An empirical study on Java method name suggestion: are we there yet?

Weidong Wang  
Faculty of Information Technology  
Beijing University of Technology  
Beijing, China  
wangweidong@bjut.edu.cn

Yuqian Kang  
Faculty of Information Technology  
Beijing University of Technology  
Beijing, China  
kangyuqian@gmail.com

Dian Li  
Faculty of Information Technology  
Beijing University of Technology  
Beijing, China  
iamlidian@emails.bjut.edu.cn

Abstract—A large-scale evaluation for current naming approaches substantiates that such approaches are accurate. However, it is less known about which categories of method names work well via such naming approaches and how’s the performance of naming approaches. To point out the superiority of the current naming approach, in this paper, we conduct an empirical study on such approaches in a new dataset. Moreover, we analyze the successful naming approaches above and find that: 1) around 60% of the accepted recommendation names are made on prefixes within get, set, is, and test. 2) a large portion (19.3%) of method names successfully recommended could be derived from the given method bodies. The comparisons also demonstrate the superior performance of the empirical study.

Keywords—Method Names, Machine Learning, empirical study

I. INTRODUCTION

In software project, while coding a function or method body, developers give a name to the body as method name. A good method name not only helps programmers understand the program code better [1], [2], but also prevents method APIs from being operated incorrectly [3]. Since function call using specified method name is commonplace in program code, more and more software companies emphasize method naming conventions and programming standards. As emphasized by the industrial experts including McConnell [4] and Beck [5], method naming is one of the significant software activities.

While there are an army of implied standards for method naming such as the CamelCase principle [6], [7], there is no broad understanding of which categories of method names are prone to be used or accepted by developers. We investigate the current naming approaches [8]–[11] in the past three years and find that in the correctly suggested method names, the categories within prefixes such as get, set, is, and test account for 60%. If we remove such method names within the prefixes above, the precision and recall of suggested method names could be downgraded dramatically. Actually, in coding process, we commonly need to name a potential method body that is far beyond the category including the above prefixes.

Furthermore, we don’t know which naming approach is outstanding under the same experiment environment since the evaluation metrics of such naming approaches are not uniform as well [12]. For example, the actual method name suggested is likely to be locally different from the given method name, however, the meanings of both method names are the same. To evaluate the above method names suggested, code2seq [8] and code2vec [11] redefined the loose evaluation metrics, while the HeMa [10] adopts tightened evaluation metrics i.e., F1-score, recall, and precision. Meanwhile, the HeMa approach also uses panel review to evaluate the method names suggested by inviting domain experts.

Although such approaches are overall accurate in suggesting method name, little is known about which categories and why they work or don’t work. And we also know little about the actual performance on a new dataset. Indeed, it is hard to automatically generate high-quality method name including answering why an accurate method name occurring frequently requires synthesis of different kinds of knowledge and context. Hence, we conduct empirical study among those famous approaches related with method name suggestion.

The rest of this paper is organized as follows. Section II introduces the new datasets. Section III describes the experiment results. Section IV investigates the related work. Section V concludes this paper.

II. NEW DATASETS

To make the following empirical study more objective and fair, we built 3 datasets for the purpose of experimental evaluation namely “Liu-kui dataset”, “new-100 dataset”, and “new-400 dataset”, respectively. The details of these datasets are as follows.

- Liu-kui dataset: This dataset was collected from 4 highly rated open source organizations, Apache, Spring, Hibernate and Google. 430 projects with the latest versions of Java source code were employed, each of which has 100 times submissions at least. This dataset totally contains 2,116,413 methods. The further details of the dataset can be found in Liu’s literature [13].

- New-100 dataset: We randomly selected 100 projects from the open source Java projects on the former 10K GitHub, and each of these projects has been submitted at least 100 times. The dataset contains 47,098 Java methods.
III. EMPIRICAL STUDY

To evaluate the performance of such naming approaches, we conducted a series of experiments. The following parts mainly consist of the parameter settings of experiments, the choices of evaluation metrics, and the answers to the research questions from RQ1 to RQ3.

1) In order to study the advanced approaches to Java method name recommendation in this paper, we propose the following research questions (RQs) as follows.

- **RQ1**: Can code2seq perform as well on the new dataset as the original dataset in this paper?
- **RQ2**: If the AST paths of a method do not contain a token in the method name, what is the performance?
- **RQ3**: Can the length of the method names recommended fulfill the standard of method name length?

To ensure the accuracy of the experimental results, we divide the training set and the test set based on 10-cross validation. In the evaluation process, we take the average of the results with 10-cross validation as the final result.

The evaluation results are in the following table. As in code2vec, we choose Precision, Recall and F1-score as evaluation indexes. The right three columns correspond to the performance of code2seq on the three different datasets. Precision, Recall, and F1-score in the table are the average values of each item in the test set.

| Method           | Liu-kui dataset | new-100 dataset |
|------------------|-----------------|-----------------|
| **Precision**    | 0.70            | 0.67            |
| **Recall**       | 0.63            | 0.56            |
| **F1-score**     | 0.66            | 0.60            |

From the table, we have the following findings:

- The volume of datasets may have an influence on the performance of code2seq. Changing the volume of new-400 dataset, Liu-kui dataset, new-100 dataset, we can find that the performance of code2seq improves significantly with the increase of dataset volume.
- Compared with Liu’s approach [13], the precision, recall and F1-score of code2seq on the largest new-400 dataset are improved by 3%, 7% and 6% respectively, while the precision, recall and F1-score of code2seq on the new-100 dataset are improved by 6%, 10%, and 8%, respectively.

From the above experiments and analysis, we can draw the conclusion that code2seq will still keep certain accuracy after transforming datasets. Nevertheless, different datasets have great influence on the performance of code2seq, and its performance on larger dataset is obviously better than that on smaller datasets due to the significant increase of training samples on larger datasets.

**RQ2: If the AST paths of a method do not contain a token in the method name, what is the performance?**

To study RQ2, we evaluated the performance of code2seq on visible and invisible methods. We will give the definitions of visible method and invisible method as follows.

For each method name $n_i$ in the Liu-kui dataset, we divide it into a token stream according to the CamelCase principle, $T_i = \{t_1, t_2, t_3, \ldots, t_m\}$, where $t_i$ is the $i$-th token in the token stream. The method body $b_i$ corresponding to the method name $n_i$ is parsed into an array of AST paths, where $p_j$ is the $j$-th AST path, consisting of several tokens, $p_j = \{s_1, s_2, \ldots, s_k\}$, and $s_k$ is the $k$-th token. $Q_i$ is the set of all AST path tokens belonging to $b_i$, $Q_i = \{p_1, p_2, \ldots, p_j\}$. If each token in $T_i$ does not appear in $Q_i$, $n_i$ is an invisible method, otherwise, it’s a visible method.

Next, we evaluate the performance of code2seq on both visible and invisible methods, we select 918k invisible methods from the Liu-kui dataset as a new dataset called “invisible dataset”. Accordingly, we randomly choose the same number of visible methods from the Liu-kui dataset to form the corresponding “visible dataset”. On both datasets above, we evaluate the code2seq’s performance via precision, recall and F1-score metrics.

Here, we have several findings as follows.
First of all, as shown in Table IV, a large proportion of method name tokens (accounting for 43%) can not be found in their corresponding method bodies. Therefore, code2seq based on machine learning is often limited by its replication mechanism and vocabulary size, which results in performance decline.

| Number | Total | Invisible dataset |
|--------|-------|-------------------|
| Rate   | 2116413 | 918545            |

Table IV: The proportion of the number of invisible methods to the total number of methods.

What’s more, even under the limitation mentioned in the first point, code2seq still performs well in the field of method name recommendation on the invisible datasets. As shown in Table V it maintains 39% precision, 37% recall and 38% F1-score. Compared with code2seq’s recommendation effect on the visible methods, the corresponding precision, recall, and F1-score only have a significant decrease of 35%, 25%, and 29%, respectively.

The code2seq can generate method names for invisible methods. The main reason is that it is not just to select some tokens from the input tokens to form a new method name, but to extract the features of AST corresponding to the method body and establish the relationship between these features and the tokens of the method name. If the AST structure of the given method $m_1$ and $m_2$ in the training set is highly similar, then the code2seq probably recommends the method name token of $m_2$ to $m_1$, even if the token in the corresponding method name does not appear in the AST path token of $m_1$.

A typical case is shown in the figure 1 where the code2seq successfully recommends method names for invisible methods in the test set. We found that both visible and invisible methods have very similar AST trees. Moreover, the code2seq can strongly identify the relationship between AST features and method name tokens, and favourably recommend the corresponding method names.

After the analysis in the previous figure, we can conclude that the performance of code2seq in invisible methods slightly decreases. Still, it can accurately recommend method names.

RQ3: Can the length of the method names recommended fulfill the standard of method name length?

A method name is usually regarded as a brief description of the function of a piece of code. First, we expect that it can describe the function of a method in sufficient detail. Second, we hope it can help developers understand the function of a program quickly and concisely. Therefore, the length of the sequence contained in the method name recommended by the code2seq should be within a reasonable range. Empirical studies show that the length of method name sequence in the range of 3 to 7 is the most conducive to improve the readability of the code, meanwhile the range conforms to the habit of human short-term memory.

To evaluate whether the length of the token sequence contained in the method name generated by code2seq meets the custom of the method name length, we train code2seq model on the training set derived from three datasets, i.e., Liu-kui dataset, new-100 dataset, and new-400 dataset, respectively. Then, we test the method names generated on their corresponding test sets, and count the length distribution of the corresponding method name token sequences.

As shown in Table VI the satisfaction rates are generated by the code2seq on the test sets from three datasets (Liu-kui dataset, new-100 dataset, new-400 dataset).

| Satisfaction | new-400 dataset | Liu-kui dataset | new-100 dataset |
|--------------|-----------------|-----------------|-----------------|
| Correct      | 2116467         | 88889           | 2116            |
| Dissatisfaction | 264572       | 122752          | 2594            |
| Satisfaction rate | 0.44          | 0.41            | 0.44            |

Table VI: The satisfaction rates on the three datasets.

The box diagram of the length distribution of code2seq’s method name sequence generated on the test set from three datasets (Liu-kui dataset, new-100 dataset and new-400 dataset) is shown in figure 2. The length of most method name sequences generated by code2seq on the three datasets is between 1 to 5, 1 to 7, and 1 to 5, respectively. According to the experimental results above, we can conclude that most of the method name sequences generated by code2seq fulfill the standard of method name length.
We define satisfaction rate as follows.

\[
\text{Satisfaction rate} = \frac{N_{\text{satisfaction}}}{N_{\text{satisfaction}} + N_{\text{dissatisfaction}}} \quad (1)
\]

\(N_{\text{satisfaction}}\) represents the number of methods that fulfill the standard, and \(N_{\text{dissatisfaction}}\) means the number of methods that do not fulfill the standard. Satisfaction rate indicates the proportion of the number of methods that fulfill the standard of method name length to the total number of methods. The second to fourth columns show the number of methods that fulfill the standard of method name length \((N_{\text{satisfaction}})\), the number of methods that do not fulfill the standard of method name length \((N_{\text{dissatisfaction}})\), and the proportion of the number of methods that meet the length standard to the total number of methods (Satisfaction rate).

From the table, we have the following findings:

- Firstly, from the point of length distribution, the length of the method name sequences generated by code2seq on the three datasets is in the range of 1 to 5, 1 to 7 and 1 to 5, respectively. It is true that the sequence length of most method names ranges from 1 to 4 tested on the three datasets. The result fulfills the standard of method name length.

- Secondly, in terms of satisfaction rate, the number of method name sequences generated by code2seq on the three datasets accounts for 44\%, 41\% and 44\%, respectively. Among the method names that do not fulfill the standard of method name length, the proportion of method names, whose length equals 2, is 32\%.

After the previous analysis, we can draw conclusion that most method names generated by the code2seq fulfill the standard of method name length.

IV. DISCUSSION OF RELATED WORK

To study the performance of naming methods, we investigate the primary naming methods for the program code entity. One is to use machine translation methods such as neural-translation framework seq2seq [9], code2seq [8], and the method name is finally generated by translating the method body. The code2seq [8] models a code snippet [14] as the set of compositional abstract syntax tree (AST) [15] paths and uses attention mechanism to select the relevant paths while decoding. Nguyen et al [9] presented MNire, a machine learning approach to suggest a method name and to detect method name inconsistencies. First, they found high proportions of the tokens of method names can be found in the three contexts of a given method including its body, the interface (the method’s parameter types and return type), and the enclosing class’s name. Second, to suggest a good name for a method, they used seq2seq to the naturalness of the program entities in the contexts.

Alon [15] presented a general path-based representation for predicting method names by combining abstract syntax tree (AST) [15]. Furthermore, Alon et al. proposed code2vec [11] to represent a method body into code embeddings, i.e., converting method body into fixed-length, continuously distributed vectors, by aggregating the set of its corresponding AST paths with attentional network. The vectors of methods with similar AST structures are close to each other in the same space measured by semantic similarity among method names. The code2vec can retrieve similar method bodies and reuse their method names. Jiang et al. [10] conducted an empirical study on the method name recommendation approach code2vec. Their evaluation results suggest that code2vec deserves significant improvement. However, to investigate the possibility of designing simple and straight-forward alternative approaches, they also proposed a heuristics based approach to recommending method names according to given method bodies. The experimental results show that their approach outperforms code2vec significantly.

V. CONCLUSIONS

In this paper, we conduct an empirical study on such top-level method name recommendation approaches. Our evaluation suggests that we should adopt different naming approaches for different categories of method names. To intuitively reveal the advantages of such different naming approaches, we also empirically evaluate the code2seq, HeMa, and code2vec especially in the part of naming categories, precision and recall.

REFERENCES

[1] R. Oliveira, “When more heads are better than one? understanding and improving collaborative identification of code smells,” in IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), 2016, pp. 879–882.
[2] J. Czerwonka, M. Greiler, and J. Tilford, “Code reviews do not find bugs: how the current code review best practice slows us down,” in IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2, 2015, pp. 27–28.
[3] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshvyanyk, “Recovering clear, natural identifiers from obfuscated js names,” in proceeding of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), 2017, pp. 542–553.
[4] S. McConnell, “Code complete,” Pearson Education, 2004.
[5] K. Beck, “Clean code: a handbook of agile software craftsmanship,” Pearson Education, 2009.
[6] B. Sharif and J. I. Maletic, “An eye tracking study on camelcase and underscore identifier styles,” in IEEE 18th International Conference on Program Comprehension, 2010, pp. 196–205.
[7] D. W. Binkley, M. Davis, D. J. Lawrie, and C. Morrell, “To camelcase or underscore,” in IEEE/ACM 27th International Conference on Program Comprehension (ICPC), 2019, pp. 177–177.
[8] U. Alen, S. Brody, O. Levy, and E. Yahav, “Code2seq: Generating sequences from structured representations of code,” in International Conference on Learning Representations of Code (ICLR), 2019, pp. 111–124. [Online]. Available: https://openreview.net/pdf?id=H1gKYo09tX
[9] S. Nguyen, H. Phan, T. Le, and T. N. Nguyen, “Suggesting natural method names to check name consistencies,” in 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), 2020, pp. 1372–1384.
[10] L. Jiang, H. Liu, and H. Jiang, “Machine learning based recommendation of method names: How far are we,” in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 602–614.
[11] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “code2vec: Learning distributed representations of code,” in Proceedings of the ACM on Programming Languages, 2019, pp. 1–27.
[12] R. S. Alsuhaibani, C. D. Newman, M. J. Decker, M. L. Collard, and J. I. Maletic, “On the naming of methods: A survey of professional developers,” CoRR, vol. abs/2102.13555, 2021. [Online]. Available: https://arxiv.org/abs/2102.13555
[13] K. Liu, D. Kim, T. F. Bissyand, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. Le Traon, “Learning to spot and refactor inconsistent method names,” in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 1–12.
[14] M. Raghothaman, Y. Wei, and Y. Hamadi, “Swim: Synthesizing what i mean: Code search and idiomatic snippet synthesis,” in IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016, pp. 357–367.
[15] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “A general path-based representation for predicting program properties,” in ACM 39th SIGPLAN Conference on Programming Language Design and Implementation, 2018, pp. 404–419.