Poster: Guiding Developers to Make Informative Commenting Decisions in Source Code

Yuan Huang
Sun Yat-sen University
Guangzhou, GuangDong, China
huangyjn@gmail.com

Nan Jia
Hebei GEO University
Shijiazhuang, HeBei, China
jiasysu@gmail.com

Qiang Zhou
Sun Yat-sen University
Guangzhou, GuangDong, China
zhouq865@gmail.com

Xiangping Chen*
Sun Yat-sen University
Guangzhou, GuangDong, China
chenxp8@mail.sysu.edu.cn

Yingfei Xiong
Peking University
Haidian Qu, Beijing, China
xiongyf@pku.edu.cn

Xiaonan Luo
Sun Yat-sen University
Guangzhou, GuangDong, China
lnslxn8@mail.sysu.edu.cn

ABSTRACT

Code commenting is a common programming practice of practical importance to help developers review and comprehend source code. However, there is a lack of thorough specifications to help developers make their commenting decisions in current practice. To reduce the effort of making commenting decisions, we propose a novel method, *CommentSuggester*, to guide developers regarding appropriate commenting locations in the source code. We extract context information of source code and employ machine learning techniques to identify possible commenting locations in the source code. The encouraging experimental results demonstrated the feasibility and effectiveness of our commenting suggestion method.

ACM Reference Format:
Yuan Huang, Nan Jia, Qiang Zhou, Xiangping Chen, Yingfei Xiong, and Xiaonan Luo. 2018. Poster: Guiding Developers to Make Informative Commenting Decisions in Source Code. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3183440.3194960

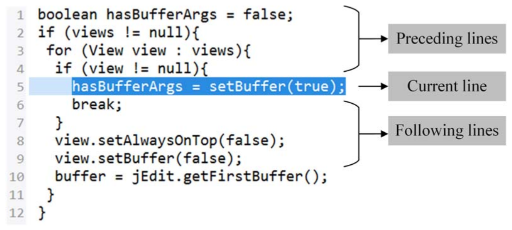
1 INTRODUCTION

Code comment is an integral part of a software project that demonstrates the logic and functional implementation of the source code in a natural language form [1]. High-quality code comment plays an important role in code review and comprehension [2]. However, in our developer survey, we found that even in a known social networking company like Tencent[1] (WeChat[2] is one of the highly popular messenger app released by Tencent with over 900 million monthly active users in the world), it is difficult to find rigorous specifications for developers to guide their commenting behaviors. In most cases, developers make commenting decisions according to their own programming experience and domain knowledge. Therefore, commenting during development becomes an important yet tough decision, especially for novice programmers with little programming experience.

In this work, we propose a novel method, *CommentSuggester*, that helps developers make commenting decisions by analyzing code context information. We expect that *CommentSuggester* can acquire common commenting knowledge from well-commented projects and then guide developers in making commenting decisions.

2 LEARNING TO COMMENT

Our goal, referred to as "learning to comment", is to automatically learn common commenting practices as a machine learning model and then employ the model to guide developers in make commenting decisions during development. Figure 1 shows the main idea behind our approach, that is, predicting whether the current line is an appropriate commenting location based on the information of its preceding and following lines.



```
boolean hasBufferArgs = false;
if (views != null){
 for (View view : views){
  if (view != null){
    hasBufferArgs = setBuffer(true);
    break;
  }
  view.setAlwaysOnTop(false);
  view.setBuffer(false);
  buffer = jEdit.getFirstBuffer();
 }
}
```

Figure 1: Code snippet

Software developers tend to make comments on a functionally (or logically) independent code snippet [2]. Therefore, if the current line is the start line of a functionally independent code snippet, it is very likely to be a comment location. To assess whether the current line is the start line, we need to analyze the logical strength between the current line and its following lines. A stronger logical connection indicates a higher possibility for the current line to be the start line. Besides, the logical strength between the current line and its preceding lines is also important for identifying the start line. The weaker the logical relationship between them, the more likely it is that the current line is the start line. In this paper, we extract context features from three different perspectives to measure the logical strength between code lines, they are: code structural features, syntactic features, and semantic features.

*Corresponding author.
[1]https://www.tencent.com/en-us/index.html
[2]http://www.wechat.com/en/

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3194960

2.1 Structural Context Features

Source code has a well-defined structure [3]. It is accepted that code line itself has prominent structural features, known as *Intra-Structural Context Feature*. Additionally, strong structural features exist between code lines, these are known as *Inter-Structural Context Feature*. The *Inter-Structural Context Feature* represents the relationships between the current code line and its context code lines in terms of locations, distances, and coupling relations and, more importantly, expresses the strength of the logical relationships between them. TABLE 1 reports the detailed information of the structural context features.

Table 1: Structural Context Features

Category	Features	Descriptions
Intra-Structural Context Features	C_{loca}	the absolute location of current line in a method
	C_{type}	the type of current line, e.g., control and non-control statements
	C_{cove}	the cover range of current line
	C_{iden}	the number of identifiers containing in current line
	C_{icove}	the total number of identifiers within the scope of current line
Inter-Structural Context Features	C_{cvar}	whether current line and its preceding (or following) line invoke the same variable
	C_{cmet}	whether current line and its preceding (or following) line invoke the same method
	C_{dist}	the distance between current line and its preceding (or following) line
	C_{dcontr}	whether current line and its preceding (or following) line are located within the scope of the same control statement
	C_{icontr}	similar to C_{dcontr}, however, other control statements exist between current line and its preceding (or following) line
	C_{contrw}	the sum of the weights of the control statements that exists between current line and its preceding (or following) line

Note that, to obtain the *Inter-Structural Context Feature*, we must generate the features between the current line and the η preceding lines, as well as the η following lines. Because the current line has 6 *Inter-Structural Context Feature* with any one preceding and following code line, it will produce 60-dimensional vector if η equals 5, when coupled with the five *Intra-Structural Context Feature* of the current line, there provides a total of 65 dimensions.

2.2 Syntactic Context Features

In this study, the code syntax is subdivided into 96 types (e.g., *IfStatement*, *ForStatement*, *ReturnType*, *VariableDeclaration*, etc) [4]. We employ the abstract syntax tree to parse the syntax structure of each code line in source code. It is worth noting that a single code line may contain multiple syntax types. For example, a `if` code line "`if(epu.getName() == null)`" contains four syntax types: *IfStatement*, *MethodInvocation*, *ConditionalExpression*, and *NullLiteral*.

To apply the syntax distribution of the context lines to the learning based method, we use a unique floating number (10-bit, lower than 1.0) to encode each code syntax type. According to the natural order of the context lines, their syntax can be converted to a vector by using their encoding values.

2.3 Semantic Context Features

We apply word embeddings to encode words in the source code [5]. Word embeddings only require large amounts of unlabeled text to learn, and we collect the method body content as software engineering text. Then, we use continuous skip-gram model [5] to learn the word embedding of a central word (i.e., w_i). The objective function of the skip-gram model aims at maximizing the sum of log probabilities of the surrounding context words conditioned on the central word [5]: $\sum_{i=1}^{n}\sum_{-k\leq j\leq k, j\neq 0}\log p(w_{i+j}|w_i)$, where w_i and w_{i+j} denote the central word and the context word, respectively, in a context window of length $2k$+1 and n denotes the length of the word sequence.

After training the model, each word in the corpus is associated with a vector representation and formed a word dictionary. To obtain the semantic information of the following and preceding lines of the current code line, we first find the vector representation of each identifier in the following and preceding lines. Then, we sum the vectors of all identifiers dimension by dimension to generate the semantic feature vector of the current code line.

3 DISCUSSIONS AND CONCLUSION

Code commenting plays an important role in program comprehension and strategic commenting decision is a goal pursued by software developers. This paper proposes a novel method, *CommentSuggester*, of making suitable commenting decisions in software development.

Table 2: Features choice effect on positive instances

Features	Positive Instances		
	PRE	***REC***	***FM***
Structural	0.771	0.502	0.601
Structural+Syntactic	0.814	0.518	0.627
Structural+Syntactic+Semantic	0.805	0.707	0.748

We evaluate *CommentSuggester* using 10 data sets in GitHub, and the results are shown in TABLE 2. We can observe that the prediction accuracy presents a continuous step-like increase when adding new types of features into the learning model. This continuous accuracy increase can also confirm that the three types of features are useful in characterizing the commenting decisions.

ACKNOWLEDGMENTS

This research is supported by the National Natural Science Foundation of China (61672545), NSFC-Guangdong Joint Fund (U1611261), Science and Technology Planning Project of Zhongshan (No. 2016A-1044), Science and Technology Planning Project of Guangzhou (201-604016136).

REFERENCES

[1] Jessica Keyes. *Software engineering handbook*. CRC Press, 2002.
[2] D. Steidl, B. Hummel, and E. Juergens. Quality analysis of source code comments. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 83–92, May 2013.
[3] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 1287–1293, 2016.
[4] Y. Huang, Q. Zheng, X. Chen, Y. Xiong, Z. Liu, and X. Luo. Mining version control system for automatically generating commit comment. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 414–423, Nov 2017.
[5] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems*, NIPS'13, pages 3111–3119, 2013.