SCIENCE CHINA
Information Sciences

• LETTER •

September 2020, Vol. 63 190106:1–190106:3
https://doi.org/10.1007/s11432-019-2777-2

Special Focus on Big Data-Driven Intelligent Software Development for Onsite Programming

Code line generation based on deep context-awareness of onsite programming

Chuanqi TAO[1,2,3*], Panpan BAO[1] & Zhiqiu HUANG[1,2]

[1]*College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing* 211100*, China;*
[2]*Ministry Key Laboratory for Safety-Critical Software Development and Verification, Nanjing University of Aeronautics and Astronautics, Nanjing* 211100*, China;*
[3]*National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing* 210023*, China*

Received 24 September 2019/Revised 26 December 2019/Accepted 20 January 2020/Published online 7 August 2020

Citation Tao C Q, Bao P P, Huang Z Q. Code line generation based on deep context-awareness of onsite programming. Sci China Inf Sci, 2020, 63(9): 190106, https://doi.org/10.1007/s11432-019-2777-2

Dear editor,
Intelligent code generation has become an essential research task to accelerate modern software development. To facilitate effective code generation for programming languages, numerous approaches have been proposed to generate token or tokens by mining existing open software repositories, e.g., Nguyen et al. [1] and White et al. [2]. Currently code token or tokens generation are used to recommend specific API or variables. There is a lack of code generation strategy for developers to finish a complete code line. A feasible solution to code line generation is to extract relevant context factors between code lines and mine hidden context information based on the existing massive source code using deep learning. The trained model can generate target code lines by using the existing source code data and task data in onsite programming. Onsite programming here refers to the set of elements related to the current programming in software production, such as source code data, task data, and other related data. In this letter, we only use source code data and task data according to the actual requirements. Besides, owing to the variety of software programmers, there exists a clear quality issue in projects on GitHub frequently. To address these issues, we propose a context-sensitive approach named DA4CLG (deep context-awareness for code line generation) to generate code lines. The deep context-awareness in title means our method uses deep learning to extract the relevant contextual factors in code lines. Our contributions are as follows. (1) An approach to data quality analysis for open source code is proposed to ensure high quality of training data set. An evaluation framework is proposed to analyze code data quality of open source projects and construct data set with quality assurance. (2) The potential pattern of source code context from the existing large-scale open-source data sets is obtained based on deep learning models. Then we use the existing source code data in onsite programming to generate code lines. (3) Developer intention is acquired based on task data in onsite programming and code lines are re-ranked through semantic similarity matching, to ensure the rank of satisfying recommended items for developers are higher in the recommended code line list.

DA4CLG consists of three main processes (marked as (1), (2), (3) in Figure 1). The detailed process is shown below.

(1) Constructing a large-scale data set including methods with quality assurance. The proposed approach is based on deep learning model. According to 'garbage in and garbage out', the model performance depends on training data. We prefer to select methods (code segment) with fewer bugs, good structure, definite functions (e.g, 'read

* Corresponding author (email: taochuanqi@nuaa.edu.cn)

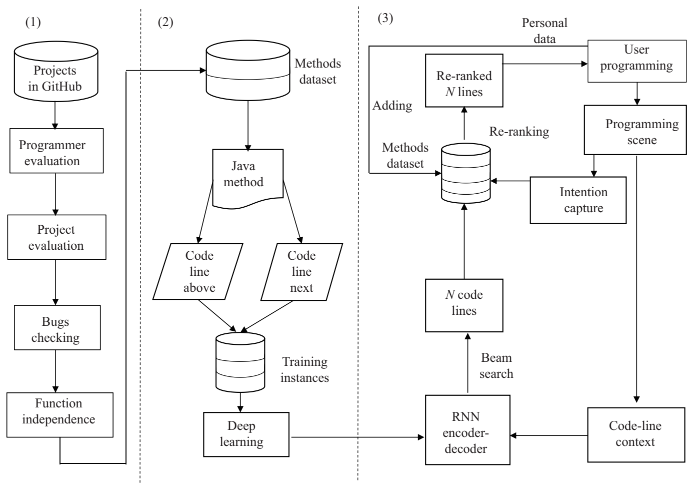

Figure 1 Overview of DA4CLG.

file content') as well as not invoking other methods programmed in the same project. Aiming to address the quality problem, an evaluation framework is proposed to analyze code data quality of open source projects on GitHub based on defined quality dimensions from different perspectives.

We evaluate the quality of source code structure by gathering information of programmers and projects on GitHub. Specifically, we calculate the value of Watch/(Watch+Star+Fork) to evaluate whether the programmer is experienced or not. Because when we investigate the behavior to get more artificial stars in GitHub project, we found that there may be the behavior to get more artificial stars and forks at the same time, but no artificial watches. Therefore, we use the watch number as the numerator of the division. To calculate the threshold, we select 365 well-established programmers and calculate total Watch, Star and Fork numbers of their projects. After removing the top 30 values and bottom 30 values from 365 Watch/(Watch+Star+Fork) values, the average value of the remaining is used as a reference threshold value. The threshold is 0.058. Besides, we use Watch, Star, Fork, Issues, Pull requests and Commits on GitHub to evaluate project quality. To calculate the threshold for each of them, we gather 6000 projects randomly. To reduce the influence of extreme value, we calculate the corresponding value of each indicator on 6000 items, sort them from high to low, and then obtain the average values after removing the top 10% values and the bottom 10% values. Corresponding threshold for each metric is 11, 76, 28, 4, 1, and 58. Thus, projects with higher corresponding metric value than threshold perform better on GitHub. To construct a project data set with higher quality, we select projects as follows: (i) Programmer's metric on Watch/(Watch+Star+Fork) is greater than 0.058. (ii) Projects have higher corresponding metric value than threshold value 11, 76, 28, 4, 1, and 58.

We use PMD (an extensible cross-language static code analyzer) to check source code. PMD checks Java code through its built-in rules, including the potential bugs, unused code and repetitive code and so on. These rules cover the possible defects in source code. According to the opinions of 30 programmers with 4–6 years of java development experience, 8 important PMD rules, including basic, braces, unused code, string, strict exception, naming, design, and coupling, are finally selected. To compute each number of defects in a java method, we perform PMD on 45 java projects with high quality. Corresponding number for each rule is 2, 1, 1, 3, 1, 4, 0, and 1, respectively.

Through parsing source code files into ASTs (abstract syntax trees), we can validate function independence relationship through checking whether a java method invokes other methods in the project. Finally, we construct a large-scale java method data set with higher quality.

(2) Training instances construct and model training. In the second step, java methods selected in step (1) are used to extract training instances and train our deep learning model. The number of java projects is about 6000 and some of the popular java projects and some common jar package projects are also included in these projects.

We analyze all classes, record field declarations together with their type bindings and unify different variable values. We replace all object types with their real class types. The process above is beneficial to the learning of the context of code lines. When building training instances, we ignore the method declaration and start from line 4 to the last line of a method. For example, we have code lines in a method:

1: String line = "stringValue";
2: for (String str: list⟨String⟩);
3: if (str.contains ("stringValue"));
4: line = line + str.trim().

The first 3 lines, String line = "stringValue", for (String str: list⟨String⟩), if (str.contains ("stringValue")), act as model input, and the code line in line 4, line = line + str.trim(), acts as output (label) of model.

To learn the potential pattern of source code context, we use the attention-based recurrent neural network (RNN) encoder-decoder model and long short-term memory (LSTM). As a specific application of RNN with better performance, LSTM has achieved good results in many applications [3]. Specifically, we construct the model as follows: in the decoder, we use two RNNs for the encoder, including a forward RNN that directly encodes the source sentence and a backward RNN that encodes the reversed source sentence. We set all RNN here with 1000 hidden units and the dimension of word embedding with 120.

(3) Generating code lines and re-ranking. We have discussed the model training in step (2). Now we present the approach to applying the trained RNN encoder-decoder model to generate code lines. In the programming language, there is a certain disorder between code lines. It means exchanging the order of some lines will not affect the realization of program functions. Therefore, based on existing code lines, the target lines may not be unique. Hence, a good model should generate N target code lines which might be correct and rank them according to possible priorities. Our method collects the code lines that the developer has already entered in onsite programming, then the trained model generates the most likely target code lines based on Beam Search [4]. To enhance the rank of items from a developer perspective in the recommended code line list, we need to re-rank the recommended results based on the similarity. We use latent semantic analysis (LSA) to measure semantic similarity to improve the accuracy of information retrieval. We calculate the similarity between methods in onsite programming and methods programmed by the same writer in history. The code line is re-ranked if the similarity value is more than 0.7.

To evaluate DA4CLG, we compare our results with Lexical n-gram and RNN. To reduce bias or mistakes, we check all results (500 sets of test cases) carefully by two authors and two non-authors. In all cases, in Top1, DA4CLR achieves 0.18 higher than the Lexical n-gram and 0.13 higher than RNN on BLEU (bilingual evaluation understudy) measure. The BLEU measures how close a candidate sequence is to a reference sequence. When the recommended number of items is 5 and 10, the average BLEU score between DA4CLR, Lexical n-gram, and RNN is similar. We discover that only a few code lines are useful in all 10 lines. When generating 10 code lines, the corresponding mean reciprocal rank (MRR) score of RNN and Lexical n-gram is much smaller than DA4CLR. The higher MRR value represents the overall ranking of the recommended items list is better. This indicates that our re-ranking method based on similarity achieves good performance. Overall, the study results show DA4CLR can improve the precision and enhance the rank of code line list.

Conclusion. We propose a code line generation approach based on deep context-awareness of onsite programming. We take into account data quality of open source projects and information in onsite programming. Besides, we re-rank the code lines list by semantic similarity matching, thereby the items needed by the developer are set higher in the rank list. This approach focuses on code lines and task data in onsite programming and limits to java language. In future work, we will improve the proposed DA4CLR through extension to other programming languages and apply it to different granularity from segment to API. Moreover, we will continue the related work of intelligent programming with data model quality assurance.

Acknowledgements This work was supported by National Key R&D Program of China (Grant No. 2018YFB1003900), National Natural Science Foundation of China (Grant Nos. 61602267, 61402229), Open Fund of the State Key Laboratory for Novel Software Technology (Grant No. KFKT2018B19), and Fundamental Research Funds for the Central Universities (Grant No. NS2019058).

References

1 Nguyen T T, Nguyen A T, Nguyen H A, et al. A statistical semantic language model for source code. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, Saint Petersburg, 2013. 532–542

2 White M, Vendome C, Linares-Vásquez M, et al. Toward deep learning software repositories. In: Proceedings of the 12th Working Conference on Mining Software Repositories, Florence, 2015. 334–345

3 Greff K, Srivastava R K, Koutník J, et al. LSTM: a search space odyssey. IEEE Trans Neur Net Lear Syst, 2016, 28: 2222–2232

4 Gu X, Zhang H, Zhang D, et al. Deep API learning. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, 2016. 631–642