ASTToken2Vec: An Embedding Method for Neural Code Completion

Li Dongfang　Masuhara Hidehiko

Code completion systems help the programmers to write code more efficiently and to reduce typographical errors by automatically suggesting code fragments that the programmers likely to write next. This work attempts to improve an LSTM neural network-based code completion system by proposing a new embedding method called ASTToken2Vec for program tokens. ASTToken2Vec is inspired by Word2Vec, and gives a vector representation to a program token so that tokens often appear in similar contexts are mapped to positions close to each other. We integrated ASTToken2Vec with the LSTM-based code completion model and evaluated its prediction performance by using a dataset consisting of 150,000 open-source JavaScript program files.

1 Introduction

Code completion is a feature of programming editors that suggests code fragments that are likely to be typed in by the programmer following to the program text just before the cursor position. It helps the programmer to write code more quickly, to remember rarely used API names, and to reduce typographical errors.

In general, code completion systems make suggestions (we hereafter call *predictions*) based on the program text around the cursor position (we hereafter call a *context*). Predictions are usually a list of identifiers (e.g., function, method and variable names), but some systems predict a sequence of tokens or a few lines of statements.

Context information determine the prediction performance. For example, a standard completion feature in Eclipse JDT merely the type of the receiver expression as the context when it predict a method name of a method invocation expression. Eclipse code recommender [3] additionally uses a sequence of method calls preceding to the cursor line as the context so that it can recognize conventions of APIs.

李 東方, 増原 英彦, 東京工業大学数理計算科学系, Dept. of Mathematical and Computing Science, Tokyo Institute of Technology.

Traditional code completion systems are not intelligent enough. Most of them are based on statistical methods and suggest code only by simple term frequency which often relatively have a higher error rate. Another problem is that they rely on static types (like Eclipse for Java) to filter out candidates. However, the dependency on typing information limits their applicability to widely used in dynamically typed languages like Python.etc.

With the rapid development of deep learning techniques, more and more research pays attention to applying deep learning models, especially recurrent neural networks (RNNs) to automatic code completion. RNN models achieved excellent performance in the natural language processing (NLP) field. Intuitively, a programming language can be considered as a special language just with stricter semantic and syntax. Hence, it is straightforward to apply RNNs to programming languages tasks like code completion and may have an exciting performance.

In this paper, we propose an embedding method for abstract syntax trees (ASTs) nodes called ASTToken2Vec to improve the prediction performance of the deep learning-based code completion. It consists of a four-layer neural network for encoding the context information of ASTs to generate semantic-based embedding representation vectors that contain more knowledge hidden behind ASTs. We use

this ASTToken2Vec model as a pre-trained model and integrate it with an LSTM based model as a code completion system to predict next tokens. We name our integration model as AT2V-LSTM model and the overview of our model illustrates in Figure 1.

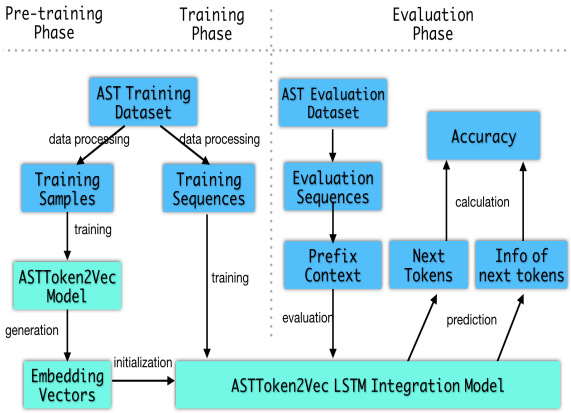

Fig. 1 Overview of ASTToken2Vec LSTM integration model

As an experiment, we implement both ASTToken2Vec embedding and AT2V-LSTM model with a JavaScript AST dataset [11] collected from open-source programs containing a total of 150,000 JavaScript files. We visualize representation vectors of several terminal tokens and evaluate the performance of predicting next tokens by AT2V-LSTM integration model From the evaluation and results analysis, we conclude that the ASTToken2Vec method is able to generate semantic-based representation vectors of AST nodes and the AT2V-LSTM integrated model predicts tokens more correctly.

2 Background

Hindle et al. [4] 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. Nguyen et al. [7] 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. [9] extends the state-of-the-art *n-gram* approach which is called SLAMC by incorporating semantic information into code tokens.

Comparing with *n-gram* model, a statistical nonparametric Bayesian probabilistic tree-based system [1] is proposed by Allamanis et al. which is able to extract code idioms from the existing written code files. Liang et al. [5] focus on learning programs for multiple related tasks with a few training samples and propose a nonparametric hierarchical Bayesian model which is able to share the statistical information across multiple tasks for code completion.

Bielik et al. [2] introduce a generative model for code called probabilistic higher-order grammar (PHOG). This model is able to capture the rich context information between tokens by allowing conditioning of a production rule. Raychev et al. [10] create a domain-specific language (DSL) over abstract syntax trees (ASTs) called TGEN which can encode an AST to a specific language context. They also propose a special decision tree called DEEP3 which can make code predictions leveraging the AST context encoded by the TGEN model.

Raychev et al. [12] and White et al. [13] explore how to apply RNN models on sequences of tokens to facilitate the task of code completion. Chang et al. [6] propose several LSTM-based models for code completion with an AST dataset and they leverage the ASTs by converting ASTs to sequences of training samples. Their work gives us inspiration about how to convert an AST to a sequence and how to train an LSTM model with AST dataset.

3 Abstract Syntax Tree

Abstract Syntax Trees (AST) are a tree structure that represents structural information of programs. It is widely used in code completion engines.

There are two kinds of a node in ASTs, non-terminal nodes and terminal nodes. A non-terminal node has a children node list denoting all its children nodes. 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 represent more knowledge about the structure and logical judgment of a program

like “ForStatement”, “IfStatement”, “WhileStatement”, etc.

A terminal node has “value” to represent the value of terminals and does not have any children node. In JavaScript, for instance, the type of terminal tokens could be “LiteralString”, “LiteralNumber”, “Identifier” etc. Due to programmers can specify any strings, variables and function’s name in a program, it is obvious that there are infinite possibilities for terminal tokens.

Due to ASTs contain more semantic knowledge about source code, we use ASTs as the basic data to generate two sets of training samples for our models’ training.

4 ASTToken2Vec Embedding

ASTToken2Vec model is an embedding model for AST nodes. It is inspired by the embedding method for natural languages, namely Word2Vec [8]. It trains a four-layers neural network to generate the semantic-based representation vectors of AST nodes which enable the LSTM-based code completion model to leverage more structural knowledge to predict next tokens. In order to do that, we give a basic hypothesis of ASTToken2Vec which is same as 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. We specified the surrounding non-terminal nodes of a target node as the non-terminal context and surrounding terminal nodes as the terminal context. The details of contexts is explained in subsection4. 2 and subsection4. 3.

4. 1 Model Architecture

The ASTToken2Vec model is a four layers neural network contains one input layer, one single hidden layer, and two output layers. Due to there are two kinds of tokens non-terminal and terminal in ASTs, 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.

The input layer is the one-hot encoding representation vector of a non-terminal node for the NT2V model and a terminal node for the TT2V model. The ASTToken2Vec model has two output layers: non-terminal output layer for non-terminal context representation and terminal output layer representing the terminal context of the input node. The length of these two output layers is equal to the size of the terminal vocabulary and non-terminal vocabulary separately. We use the values in the hidden layer as the embedding vectors to represent the input AST node and the length of the hidden layer is a hyperparameter D specified by users. In the training phase, we calculate a joint loss function to update the model.

Figure 2(1) illustrates the architecture of the NT2V model and Figure 2(2) is the architectrue of TT2V model.

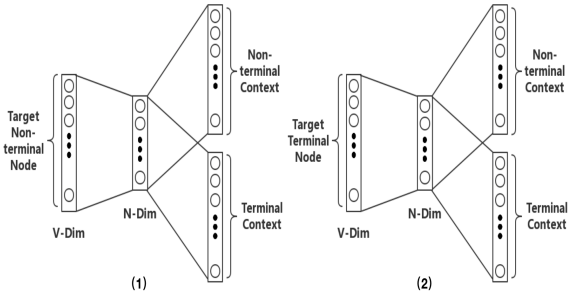

Fig. 2 Architecture of NT2V (1) and TT2V (2)

4. 2 Embedding for Non-terminals

The ASTToken2Vec model for encoding non-terminals to vectors is abbreviated as NT2V. It employs both the terminal context and the non-terminal context of a target non-terminal to generate embedding representation vectors. A training tuple for NT2V contains three elements: (*target non-terminal, non-terminal context, terminal context*) where the *target non-terminal* is the non-terminal token which the NT2V model generates embedding vector for. The second and third elements are lists representing the contexts of the input *target non-terminal*. We define these two contexts as follows.

4. 2. 1 Non-terminal context

The non-terminal context for a non-terminal token means the surrounding non-terminal nodes of it in an AST. Concretely, we define the n parent non-terminal nodes of a target non-terminal and its all non-terminal children nodes as its non-terminal

context. 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 model does not consider the surrounding non-terminal tokens which are far from the target node as the non-terminal context.

4. 2. 2 Terminal context

We define the terminal context of a target non-terminal as all its children terminal nodes. If a non-terminal node does not have any terminal children nodes (all its children nodes are non-terminal tokens or it does not have any children), we use a special terminal token: *TT-EMPTY* to declare an empty terminal context for it.

We use an example of partial AST to illustrate the contexts of a non-terminal in the Figure 3(1). The nodes whose name starts with "NT" are non-terminal nodes and nodes whose name starts with "TT" represent terminal nodes. In this AST, We assume a target non-terminal node "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 "NT-4". Hyper-parameter n here is specified as two.

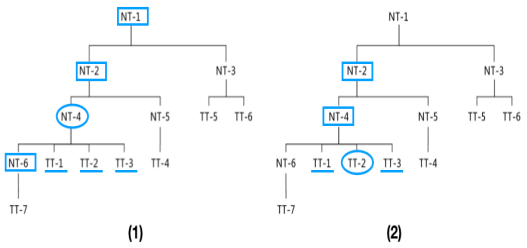

Fig. 3 Non-terminal and terminal context for non-terminal nodes (1) and terminal nodes (2)

4. 3 Embedding for Terminals

The ASTToken2Vec model for terminal tokens' embedding vectors generation is abbreviated as TT2V. Similar with NT2V model, the structure of training tuples for TT2V is (*target terminal, non-terminal context, terminal context*) NT2V model leverage both non-terminal context and terminal context to generate the embedding vectors for the *target terminal*. The non-terminal context and terminal context of a target terminal are described as below.

4. 3. 1 Non-terminal context

Because there is no child node of a terminal node, we only consider the non-terminal context of a terminal as only its n parent non-terminal nodes. Same with NT2V, hyperparameter n is used to define the scope of non-terminal context.

4. 3. 2 Terminal context

We define the terminal context of a target terminal node as m neighbor terminal nodes in an AST. Neighbor terminal nodes of a target terminal mean its sibling terminals which have the same parent non-terminal with the target terminal. Here m is a hyperparameter 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 terminal node does not have any neighbor terminal nodes which means its parent node only has one single terminal child, in this case, we use a special terminal node: *TT-EMPTY* to represent an empty terminal context.

The partial AST in the Figure 3(2) shows a concrete example of what is contexts for terminals. We specify a target terminal node "TT-2" which is surrounded by an oval. Non-terminal nodes: "NT-2" and "NT-4" which are emphasized by a rectangle, represent the non-terminal context and terminal nodes "TT-1" and "TT-3" are the terminal context of the target node. In this example, hyperparameter n is specified as two and m is equal to one.

4. 4 Joint Loss Function

Due to there are two output layers in our ASTToken2Vec model, we design a joint loss function combining the non-terminal context output and terminal context output.

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

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

Equation1 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 $\hat{y}_{nt-context}$ and $y^{i}_{nt-context}$ represents its ground-truth non-terminal context. N is the size of the non-terminal vocabulary.

$$Loss_{tt} = -\sum_{j=1}^{M}(y^{j}_{tt-context} \times log(\hat{y}^{j}_{tt-context})) \quad (2)$$

Equation2 illustrates the formula of $Loss_{tt}$ calculation which also a *log* loss function. $\hat{y}_{tt-context}$ is the terminal output of our model. and $y_{tt-context}$ is the ground truth label of the terminal context. M is the size of terminal vocabulary.

$$Loss_{total} = \alpha * Loss_{nt} + (1-\alpha) \times Loss_{tt} \quad (3)$$

Equation3 is the joint loss function combining $Loss_{nt}$ and $Loss_{tt}$. We utilize a hyperparameter α whose range is from zero to one to adjust the importance between the loss of non-terminal context output $Loss_{nt}$ and terminal context output $Loss_{tt}$.

5 AT2V-LSTM Integration

We integrate a basic LSTM model with our ASTToken2Vec embedding method. This integration model is called AT2V-LSTM which is able to leverage the semantic-based information extracted by ASTToken2Vec embedding and predict the next tokens as code completion.

5.1 Sequences of Training Samples

In order to train the linear-structured LSTM model, we convert ASTs to sequences of training samples. Basically, we first convert an AST to a left-child-right-sibling (LC-RS) binary tree. Then, we transform this LC-RS binary into a complete binary tree by padding a special non-terminal node *NT-EMPTY*. Next, we apply a deep-first in-order traversal on this complete binary tree to generate a visiting sequence of training samples. There are four elements in a sample: (*non-terminal, terminal, node-or-leaf, right-or-left*). Concretely, when a non-terminal node is visited, it is considered as a target *non-terminal* and the first element in the sample. The second element: *terminal* is the children terminals of the target *non-terminal*. If the target *non-terminal* does not have any terminal child, we use a specified terminal token *TT-EMPTY* to represent its empty child. The last two elements: *node-or-leaf* is used to declare whether the *non-terminal* is a leaf or not in the complete binary tree and *right-or-left* represents the position relationship between the target *non-terminal* and its parent node. These two elements are used to reconstruct the predicting AST from the sequence of training samples.

5.2 Model Architecture

The architecture of our integration model is illustrated in the Figure 4. It contains one input layer, one LSTM layer, and an output layer.

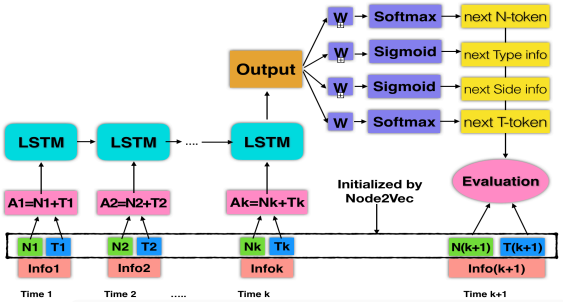

Fig. 4 The architectrue of AT2V-LSTM model

5.2.1 Input layer

The input layer is a combination layer of the representation vectors of feeding elements and these vectors are initialized by ASTToken2Vec embedding. The input is a sequence of training samples and one sample has four elements: (N_i, T_i, NL_i, RL_i) where N_i is a non-terminal, T_i is a terminal, NL_i is the type information of N_i and RL_i is the side information of N_i. All these four elements are encoded by one-hot encoding and multiply embedding matrices as below:

$$Input_i = Concat(A \cdot N_i + B \cdot T_i, \quad C \cdot NL_i + D \cdot RL_i) \quad (4)$$

where A, B are the embedding matrices for non-terminals, terminals and initialized by embedding

vectors generated by ASTToken2Vec. A is a $K \times V_N$ matrix and the shape of matrix B is $K \times V_T$. K is the length of embedding vectors. V_N and V_T are the size of non-terminal vocabulary and terminal vocabulary respectively. C, D are embedding matrices for type information, and side information.

5. 2. 2 LSTM layer

The LSTM layer receives the embedding vectors from the input layer as x_i and takes the output h_{t-1} and hidden state c_{t-1} from the previous state of LSTM layer. Then, the LSTM layer computes three operating gates: forget gate, update gate and output gate to calculate a new hidden state as h_t and the new output as c_t.

5. 2. 3 Output layer

The output layer has four trainable matrices as the linear mapping between the output of the LSTM layer and the prediction. There are four instances our model predicting: next non-terminal N_{i+1}, next terminal T_{i+1}, type and side information of the next non-terminal, NL_{i+1} and RL_{i+1}. The formula of output layer is as below:

$$P_i = softmax(W \times h_i + b) \qquad (5)$$

where P is the prediction of next training sample including p_n, p_t, p_{nl} and p_{rl} representing the prediction of next non-terminal, next terminal, the side and type information of p_n. Ws are four trainable matrices for linear mapping and h_i is the output of the LSTM layer. The softmax function returns the possibility of the next tokens predicting.

6 Experiments

6. 1 Dataset Details

The data we use for both the ASTToken2Vec embedding model training and AT2V-LSTM integration model training is from the same dataset which is a JavaScript AST dataset provided by Raychev et al. [11]. There are 100,000 ASTs as training dataset and 50,000 ASTs as evaluation dataset.

6. 1. 1 Non-terminal Vocabulary

There are 44 different kinds of non-terminal tokens specified by the JavaScript programming language grammar. Base on these 44 non-terminals, we add two more bits of information: whether the non-terminal token has a child token; whether this non-terminal has a right sibling or not. These two bits care more about the surrounding context of non-terminals and make the task of non-terminal predict become more challenge. This adjunction is also used in the previous work [6] [11]. There are 98 kinds of bits-information combination non-terminal tokens in total as the elements in the non-terminal vocabulary including the special non-terminal token:*NT-EMPTY* we use as a padding token to build a complete binary tree from an AST.

6. 1. 2 Terminal Vocabulary

Theoretically, there are infinite kinds of terminal tokens may be included in programs. So, 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 infrequent terminal tokens (out of our terminal vocabulary bag), we use a special terminal token *UNK* to represent these terminals. In total, we have 50,002 tokens in the vocabulary of the terminal including particular terminal *UNK* and *TT-EMPTY* which is used to represent a child terminal for a non-terminal who does not have a terminal child.

6. 2 Experiment of ASTToken2Vec

6. 2. 1 Training details

We implement ASTToken2Vec models to generate embedding vectors for both non-terminals and terminals. We define the size of the hidden layer D is equal to 1,000 so that the length of embedding vectors is 1,000. We specify the adjuster α in the joint loss function as 0.6. We use the Adam optimization algorithm with the learning rate of 0.002 to train models. The size of the training batch for ASTToken2Vec is $b = 100$ and the training epoch is $e = 10$.

6. 2. 2 Visualization

We visualize the representation vectors of several terminal tokens to show the performance of ASTToken2Vec. We first apply principal component analysis (PCA) algorithm to these vectors to reduce the dimension from 1000-d to 2-d. Then we normalize the 2-d vectors with min-max normalization so that the entire range of values of elements is -2 to 2.

We pick up several terminal tokens to visualize, the visualization is shown in Figure 5. Terminal token *Identifiers* are blue, *LiteralNumbers* are rep-

resented by purple, *Property* terminals are green token and red tokens are *LiteralString* in the figure.

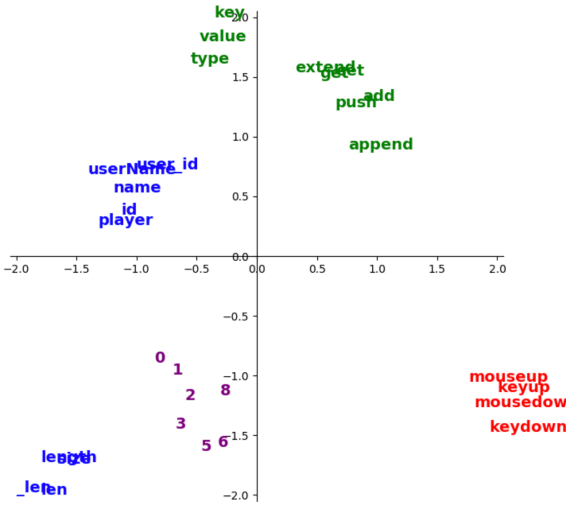

Fig. 5 The visualization of embedding vectors

In the figure, there are several clusters like: "literal string cluster", "property cluster" which represent different types of terminals. We also find that even tokens are the same type, if their meaning is different, they are still not in the same cluster. For example, even the type of "append" and "value" are *Property*, they are still far from each other because "append" is a property which can add some elements to a container in the most cases. However "value" usually is a member in a class without some operation functionality of a container. Another example is "length" and "userName". Due to they have a different meaning, they are not in the same cluster even they are all *Identifiers*

We also calculate the cosine similarity between these embedding vectors and the result meets our conclusion of visualization.

6.3 Experiment of AT2V-LSTM

6.3.1 Training details

We implement two models for code completion: basic LSTM model and our AT2V-LSTM integration model to compare the predicting performance. We use Adam optimization algorithm to train our model with base learning rate 0.0025 and it multiplies 0.9 every epoch as learning rate decay. We clip the gradient which is more than 6 to 6 and less than -6 to -6 to avoid the gradient exploding problem. We specify the time sequence $s = 50$ and the batch size is $b = 100$, therefore, there are $s \times b = 5000$ training instances for one training batch. We train two models $e = 10$ epochs.

Models	Top 1 accuracy	Top 3 accuracy
Basic LSTM	83.5 ± 0.2%	92.6 ± 0.2%
AT2V-LSTM	85.2 ± 0.2%	94.4 ± 0.2%

Table 1 Non-terminal evaluation accuracy

6.3.2 Next Non-terminal Prediction

Valid accuracy curve of the next non-terminal token prediction during the training phase is illustrated in Figure 6. The blue curve represents the validation accuracy of the basic LSTM model and the orange curve is our AT2V-LSTM integration model.

From the validation accuracy curves, we find the non-terminal prediction accuracy of AT2V-LSTM integration model is a little higher than the accuracy of the basic LSTM model. The evaluation result illustrated in the table1 also shows that the accuracy of the integration model is 1.5% higher than the basic baseline model.

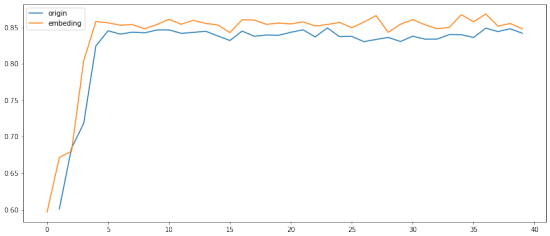

Fig. 6 Validation accuracy for non-terminal prediction during the training phase

6.3.3 Next Terminal Prediction

Figure 7 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 the basic LSTM model. The evaluation accuracy for the terminal in the test phase is shown in Tabel2. From both the evaluation result, we can find that the AT2V-LSTM integration model has a better performance with the predicting accuracy of

Models	Top 1 accuracy	Top 3 accuracy
Basic LSTM	75.8 ± 0.2%	87.7 ± 0.2%
AT2V-LSTM	78.9 ± 0.2%	89.2 ± 0.2%

Table 2 Terminal evaluation accuracy

Models	Type accuracy	Side accuracy
Basic LSTM	97.6 ± 0.2%	94.8 ± 0.2%
AT2V-LSTM	97.8 ± 0.2%	95.1 ± 0.2%

Table 3 Type and side evaluation accuracy

78.9% than the basic baseline model 75.8%.

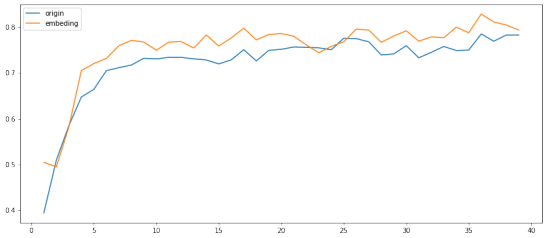

Fig. 7 Validation accuracy for terminal prediction during the training phase

6.3.4 Next Token Information Prediction

The evaluation result of the type (non-leaf or leaf) and the side (right-child or left-child) is shown in the Table3. We can find that both two models achieve a good performance that the accuracy of type information is near to 97% and side information is near to 95%.

6.3.5 Uncommon terminal repeation

We analyze the possible reasons that may cause our integration model to work better for some evaluation cases.

The code snippet in the figure8 is a test case and models predict what token should be filled in the hole. The expecting token is "Property pageY". The basic model predicts terminal "UNK" which means this model consider the token appearing in the hole is a quite uncommon terminal. However, our AT2V-LSTM model gives a correct prediction. From the programming habits and the statistic of the dataset, the "property pageY" is an uncommon terminal only appears in several files but repeats many times in one file. Due to it is uncommon, the basic LSTM model is hard o ltearn enough infor-

```
_move:function(ev) {
  if (this.get("lock")) {
        return false
  } else {
        this.mouseXY =[ev.pageX, ev.pageY,]
        if (!this._dragThreshMet) {
              var diffX = Math.abs(this.startXY[0] -
ev.pageX) diffY = Math.abs(this.startXY[1] - ev.pageY)
              if (diffX > this.get("clickPixelThresh")
|| diffY > this.get("clickPixelThresh")) {
                    this._dragThreshMet = true
                    this.start()
                    this._alignNode([ev.pageX,
ev.pageY,])
              }
        } else {
              if (this._clickTimeout) {
                    this._clickTimeout.cancel()
              }
              this._alignNode([ev.pageX, ev._____,])
        }
  }
}
```

Basic LSTM: UNK
AT2V-LSTM: Property = $$ = pageY
Expect Token: Property = $$ = pageY

Fig. 8 Code snippets for prediction result analysis

matnio and give an incorrect prediction. On the contrary, the AT2V-LSTM integration model can leverage the semantic information of "pageY" extracted by the ASTToken2Vec embedding and give a correct prediction.

7 Conclusion

In this paper, we propose an embedding method for AST nodes called ASTToken2Vec. We integrate it with a basic LSTM model to build an integration model called AT2V-LSTM. From the results of our experiments, we conclude that the ASTToken2Vec model is able to generate the semantic-based embedding representation vectors for tokens. These embedding vectors enable our AT2V-LSTM integration model to leverage more semantic knowledge hidden behind ASTs and to complete code with a higher possibility of predicting next tokens correctly.

References

[1] Allamanis, M. and Sutton, C.: Mining idioms from source code, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 2014, pp. 472–483.

[2] Bielik, P., Raychev, V., and Vechev, M.: PHOG: probabilistic model for code, *International Conference on Machine Learning*, 2016, pp. 2933–2942.

[3] Heinemann, L. and Hummel, B.: Recommending API methods based on identifier contexts, *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, SUITE '11, New York, NY, USA, ACM, 2011, pp. 1–4.

[4] Hindle, A., Barr, E. T., Su, Z., Gabel, M., and Devanbu, P.: On the naturalness of software, *2012 34th International Conference on Software Engineering (ICSE)*, IEEE, 2012, pp. 837–847.

[5] Liang, P., Jordan, M. I., and Klein, D.: Learning programs: A hierarchical Bayesian approach, *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 2010, pp. 639–646.

[6] Liu, C., Wang, X., Shin, R., Gonzalez, J. E., and Song, D.: Neural code completion, (2016).

[7] Maddison, C. J. and Tarlow, D.: Structured Generative Models of Natural Source Code, *CoRR*, Vol. abs/1401.0514(2014).

[8] Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J.: Distributed Representations of Words and Phrases and their Compositionality, *CoRR*, Vol. abs/1310.4546(2013).

[9] Nguyen, T. T., Nguyen, A. T., Nguyen, H. A., and Nguyen, T. N.: A Statistical Semantic Language Model for Source Code, *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, New York, NY, USA, ACM, 2013, pp. 532–542.

[10] Raychev, V., Bielik, P., and Vechev, M.: Probabilistic model for code with decision trees, *ACM SIGPLAN Notices*, Vol. 51, No. 10, ACM, 2016, pp. 731–747.

[11] Raychev, V., Bielik, P., Vechev, M., and Krause, A.: Learning Programs from Noisy Data, *SIGPLAN Not.*, Vol. 51, No. 1(2016), pp. 761–774.

[12] Raychev, V., Vechev, M., and Yahav, E.: Code completion with statistical language models, *Acm Sigplan Notices*, Vol. 49, No. 6, ACM, 2014, pp. 419–428.

[13] White, M., Vendome, C., Linares-Vásquez, M., and Poshyvanyk, D.: Toward deep learning software repositories, *Proceedings of the 12th Working Conference on Mining Software Repositories*, IEEE Press, 2015, pp. 334–345.