format of dataset either.

3.1.3 Abstract Syntax Tree

After lexers convert the source code text to a lexical token sequence, parsers in compilers could receive the sequence of tokens created by lexers as the input and convert it into an abstract syntax tree(AST) with the unambiguous

context-free grammar of programming languages. An AST is a tree data structure which is able to represent the abstract syntactic of the source code. It is usually considered as the result of the syntax analysis phase of a compiler and it contains rich structural information about the source code due to its hierarchical tree structure. This information makes it is widely used in program analysis and program transformation systems.

AST is a tree abstract data structure which represents structural, content-related details about the syntax and grammar of the given programming language. There are two kinds of the node in an AST, non-leaf node and leaf node, and each node in the AST denotes a construct token in the source code.

A non-leaf node has a children list which represents all its children node of the non-leaf node. Each non-leaf node corresponds to a non-terminal token in context-free grammar specifying information in a program. For example, in JavaScript, a non-terminal token could be "FunctionDeclaration", "VariableDeclarator", etc. These non-terminal tokens declare what kind of functions or variables specified in the program. Other kinds of non-terminal tokens contain more knowledge about the structure and logical judgment of a program like "ForStatement", "IfStatement", "WhileStatement", etc. A leaf node does not have any children node which corresponds to a terminal in the context-free grammar of a programming language. It is always called a terminal node which represents code text and contains more knowledge about the content of a program. In JavaScript, for instance, the type of terminal tokens could be "LiteralString", "LiteralNumber", "Identifier" etc. Due to programmers can specify any string and numerical literals or variable and function names in a program, it is obvious that there are infinite possibilities for terminal tokens.

Figure 3.1 illustrates a simple JavaScript source code snippet, it contains a loop statement and will print "Hello World" for three times. The corresponding sequence under this code snippet is its AST format representation generated by a parser. Each element in the sequence represents a node in the AST. From this figure, we can find it is definite that each node at least has two elements: the index of the node and its type information no matter it is a non-terminal or terminal node.

A non-terminal node has a children node list which is an array of integers denoting the index of all its children node. For example, the second node is a "ForStatement" and it has four children nodes which are listed as "[2, 5, 8, 10]". One thing to emphasis here is that not all non-terminal tokens have a

children list. For instance, “BreakStatement” and “ContunueStatement” do not have any children nodes but they are non-terminal tokens in terms of the context-free grammar of JavaScript language. Other kinds of non-terminal tokens may have a children node list and sometimes may not have it like “ReturnStatement”.

A terminal node, on the contrary, does not have any node as a child. Furthermore, all terminal nodes have a “value” element to represent the value of the node. For example, the seventh node in the sequence is an identifier and value of this node is “i” which is the name of this identifier. Figure 3.2 is the visualization of the sample AST in Figure 3.1. In this tree, a node without a surrounding box (e.g. ForStatement, VariableDeclaration, etc.) denotes a non-terminal node. A terminal node (e.g. LiteralString, Identifier, etc.) is surrounded by a box.

```
for (var i = 0; i < 3; i++) {
    console.log("Hello World")
}

[ { "id":0, "type":"Program", "children":[1] },
{ "id":1, "type":"ForStatement", "children":[2,5,8,10] },
{ "id":2, "type":"VariableDeclaration", "children":[3] },
{ "id":3, "type":"VariableDeclarator", "value":"i", "children":[4] },
{ "id":4, "type":"LiteralNumber", "value":"0" },
{ "id":5, "type":"BinaryExpression", "value":"<", "children":[6,7] },
{ "id":6, "type":"Identifier", "value":"i" },
{ "id":7, "type":"LiteralNumber", "value":"2" },
{ "id":8, "type":"UpdateExpression", "value":"?++", "children":[9] },
{ "id":9, "type":"Identifier", "value":"i" },
{ "id":10, "type":"BlockStatement", "children":[11] },
{ "id":11, "type":"ExpressionStatement", "children":[12] },
{ "id":12, "type":"CallExpression", "children":[13,16] },
{ "id":13, "type":"MemberExpression", "children":[14,15] },
{ "id":14, "type":"Identifier", "value":"console" },
{ "id":15, "type":"Property", "value":"log" },
{ "id":16, "type":"LiteralString", "value":"Hello World" }, 0]
```

Figure 3.1: A JavaScript ’loop statement’ and its AST format

ASTs are the data format which is the most widely used in the traditional code completion engines. The Advantage of ASTs is pretty apparent: they contain a wealth of knowledge about the tokens and structure in the source code. An AST node contains ’type’ information and ’value’ information which can represent the content of tokens in the source code. And the position of the node denotes the structural information of code snippets, especially non-terminal nodes. Is it a terminal token or a non-terminal node?

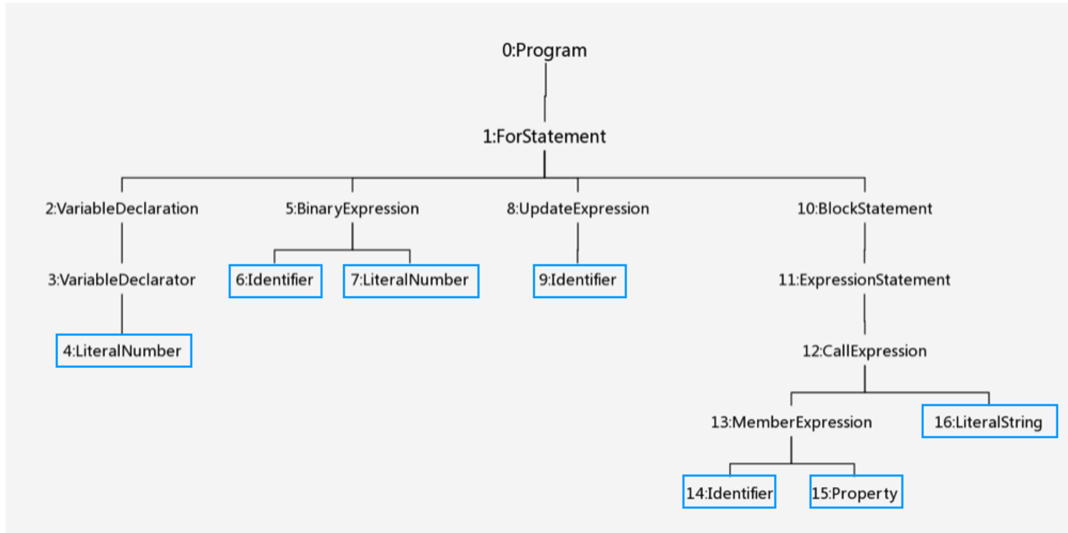


Figure 3.2: One AST sample of a JavaScript code snippet

how many children nodes does it have? What is the type of its parent node? All of this information is particularly useful in code completion tasks and can help deep learning models discover more implicit knowledge hidden behind the source code. Information given by ASTs is extremely important to help our deep learning-based code completion system to improve the performance of the next token prediction.

However, even the deep learning models for code prediction tasks can benefit from the extensive knowledge if the training dataset is composed of ASTs, the disadvantages of the AST is also obviously. First, we need to pay attention to the way to convert ASTs to sequences structure if we want to use ASTs as the training dataset. The reason for this transformation is that LSTM models are a sequence-based model which means the input and the output of them should be sequence-like data. However, an AST is just like its name, it is a tree structure and can not be fed into LSTM models directly. It is necessary to consider how to convert an AST to sequence-like data without losing too much structural information in the AST. Another shortcoming of ASTs usage is that ASTs are much more complex than both source code texts and the sequences of tokens. The generation of ASTs needs parsers in compilers. Different parsers generate the different format of ASTs which cause the way to convert ASTs to sequences also need to be changed at the same time and this change has a significant influence on the performance of

code completion. Finally, AST based deep learning code completion models cannot predict next token appearing after the current given code snippets directly. They can only predict the next elements in the sequence which is generated from an AST. So, the way to reconstruct an AST for this predicting sequence effectively and easily is also a question we need to handle if the AST dataset is used.

Even the shortcoming of ASTs usage is obvious because of their complex processing and parsers dependency. We still use ASTs as our dataset for our deep learning-based code completion system due to ASTs contain rich structural information. And the goal of our code completion system is the improvement of the code suggestion accuracy, we believe that the syntax knowledge hidden behind ASTs enables the system to achieve a better performance of code completion.

3.2 Data Processing for the LSTM Model

There are two parts of our code completion system and each part is a neural network model. A pre-trained embedding model called ASTToken2Vec which is introduced in the chapter4. A LSTM based model for code completion which is introduced in the chapter5. Both of these two models need a dataset to train. Due to the obvious advantages of ASTs, the format of our data is AST so that the rich structural information of source code can be used by our models. However, ASTs are tree structure and hard to be fed into either the LSTM model and ASTToken2Vec model. In this section, we explain how we convert the original ASTs to sequences of training samples for the LSTM model as the training dataset. The way to process ASTs to a huge set of training samples for ASTToken2Vec model training is declared in the ASTToken2Vec chapter4.

3.2.1 AST to Sequence Conversion

Due to ASTs are tree structure, in order to apply the chain-structure LSTM model in our code completion system, the first step of data processing is to convert the AST, a tree-like structure, to a sequence. We extend the way this transformation proposed in the previous work[4]. An AST is a multi-way tree structure, so we first convert the original AST to a left-child-right-sibling (LC-RS) tree which is a binary tree. During this conversion, we only consider

non-leaf nodes(also means non-terminal nodes) to build the LC-RS binary tree and leaf nodes(terminal nodes) only serve as an element in its parents non-terminal's children list.

The process of converting an AST, a multi-way tree, to an LC-RS binary tree works like this: first, using the root node of the original AST to create the root of the new binary tree. The root node of an AST is always a non-terminal node in terms of programming languages' grammar. Then, starting with this root, given a non-terminal node, its leftmost child in the original tree is used to make the left child of the given non-terminal node in the new binary tree, and its nearest non-terminal sibling to the right in the original tree is made the right child of the given non-terminal in the binary tree. During this binary tree build process, we ignore all terminal nodes and only record them as a children list of the element of their parent's non-terminals.

In the converting way described above, an AST can be converted to a binary tree. Next step, we build a complete binary tree based on generating LC-RS tree. For each non-leaf non-terminal node in the LC-RS tree, if it only has one non-terminal child node(either right child or left child), we give it a special non-terminal node *NT-EMPTY* leaf node to make sure all of the non-terminal nodes in the LC-RS tree have either two non-terminal child nodes or do not have any child nodes as leaves in the tree. For instance, if a non-terminal node in the LC-RS tree only has a right child node, we give it a *NT-EMPTY* non-terminal node to represent its left child. On the contrary, if a non-terminal node only has a left child node, we pad a right *NT-EMPTY* child node to make sure this binary tree is a complete binary tree. The reason why converting a general LC-RS binary tree into a complete binary tree is that it is much easier to reconstruct the AST from the predicting sequence with this *NT-EMPTY* node padding method. We declare the way to reconstruct AST with more details in the next section.

A complete binary tree is generated from the transformation above. In order to obtain a sequence, we apply a deep-first in-order traversal on this binary tree. When a non-terminal node in the binary tree is visited, we consider this visited node as the target non-terminal node and scan the children list of the target non-terminal to generate n training samples as the elements in the generation sequence on the basis of this target non-terminal node. n is equal with the number of children terminal nodes this visited non-terminal node has. If the target non-terminal node does not have any terminal child, we give a specified terminal node *TT-EMPTY* as its child terminal node. After all non-terminal nodes in the binary tree are visited, a traversal se-

quence is generated to build the dataset for deep learning model as the input of our deep learning model for code completion.

There are four elements in one training sample, the first two elements are about the content knowledge of ASTs: a target non-terminal token and a child terminal token of it. The last two elements are used to describe the structural information of the target non-terminal token: **node-or-leaf** and **right-or-left**. Both of these last two elements are bits information which means there are only two possible values of them: 0 or 1. "node-or-leaf" is used to declare whether the target non-terminal node is a leaf in the complete binary tree or not. 0 represents leaf which means there is no non-terminal child node and 1 represents it is not a leaf in the tree. The fourth-bit element is "right-or-left" which is used to represent the position relationship between the target non-terminal node and its parent node. If the target non-terminal node is the right child of its parent non-terminal node, 0 is labeled to "right-or-left". If the target non-terminal node is the right child of its parent non-terminal node, "right" will be labeled. On the contrary, the value of "right-or-left" is 1 if the target non-terminal node is the left child of its parent node.

So, the structure of a training sample looks like this: (*non-terminal, terminal, node-or-leaf, right-or-left*). From the perspective of the model prediction, these four elements are fed into the deep learning model every training moment and the model makes predictions about the elements in the next training samples in the inputting sequence. There are two different points with the way to process AST data described in the previous work[4][18]. We not only convert an AST to an LC-RS binary tree but also create a complete binary tree with *NT-EMPTY* padding. The second one is, in the previous work, the training samples are pairs which only contain two elements: non-terminal and terminal. We extend them with two more bits of information. The reason for our extension is that the AST reconstruction from the sequence is easier with this another two bits information.

Figure 3.3 illustrates an example of how to generate training samples from a target non-terminal node in an AST. In this example, "non-terminal 2" is the target node which is surrounded by a box in the figure. And three terminal nodes with underline are the children terminal nodes of the target non-terminal node. We generate three training samples on the basis of this target node. The first element in these samples is the target non-terminal node: "non-terminal 2", the second element which is the only difference between these three samples, is the different terminal child of the target

node: “terminal 1”, “terminal 2” and “terminal 3”. Because the target non-terminal node is a leaf node in the given binary tree, the third bit in these samples is “leaf”, and the fourth element is “left” due to the target node is the left child of its parent node.

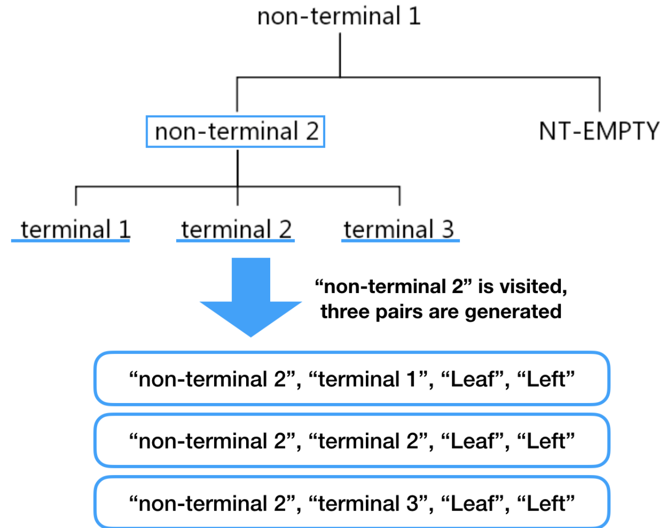


Figure 3.3: Training samples generated from a toy non-terminal binary tree

Figure3.2 illustrates the AST format of a JavaScript “loop printing” code snippets shown in the figure3.1. And figure 3.4 is the complete binary tree of the AST in the figure 3.2 after the conversion. On the basis of this binary complete tree, we visit each non-terminal nodes in a deep first in-order-traversal order and generate a sequence of training samples. Table 3.1 illustrates the generated training samples.

3.2.2 Conclusion

To summarise the AST processing in our work: There are three main steps in this AST transformation: First, an original AST is converted to an LC-RS binary tree. Then, we pad a special non-terminal node *NT-EMPTY* to build a complete binary tree. all nodes in this tree are non-terminal nodes in the original AST. Finally, a deep-first in-order traversal is applied in this complete binary tree so that each non-terminal node is visited to generate a sequence of training samples. Each training sample contains four elements.

Index	Non-terminal	Terminal	Node-or-leaf	Right-or-left
1	3:VariableDeclarator	4:LiteralNumber	Leaf	Left
2	2:VariableDeclaration	TT-EMPTY	Node	Left
3	NT-EMPTY	TT-EMPTY	Leaf	Left
4	5:BinaryExpression	6:Identifier	Node	Right
5	5:BinaryExpression	7:LiteralNumber	Node	Right
6	NT-EMPTY	TT-EMPTY	Leaf	Left
7	8:UpdateExpression	9:Identifier	Node	Right
8	13:MemberExpression	14:Identifier	Leaf	Left
9	13:MemberExpression	15:Property	Leaf	Left
10	12:CallExpression	TT-EMPTY	Node	Left
11	NT-EMPTY	TT-EMPTY	Leaf	Right
12	11:ExpressionStatement	TT-EMPTY	Node	Left
13	NT-EMPTY	TT-EMPTY	Leaf	Right
14	10:BlockStatement	TT-EMPTY	Node	Right
15	NT-EMPTY	TT-EMPTY	Leaf	Right
16	1:Forstatement	TT-EMPTY	Node	Left
17	NT-EMPTY	TT-EMPTY	Leaf	Right
18	0:Program	TT-EMPTY	Node	Left
19	NT-EMPTY	TT-EMPTY	Leaf	Right

Table 3.1: Training samples generated from the AST representation complete binary tree in the figure3.4

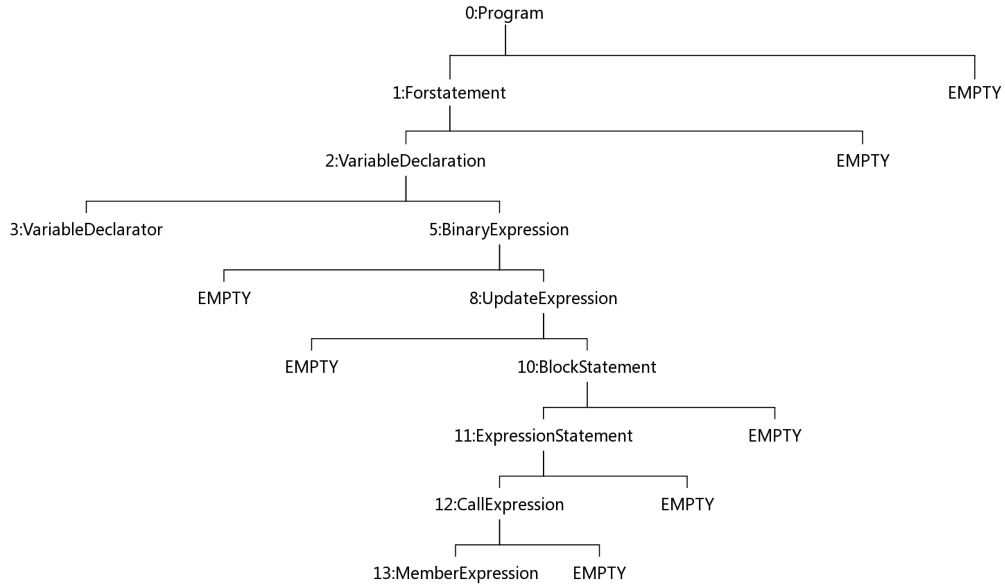


Figure 3.4: The complete binary tree of JavaScript 'loop printing' code snippets

This sequence is fed into the LSTM model to train the model. We will scan the entire original AST dataset to produce a tremendous number of training samples in terms of our AST processing regulation. More details of AST processing are introduced in the experiment chapter6. This AST processing is used to generate the sequences of training samples for the LSTM code completion model. The way to process AST for ASTToken2Vec model, our pre-trained embedding model for code completion is described in the chapter4 of ASTToken2Vec model.

3.3 AST Reconstruction

The original ASTs are transformed into sequences of training samples to build the dataset for the code completion model in terms of previous AST processing rules. With this dataset, we train an LSTM model with ASTToken2Vec embedding to predict next training samples in the given sequence as code prediction. After this prediction, a sequence with the new completed

code is generated. However, this sequence is not the final result of code completion due to code completion systems serve programmers and they cannot understand the completed code in the predicting sequence. This predicting sequence should be reconstructed back to an AST and this AST is parsed back to source code text as the final result of code completion. In the previous work of LSTM based code completion, there is no effective way to reconstruct. In our work, we propose a method to reconstruct the AST from the predicting sequence easily with our extension training samples in the sequence.

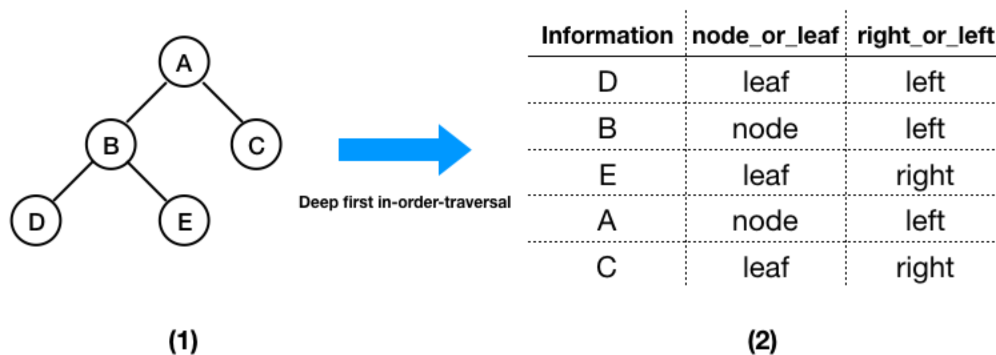
The way to reconstruct an AST from the sequence includes the following steps: First we reconstruct an LC-RS complete binary tree from predicting the sequence of training samples. Then we delete unnecessary padding non-terminal nodes ("NT-EMPTY" placeholder non-terminal nodes we pad to build complete binary trees during the AST processing phase) in the LC-RS complete binary tree. Finally, we convert this LC-RS binary tree back to an AST. The second and third steps are quite easy to be understood. We introduce how to reconstruct a binary complete tree from the predicting sequence with more details in this section.

In order to reconstruct the AST, the last two elements: *node-or-leaf, right-or-left* in the training sample are used to locate the position of non-terminal nodes in the LC-RS binary tree with the help of a stack auxiliary tool. Stacks are an abstract data structure widely used in computer programming which is a linear data type and serves as a collection of elements following the "last in, first out"(LIFO) principal operations. The reconstruction method would be implemented with the help of a stack.

This is how does our reconstruction stack work: Training samples in a predicting sequence are pushed into this stack and elements in the stack are reduced recursively. Concretely, we push each training sample in the predicting sequence into the stack as an element, after a sample is pushed into the stack, the top element in the stack is checked by our reduction regulation: If the third bit in the element, "node-or-leaf", is "leaf" and the fourth bit "right-or-left" is "left", reduction of the stack operation occurs: three elements on the top of the stack are popped and build a binary subtree which contains three nodes: The first element on the top of the stack poped is the right child node of the binary subtree. The second element is the root node of the subtree and the third one is the left child node. After this subtree reconstruction, we push this subtree back to the stack as the top element. The two bits information: "node-or-leaf" and "right-or-left" is the

same with these two bits information of the root node in the subtree. Then, we continue to check whether stack still conforms to the reduction regulation. Tf the reduction condition meets, we continue to do reduction operation in the stack described above and a larger subtree is created recursively until there is no necessary to reduce. Then, the next sample in the predicting sequence is pushed to the stack and the reduction operation keep working recursively until there is no sample in the sequence. When the predicting sequence is empty, the last element in the stack is the AST we reconstruct.

We use a concrete example to illustrate how to reconstruct a binary tree from a sequence of samples. Figure 3.5 shows a simple AST and it deep first in-order-traversal visiting sequence. When a node in the AST is visited, two bits of information *node-or-leaf, right-or-left* are added to generate samples.



Information	node_or_leaf	right_or_left
D	leaf	left
B	node	left
E	leaf	right
A	node	left
C	leaf	right

Figure 3.5: (1) is a toy complete binary tree and (2) is the sequence of samples generated in terms of AST processing regulations

Figure 3.6 illustrates how to reconstruct an binary tree from the sequence of samples in the figure 3.5. We push the samples in the sequence into the stack or reduce the stack depending on the bits information of the top element in the stack. In this example, we first check whether the elements "D" and "B" are a "leaf" node and "left" child of their parent node. Due to they do not meet the reduction regulation, we push them into the stack. When the token "E" is on the top of the stack, the reduction occurs and three elements on the top of the stack are popped to build a subtree. Then we push this subtree back to the stack and the token "A" is pushed to the stack due to it does not mean the reduction regulation. "C" is the last sample in the sequence and we use 'C', "A" and the subtree in the stack to construct a

new tree and this tree this the final result of our reconstruction.

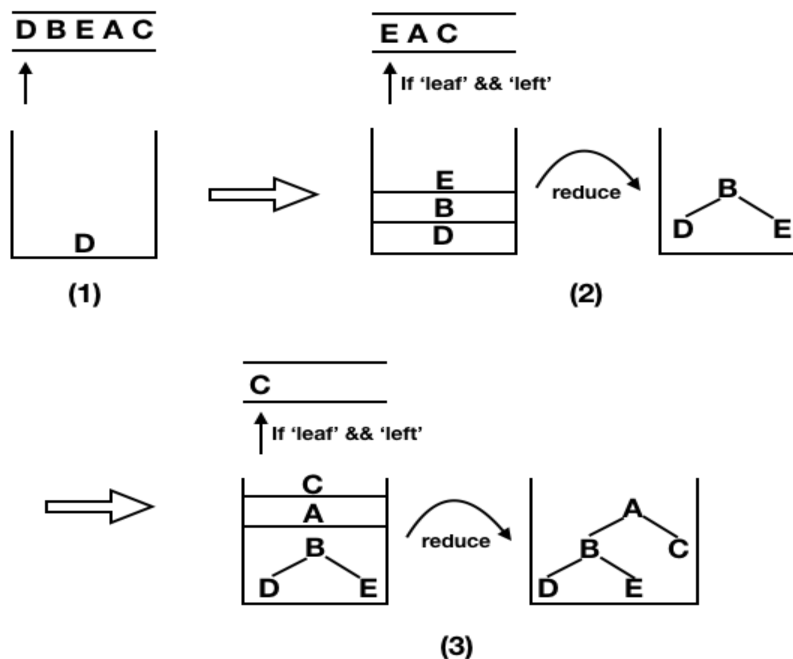


Figure 3.6: AST reconstruction from a sequence

3.4 Identifiers Renaming

From the application perspective of code completion systems, the most useful prediction that the programmers really want might be the prediction of *Identifiers*. *Identifiers* are the names of both *variables* and *functions*, the prediction of them can not only observably reduce the possibility of writing errors, but also inspire the ideas of programming. From the dataset perspective, same as above, They also play a significant role in the terminal tokens. According to the statistics of terminal tokens number in the training dataset we used, the proportion of *Identifiers* have the largest percentage with 43 percent. From both application and experiment ideas, if there are some methods can make the predicting of *Identifiers* have a better performance, it can be sure that not only the users of our code completion system

have a much better user experience, but also the performance of prediction is going to increase in our experiment.

Renaming identifiers can help us to achieve a better prediction performance of identifiers. In theory, there are infinite kinds of identifiers that may occur in the source code because programmers can specify identifiers with any names they want just follow the regulation of the programming language. This huge size of candidate identifier vocabulary makes the deep learning models become quite cumbersome and causes a lower performance of prediction. Usually, the size of the token vocabulary is specified by a not too huge number according to the frequency of occurrence of each token in the dataset, other infrequent tokens are represented by a special *UNKNOWN* token. This specification makes our model unable to predict some unusual terminal identifiers which occur in the dataset infrequent. Renaming identifiers can help us to solve this problem.

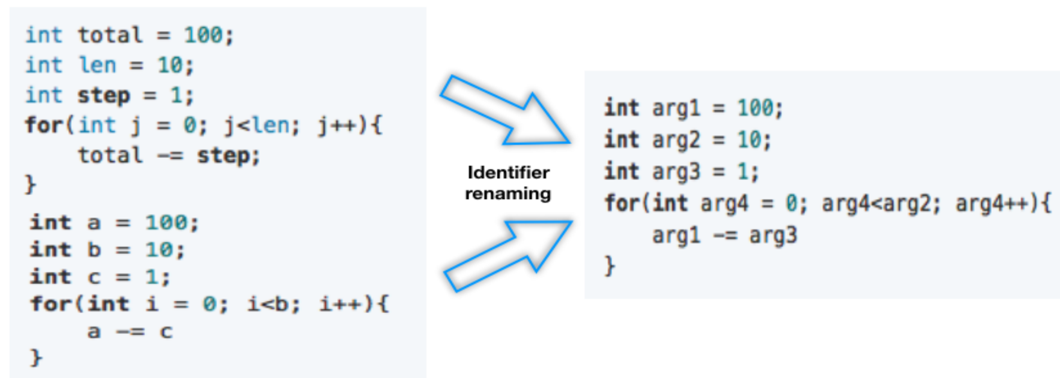


Figure 3.7: Identifier renaming

The way to rename identifiers is that we rename all identifiers as “arg1”, “arg2”... which means we give each identifier in the source code text an order-including uniform name. Concretely, we use a mapping dictionary to record the order of appearance of each identifier in a code file, and swap the identifiers’ name with this order information and use this renamed data to train our deep learning model. The mapping dictionary is recorded in order to get the original identifiers’ name. When the model is evaluated with some test queries, for the identifier completion, the model can predict the appearance order of identifiers in the evaluation queries and we use the mapping dictionary to find the real name of these prediction identifiers. With identifiers renaming, the number of identifiers type have a great reduction.

According to our statistics of experiment dataset, there are near 485,000 kinds of identifiers in the dataset before renaming, and only 8,150 kinds of identifiers after renaming. With identifiers renaming, we can predict infrequent identifiers and achieve better performance. Figure3.7 shows how to rename two code snippets to a single code file.

Chapter 4

ASTToken2Vec EMBEDDING

In this chapter, we discuss the ways to represent data, we introduce several common methods of data representation including one-hot encoding and word2vec embedding. We also talk about the representation method used in the previous work of deep learning-based code completion. Finally, we present the embedding model proposed by us: a pre-trained embedding model for AST nodes called ASTToken2Vec. ASTToken2Vec is able to generate syntax-based representation vectors. Due to there are two kinds of tokens in ASTs: non-terminal and terminal. We design two application variant of our embedding model: ASTToken2Vec for terminal token abbreviated as TT2V and ASTToken2Vec for a non-terminal token called NT2V. We introduce the mechanism and details of our embedding models and how to integrate it with our LSTM code completion to improve the predicting performance.

4.1 Data Represention

Most of machine learning models, especially deep learning models, have a requirement for the format of input data. Due to the input layer of deep learning models is a vector, the feeding data of them should also be vector format. However, almost all data we face in real life are not vectors(natural words, speeches, images, etc.). Methods to encode the original data to vectors for deep learning models are called data representation. In our task, the data we use is nodes in ASTs which are a composite data structure contains information about tokens(the type of a token, children of a token, etc). And due to the similarity between tokens in code and words in natural languages,

we consider using data representation methods for natural languages. In this section, we introduce several representation methods widely used in machine learning models and how to apply them to AST nodes.

4.1.1 One-hot Encoding

The simplest method to represent AST nodes is *one-hot encoding*. This method gives a unique number of each token in the training dataset vocabulary. Then it creates a special representation vector to distinguish each token from every other token in the vocabulary. The length of these vectors is equal to the size of the token vocabulary. All elements in a vector consisting of zeros with the exception of a single one in the position of the unique number of a token to identify it. In other words, the position of one in the vector is equal to the unique number of the token which this vector wants to represent.

The most intuitive strength of one-hot encoding is that it is the most uncomplicated way to represent tokens. It is easy to design and modify and users do not need to pay much attention to it. However, the disadvantage of one-hot encoding is also equally obvious: The length of the representation vector is out of control. It must be equal to the size of the token vocabulary. If it is quite large(There are infinite kinds of terminal tokens theoretically), the length of one-hot encoding representation vectors is going to become very huge and it may cause a dimension disaster for deep learning models. Another shortcoming is almost all elements in vectors are 0s but only single 1 which means the content of each vector is the same only the order of 1 in the vector is different. It causes that the one-hot encoding vectors can only distinguish each token in the vocabulary but they are not able to represent the meaning of nodes which, however, is much more significant for token representation. One-hot encoding is the simplest way to identify tokens, but not a good way to represent them.

4.1.2 Embedding Representation

Due to the disadvantages of one-hot encoding, it is not directly used in machine learning models. Rather than using one-hot encoding directly, learning-based embedding representation has become one of the most popular representation methods which can solve the two problems of one-hot encoding: The length of the vector generated by the embedding learning method can be defined by users as a hyper-parameter comparing with the uncontrolled

length of the one-hot encoding representation vector. The elements in the embedding vectors not either 0 or 1 but any float numbers which enable vectors to represent the meaning of data. Embedding methods for natural language processing tasks are a more efficient solution which is capable to capture the context information of target natural words like syntactic similarity, semantic similarity, and the relationship between words. With this similarity and relationship knowledge, representation vectors generated by learning the embedding method become more meaningful and are able to improve the performance of machine learning models.

Tomas et al.[2] propose a learning-based embedding representation method for words in natural sentences in 2013 which is still widely applied in many machine learning model based natural language processing tasks as a pre-trained model. Our embedding model for AST nodes is also inspired by their method.

4.1.3 Word2Vec Embedding

The embedding learning model proposed by Tmoas et al.[2] is called Word2Vec which means it is a method which is able to encode the words to embedding representation vectors with more semantic meaning by the usage of the context information in a sentence of the target words.

In order to generate the semantic-based representation vectors of words, Word2Vec[2] model declares a basic hypothesis that "if two target words have a similar meaning, their context would also be similar". A target word means the word that the generation embedding vector wants to represent, the context is n surrounding words of the target word in a sentence, n is a hyper-parameter which can be specified by users to control the size of surrounding context Word2Vec model considered. For example, let's assume that there are two sentences in the whole language corpus: "I really like eating apples because it is delicious."; "I really like eating bananas, it is tasty". We consider two words in these sentences: "apples" and "bananas" as target words to generate their representation vectors. It is easy to find that the contexts(surrounding words) of these two target words have a high-level similarity. In terms of the hypothesis of Word2Vec, these two target words: "apples" and "bananas" are considered that they are similar to each other. If we visualize the representation vectors of these two words generated by Word2Vec, the location of vectors near to each other. In other words, Word2Vec leverages the context similarity of the words in a sentence

to generate a more significant representation vector.

For the purpose of semantic-based representation generation with surrounding contexts, Word2Vec puts forward two neural network architectures: *Skip-gram* and *Continuous Bag of Words (CBOW)*, both of these two models contain three neural layers: an input layer, a single hidden layer, and an output layer. The size of the hidden layer is much smaller than the input and output layer. The input of *Skip-gram* model is the one-hot encoding representation of a target word and the output of it is the addition of the one-hot encoding representation of surrounding context words of the input word. On the contrary, the input and output of *CBOW* is opposite to *Skip-gram*. In order to train the Word2Vec model, sentences in the corpus are processed to a huge number of training pairs. A training pair looks like this: "(target-x, context-y)" for *Skip-gram* and "(context-x, target-y)" for *CBOW*. After models are trained well, for *Skip-gram*, the embedding representation vector can be obtained by extracting the value in the hidden layer after feeding the one-hot encoding representation of a target word into the model. Due to the size of the hidden layer can be specified as a hyper-parameter n, with Word2Vec, the dimension of the representation vector decreases from the length of input layer which is equal to the size of word vocabulary to the length of the hidden layer. This reduction is quite useful for some natural language processing tasks with huge size of the vocabulary and is also helpful for AST node dataset because of the infinite possibility of terminal tokens. Furthermore, these representation vectors contain model semantic and syntactic knowledge of a natural language hidden behind the sentences due to the model leverages the surrounding context information of the words. The evidence of this semantic information included is that the distance between embedding vectors is able to describe the relationship between words quite robustly and well. If two words are similar to each other, their representation vectors are also near to each other in the high dimensional space.

4.2 AST Nodes Embedding

In this section, we first discuss the way to initialize the vectors in the previous work. Then, we introduce details about our embedding model: ASTToken2Vec. Its architecture, how we define the context of an AST node and its joint loss function.

Chang Liu.at.al[4] used vanilla LSTM models to predict next non-terminal

and terminal tokens. However, they initialize the embedding representation vectors of nodes in ASTs randomly rather than employ any pre-trained models. Random initialization is not a good way compare with the embedding initialization because a groovy pre-trained embedding model can generate more meaningful embedding vectors which are able to speed up the model training and improve the performance of the deep learning model. In this case, we propose a neural pre-trained embedding model which can be used to generate the semantic and syntax-based representation vector for AST nodes called ASTToken2Vec. This model not only is available in our code completion task but also can be applied to other deep learning-based source-code-related tasks (Inferring coding conventions, code migration.etc.[19] which employ AST based datasets.

Because of the excellent performance of Word2Vec, it is widely used as a pre-trained model for most natural language processing tasks. The idea of our ASTToken2Vec model is inspired by Word2Vec embedding, ASTToken2Vec is also a neural pre-trained model for nodes in ASTs with learning method and is able to be used to generate embedding representation vectors of AST nodes. In order to do that, we have a basic hypothesis of ASTToken2Vec which is same to Word2Vec's: we assume that if two nodes in an AST have a similar context, the meaning of these two nodes also has a high-level similarity. The meaning of AST nodes is not as obvious as natural words, it is the semantic and syntax of the token specified by the context-free grammar of programming languages. The context of an AST node we defined is the surrounding AST nodes. As for the architecture of our model, similar to Word2Vec, it is a three layers neural network model with a single hidden layer.

There are three points of our ASTToken2Vec that are different from Word2Vec. The first one is that the way we define the context. Due to natural sentences are one dimension linear structure, it is easy to define the context of a word as some surrounding words in a sentence for natural languages, Nevertheless, the data in use by our model is AST nodes and AST are tree data structure. So the context of a node in an AST we define is some tokens surrounding with the target node but not same as the context in a natural sentence due to the special 2-dimensions tree structure of an AST. "Surrounding context" here is not some preceding tokens and incoming tokens, we define it as some parent nodes and some children nodes surrounding with the target node in an AST. Namely, we assume if two nodes in an AST have similar parent nodes and similar children nodes, the meaning of these

two nodes also have a high-level similarity. Due two there are two kinds of tokens in a programming language, we define two different contexts for a target node: Non-terminal context and terminal context.

The second different point of our ASTToken2Vec is that we create two ASTToken2Vec models: non-terminal node to vector(NT2V) model and terminal node to vector(TT2V) model due to there are two kinds of the node in ASTs: non-terminal nodes and terminal nodes in terms of programming languages grammar. NT2V is able to generate the representation vectors for non-terminal nodes and TT2V is for terminal representation vectors generation. Non-terminal nodes are the non-leaf nodes in AST which are more about the control statement and structure of the source code like *ForStatement* and *IfStatement*. The number of non-terminal tokens is specified by the context-free grammar of programming languages. On the contrary, Terminal nodes are the leaf nodes in ASTs which are able to record the content of the source code like *LiteralString* and *Identifier*. The huge difference between non-terminal and terminal is the reason we define two different ASTToken2Vec models.

The third difference is the architecture of the model. Both our NT2V and TT2V models extend the *Skip-gram* model which means the input is a target token and the output is the context of the input token. However, because the number of contexts is two for each model, These two contexts make both NT2V and TT2V models have two output layers to represent the non-terminal context and terminal context of the input target node separately rather than one single layer to represent the context in *Skip-gram* model. For the NT2V model, the input is a target non-terminal token and the two outputs are non-terminal context and terminal context of the input non-terminal. TT2V model is contrary to NT2V, the input of it is terminal tokens.

4.2.1 Contexts of Non-terminal Tokens

The ASTToken2Vec model for non-terminal tokens to vectors is abbreviated as NT2V which is able to generate embedding representation vectors for non-terminal tokens. NT2V employs both the terminal context and the non-terminal context of the target non-terminal tokens and we define these two contexts as follows.

Non-terminal context

The non-terminal context for a non-terminal token means the surrounding non-terminal node of it in an AST. Concretely, we define the n parent non-terminal nodes of the target non-terminal as the parent non-terminal context. And all non-terminal child nodes of the target non-terminal as the child non-terminal context if it has. The combination of parent non-terminal context and child non-terminal context is used to represent the non-terminal context of a target non-terminal.

If a target non-terminal node does not have any parent node(the root node in an AST), we use the special non-terminal node: *"NT-EMPTY"* to represent its parent non-terminal context which has been used as the padding non-terminal to build a complete binary tree in the AST processing phase. If a non-terminal node does not have any non-terminal children node(all its children nodes are terminal nodes), The child non-terminal context is ignored and the non-terminal context of this target non-terminal is only its n parent non-terminal nodes. Here, n is a hyper-parameter which declares the scope of the parent non-terminal context employed by the NT2V model. If n is relatively small, it means NT2V does not consider the surrounding non-terminal tokens which are far from the target node as the non-terminal context. On the contrary, if n is relatively large, more surrounding non-terminals are covered as the non-terminal context.

Terminal context

Terminal context of a target non-terminal is all terminal nodes surround the target node. Specifically, surrounding terminal nodes are all children terminal nodes of the target node. We specify these terminal nodes as the terminal context of a non-terminal node. If a non-terminal node does not have any terminal children nodes(all its children nodes are non-terminal tokens), we use the special terminal token: *"TT-EMPTY"* to declare the terminal context of this target non-terminal node is empty.

Figure4.1(1) is an example of partial AST which is able to illustrate the terminal context and the non-terminal context of a target non-terminal node. Nodes whose name starts with "NT" are non-terminal nodes and nodes whose name starts with "TT" represent terminal nodes. In this particular AST, the target non-terminal node is 'NT-4' which is surrounded by an oval. Non-terminal nodes surrounded by a rectangle are the non-terminal context of

the target node including “NT-2” and “NT-1”. Terminal nodes: “TT-1”, “TT-2” and “TT-3” which have an underline mean the terminal context of the target node. Hyper-parameter n here is specified as two.

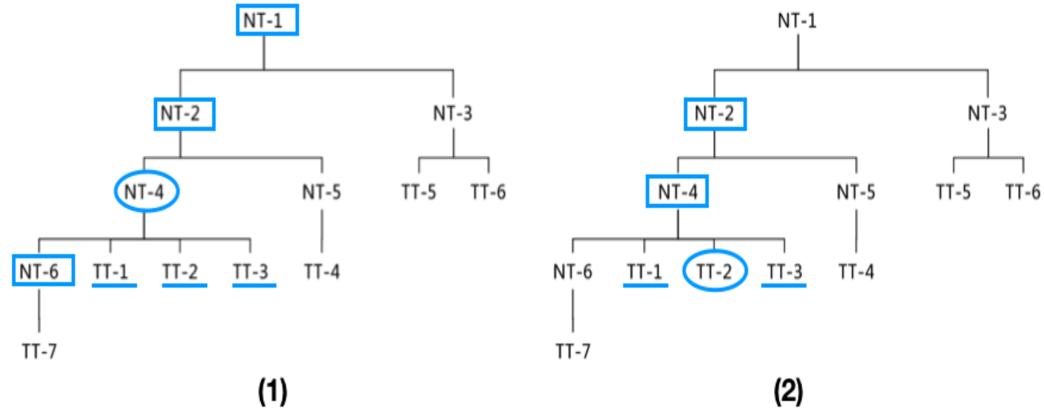


Figure 4.1: Non-terminal and terminal context for nodes. (1) is the context for non-terminal nodes, (2) is the context for terminal nodes

4.2.2 Contexts of Terminal Tokens

TT2V is the abbreviated name of ASTToken2Vec model for terminal tokens to vectors. TT2V model is able to generate embedding representation vectors for terminal nodes. Same with NT2V model, TT2V uses both the non-terminal context and terminal context to generate the representation vectors.

Non-terminal context

We only consider the non-terminal context of a target terminal node as the parent non-terminal context: n parent non-terminal nodes because there is no children node of a terminal node. Here, same with the NT2V model, hyper-parameter n is used to control how many surrounding parent non-terminal tokens are considered as the non-terminal context. Due to a terminal node must have at least one parent non-terminal node, we do not need to specify any “NT-EMPTY” non-terminal as a padding token.

Terminal context

Because a terminal node in ASTs does not have any children nodes, we define the terminal context of a target terminal node as m neighbor terminal nodes in an AST. Neighbor terminal nodes mean the other terminal children nodes of the non-terminal node which is the parent node of the target terminal node. Here m is a hyper-parameter which is used to specify the size of the terminal context. A relatively small m means TT2V model does not consider too many surrounding terminal nodes as the terminal context.

If a target terminal node does not have any neighbor terminal nodes which means its parent non-terminal node only has it as the single terminal child, in this case, we use the special terminal node: *"TT-EMPTY"* to represent an empty terminal context.

Figure4.1(2) shows an example of partial AST to illustrate the non-terminal context and terminal context of a target terminal node. In this example, terminal node "TT-2" is the target node surrounded by an oval. Non-terminal nodes: "NT-2" and "NT-4" which are emphasized by a rectangle represent the non-terminal context of the target node. The neighbor terminal nodes of the target node are "TT-1" and "TT-3" which are the terminal context of the target node. In this example, hyper-parameter n is specified as two and m is equal to one.

4.3 Data Processing

In this section, we explain how we generate training samples from ASTs for model training. Training samples for ASTToken2Vec model is different with training samples generated for LSTM based code completion introduced in chapter3 but both of them are generated from the same JavaScript AST dataset.

There are two kinds of training samples we generate, non-terminal training samples for NT2V model training we name it as NTS and terminal training samples for TT2V model training called TTS. For a given AST, we generate x NTS and y TTS in terms of processing regulation. x is the number of non-terminal nodes in the AST and y is equal to the number of terminal nodes in the AST. We scan all ASTs in the dataset and generate a set of NTS for NT2V model and TTS for TT2V model.

Index	Target non-terminal	Non-terminal context	Terminal context
1	NT-1	[NT-EMPTY]	[TT-EMPTY]
2	NT-2	[NT-1, NT-4]	[TT-EMPTY]
3	NT-3	[NT-1]	[TT-5, TT-6]
4	NT-4	[NT-1, NT-2, NT-6]	[TT-1, TT-2, TT-3]
5	NT-5	[NT-1, NT-2]	[TT-4]
6	NT-6	[NT-2, NT-4]	[TT-7]

Table 4.1: Training samples for NT2V model

Structure of NTS The structure of a NTS is a tuple which includes three elements: *(target non-terminal token, non-terminal context, terminal context)*. A target non-terminal token is the training x which is going to be fed into NT2V as the input. Both *non-terminal context* and *terminal context* are a list which represents the non-terminal context and terminal context of the target non-terminal token. These two contexts are used as the ground truth label of the input target non-terminal to calculate the loss.

Table4.1 shows all NTS generated from the example AST in the figure4.1. We specify the hyperparameter **n** as two which is the size of non-terminal context. Due to there are six non-terminal nodes in the AST, there are six NTS generated in terms of the definition of generation regulation.

The structure of TTS for TT2V model training is similar to NTS. The difference is the first element in TTS is a target terminal token. Tuple TTS looks like *(target terminal node, non-terminal context, terminal context)*. Same with NTS, the second element is a list to represent the non-terminal context of target terminal node and the third element is used to represent the terminal context. Table4.2 shows all TTS for TT2V model training generated from the example AST in figure4.1. For this generation, we declare the hyperparameter n is 2 which represent how many surrounding non-terminal tokens are considered as non-terminal context and m is equal to 1 which means the size of the terminal context of target terminal node to be considered.

4.4 Model Structure

After the introduction of what is a target node, the non-terminal, terminal contexts of it and two variants of ASTToken2Vec: NT2V and TT2V. We explain the structure of the model in this section. The structure of NT2V

Index	Target terminal	Non-terminal context	Terminal context
1	TT-1	[NT-2, NT-4]	[TT-2]
2	TT-2	[NT-2, NT-4]	[TT-1, TT-3]
3	TT-3	[NT-2, NT-4]	[TT-2]
4	TT-4	[NT-2, NT-5]	[TT-EMPTY]
5	TT-5	[NT-1, NT-3]	[TT-6]
6	TT-6	[NT-1, NT-3]	[TT-5]
7	TT-7	[NT-4, NT-6]	[TT-EMPTY]

Table 4.2: Training samples for TT2V Model

and TT2V is the same to each other only the input and output is different, we introduce both of them as the structure of ASTToken2Vec.

The structure of ASTToken2Vec model is a four layers neural network contains one input layer, one single hidden layer, and two output layers which similar to *Skip-gram* in Word2Vec. The length of the input layer is equal to the size of non-terminal vocabulary for NT2V model and is equal to the size of the terminal vocabulary for TT2V model. The input of ASTToken2Vec is the one-hot encoding representation vector of a target non-terminal node for NT2V and a target terminal node for TT2V.

The length of the hidden layer is the length of the embedding representation vectors which is a hyperparameter d specified by users. We use the values in the hidden layer as the embedding vectors to represent a target token after it is fed into the ASTToken2Vec model.

The ASTToken2Vec model contains two output layers: non-terminal output layer for non-terminal context and terminal output layer to represent the terminal context of the input target node. The length of these two output layers is equal to the size of the terminal vocabulary and non-terminal vocabulary. Context output layers are the addiction of one-hot encoding representation vectors of context tokens as the training target of the input.

Figure4.2 illustrates the structure of ASTToken2Vec models, figure(1) is the NT2V model, the one-hot encoding representation of a target non-terminal node is fed into the model and the output is two contexts of this target non-terminal node. Figure(2) is the structure of TT2V model which is quite similar with NT2V only the input layer is different.

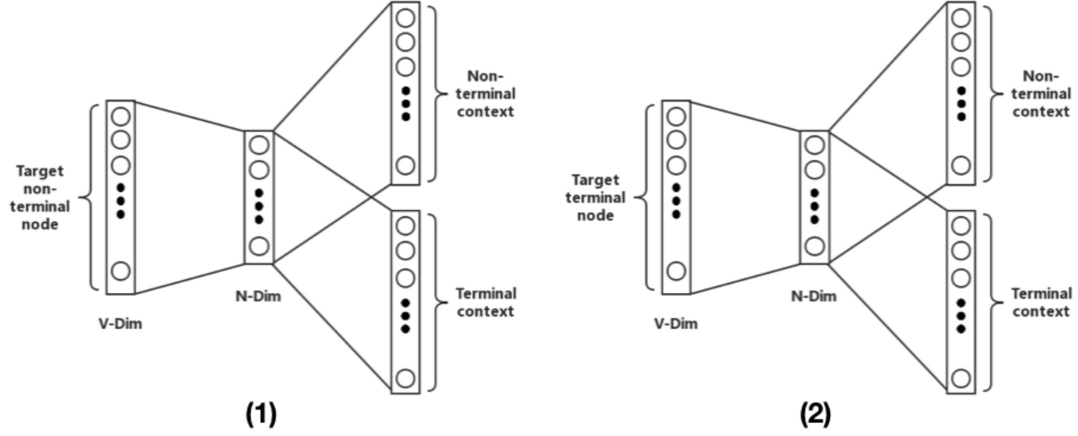


Figure 4.2: model structure for ASTToken2Vec. (1) is the structure of NT2V, (2) is the structure of TT2V

4.5 Joint Loss Function

In this section, we introduce the loss function in the ASTToken2Vec model. We specify a joint loss function of non-terminal context output and terminal context output to our ASTToken2Vec model to update the parameters in the model. This joint loss function combines the loss value of the terminal context output layer and the loss value of the non-terminal context output layer.

There are three parts of the loss function calculation. $Loss_{nt}$ is used to represent the loss of non-terminal context output and the loss of terminal context output is called $Loss_{tt}$. Both of these two are multi-labels loss calculations because there are more than one surrounding tokens as the context output. And $Loss_{total}$ is the final joint loss function for our model's training. The joint loss calculation is the same between NT2V model and TT2V.

$$Loss_{nt} = -\sum_{i=1}^{N} y^i_{nt-context} \times log(\hat{y}^i_{nt-context}) \tag{4.1}$$

Equation4.1 is the $Loss_{nt}$ calculation formula which is a *log* loss function. Concretely, for an input token x, ASTToken2Vec model calculates the non-terminal context output as $\hat{y}_{nt-context}$. The value range of $\hat{y}_{nt-context}$ is from zero to one which represents the possibilities of non-terminal token i as the

non-terminal context of input x predicted by the model. $y^i_{nt-context}$ is the one-hot encoding vectors of the non-terminal context ground-truth label of the input token x. N is the size of the non-terminal vocabulary. *log* loss between the output of the model and the ground truth non-terminal context is calculated and summed as the non-terminal loss of the input token.

$$Loss_{tt} = -\sum_{j=1}^{M} y^j_{tt-context} \times log(\hat{y}^j_{tt-context}) \tag{4.2}$$

Equation4.2 illustrates the formula of $Loss_{tt}$ calculation which also a *log* loss formula. M is the size of terminal vocabulary and the model predicts the possibilities of terminal token j as the terminal context of the input token x. Then, we calculate the *log* loss between the output terminal context of the model $\hat{y}_{tt-context}$ and the ground truth terminal context $y_{tt-context}$.

$$Loss_{total} = \alpha * Loss_{nt} + (1-\alpha) \times Loss_{tt} \tag{4.3}$$

Equation4.3 is the joint loss function combining $Loss_{nt}$ and $Loss_{tt}$, here we utilize a hyperparameter α which is used to adjust the influence of $Loss_{nt}$ and $Loss_{tt}$. The range of α is from zero to one to emphasize the importance between non-terminal context and terminal context. When the value of α is zero, the loss of non-terminal context output is ignored. On the contrary, if α is equal to one, only $Loss_{tt}$ is considered.

Sampled Softmax Sebastien Jean et al.[20] propose a method called sampled softmax which is a sample method to calculate the loss function. It can train neural network models with a very large output target vocabulary faster and keep training complexity not increase even the size of vocabulary increasing. This method is widely used by many deep learning models for natural language processing tasks due to the large size of vocabularies. Both *Skip-gram* and *CBOW* model of Word2Vec are able to leverage the sampled softmax trick to speed up the training efficiency. Our ASTToken2Vec model also has a huge size of terminal token vocabulary, it is also possible to apply sampled softmax to our ASTToken2Vec model for training.

4.6 Conclusion

In this chapter, we propose a pre-trained model extending from Word2Vec model called ASTToken2Vec. It is able to generate semantic and syntax-based representation vectors for AST nodes. Due to there are two kinds of tokens specified by programming language grammar, we design two different variants of ASTToken2Vec: NT2V model for non-terminal representation vectors generation and TT2V model for terminal representation vectors generation. We introduce the structure of ASTToken2Vec model, it is a neural network which has four layers: one input layer, one single hidden layer and two output layers which represent two different contexts of an input node. We explain the joint loss function we specify and how to generate two kinds of training samples from ASTs: NTS for NT2V model training and TTS for TT2V model training.

Embedding representation vectors generated by ASTToken2Vec model not only can be used as the pre-initialized representation for our LSTM-based code completion system and enhance the performance of token prediction due to the rich structural information of ASTs but also are able to serve in many other machine learning-based tasks with an AST dataset. We introduce how to integrate our ASTToken2Vec embedding model and an LSTM model to predict the next token as code completion in the next chapter. The evaluation of performance and visualization of the ASTToken2Vec model in a JavaScript dataset illustrates in the experiment chapter.

Chapter 5

LSTM INTEGRATION MODEL

In this chapter, we introduce what is recurrent neural network(RNN) model, how does it work and the shortcoming of traditional RNNs. We also introduce an extension variant of RNNs called long-short-term memory(LSTM) model which is widely used in sequence-based deep learning tasks and how do LSTM models overcome the shortcoming of RNN models. Then, we explain how to apply an LSTM model for code completion with an AST dataset which is our basic model to predict next tokens. Furthermore, we propose a method to integrate ASTToken2Vec embedding model introduced in chapter4 and an LSTM model together in details. This integration model is abbreviated as AT2V-LSTM model which is able to predict the next non-terminal, next terminal and the position of the next non-terminal as code completion.

5.1 Recurrent Neural Network

Programming languages can be considered as special natural languages with more grammar constraints, programmers type code just like human write natural sentences. Both code texts and sentences are linear time-sequence structure data. When traditional artificial neural networks are applied to handle this linear structure data like predicting the word in a sentence or classifying the sentiment information of a sentence, the traditional neural networks always achieve a not good performance because they are hard to handle this sequence-structure data and cannot leverage the information of

previous words in a sentence. This is a major shortcoming of traditional neural networks: Models cannot record the previous input information for the current output but this information is much more important for linear sequence-structure data-based prediction tasks.

Recurrent neural work(RNN) models[13] is proposed by Rumelhart et al. in 1988 which can address this issue pretty well. RNNs extend a new feature from the traditional neural networks. They contain many loops on their hidden units which are able to record the previous input of models. When an RNN model is unrolled, it becomes a chain-like structure neural network and each two adjacent hidden units on time series is connected. This connection enables the model to use the previous internal state(previous input) for the training of input. The input of each hidden unit is not only the output of units in the previous layer but also the output of hidden unit itself at the previous input moment. This feature allows RNN models becoming more applicable to linear sequence-structured data-based problems like natural language processing tasks.

However, even there is incredible success applying RNN models to a variety of sequence tasks, it still has two fatal flaws: long-term dependencies problem and gradient vanishing problem. The long-term dependencies problem means the model is hard to leverage the further input information of the sequence, further information is covered by the recent input even the further information is more useful for the following model prediction. Gradient vanishing is a more serious problem for RNNs. When gradient vanishing problem happens, the gradient which is used to update the parameters in models is vanishingly small and it many completely stop RNN models from training.

5.1.1 Long Short Term Memory(LSTM)

Long short term memory (LSTM) model[1] is a variant of the standard RNN models introduced by Hochreiter et al. in 1997. LSTM is capable of solving both gradient vanishing problem and learning long-term dependencies problems. It is also a chain-like neural network with a special state record structure: cell memory and three operation gates: forget gate, update gate and output gate. Instead of only using a linear connection of each adjacent hidden unit in a traditional RNN model, the LSTM model leverages the cell memory as a “conveyor belt” running straight down the entire sequence model chain. Cell memory also serves as a structure to record the infor-

mation of previous inputting. The three special operation gates are able to change the value of cell memory. The forget gate decides what information in the cell memory should be discarded. The update gate determines the new information be added into the cell memory and the output gate chooses the output of the current moment to the next moment in terms of the cell memory. At each model running moment, the value of cell memory is changed by three operation gates in terms of the output of the last moment and the input at the current moment. With the help of cell memory structure, the significative further inputting context in a sequence could be recorded and the information in the cell can be updated by three operation gates automatically and intelligently. LSTM models have a quite great achievement and improve performance of linear-sequence data-based tasks especially natural language processing comparing with standard RNN models.

Due to the great achievement of LSTM models in natural languages tasks and the similarity between source code files and natural sentences texts. More and more research explores the way to apply LSTM models to source code based tasks like code completion systems. Our code completion system is also an LSTM based model extending from the previous work[4]. We integrate the LSTM based model with our ASTToken2Vec embedding to predict the next code.

5.2 LSTM for Code Completion

In this section, we introduce the basic LSTM model we use for code completion which is called NTI2P, how to train this model with the AST dataset, The details about the structure of our model. Finally, we introduce how to integrate the LSTM based model with ASTToken2Vec embedding which is named AT2V-LSTM integration model.

The data format of dataset for our code completion system training is the AST, a tree-structure data. We introduce how to convert an AST to a sequence of training samples in the chapter3. As the description in this chapter, A sequence is composed of training samples which contain four elements each sample: *(non-terminal, terminal, node-or-leaf, right-or-left)*. Thus, we consider the input of our LSTM model as a sequence like $(N_1, T_1, NL_1, RL_1), (N_2, T_2, NL_2, RL_2), ...(N_k, T_k, NL_k, RL_k)$. In this example sequence, there are k training samples and i is the ith training sample in the sequence. N_i is a non-terminal and T_i is one of the children termi-

nal node of N_i. NL_i is called type infomation which represents the N_i is a non-leaf non-terminal node(has at least one non-terminal children) or leaf non-terminal node(has no non-terminal children). RL_i is the side information which is used to describe the N_i is the right child of its parent node or left child. These four bits of information are introduced in the chapter3 in details. Both N_i and T_i are represented by randomly initialized vectors. The value of NL_i and RL_i is either 1 or 0 which are used to represent *node or leaf* for NL_i and *right or left* for RL_i.

We name our basic LSTM model as NTI2P which means we are going to use the sequence of training samples contain non-terminals, terminals and two bits of information about type/side to predict the next sample. We introduce our AT2V-LSTM model in details in the next subsection.

5.2.1 Model Structure

The structure of NTI2P is illustrated in Figure5.1. It contains an input layer, an LSTM layer, and an output layer. The input layer is a combination layer of the representation vectors of feeding elements, and the output layer has four trainable matrices as the linear mapping between the output of the LSTM layer and our output prediction.

Input layer

As mentioned above, there are two tokens and two bits of information fed into the model and all of these inputting elements are encoded with the one-hot encoding method. There is another step for non-terminal N_i and terminal T_i inputting rather than feeding the one-hot encoding vectors into the input layer directly. We use two matrices to map the non-terminal and terminal to embedding vectors with the same length linearly. Concretely, for the basic NTI2P model, the one-hot encoding vectors of non-terminal and terminal multiply two embedding matrices separately to generate the embedding representation of tokens. Both of the mapping matrix for non-terminals and matrix for terminals is trainable by the model and randomly initialized before the model training(This random initialization is replaced by the representation vectors generated by ASTToken2Vec model in our integration model). After the one-hot encoding vectors multiply the embedding matrix and generate the embedding vectors for non-terminal and terminal

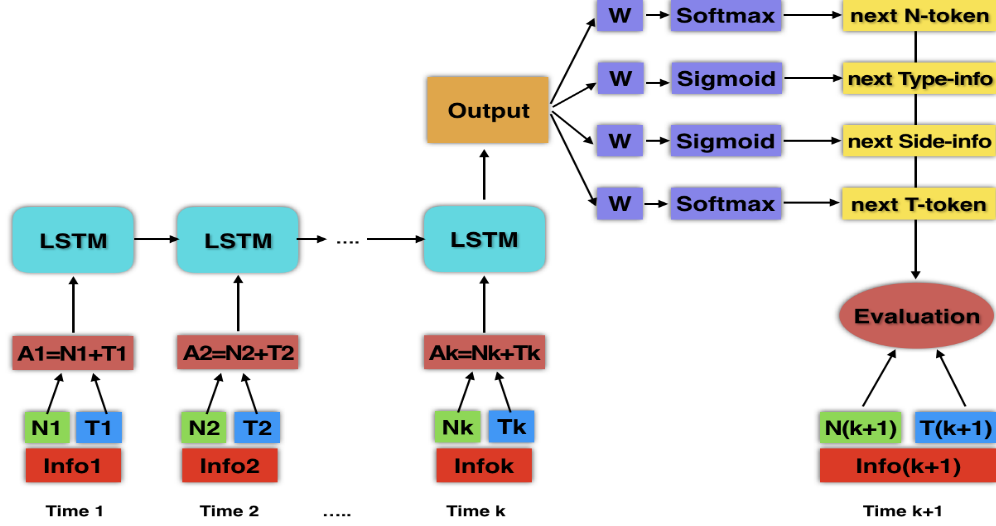


Figure 5.1: The structure of NTI2P model

separately, we add these two vectors together as the representation of token content in a training sample.

As for NL_i and RL_i, we also use two relatively small embedding matrice as a linear mapping. Then, we add the embedding vectors of NL_i and RL_i together to represent the structural postion information of N_i. Finally, we concatenate the addition embedding vectors for token content and addition embedding vectors for token structure together as the final representation vectors of a training sample. This vectors is the input of LSTM layer. The input layer of NTI2P model is computed as below:

$$input_i = concat(A \cdot N_i + B \cdot N_i, \quad C \cdot NL_i + D \cdot RL_i) \tag{5.1}$$

where A, B, C, and D are the embedding matrices for non-terminals, terminal, type information, and side information in a training sample. A is a $K \times V_N$ matrix and The shape of matrix B is $K \times V_T$. K is the length of the embedding vector for tokens which is a hyper-parameter specified by users. V_N and V_T are the size of non-terminal vocabulary and terminal vocabulary respectively. The shape of C and D is $J \times 2$ where J is the length of embedding vector for NL_i and RL_i. After the concatenation of these embedding

vectors, $input_i$ is a combination embedding vector which serves as the representation of the whole four elements in a training sample in an inputting sequence during the training phase. From the formula above, it is easy to calculate that the length of $input_i$ is $K + J$.

LSTM layer

After the calculation of input layer, the LSTM layer receives the embedding representation vectors from the input layer as x_t and takes the output h_{t-1} and hidden state c_{t-1} from the previous state of LSTM layer. With x_t and h_{t-1}, this layer calcuates three operating gates(input gate, update gate, output gate). These three operating gates are used to change the state of cell memory at the current moment as h_t and calcuates the output as c_t. The calcuation of the LSTM layer in our NTI2P model is same with the vanilla LSTM introduced in 1997[1]. Three operating gates is computed as below:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{5.2}$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{5.3}$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{5.4}$$

where f_t is the forget gate, i_t is the input gate and o_t is the output gate. σ is the sigmoid function which makes certain that the range of the result of these gates is from 0 to 1. Matrices W, U and bias vector b are the parameters of the model which are trained during the training phase. These three operating gates change the hidden state of the LSTM cell depending on the input at the current model x_t and the output h_{t-1} of the LSTM layer at the previous moment. The calculation formula of the state of the LSTM hidden cell c_t and output h_t is computed as:

$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma(W_c x_t + b_c) \tag{5.5}$$

$$h_t = o_t \circ \sigma(c_t) \tag{5.6}$$

where the forget gate f_t and the input gate i_t control what information should be forgotten from the hidden cell c_t and what needs to be updated to it. The output gate o_t is used to compute the output of the LSTM model h_t at the current moment which is one of the input elements of the LSTM layer at the next moment.

Output layer

The output of the LSTM layer h_t is a vector calculated from the state of the hidden cell and the output gate. Output layer of our model use this output vector to compute four predictions: next non-terminal N_{i+1}, next terminal T_{i+1}, side information of the next non-terminal NL_{i+1} and type information of the next non-terminal RL_{i+1}. Concretely, linear mapping is applied on h_t. h_t is going to multiply four different mapping matrices separately, and the result of linear mapping are four vectors which are fed into two softmax classifiers and two sigmod classifiers to calcuate the possibilities of prediction. Formula of the output layer is as:

$$p_{nt} = softmax(W_{nt} \times h_t + b_{nt}) \tag{5.7}$$

$$p_{tt} = softmax(W_{tt} \times h_t + b_{tt}) \tag{5.8}$$

$$p_{si} = softmax(W_{si} \times h_t + b_{si}) \tag{5.9}$$

$$p_{ti} = softmax(W_{ti} \times h_t + b_{ti}) \tag{5.10}$$

where p_{nt}, p_{tt}, p_{si} and p_{ti} represent the prediction of next non-terminal, next terminal, the side (right or left) information of the next non-terminal p_{nt} and the type(non-leaf or leaf) of p_{nt}. Ws are trainable matrices for linear mapping. The shape of W_{nt} is $K \times V_N$ and the shape of W_{tt} is $K \times V_T$. Concretely, K is the length of the hidden cell in the LSTM layer and V_N and V_T are equal to the size of non-terminal vocabulary and terminal vocabulary. W_{si} and W_{ii} are the linear mapping matrices whose shape is equal to $K \times 2$. Then these four outputs are fed into softmax classifiers. The softmax function could return the possibility of prediction which is also known as the confidence for the predicting result.

5.2.2 Conclusion

We introduce the details about the structure of our basic LSTM model for code completion. There are four elements are fed into the model at every training moment and the model predicts the next four elements in the training sequence. Our basic model is the extension of the previous work[4]. In the experiment, we use this basic model as the baseline to compare the performance of our integration model. The way to integrate this basic LSTM model and ASTToken2Vec embedding method is introduced in the next section.

5.3 Integration

As mentioned in the previous section, the embedding vectors for non-terminals and terminals in the input layer of the basic NTI2P model are the result of the one-hot encoding vectors multiplies two embedding matrices separately. The embedding matrices are randomly initialized and the embedding vectors for both non-terminal and terminal generated by them do not contain any semantic and syntax information of the nodes because of random initialization. In order to make the representation vectors become more meaningful, we integrate the ASTToken2Vec embedding model which is described in the chapter4 as a pre-trained model and the basic LSTM model for code completion NTI2P so that the integration model is able to leverage more semantic and syntax-based information to improve the performance of predicting the next token. This integration model is named AT2V-LSTM model which means the combination between ASTToken2Vec and basic LSTM model.

This the way to integrate two models: We first process the original ASTs to training samples for ASTToken2Vec model and then train two ASTToken2Vec models: NT2V model is used to generate representation vectors for non-terminals and TT2V model generates the representation vectors for terminals. The way to generate training samples from ASTs and the details of ASTToken2Vec models are introduced in the chapter4. From the mechanism of ASTToken2Vec model, the representation vectors contain more semantic and syntax of AST nodes. Then, we use these representation vectors to build the embedding matrix for non-terminals and embedding matrix for terminals which are used to replace the matrix A and B in the equation5.1 in the input layer of basic NTI2P model. With this initialization, Our AT2V-LSTM integration model is able to leverage both the order information of the input sequences of training samples and the semantic structural knowledge for each non-terminal and terminal hidden in ASTs to complete code.

5.4 Conclusion

In this chapter, we first introduce RNN models and an extension variant of them, the LSTM model. The reason why they are powerful in handling the tasks based on linear sequence data. Then, we explain the idea of using LSTM models for code completion tasks. We propose a basic LSTM model which is named NTI2P to predict the next training samples in the sequence

and the way to generate sequences of training samples for this model is introduced in the chapter3. We introduce details about the structure of NTI2P model, the mechanism of it and how does it work. We explain the difference between our basic NTI2P model and the LSTM model described in the previous work[4]. Finally, we introduce the shortcoming of the vanilla NTI2P model and how to integrate the ASTToken2Vec embedding model with the NTI2P model which is called AT2V-LSTM model. How does this integration model leverage the semantic and syntax knowledge of AST nodes and why it may have a better performance of the next tokens prediction.

Chapter 6

EXPERIMENT

In this chapter, we describe the experiments we do for both ASTToken2Vec model and AT2V-LSTM integration model. There are four models we implement, we first train an NT2V model for non-terminal embedding vectors generation and a TT2V model for terminal embedding vectors generation. Then we analyze the result of these embedding vectors by visualizing representation vectors of some tokens as the evaluation of ASTToken2Vec models. Then we implement another two models for code completion tasks: basic NTI2P model as the baseline and our AT2V-LSTM integration model to compare the performance of them. We also analyze in what situations our AT2V-LSTM model works better than the NTI2P model which means for what kind of test cases the integration model can predict right but the basic model predicts wrong. In order to explain experiments better, we also introduce the dataset and the result of our data processing in details.

6.1 Dataset and Data Processing

In this section, we introduce the details about the dataset we use, how we build the vocabularies and the statistical information about our training dataset.

6.1.1 Dataset Details

The data we use for both the ASTToken2Vec embedding model training and AT2V-LSTM integration model for code completion training is the same

size	10.77GB
programs	100,000
total terminals	8.9×10^7
total non-terminals	8.3×10^7

(a) Training set

size	5.15GB
programs	50,000
total terminals	4.3×10^7
total non-terminals	3.9×10^7

(b) Evalution set

size	15.92GB
programs	150,000
total terminals	1.3×10^8
total non-terminals	1.2×10^8

(c) Overall

Table 6.1: Dataset

dataset which is a JavaScript AST dataset provided by Raychev et al.[3]. This dataset is collected from online open-source programs and it contains 100,000 JavaScript programs as the training dataset and 50,000 programs as evaluation dataset. All of this source code has been parsed to AST format. This dataset is also used by Raychev et al[3], the PHOG model for next token prediction of code completion and Liu et al[4] also use this dataset to train several LSTM-based models for code completion. We use the training part of this dataset to generate training samples for two ASTToken2Vec models training described in the chapter4. After ASTToken2Vec models are trained well and representation vectors are generated by these models. We process the same training part of the dataset to generate sequences of training samples for both basic NTI2P model and AT2V-LSTM model for code completion. Finally, we evaluate our trained NTI2P model and AT2V-LSTM model with the test queries generated from the test part of this dataset and analyze the performance of the models.

The statistics details of the dataset can be found in Table6.1. Subtable(a) is the information about the training set which contains 100,000 ASTs, and subtable(b) is the test set we use to evaluate the performance of the predicting of the next tokens.

6.1.2 Data Processing

During the processing of AST data, we first build the non-terminal vocabulary and terminal vocabulary. The specification of vocabularies determines what kind of tokens is considered by our models. For non-terminal vocabulary, we add another two bits in each non-terminal.

Non-terminal Vocabulary Because the dataset we use is a JavaScript AST based dataset, we consider that there are 44 different kinds of non-terminal tokens specified by the JavaScript programming language grammar. Base on these 44 non-terminal tokens, we add two more bits of information to each token: whether the non-terminal token has a child token; whether this non-terminal has a right sibling or not. These two bits of information care more about the surrounding context(a child or a sibling) of each non-terminal and makes the task of non-terminal predict become more challenge. This adjunction is also used in the previous work[3][4]. There are 97 kinds of bits-information combination non-terminal tokens. From the method to convert an AST to a sequence described in chapter3, there is a special non-terminal token:*NT-EMPTY* we use as a padding token to build a complete binary tree from an AST. We also consider this special non-terminal as an element in the vocabulary of non-terminal which the model needs to predict. In total, we have 98 non-terminal tokens as the elements in the non-terminal vocabulary. Our models predict non-terminal tokens inside this non-terminal vocabulary.

Terminal Vocabulary Due to the terminal tokens of programming languages are symbols specified by programmers themselves like *LiteralString, Identifier, LiteralNumber, and Property, etc.*, in theory, there are infinite kinds of terminal tokens may be included in programs because programmers are able to specify any value and name of variables they want. It is very hard for deep learning models to predict all of these tokens directly because of this infinity and meaningless(In fact, programmers do not want code completion systems to predict next numeric value even it is a terminal *LiteralNumber*). In order to predict terminal tokens in a more reasonable way, we use the idea of *Word of Bag* to specify the terminal vocabulary. Concretely, we sort all terminal tokens appearing in the training dataset by their frequencies of occurrence. Then we choose the 50,000 most frequent terminal tokens as the vocabulary of the terminal. For the terminal tokens which have a lower

frequency in our dataset which means they are out of our terminal vocabulary bag, we use a special terminal token *UNK* to represent these infrequent terminal tokens. Same with the vocabulary of non-terminal, there is a particular terminal token *TT-EMPTY* described in the chapter3 which serves as the padding terminal token to represent the case of a non-terminal who does not have a terminal child. In total, we have 50,002 tokens in the vocabulary of the terminal.

Then, we generate a mass of training samples for ASTToken2Vec models training: both NT2V and TT2V model. This generation for ASTToken2Vec model described in the section4. The structure of training samples for NT2V model is *target non-terminal, non-terminal context, terminal context* where target non-terminal is the input and non-terminal context and terminal context are the ground truth of the model. The structure of training samples for TT2V model is *target terminal, non-terminal context, terminal context.* Ather the processing for ASTToken2Vec model, there are 8.9×10^7 training samples for NT2V model and the training set for TT2V model contains 8.3×10^7 training samples.

Finally, we transform all ASTs in the training dataset to the sequences of training samples for both basic NTI2P model and AT2V-LSTM integration model which is explained in the chapter3. After the data processing, there are 100,000 sequences for models training which is equal to the number of ASTs. And there are 1.6×10^8 training samples in sequences totally.

6.2 Experiment of ASTToken2Vec

In this section, we introduce the experiment for ASTToken2Vec embedding models. We implement NT2V model and TT2V model with TensorFlow[21] deep learning framework. We train both the NT2V model and TN2V model with the training samples generated from ASTs described as above. We discuss details of ASTToken2Vec models training and visualize the representation vectors of some terminal tokens and compute the similarity of tokens.

6.2.1 Training details

Due to the architectures of NT2V and TT2V are similar to each other only the input layer and output layers are different, we explain the training details of these two models together. We define the size of the hidden layer is equal

to 1,000 for both NT2V and TT2V model which is the same as the length of representation vectors for tokens.

Because there are two output layers of the ASTToken2Vec model which represents the terminal context and the non-terminal context of the input token, we specify a hyper-parameter α in the definition of our loss function in the chapter4 which is able to adjust the proportion of terminal context and non-terminal context. The choice of α has a significant influence on the performance of ASTToken2Vec models. In our NT2V model for non-terminal representation vectors generation, we specify the adjuster α equal to 0.5 so that the NT2V treat both non-terminal context and terminal context equally. In the TT2V model for terminal representation vectors, however, we define α equal to 0.7 which means we ask the model to care more(70%) about the non-terminal context comparing with the terminal context(30%). The reason why we choose α for TT2V model as 0.7 is that the non-terminal context contains more knowledge about the structure of the AST which is much more important than the neighbor terminal context. For example, non-terminal tokens like *ForStatement*, *IfStatement* and so on represent the structure of a code snippet in a source code, and identifier terminal tokens like *Identifer i* or *Identifer index* are more possible appearing near to non-terminal *ForStatement* from the programming habit of coders, Programmers are more interested in writing a 'for' expression like "for(var i = 0; ...; ...)...". So, if the TT2V model cares more about the non-terminal context, representation vectors containing more semantic information are more possible to be generated, This assumption is also confirmed by our experiment.

In the training phase, we use the Adam optimization algorithm[22] with learning rate 0.01 to train models, this optimization algorithm has a better performance comparing with stochastic gradient descent algorithm in our ASTToken2Vec model training. The size of the training batch in our model is 100 and we train both two models with 10 epochs. We train our models with the GPU supporting and the time cost for each epoch is near to five hours with a GeForce 980 NVIDIA GPU.

6.2.2 Visualization

What ASTToken2Vec models generate is the embedding representation vectors for AST nodes. In order to illustrate the performance of ASTToken2Vec models, we visualize embedding vectors of several terminal tokens generated by our TT2V model. Due to the size of the hidden layer in the TT2V model

we trained is equal to 1,000, the length of the representation vectors is 1,000 too.

We first apply principal component analysis(PCA) algorithm[23] to our embedding vectors. PCA algorithm is a dimensionality reduction method which is widely used to reduce the dimension of a vector to a relatively lower dimension one without losing much information of the vector. In this case, we use PCA to reduce the dimension of the embedding vectors from 1000 dimensions to 2 dimensions so that they are much easier to be visualized because two value x and y in the dimension vector are considered easily as the coordinate of the representation tokens. After the dimensionality reduction, we normalize the two-dimension vectors with min-max normalization. Basically, min-max normalization is a normalization strategy which linearly transforms x to $x_new = (x - min)/(x_{max} - x_{min})$ where x_{max} and x_{min} is the maximum and minimum value of all x and y to $y_mew = (y - y_{min}/(y_{max} - y_{min})$. Our normalization is different with this standard normalization, in our visualization, we transform the entire range of values of elements in the 2 dimensions vectors from min to max are mapped to the range -2 to 2 rather than -1 to 1. These processed vectors are used as the coordinate of terminal tokens.

We pick up several terminal tokens to visualize, the visualization is shown in Figure6.1. Terminal token *Identifiers* are blue including “userName”,“id”, “size”, “length”, “user_id” and so on. *LiteralNumber* is represented by purple. Tokens like “add”, “append”, “value”, “key” are terminal *Property* which is green token in the figure. Red tokens are *LiteralString* like “mouseup”,“mousedown”, “keyup” and “keydown”.

From the visualization of tokens in the figure6.1, it is not hard to find that terminal tokens with the similar feature(tokens belong to the same type like LiteralString or Property) are near to each other but far from other different types of tokens. We can find there are several clusters like: “literal string cluster”, “property cluster” and so on. Another valuable thing is that even tokens belong to the same type, if another feature is different, the 2 dimension reduction representation vectors are also far from each other. For example, even both “append” and “value” are terminal token *Property*, they are still not in the same cluster because “property append” is a function which is able to add some elements to a container in the most cases, however “property value” serves as a member in a class without some operation functionality of a container. Another example is *Identifiers*, in the most instances, “identifier length”, “identifier len” and “identifier size” are used to express length or scope of a class or a container and they are also bonded

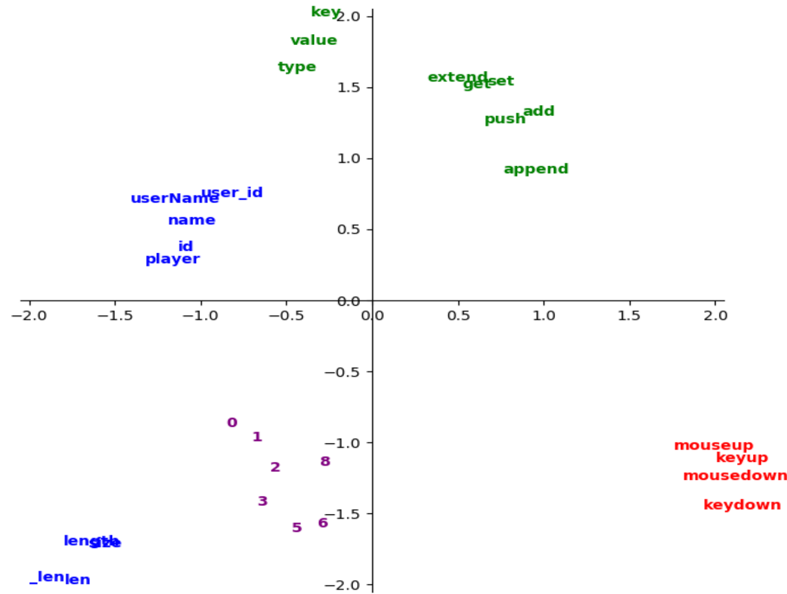


Figure 6.1: The visualization for the 2-D terminal representation vectors generated by ASTToken2Vec

with a *LiteralNumber*. Their semantic meaning is similar and their syntax location in an AST is also nearly similar to each other. This is the reason why their representation vectors are in the same cluster and near to each other. Nevertheless, even "userName", "user_id" and "name" are also belong to the type *Identifiers*, their 2 dimension representation vectors are still far from the "identifier length" cluster in our figure due to they have different semantic meaning and different surrounding context in an AST("userName" and "user_id" are used to record the information about a user in most instances).

6.2.3 Similarity Calculation

We also calculate the cosine similarity between embedding vectors of several terminal tokens. The embedding vectors we use for similarity calculation is the original 1000 dimension vectors rather than the dimensionality reduction

vectors. The cosine similarity is a measuring method of similarity between two vectors. The range of cosine similarity is from 0 to 1. If two vectors have the same orientation, their similarity is equal to 1. On the contrary, if the similarity is 0, it means these two vectors are perpendicular to each other(there is no similarity between them). Cosine similarity of two vectors A and B is calculated with the formula:

$$similarity = \frac{A \cdot B}{\|A\| \times \|B\|} \tag{6.1}$$

The result of the cosine similarity calculation meets our conclusion of ASTToken2Vec models. The most similar token to "identifier length" is "identifier size" with the similarity 0.927. The second similar token is "identifier len" with the similarity 0.883. The similarity between "identifier length" and "property push" is equal to 0.327 which means these two embedding vectors are not near to each other. The analysis conclusion of similarity calculation is the same as the conclusion of vector visualization.

6.2.4 Conculsion

In this section, we introduce the implementation of our ASTToken2Vec models, the hyper-parameters we specified and the training details of the experiment. After models are trained well, they generate the representation vectors for both non-terminal tokens and terminal tokens. We also leverage the dimensionality reduction method: PCA to reduce the dimension of the vectors from 1,000 dimensions to 2 dimensions and normalize the 2 dimensions representation vectors to the range of value from -2 to 2. Then, we visualize reduction dimension embedding vectors of several terminal tokens by drawing the values of their 2 dimension vectors as the coordinate of these terminal tokens. We also calculate the cosine similarity between these embedding vectors. From the analysis of the visualization and similarity calculation, we can find that the embedding representation vectors of the terminal token are semantically meaningful. These embedding vectors are used to initialize the representation of tokens in our AT2V-LSTM integration model.

6.3 Experiment of AT2V-LSTM Model

In this section, we implement two models for code completion: basic NTI2P model and AT2V-LSTM integration model. The basic NTI2P model works

as the baseline to compare the performance of the next token prediction. The details about these two models are described in chapter5. We use TensorFlow[21] deep learning framework to implement two models for the next token prediction.

6.3.1 Training details

The RNN core we use in our models is a basic LSTM core rather than the gated recurrent unit(GRU)[24] core. The reason is from our control experiment of LSTM and GRU, the GRU is a little bit faster than LSTM core during the training phase but losing 1.2 percent accuracy of the evaluation performance. Due to the degree of speedup is so small that can be ignored comparing with the losing of accuracy, we choose basic LSTM as the hidden core of our models rather than GRU core. We use Adam[22] optimization algorithm to train our model with base learning rate 0.0025. Due to we use learning rate decay trick for model training, this basic learning rate is multiplied by 0.85 every 0.5 epoch. Even the LSTM can avoid the gradient vanishing problem and prevent the gradient exploding problem in most cases, the gradient exploding problem still occurs one time among our experiments. So we use gradient clipping method to forbid the occurrence of gradient exploding problem. Basically, we clip the gradient which is more than 6 to 6 and less than -6 to -6 to avoid the gradient becoming too large or too small. We use randomly uniformly initialize the initial state of LSTM cell with values from -0.1 to 0.1. Due to the training of RNN models is backpropagate through time, we unroll the LSTM model with the time sequence $s = 50$ to take a subsequence of length 50 in each input batch and the batch size is $b = 100$. Therefore, there are $s \times b = 5000$ training samples for one training batch. The training epoch is $e = 10$. We also train these models with the supporting of a GeForce 980 NVIDIA GPU and each epoch costs 8 hours for model training.

6.3.2 Next Token Prediction

We evaluate our AT2V-LSTM integration model and the basic NTI2P model with the test dataset after the models are trained well. We present the performance of prediction accuracy including the next non-terminal prediction, next terminal prediction and the prediction of the type and side information of the predicting non-terminal. And then, we compare the performance of

these two models and analyze the comparing predicting result to explore in which case our AT2V-LSTM integration model is more accurate than the basic NTI2P model.

Next non-terminal prediction The valid accuracy curve of the next non-terminal token prediction during the training phase is illustrated in Figure6.2, the blue curve represents the validation accuracy of the basic NTI2P model and the orange curve is our AT2V-LSTM integration model. the x-axis represents the validation checkpoint among training step, there are four checkpoints for each training epoch. Due to we train our models 10 epochs totally in the experiment, there are 40 checkpoints to record the validation accuracy.

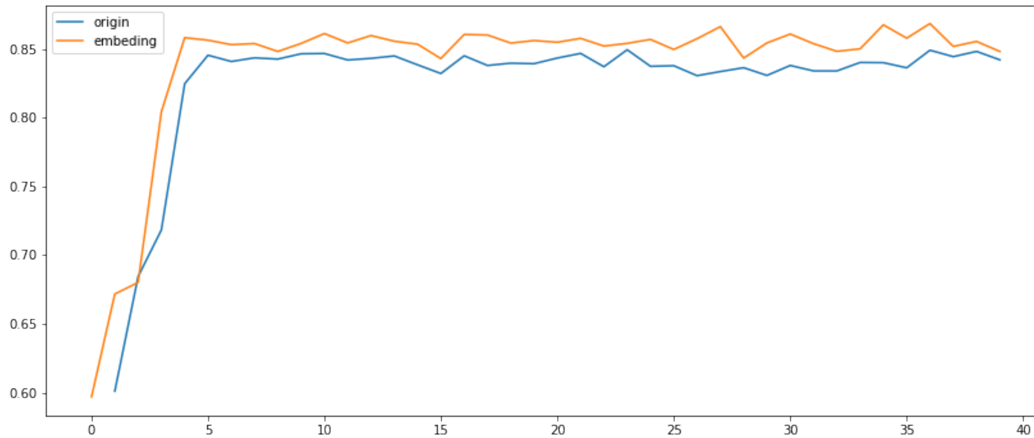


Figure 6.2: Validation accuracy for non-terminal prediction during the training phase

From the validation accuracy curve in Figure6.2, we find the accuracy of AT2V-LSTM integration model is little higher than the accuracy of the basic NTI2P model. The evaluation result illustrated in the table6.2 also show that the non-terminal accuracy of the integration model is 1.5% higher than the basic baseline model.

Next terminal prediction Figure6.3 illustrates the validation accuracy curve for next terminal token prediction during the training phase. Orange curve represents our AT2V-LSTM integration model and the blue curve is the accuracy of our baseline: the basic NTI2P model. The evaluation accuracy

Models	Top one accuracy	Top 3 accuracy
Vanilla LSTM	$83.5 \pm 0.2\%$	$92.6 \pm 0.2\%$
AT2V-LSTM	$85.2 \pm 0.2\%$	$94.4 \pm 0.2\%$

Table 6.2: Non-terminal Evaluation Accuracy

Models	Top one accuracy	Top 3 accuracy
Vanilla LSTM	$75.8 \pm 0.2\%$	$87.7 \pm 0.2\%$
AT2V-LSTM	$78.9 \pm 0.2\%$	$89.2 \pm 0.2\%$

Table 6.3: Terminal Evaluation Accuracy

for the terminal in the test phase is shown in Tabel6.3. From both validation result and evaluation result of terminal prediction, we can find that the AT2V-LSTM integration model has a better performance with the predicting accuracy of 78.9% than the basic baseline model.

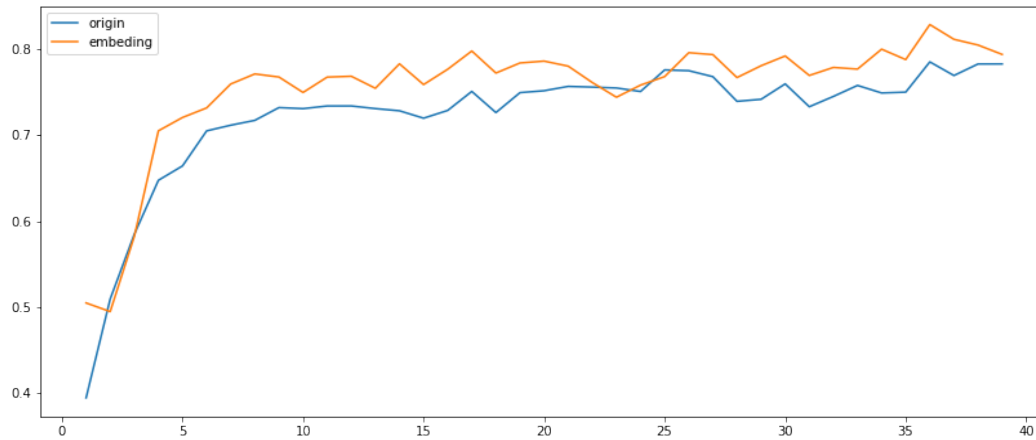


Figure 6.3: Validation accuracy for terminal prediction during the training phase

Models	Token type accuracy	Token side accuracy
Vanilla LSTM	$97.6 \pm 0.2\%$	$94.8 \pm 0.2\%$
AT2V-LSTM	$97.8 \pm 0.2\%$	$95.1 \pm 0.2\%$

Table 6.4: Node Information Evaluation Accuracy

Next token information prediction There are another two bits of information our models need to predict which is the type(non-leaf or leaf) and the side(right-child or left-child) of the next predicting non-terminal. Both of these two information is binary which means there are only two possible value:0 or 1 to predict by our models. The evaluation accuracy of these two bits prediction is shown in the Table6.4, from this result, we can find that both basic NTI2P model and AT2V-LSTM integration model achieve a good performance that the accuracy of type information is near to 97% and the accuracy of side information is near to 95%. There is not too much difference between the performance of the two models.

6.3.3 Prediction Analysis

From the evaluation result of the predicting performance, we find that the AT2V-LSTM integration model has a better performance for the next terminal prediction. In order to figure out in which case the integration model is more correct, we extract some featured evaluation queries where the basic NTI2P model predicts wrong but our AT2V-LSTM model predicts correctly. We analyze the possible reasons that may cause our integration model to work better.

Infrequent Terminal Repeation

When programmers specify variables or functions, they may not use some common name to define the name of identifiers due to the special purpose of these identifiers and these special identifiers always used only in one program files repeatedly(An example is an identifier: 'shouldBe' in the code snippet shown in the figure6.4(1)). This situation usually happens in Web-related programs. However, these special identifiers are quite infrequent both in the daily programming and our dataset. Basic deep learning models for code completion are hard to learn enough knowledge of these infrequent terminals from the training dataset because they only appear in several program files. But for our AT2V-LSTM model, because the ASTToken2Vec model is able to learn syntax knowledge of a repeat terminal from ASTs even this terminal token is only included in several training ASTs.

The code snippet in the figure6.4(1) is an evaluation source code file in the test dataset. In this file, the programmer specifies a function called "shouldBe" which is a terminal token "identifier" in terms of the program-

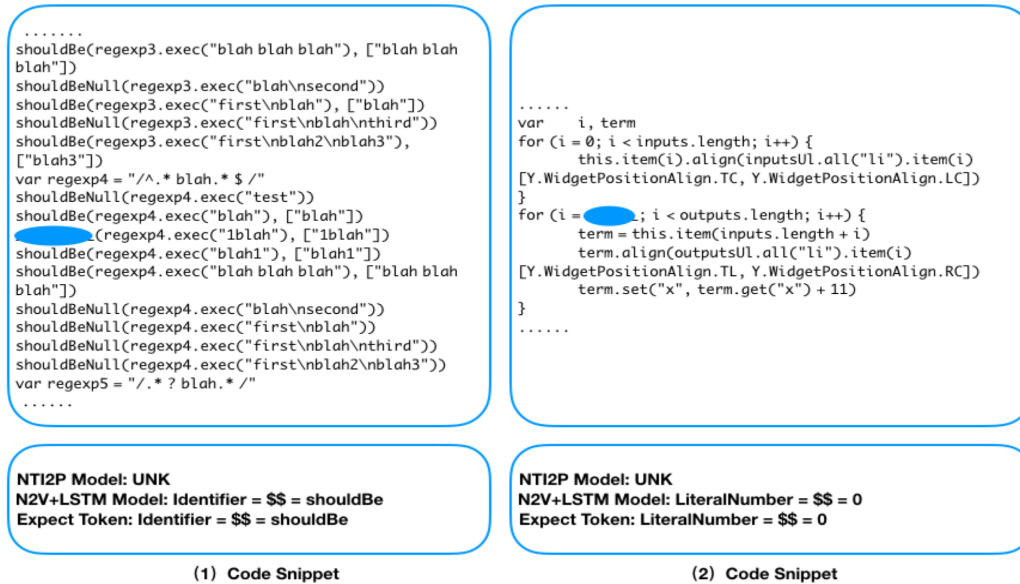


Figure 6.4: Code snippets for prediction result analysis

ming language grammar. From the programming habits of most coders, identifier "shouldBe" is an infrequent terminal. In this figure, we create a hole which is covered by a blue oval and transforms the previous context of the hole as a sequence. Both trained NTI2P model and AT2V-LSTM model predict what kind of terminals may appear in the hole as the code completion. In this example, the basic NTI2P model predicts "UNK" which means this model consider the terminal token appearing in the hole is a quite infrequent terminal token and this terminal is out of the terminal vocabulary we specified. On the contrary, our AT2V-LSTM integration model gives a correct prediction: "shouldBe" identifier. From our analysis of this test case, we can conclude that the ASTToken2Vec model is useful to improve the performance of code completion and our AT2V-LSTM integration model is more powerful to predict the infrequent terminal tokens which appear in a single source code file repeatedly.

Literal Number Prediction

The prediction for the literal number is hard for our model because there are infinite numeric values. However, some numbers determined appear in some

code snippets. For example, the literal number zeros most possible appears in the variable specification of identifier “i” expression in the “for loop” statement. The example code snippet shown in the figure6.4(2) is a test case in our test dataset. We ask our models to predict what terminal token should be filled in the hole covered by a blue oval. The ground truth label in this test case is “LiteralNumber 0”. Our AT2V-LSTM model predicts correctly comparing with the NTI2P model predicts unknown terminal tokens here. From the analysis of this test case, we can conclude that the representation embedding vector of “ LiteralNumber 0” is able to learn from its non-terminal context “ForStatement” so that the embedding vector contains syntax information of “for loop statement”. And when our AT2V-LSTM model leverage this syntax information included by the embedding vector and predict the terminal token in this hole, it has a higher possibility to predict correctly.

6.3.4 Conclusion

In this experiment, we implement both NTI2P model and the AT2V-LSTM integration model. We feed the same sequences of training samples to these two models and ask them to predict the next non-terminal token, next terminal token, the type and side information of the next non-terminal token. We compare the predicting performance of two models and find that our AT2V-LSTM model has a better performance of next token prediction especially the predicting of next terminal tokens. Finally, we analyze in which case our integration model is more likely to predict correctly than the basic model and its reason. We conclude that our AT2V-LSTM model has a higher possibility to predict infrequent repetition identifiers and literal number terminal tokens which the usage of representation embedding vectors generated by ASTToken2Vec model.

From the experiment of code completion and the experiment of ASTToken2Vec models, we can conclude that ASTToken2Vec is a useful pre-trained embedding model which is able to generate more semantic-based representation vectors for AST nodes. And these representation vectors contain more information hidden in the structure of ASTs and they can be used to help the LSTM based model achieve a better performance of the next token prediction.

Chapter 7

CONCLUSION

In this chapter, we summarize the work we do for deep learning-based code completion and analyze its performance, we also discuss the future research direction about code completion with deep learning models.

7.1 Summary

AST reconstruct from the sequence We extend the method to transform ASTs to sequences in our work so that an AST can be reconstructed from a sequence directly. Deep learning-based code completion research in the previous work provides the method to transform ASTs in the dataset to sequences of training samples as the data processing. Each element in this sequence is a training pair which contains two tokens: non-terminal token and terminal token. Each training pair is fed to an LSTM model to predict the next training pair.

We consider the shortcoming of this AST processing that: after the model outputs the predicting sequence of pairs, it is hard to reconstruct the AST from this predicting sequence directly. Our method, rather than only two tokens: non-terminal and terminal in training pair in the sequence, we give another two bits of information: the type information and side information the non-terminal. This information is more about the structural information of the AST and the position of the non-terminal tokens. It is used to reconstruct the AST from the predicting sequence. So we enhance the content of the original training pair to a four elements training samples: *non-terminal, terminal, node-or-leaf, right-or-left.* Our model is able to predict the next

non-terminal and next terminal token in the training sequence, but also predict the two bits of information.

Embedding model for AST nodes: ASTToken2Vec The most significant part of this thesis is that we propose a pre-trained model called ASTToken2Vec which is inspired by a widely used embedding method for natural languages called Word2Vec[2]. The ASTToken2Vec model is also an embedding method which is able to generate the representation vectors for both non-terminal and terminal AST nodes. The generation embedding vectors contain more semantic-based information of AST nodes. There are two variants of the ASTToken2Vec model: ASTToken2Vec for non-terminal is called NT2V and ASTToken2Vec for the terminal is abbreviated as TT2V. The ASTToken2Vec model is a neural network model which contains one input layer, one single hidden layer, and two output layers. The mechanism of ASTToken2Vec is using the surrounding context of a target non-terminal/terminal token in an AST to generate the representation vectors. ASTToken2Vec model not only can be applied in deep learning-based code completion systems but also is useful for other learning-based tasks with an AST dataset as a pre-trained model. In the experiment chapter, we also train both of two ASTToken2Vec models and visualize the representation vectors of several terminal tokens. We reduce the dimension of the embedding vectors by PCA algorithm and analyze the visualization result of these terminal tokens.

AT2V-LSTM Integration Model We propose a basic LSTM model as the baseline for code completion named NTI2P model. Then, we integrate this basic model and the ASTToken2Vec method as a pre-trained model to predict the next tokens. We name this integration as AT2V-LSTM model. Basically, we use the embedding vectors of non-terminal and terminal tokens generated by ASTToken2Vec method to initialize the representation vectors of all tokens in the vocabulary rather than random initialization. In the experiment phase, we train our AT2V-LSTM integration model with a JavaScript AST dataset and evaluate the performance of both NTI2P model and AT2V-LSTM model with test queries. We find that our integration model has better accuracy on the next terminal token prediction. We also analyze in what situation the integration model is more possible to predict correctly compared with the basic model. From the analysis, we conclude

that the integration model is more powerful to predict infrequent identifier terminal tokens.

7.2 Future Research Directions

Code completion with deep learning models is still an exciting research direction. Code completion is one of the most useful tools for programmers to write code and it has great research value. Deep learning models have a great achievement in many areas especially handling linear sequence-structure data like natural languages. It is possible to apply deep learning models to source code based tasks like code completion due to the similarity of natural languages and programming languages. Our work is an extension of the existing LSTM based code completion research and there are more available research methods are worth to try and can be applied in code completion systems.

Tree-based LSTM models[17] have been applied in natural languages and have a good performance on semantic analysis of natural languages. This model is able to leverage the syntax tree data directly rather than transform a syntax tree of a sentence to a sequence of tokens. Due to the abstract syntax tree of a program is also a semantic-based tree structure which is similar to the syntax tree of a sentence, this similarity makes it possible to try to apply this tree-structured LSTM model to some AST data-based tasks like code completion. The challenge of it is how to feed a dynamic AST to the tree-LSTM model and predict the next node appearing in the AST.

Generative Adversarial Nets(GAN) have a rapid development in recent years. GAN models are able to generate data from the confrontation between two neural networks. It has been applied to generate natural sentences. William Fedus et al.[25] propose the MaskGAN model as a text generation model which have a great performance of natural sentence generation. Its success inspires us to consider code completion tasks as code generation problems and try to apply GAN models to predict the next tokens of grograms.

ACKNOWLEDGEMENTS

First of all, I'd like to express my gratitude to my supervisor: Prof.Masuhara Hidehiko. With his guidance, extraordinary patience for my asking, great encouragement when I am confused and pressured and valuable suggestions, I spend two years of an excellent time in Japan and complete graduation thesis. This thesis would have been impossible without him.

I am thankful to all the present and past members of the Programming Languages Lab for their support, and advice especially Doctor.Matthias Springer who give me inspiration about this research. Their friendly and patient help gives me a harmonious and comfortable environment to work even I am a foreigner in this country.

I would like to thank all professors and staff members in the Department of Mathematical and Computing Science. They help me to solve the problems I face throughout my master's studies. I am grateful to them for their generous education support that helps me to concentrate more deeply on my research work.

I also want to say thanks to my future girlfriend. Due to her absence, I am able to put all my time and energy to the research work and finish this thesis on time. With the help of her disappearance, I find the real fun in coding and computer science which may have a significant influence on my life.

Finally, I'd like to express my heartiest gratitude to my family members especially my mother Fu Zhicui and my father Li Jun. Their love and support give me the confidence and courage to face and overcome all problems and challenges in my whole life. They encourage me to take the calculated risk, to dream and to build, to fail and to succeed. Without their love, I cannot find what kind of life I really want to live and it is impossible to complete this thesis.

Bibliography

[1] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[2] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546, 2013.

[3] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. *SIGPLAN Not.*, 51(1):761–774, January 2016.

[4] Chang Liu, Xin Wang, Richard Shin, Joseph E Gonzalez, and Dawn Song. Neural code completion. 2017.

[5] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

[6] Chris J. Maddison and Daniel Tarlow. Structured generative models of natural source code. *CoRR*, abs/1401.0514, 2014.

[7] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 532–542, New York, NY, USA, 2013. ACM.

[8] Miltiadis Allamanis and Charles Sutton. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Sym-*

posium on Foundations of Software Engineering, pages 472–483. ACM, 2014.

[9] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49. ACM, 2015.

[10] Percy Liang, Michael I Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 639–646, 2010.

[11] Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: probabilistic model for code. In *International Conference on Machine Learning*, pages 2933–2942, 2016.

[12] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. In *ACM SIGPLAN Notices*, volume 51, pages 731–747. ACM, 2016.

[13] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[14] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Acm Sigplan Notices*, volume 49, pages 419–428. ACM, 2014.

[15] Mario Linares-Va squez Martin White, Christopher Vendome and Denys Poshyvanyk. Toward deep learning software repositories. 2015.

[16] Matthew Amodio, Swarat Chaudhuri, and Thomas W. Reps. Neural attribute machines for program generation. *CoRR*, abs/1705.09231, 2017.

[17] Christopher D. Manning Kai Sheng Tai, Richard Socher*. Improved semantic representations from tree-structured long short-term memory networks. 2015.

[18] Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: Probabilistic model for code. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine*

Learning, volume 48 of *Proceedings of Machine Learning Research*, pages 2933–2942, New York, New York, USA, 20–22 Jun 2016. PMLR.

[19] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. A survey of machine learning for big code and naturalness. *CoRR*, abs/1709.06182, 2017.

[20] Sébastien Jean, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. On using very large target vocabulary for neural machine translation. *CoRR*, abs/1412.2007, 2014.

[21] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[22] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[23] Ian Jolliffe. *Principal component analysis*. Springer, 2011.

[24] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[25] William Fedus, Ian Goodfellow, and Andrew M Dai. Maskgan: Better text generation via filling in the_. *arXiv preprint arXiv:1801.07736*, 2018.