A DEEP LEARNING BASED APPROACH TO ARGUMENT RECOMMENDATION

Guangjie Li, Yi Tang, Biyi Yi, Xiang Zhang and Yan He

National Innovation Institute of Defense Technology, Beijing, China

ABSTRACT

Code completion is one of the most useful features provided by advanced IDEs and is widely used by software developers. However, as a kind of code completion, recommending arguments for method calls is less used. Most of existing argument recommendation approaches provide a long list of syntactically correct candidate arguments, which is difficult for software engineers to select the correct arguments from the long list. To this end, we propose a deep learning based approach to recommending arguments instantly when programmers type in method names they intend to invoke. First, we extract context information from a large corpus of open-source applications. Second, we preprocess the extracted dataset, which involves natural language processing and data embedding. Third, we feed the preprocessed dataset to a specially designed convolutional neural network to rank and recommend actual arguments. With the resulting CNN model trained with sample applications, we can sort the candidate arguments in a reasonable order and recommend the first one as the correct argument. We evaluate the proposed approach on 100 open-source Java applications. Results suggest that the proposed approach outperforms the state-of-the-art approaches in recommending arguments.

KEYWORDS

Argument recommendation, Code Completion, CNN, Deep Learning

1. INTRODUCTION

Code completion is one of the most widely used Eclipse features by developers. It may automatically completes the rest part of an expression or statement when developers type in the first several characters, which helps speed up coding and as a result the whole process of software development.

Argument recommendation is a special kind of code completion. Most of the mainstream IDEs recommend actual arguments for method calls when developers type in method names. However, such IDEs only provide a long list of candidate arguments according to the corresponding types of formal parameters, which may take a long time for developers to select the correct one from the list of type compatible candidate arguments.

To facilitate the process of development, a few approaches have been proposed to recommend arguments. Zhang et al. [4] recommend arguments for method invocations based on the nearest k usages of them. Raychev et al. [6] recommend arguments for method invocations based on statistical language model. Such approaches can only work well for methods with richful

David C. Wyld: CSITY, NWCOM, SIGPRO, ASOFT, AIFZ, BDIoT, ITCCMA, CLSB, DTMN, MLNLP - 2021
pp. 87-93, 2021. CS & IT - CSCP 2021 DOI: 10.5121/csit.2021.111408

invocation histories, however, a large number of methods in practice have less or no invocation history before the current method call, consequently the state-of-art approaches cannot be used to recommend arguments for such method call. For example, according to Li et al. [1], nearly one half of method invocations are non-API method invocations, i.e., methods defined within the projects.

To this end, in this paper we propose a deep learning based approach to recommend arguments for both API method invocations and non-API method invocations based on features extracted from the context of method invocations. First, we extract context information for each method invocation from a large number of open-source applications, which involves method names, formal parameters, actual arguments, type compatible variable names, type compatible and visible method names. Second, we perform natural logarithm transformation and normalization to the dataset. Third, we feed the preprocessed dataset to a specially designed Convoluntional Neural Network (CNN) so as to learn the general rules of mapping candidate arguments into parameters. Fourth, we rank the candidate arguments according to the predicted probabilities of being actual arguments in descending order, and recommend the first one as the actual argument. Evaluations on 100 open-source applications suggest that the proposed approach outperforms the state-of-art approaches in recommending arguments for method invocations.

This paper makes the following contributions:

- To the best of our knowledge, it is the first one in recommending and ranking arguments for methods.
- Evaluations on real-world open-source applications suggest that the proposed approach outperforms the state-of-the-art approach in recommending arguments for method invocations.
- We exploit natural language processing techniques to mine lexical similarities embedded in software identifiers, and exploit word embedding and deep learning techniques to mine semantic similarities between related program entities.

The rest of the paper is organized as follows. Section 2 describes the proposed approach. Section 3 presents an evaluation of the proposed approach on open-source applications. Section 4 presents related works. Section 5 provides conclusions.

2. APPROACH

2.1. Overview

The rationale of the approach is that we can select correct argument from a list of syntactically candidate arguments based on method invocation contexts. Consequently, we train an CNN (convoluntional neural network) model with training data, i.e., actual arguments and their context from open source applications, and then rank and recommend correct arguments for new call sites based on the resulting neural network. An overview of the proposed approach is presented in Figure 1.

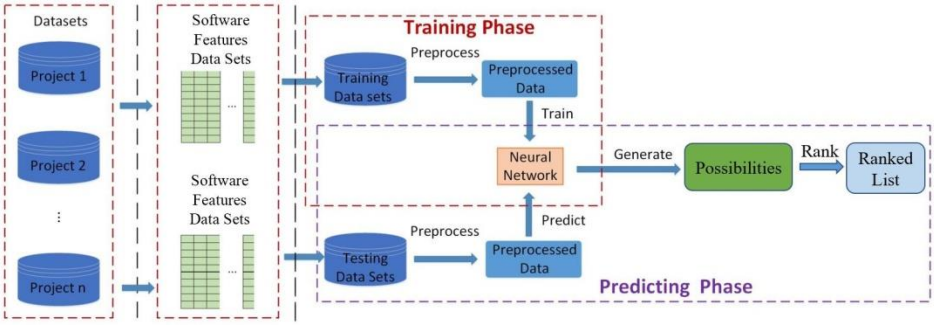

Fig. 1. Overview of the proposed approach.

For a given argument in a method invocation, we exploit the JDT (Eclipse plug-in tool) to parse the ASTs of Java files statistically and extract the following context information:

- Ar: the actual argument name.
- Fn: the method name.
- Par: the formal parameter name.
- Lvs: all variables visible and type compatible.
- Mvs: all methods visible and type compatible.

where Lvs and Mvs are a list of candidate arguments, respectively. It should be noted that, for a given recommendation position, we only consider those expressions as candidate arguments when choosing them as the actual argument will not induce syntactical errors.

2.2. Data Preprocessing

To rank and recommend candidate arguments, we need to preprocess data extracted from program and feed them into the CNN, which involves the following steps. First, we split each identifier into a sequence of tokens by exploiting underscore and capital letter as separators. Second, we exploit Word2Vec [3] to embed the token sequences of each identifier into numerical vectors. Third, if the candidate arguments are more than five, we only remain the top five candidates who are lexical similar to the parameters based on computing Jaccard similarity [13].

2.3. CNN-Based Architecture

The architecture of the neural network for argument recommendation is presented in Figure 4. The model consists of five input layers, five convolutional layers, two fully connected layers, and one output layer. Preprocessed data are divided into groups and each group is input to the corresponding convolutional neural network respectively. We set the input dimensions, output dimensions, kernel initializer and activation function for each layer as follows:

- Convolutional layers: kernel_initializer =glorot_uniform, activation function = Softmax, pooling =MaxPooling1D, and dropout = 0.25.
- First fully connected layer: output_dim = 32, activation function =Softmax, dropout = 0.5.
- Second fully connected layer: out_dim =5.

The output of the CNN-based network is the possibilities of candidates as the actual argument for a given recommendation requirement, and the proposed approach selects the one with highest

possibility as the recommended one. Each CNN layer is forwarded to a flatten layer, the merge layer merges the outputs of the flatten layers as a vector, and feed them into the fully connected dense layer. Finally, the dense layer outputs the prediction for each instance.

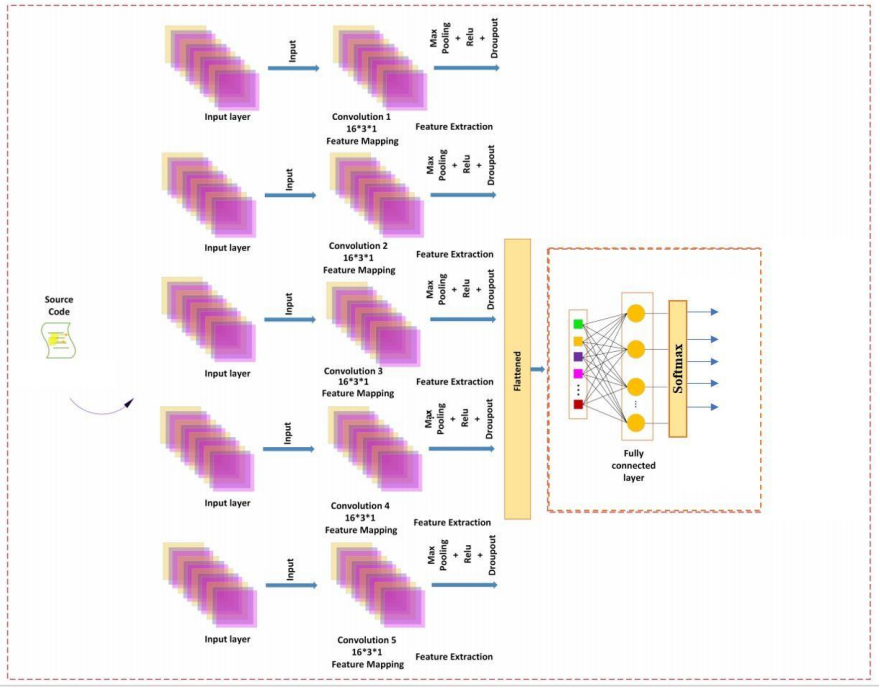

Figure 2. Overview of the CNN-based classifier

3. EVALUATION

This section specifies the setup of the evaluation, research questions, and metrics employed to evaluate the performance of the proposed approach. To evaluate the state-of-art argument recommendation approach, we select the similarity-based approach for comparison because of the following reasons. First, the similarity-based approach is the most recently proposed argument recommendation approach for method invocations. Second, the similarity-based approach does not rely on richful invocation history of the invoked method, which is similar to our approach. Third, the source code of the similarity-based approach is publicly available, which makes it easy to repeat their experiment.

We evaluate the proposed approach on real-world applications. We search for most popular (stars) 100 open-source Java applications from GitHub as the subject applications, select 90 of the resulting applications as training dataset, and the left 10 applications as testing dataset.

3.1. Research Questions

The evaluation investigates the following research questions:

- RQ1: Does the proposed approach outperform the state-of-art approach in recommending arguments for method invocations?

- RQ2: How long does it take to train the neural network model, and how long does it take to generate recommendation for a given method invocation?

- Research question RQ1 validates the performance of the proposed approach. Answering this question may reveal whether deep learning techniques outperform fine-grained heuristic rules in mining semantic relationships.

- Research question RQ2 reveals the efficiency of the proposed approach. Answering this question may help to validate whether the proposed approach can be applied in practice.

3.2. Metrics

To measure the performance of the approaches, we define precision and recall as follows:

$$\text{precision} = \frac{\text{Num}_{accepted}}{Num_{recommended}} \quad (1)$$

$$\text{recall} = \frac{\text{Num}_{accepted}}{Num_{\text{tested}}} \quad (2)$$

$$\text{F1} = 2* \frac{\text{precision} * \text{recall}}{precision + recall} \quad (3)$$

where $\text{Num}_{\text{accepted}}$ is the number of correct recommendations, $\text{Num}_{\text{recommended}}$ is the number of generated recommendations, and the $\text{Num}_{\text{tested}}$ is the number of arguments extracted from the object applications.

3.3. Results

Evaluation results are presented in Fig 3. From this figure, we observe that the proposed approach significantly outperforms existing approaches in recommending arguments.

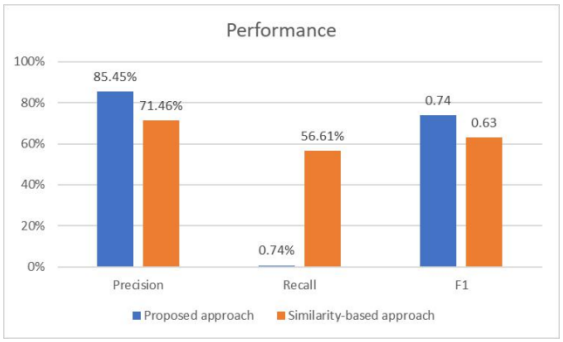

Figure 3. Evaluation Results

We also evaluate time efficiency of the proposed approach in training and testing phrase, respectively. Training task is conducted on a special workstation with the following configuration: 2.0GHz Intel Xeon E5-2683 processor, 64GB RAM, TITAN Xp GPU, with Linux installed. Testing is conducted on a personal computer with the following configuration: Intel Core i7-6700 CPU 3.4 GHz, 16 GB RAM, with Windows 7 installed. Evaluation results suggest that it takes around 89 minutes to train the CNN model and around 11 milliseconds on average to make recommendation for each argument requirement.

4. RELATED WORKS

N-gram language model, rooted in statistical natural language processing, has been proved to be successful in capturing the repetitive and predictable regularities of source code [2]. Consequently, a series of n-gram based approaches have been proposed to predict the naturalness of code. Hindle et al. [2] recommend the next code token based on the preceding n tokens by training n-gram learning model. Allamanis et al. [6] exploit n-gram models to recommend variable names, method names and class names. Tu et al. [4] exploit the localness of source code in recommending the next token by adding a cache component to the n-gram

model, i.e, assigning a higher probability to tokens occurred in the source file where the n-gram model based prediction is applied. Hellendoorn et al. [11] model and complete source code based on a nested and cached n-gram based approach. Based on the naturalness and localness of source code, they assign a higher probability to tokens most recently used by adding a cache mechanism to the n-gram model. Raychev et al. [7] exploit statistical language models and conditional random fields in predicting local variable names for JavaScript applications. Nguyen et al. [5][8] exploit graphic probability models to recommend the next API method call.

Neural network based approaches are also proposed to improve code completion. White et al. [9] exploit deep learning to model software and illustrate that deep learning based approach is more effective than n-gram based one. Murali et al. [10] train a deep learning based model to generate code fragment for program sketches that heavily dependent on APIs. Most recently, Liu et al.[12] propose a similarity-based approach to recommend arguments. They just select the candidate who has the greatest lexical similarity with the corresponding parameter as the recommended argument.

5. CONCLUSIONS

In this paper, we propose a deep learning based approach to rank candidate arguments and recommend actual argument for method invocations. By statistically parsing Java source files from open-source applications, we extract each actual argument and the corresponding context information from each method invocation, represent them in vectors, and feed them into a specially designed CNN classifier to learn the rules of selecting correct arguments. Evaluations on 100 open-source applications suggest that the proposed approach outperforms the state-of-art approaching in recommending arguments. The insight of the approach is that deep learning techniques can effectively learn the semantic similarity between related software entities, and they can be used to facilitate software engineering task.

REFERENCES

[1] Guangjie Li , H Liu, Ge Li , et al. LSTM-based argument recommendation for non-API methods[J]. Science China (Information Sciences), 2020(9).

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

[3] T. Mikolov, K. Chen, G. Corrado, and J. ean, “Efficient estimation of word representations in vector space,” Computer Science, 2013.

[4] Tu Z, Su Z, Devanbu P. On the localness of software. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2014. 269–280

[5] A. T. Nguyen and T. N. Nguyen, “Graph-based statistical language model for code,” in Proceedings of the 37th International Conference on Software Engineering - Volume 1, ser. ICSE’15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 858–868. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818858

[6] M. Allamanis and C. Sutton, “Mining source code repositories at mas sive scale using language modeling,” in 2013 10th Working Conference on Mining Software Repositories (MSR), May 2013, pp. 207–216.
[7] V. Raychev, M. Vechev, and E. Yahav, “Code completion with statistical language models,” in Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI ’14. New York, NY, USA: ACM, 2014, pp. 419–428. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594321
[8] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, “Learning api usages from bytecode: A statistical approach,” in Proceedings of the 38th International Conference on Software Engineering, ser. ICSE’16. New York, NY, USA: ACM, 2016, pp. 416–427. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884873
[9] White M, Vendome C, Linares-Vasquez M, et al. Toward deep learning software repositories. In: Proceedings of the 12th Working Conference on Mining Software Repositories. Piscataway: IEEE Press, 2015. 334–345
[10] Murali V, Qi L, Chaudhuri S, et al. Neural sketch learning for conditional program generation. 2017. ArXiv: 1703.05698
[11] Hellendoorn V J, Devanbu P. Are deep neural networks the best choice for modeling source code? In: Proceedings of Joint Meeting on Foundations of Software Engineering, 2017. 763– 773
[12] Liu H, Liu Q, Staicu C A, et al. Nomen est omen: exploring and exploiting similarities between argument and parameter names. In: Proceedings of the 38th International Conference on Software Engineering. New York: ACM, 2016. 1063–1073
[13] W. W. Cohen, P. D. Ravikumar, and S. E. Fienberg, “A comparison of string distance metrics for name-matching tasks,” in Proc. Workshop Inf. Integr. Web (IIWeb), 2003, pp. 73–78.

AUTHORS

Guangjie Li received the B.S. and M.S. degrees in educational technology from Shen Yang Normal University in 2002 and 2005, respectively. She received her Ph.D. degree in computer science from and technology from Beijing Institute of Technology in 2020. She is is interested in software quality and software evolution.