ASTToken2Vec: an Embedding Method for Neural Code Completion

Dongfang Li Hidehiko Masuhara

Code completion systems help the programmers to write code more efficiently and to reduce typographical errors by automatically suggesting code fragments that are most likely to be written next. This work attempts to improve a 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 two similar tokens will have representations closed to each other in the vector space. We integrated ASTToken2Vec with a neural network-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 before 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.

Contexts determine the prediction performance. For example, the 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. It can improve prediction since many libraries require client programs to call several functions/methods in specific orders.

Traditional code completion systems make limited use of context information. For example, Eclipse code recommender, while it uses preceding method calls, ignores names of local variables and control flow around the cursor position.

Deep learning-based code completion, which is actively studied in the last few years [6] [11] [12] [13], takes a different approach to exploit context information as well as knowledge of the programming language and APIs. It uses a neural network model, more specifically, a recurrent deep neural network (RNN) that takes a sequence of program tokens and predicts the next token. It can be understood as an application of the RNN-based sentence generation for natural languages.

When applying an RNN to programming languages, structural information in program texts matters to prediction, while typical RNN-based systems for natural languages treat a text merely as a sequence of words. There are several attempts to incorporate structural information in programs into RNN-based neural networks [2] [6] [10]. They basically encode an abstract syntax tree into a sequence of tokens with flags so that the sequence will have sufficient information to reconstruct the tree structure.

ASTToken2Vec: ニューラルネットワークによるコード補完のための記号埋め込みの一手法

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

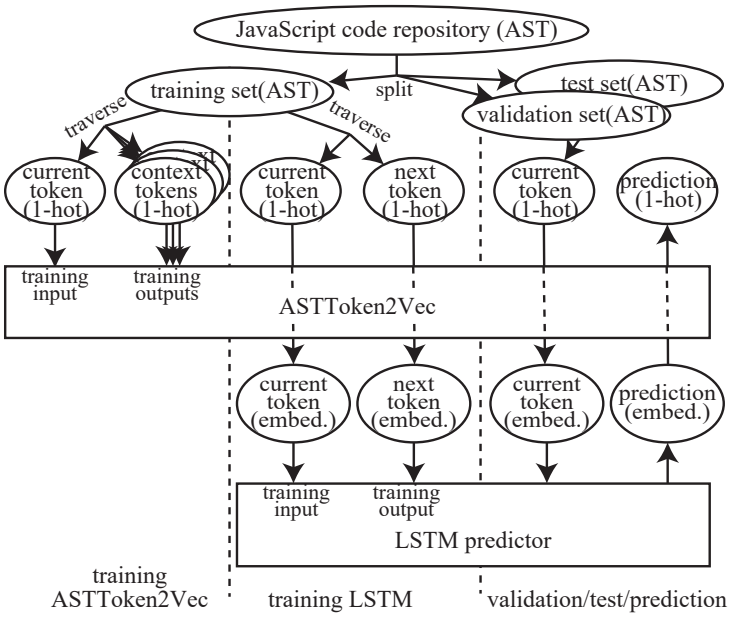

Fig. 1 Overview of ASTToken2Vec LSTM integration model

This paper proposes an embedding method called ASTToken2Vec for abstract syntax tree (ASTs) nodes of a program in order to improve the prediction performance of the deep learning-based code completion. It is a four-layer neural network that takes one AST token along with a few contextual tokens as input, and outputs one vector that represents the AST token. The idea is based on the Word2Vec embedding [8]. Similar to Word2Vec, the intuition behind the embedding is to give two semantically similar tokens vector representations that are close to each other. As a code completion system, whose overview is shown in Figure 1, we use the embedding method as a frontend to an long short-term memory (LSTM)-based neural network that predicts next AST token from a sequence of AST tokens. LSTM is a variant of RNN that explicitly manages long-term dependent information, and known to be useful to many recognition problems in natural languages.

As an experiment, we implemented both the ASTToken2Vec embedding and the AT2V-LSTM model, and trained them with a JavaScript AST dataset [11], which is a collection of open-source programs containing a total of 150,000 JavaScript files. The validity of the embedding is manually confirmed by observing a set of example cases. The experiment with the dataset showed slight improvements in accuracy with the ASTToken2Vec embedding compared to straightforward embedding.

2 Background

Hindle et al. [4] used the *n-gram* technique for code completion. Their work showed that programming code is more repetitive than natural language texts. 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.

Allamanis et al. proposed a statistical nonparametric Bayesian probabilistic tree-based system for extracting programming idioms from a source code text [1]. 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 comple-

tion.

Bielik et al. [2] proposed a generative model for code called the probabilistic higher-order grammar. This model can capture 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. Liu 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

An abstract syntax tree (AST) is a structure that represents structural information of programs. It is widely used in code completion.

An AST consists of two kinds of nodes, namely *non-terminal nodes* and *terminal nodes*. A non-terminal node is a node that has one or more children nodes. For example, in JavaScript, non-terminal nodes correspond to syntax categories such as function declarations, variable declarations, `for` statements, `if` statements, and `while` statements.

A terminal node is a node that appear only at leaf positions in an AST. We call the lexical token of a terminal code as a *value*. In JavaScript, terminal nodes correspond to to syntax categories such as identifiers (variable, field and function names), literal numbers, and literal strings.

4 ASTToken2Vec Embedding

ASTToken2Vec is an embedding neural network model for AST nodes. It converts between a program token and a point in a vector space. The latter representation can be used as an input/output of the neural network model for code completion.

Intuitively, we design ASTToken2Vec to incorporate similarities of tokens into embedded representations, where two similar programs tokens will have two points in the vector space that are close to each other. By similarity, we here mean similarity in natural language semantics (such as "`size`" and "`length`") as well as similarity in the API usage (such as `Stack.peek()` and `Stack.pop()`).

ASTToken2Vec is inspired by Word2Vec [8], which gives vector representations to natural language words. Differently from Word2Vec, ASTToken2Vec uses two identical networks for embedding non-terminal and terminal tokens. It also separately provides non-terminal and terminal tokens as context information for training.

4.1 Model Architecture

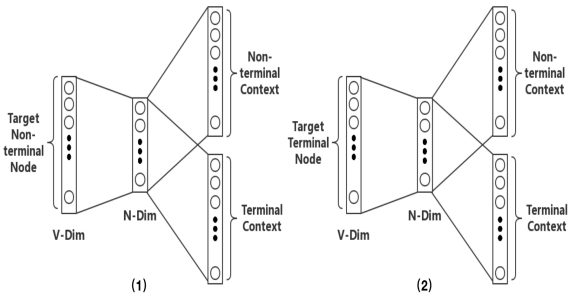

Fig. 2 Architecture of NT2V (1) and TT2V (2)

ASTToken2Vec is a pair of four layers neural networks, each contains one input layer, one hidden layer, and two output layers as shown in Figure 2. Two identical, yet independent networks are for embedding non-terminal and terminal nodes, respectively. We hereafter call the networks NT2V and TT2V, respectively.

The input layer uses the one-hot encoding representation, where i'Th token in a vocabulary of V different tokens is represented as an V-dimensional vector all but i'Th element is zero. The two output layers represent the context of non-terminal and terminal nodes, each of which has the same vector size to the non-terminal/terminal input vectors. We use the values in the hidden layer as the embedding vector of the input values. The size of the hidden layer, a hyperparameter D, is determined by heuristics. In the training phase, we calculate a joint loss function to update the model.

4.2 Embedding for Non-terminals (NT2V)

The NT2V embedding uses both the terminal and non-terminal contexts for training. 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 in an AST. Concretely, we define the n ancestor non-terminal nodes of a target non-terminal and its all non-terminal children nodes as its non-terminal context. Here, n is a hyperparameter.

4.2.2 Terminal context

We define the terminal context of a non-terminal as all the direct children terminal nodes.

We use an example of a partial AST to illustrate the contexts of a non-terminal in the Figure 3. The boxes and ovals are non-terminal and terminal nodes, respectively. Let us focus the middle non-terminal node "minv" (method invocation). The context of this node (where $n = 2$) is illustrated as a shadowed area, consisting of the two ancestor nodes, namely "if" and "block", one non-terminal child "fget" (field get) and the three terminal nodes, namely `cancel`, `flag` and `timeOut`.

4.3 Embedding for Terminals (NT2V)

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.

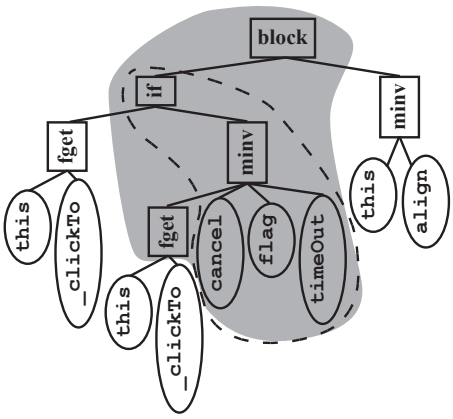

Fig. 3 An example of contexts. Boxes and ovals are non-terminal and terminal nodes, respectively. The nodes in the gray area are the context of the method invocation (minv) node in the middle. The nodes surrounded by a dashed line are the context of the terminal node `flag`.

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 shows a concrete example of what is contexts for terminals. We focus the target terminal node `flag`. The context of the node, where $n = 2$ and $m = 1$, is the area surrounded by a dashed line in the figure, consisting of two non-terminal nodes "if" and "minv", and two terminal nodes `cancel` and `timeOut`.

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. $Loss_{total}$ is the final joint loss function for our model's training.

$$Loss_{NT} = -\sum_{i=1}^{N}(y^i_{NTcontext} \times \log(\hat{y}^i_{NTcontext})) \quad (1)$$

Equation (1) is the $Loss_{NT}$ calculation formula which is a logarithmic loss function. Concretely, for an input token x, ASTToken2Vec model calculates the non-terminal context output as $\hat{y}_{NTcontext}$ and $y^i_{NTcontext}$ represents its ground-truth non-terminal context. N is the size of the non-terminal vocabulary.

$$Loss_{TT} = -\sum_{j=1}^{M}(y^j_{TTcontext} \times \log(\hat{y}^j_{TTcontext})) \quad (2)$$

Equation (2) illustrates the formula of $Loss_{TT}$ calculation which also a logarithmic loss function. $\hat{y}_{TTcontext}$ is the terminal output of our model. and $y_{TTcontext}$ is the ground truth label of the terminal context. M is the size of terminal vocabulary.

$$Loss_{total} = \alpha Loss_{NT} + (1-\alpha) Loss_{TT} \quad (3)$$

Equation (3) 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.

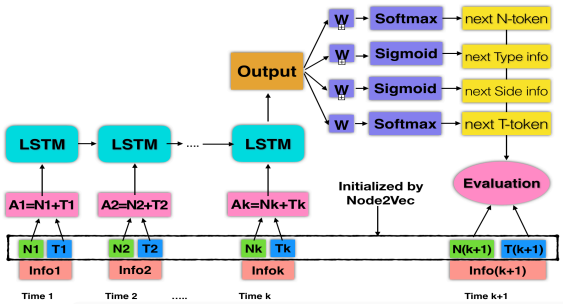

Fig. 4 The architecture of AT2V-LSTM model

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:

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

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:

$$P_i = softmax(W \times h_i + b) \tag{5}$$

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

The dataset we used for both ASTToken2Vec and AT2V-LSTM is the same one in the previous work [11]. It consists of 150,000 JavaScript ASTs, each of which corresponds to one open-source program file. We used 100,000 and 50,000 of them for training and evaluation, respectively.

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 prediction more challenging. This adjunction is also used in the previous work [6] [11]. In total, there are 98 different tokens in the vocabulary including the special token, namely *NT-EMPTY*, for representing the end of children nodes.

6.1.2 Terminal Vocabulary

In order to deal with too many different terminal tokens, we keep top 50,000 most frequently occurred terminal tokens in the original ASTs while replacing the rest tokens with the special token *UNK* (unknown). We have 50,002 tokens in the vocabulary after added another special token *TT-EMPTY* for representing absence.

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 (i.e., the vector length of the embedded representation) as 1,000. We specify the parameter 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 to 2. 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 examined distributions of several terminal to-

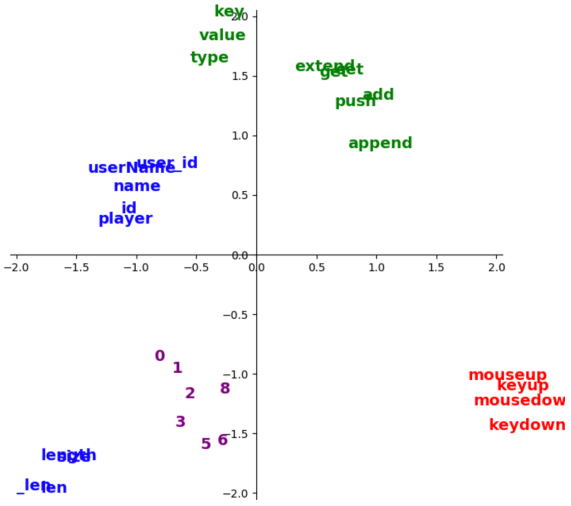

Fig. 5 The visualization of embedding vectors

kens as shown in Figure 5 in order to see whether the embedding represents intuitive similarity. In the figure, the names of tokens are written on the projected position of its vector representation. The clusters of tokens near `length` (around the bottom left corner) and near `name` (around top left) are identifiers. The numbers are clustering around the bottom right. The two clusters at the top (i.e., around `value` and `push`) are properties, which appearing as a field name of an object. The cluster at the bottom right (i.e., near `mouseDown`) are string literals[†1].

We can make the following observations.

- Different kind of terminal tokens (e.g., identifiers and property names) make different clusters. This would mean that the embedding reflects the usage of tokens.
- Tokens that have similar meaning in the natural language are placed at the positions close to each other. For example, `length`, `size`, `len` and `_len` make one cluster.
- Similar to Word2Vec, the embedded representation would be compositional. The positions of `mouseup`, `mousedown`, `keyup` and `keydown` suggest that their positions can be decomposed into the four vectors, namely "mouse," "key," "up," and "down."

†1 In JavaScript, field `f` of object `o` can be accessed by `o.f` as well as `o['f']`, some string literals are as important as field names.

Though we here can only discuss based on intuitive similarity, we would conclude the result of the embedding is reasonable.

6.3 Experiment of AT2V-LSTM

We next use the proposed embedding for code completion to see whether it can improve accuracy of prediction.

6.3.1 LSTM Training

We implemented and compared two code completion models, namely the baseline LSTM model and our AT2V-LSTM model. The two models have the LSTM model yet use different representations of tokens. 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.

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 Table 1 also shows that the accuracy of the integration model is 1.5%-point better than the baseline model.

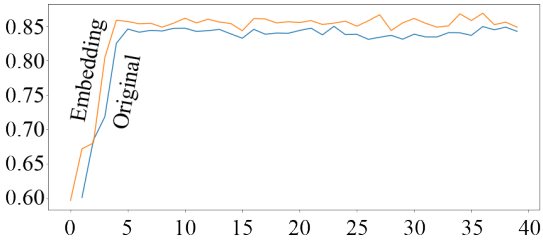

Fig. 6 Validation accuracy for non-terminal prediction during the training phase

Models	Top 1 accuracy	Top 3 accuracy
Baseline 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

Models	Type accuracy	Side accuracy
Baseline 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

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

Table 2 Terminal evaluation accuracy

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 baseline LSTM model. The evaluation accuracy for the terminal in the test phase is shown in Table2. From both the evaluation result, we can find that the AT2V-LSTM integration model has a better performance with the predicting accuracy of 78.9% than the baseline model 75.8%.

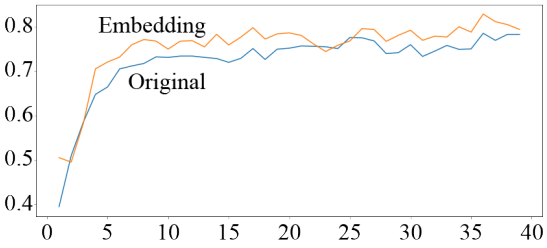

Fig. 7 Validation accuracy for terminal prediction during the training phase

6.3.4 Next Token Type Prediction

The evaluation result of the type (non-leaf or leaf) and the side (right-child or left-child) is shown in Table 3. 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 Manual comparison

To understand the effect of embedding, we manually analyzed the cases where two models gave different answers. Although this is not systematic comparison at all, some cases gave us potential strength of the ASTToken2Vec.

```
_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.[HOLE],])
      }
  }
}
```

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

Fig. 8 A case where embedding gave different results

The code snippet in Figure 8 is a test case where two models predicted different tokens. The hole, shown as a light blue box in the code, is the token to predict. The answer here was `pageY`, which was correctly predicted by ASTV-LSTM but not by the baseline model, which answered "UNK."

One possible explanations for the difference is that the number of examples that use `pageX` and `pageY`. As we can intuitively imagine those two tokens are not as common as `length` or `key`. This would mean, even if the training dataset has code with `pageX` and `pageY`, it probably does not have the code that uses them in the same way (i.e., as a pair of field accesses in an array construction). Therefore, it would be reasonable for the baseline model to answer "UNK."

With ASTToken2Vec, it would be possible to learn that `pageX` and `pageY` appear as field names of an event. Then they would probably have vector representations that are close to other field names

of events. Then, if there were code fragments in the training dataset that construct an array of two field values of an event, that knowledge can be applied to `pageX` and `pageY` thanks to similarity.

7 Conclusion

We proposed an embedding method for AST nodes called ASTToken2Vec. We integrated the method with an LSTM model for code completion, which we call AT2V-LSTM. From the experiments, we manually observed that ASTToken2Vec produces embedding that reflect intuitive similarities of token names and roles. The application of ASTToken2Vec to the LSTM code completion exhibited slight improvements in prediction accuracy.

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*, 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, 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, 2013, pp. 532–542.

[10] Raychev, V., Bielik, P., and Vechev, M.: Probabilistic Model for Code with Decision Trees, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, 2016, pp. 731–747.

[11] Raychev, V., Bielik, P., Vechev, M., and Krause, A.: Learning Programs from Noisy Data, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, 2016, pp. 761–774.

[12] Raychev, V., Vechev, M., and Yahav, E.: Code Completion with Statistical Language Models, *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, 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.