The Journal of Systems & Software 170 (2020) 110754

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

In Practice

CommtPst: Deep learning source code for commenting positions prediction

Yuan Huang [a], Xinyu Hu [a], Nan Jia [c], Xiangping Chen [b,*], Zibin Zheng [a], Xiapu Luo [d]

[a] National Engineering Research Center of Digital Life, School of Data and Computer Science, Sun Yat-sen University, Guangzhou, China
[b] Guangdong Key Laboratory for Big Data Analysis and Simulation of Public Opinion, School of Communication and Design, Sun Yat-sen University, Guangzhou, China
[c] School of Management Science and Engineering, Hebei GEO University, Shijiazhuang, China
[d] The Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China

ARTICLE INFO

Article history:
Received 30 September 2019
Received in revised form 17 July 2020
Accepted 22 July 2020
Available online 30 July 2020

MSC:
00-01
99-00

Keywords:
Comment position
LSTM
Code syntax
Code semantics
Comment generation

ABSTRACT

Existing techniques for automatic code commenting assume that the code snippet to be commented has been identified, thus requiring users to provide the code snippet in advance. A smarter commenting approach is desired to first self-determine where to comment in a given source code and then generate comments for the code snippets that need comments. To achieve the first step of this goal, we propose a novel method, *CommtPst*, to automatically find the appropriate commenting positions in the source code. Since commenting is closely related to the code syntax and semantics, we adopt neural language model (word embeddings) to capture the code semantic information, and analyze the abstract syntax trees to capture code syntactic information. Then, we employ LSTM (long short term memory) to model the long-term logical dependency of code statements over the fused semantic and syntactic information and learn the commenting patterns on the code sequence. We evaluated *CommtPst* using large data sets from dozens of open-source software systems in GitHub. The experimental results show that the precision, recall and F-Measure values achieved by *CommtPst* are 0.792, 0.602 and 0.684, respectively, which outperforms the traditional machine learning method with 11.4% improvement on F-measure.

1. Introduction

Code commenting has been an integral part of software development (Gosling et al., 2014). It has been a standard practice in the industry (Wong et al.). Comments improve software maintainability through helping developers understand the source code (Keyes, 2002). Obviously, high-quality code comment plays an important role in most software activities, such as code review (Thongtanunam et al., 2016; Moreno et al., 2014) and program comprehension (Steidl et al., 2013a; Tao et al., 2012).

Owing to the importance of code comments, many researchers have paid attention to automatic comment generation (Wong et al., 2015; Sridhara et al., 2010; Moreno et al., 2013). Current techniques can automatically generate code comments for different software entities, such as, code snippets (Wong et al., 2015), methods (Sridhara et al., 2010), and classes (Moreno et al., 2013). All these techniques for the comment generation are based on a hypothesis that the objects to be commented are for certain. That is, users should first identify the code snippet or method or class that needs comments.

Unfortunately, this assumption limits existing comment generation approaches to handle the following common scenario: given the source code of a method, how to generate the proper comments and put them at the suitable positions inside the method that can cover the core code snippets of the method (Chen et al., 2019). More precisely, in this scenario, the code snippets need to be commented become uncertain (i.e., the commenting positions are not predefined). Therefore, a smarter commenting approach is desired to self-determine the commenting positions in the source code, and further generate the comments for the code snippets inside the method body.

On the other hand, many developers often forget to add comment when they are coding as shown in previous study (Huang et al., 2019). We visited 13 programmers (with an average of 13.2 years of programming experience) via online questionnaires, and the results shown that 5 programmers often or always forget to write comments when programming, and 6 programmers occasionally forget to write comments, while only 2 programmers rarely or never forget to write comments. Therefore, it makes sense to remind programmers to add comments at appropriate positions.

* Corresponding author.
E-mail address: chenxp8@mail.sysu.edu.cn (X. Chen).

https://doi.org/10.1016/j.jss.2020.110754
0164-1212/

To fill in the gap, in this paper, we propose a novel approach named *CommtPst* to identify suitable commenting positions in the source code. It is worth noting that due to the complex logic of source code automatically identifying the appropriate commenting positions in the source code is challenging. A comment may be located at the code statements that are tightly coupled. As the Case 1 shows (the next page), the code statements have direct coupling correlation, such as calling common variable `arr[j]`. On the other hand, a comment may be located at a code statements that are low coupling. As Case 2 shows, the code statements have no coupling correlation, while they implement the initialization together. As a result, comments can appear on different forms of code statements, and it is difficult to identify the commenting positions via simply considering the code coupling correlation information. Case 1 and Case 2 also show two typical comment patterns that comments often added before control statements and variable definitions. There are also some other comment patterns, such as multi-loop code snippets that often require comments for understanding better.

```
Case 1:                          Case 2:
...                              ...
// swap array elements           // initialization
if(arr[j+1]<arr[j]){             double similar = 0.0;
  temp = arr[j];                 String path = rootPath;
  arr[j] = arr[j+1];             int sourceVersion = 65;
  arr[j+1] = temp;               int count = 0;
}                                int index = 1;
...                              ...
```

Case 1: the code statements in the snippet call same variable `arr[j]`.
Case 2: the code statements in the snippet look very loose, while they implement the initialization together.

To address this challenge, we employ LSTM (long short term memory) to model the logical dependencies of code statements from both the syntactic and semantic perspectives. Code syntactic information represents the relationships of code statements in terms of syntax types, coupling relations, etc., and code semantic information represents the semantics distributions of the source code (Huang et al., 2018). Both syntactic and semantic information reflect the logical correlations of code statements. For example, we can capture the logical correlations of code statements in Case 2 with the syntactic information, as these code statements have unified syntax types, i.e., *VariableDeclarationExpression*. Multiple code statements with the same syntax appearing together show that they probably achieve a common behavior, then these code statements are correlative. Analogously, we can use the semantic information to measure the logical correlations of code statements in Case 1, as multiple code statements visit the common variable `arr[j]`. In this paper, LSTM is designed to model the direct (explicit) and indirect (hidden) logical correlations of code statements based on the syntactic and semantic information, and further to learn the commenting patterns from the code sequence. After well training, *CommtPst* can effectively determine the commenting positions in the source code.

We evaluate *CommtPst* on ten large-scale, real-world datasets got from GitHub, and achieve a precision of about 79%, which outperforms several baseline methods. This paper contributes the following: (1) *CommtPst* can remind suitable commenting positions before generating code comments. (2) LSTM is introduced to model the logical correlations between code statements, and further to determine the commenting positions in the source code. (3) The comprehensive evaluation results demonstrate the feasibility and effectiveness of our commenting position prediction method. To facilitate research and application, our *CommtPst* and the dataset are available at https://github.com/Badorange6/DeepComment.

The rest of this paper is organized as follows. Section 2 shows the overview of main steps. Section 3 describes the subject projects and data collection. Section 4 describes feature extraction. Section 5 introduces the proposed LSTM model. The setups of the case study are discussed in Section 6. Section 7 describes the analysis of experimental results. Result discussion is presented in Section 8, while Section 9 discusses related works. Section 10 discusses the threats to validity. Section 11 summarizes our approach and outlines directions of future work.

2. Overview of main steps

Fig. 1 shows the main steps of the proposed approach. The approach includes two phases: the model training phase and the prediction phase. In the model training phase, our goal is to build a prediction model from the source code which has been well commented. In the prediction phase, this discriminant model is used to determine the commenting positions in the source code.

Our approach first analyzes the source code that has been well commented, and identifies the commenting instances from the source code. Each code line corresponds to a label, and in total we have two labels which correspond to "commented" and "uncommented". Our approach analyzes the source code to determine the "commented" and "uncommented" code statements. Then, our approach extracts features for each code line. Features are various quantifiable characteristics of code statements that could potentially differentiate the code statements that need to been commented. In this paper, we try to preserve two types of features: code syntactic features and code semantic features. To allow the LSTM algorithm to leverage these two types of features and be more expressive, we use the code syntactic and semantic features to construct an unified vector representation.

Next, the training set is constructed after the features extraction and fusion, and our approach builds a prediction model on the training set. After prediction model is constructed, it is used in the prediction phase to predict the code statements that will get commented. We use the entire code statements of a method as input to the prediction model. For each code line in the method, we first extract the syntactic and semantic features as we do in the model building phase. Then, we input the features to the prediction model. This step outputs the prediction results, which are labels corresponding to the code statements that get commented or uncommented. At last, taking the "commented" code statements back to the source code of a method, we will know the comment positions in this method.

3. Subject projects and data collection

In this section, we first introduce the subject projects under study and then, we present the data collection process.

3.1. Subject projects

In our study, we investigated ten large open-source software projects in GitHub: Ant, ANTLR, ArgoUML, conQAT, JDT, JHotDraw, JabRef, JFreeChart, Lucene, and Vuze. Each of these projects contains over ten thousand code statements written in the Java language. Table 1 summarizes the information of the studied software projects.

We chose these ten projects as our subjects for two reasons. Firstly, all of them were selected owing to their open-source availability and popularity. They are well-known Java projects and used in various researches in the field of software engineering (Xia et al., 2016; Poshyvanyk et al., 2013; Bavota et al., 2014; Yan et al., 2019). Secondly, most of the selected projects are affiliated with famous nonprofit organizations or software

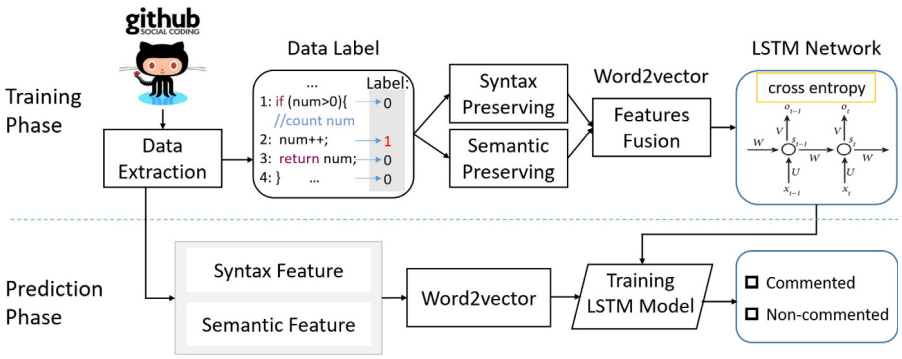


Fig. 1. Overview of main steps.

Table 1
Subject software projects.

Projects	Start time	Version	# of code lines	# of comments
Ant	2000	1.10.1	224k	1913
ANTLR	1992	4.5.2	73k	731
ArgoUML	2002	0.34	188k	1977
conQAT	2007	13.12	163k	877
JDT	2003	4.0	1126k	22,705
JHotDraw	2000	7.0.6	51k	367
JabRef	2003	3.6	168k	2100
JFreeChart	2000	1.0.19	218k	1967
Lucene	2004	6.3.0	620k	9963
Vuze	2003	5.7.4.0	803k	2002

```
/**
 * check for invalid options                                    ①
 * @throws BuildException if something goes wrong.
 */
protected void checkOptions() throws BuildException {
    // set the destination directory relative from the project if needed.   ②
    if (toDir == null) {
        toDir = task.getProject().resolveFile(".");
    } else if (!toDir.isAbsolute()) {
        toDir = task.getProject().resolveFile(toDir.getPath());  // get path   ③
    }
}
```

Fig. 2. Comment examples.

companies (e.g., Apache, IBM), and the comments in their source code are more likely to be standardized. Although the content of a few comments in these projects may be out-of-date, we focus more on the commenting positions in the source code. Namely, even if a few comments are out-of-date, it does indicate that those are commenting positions in the source code.

3.2. Data collection

There are two types of comments used in source code: header comment (i.e., JavaDoc comment) and internal comment (Steidl et al., 2013a; Huang et al., 2020). Header comment appears before a class or a method declaration (comment ① in Fig. 2). Internal comment is associated with one or several code statements, which usually precedes these associated code statements (comment ② in Fig. 2), or locates behind a code statement (on the same line) and only serve the current code statement (comment ③ in Fig. 2). Because header comment can only appear before a class or a method declaration, so there is no need to predict its position. Therefore, *CommtPst* was designed to predict commenting positions only for internal comments.

CommtPst is a learning-based approach, the data set, including positive and negative instances, was collected from the source code of the ten target projects. Specifically, for each code line in a method, if there is an internal comment in it or its preceding line is an internal comment, then the code line is labeled as "commented"; otherwise, it is labeled "uncommented". Then, the code statements with "commented" labels correspond to positive instances and those with "uncommented" labels correspond to negative instances. Note that we only regard the first line of the internal comment coverage as a positive instance. This is because we only need to know the position of a comment in the source code, and then a deep learning based approach can be used to model the logical relations between the code line with "commented" label and its context code statements, and further to identify the possible commenting patterns in the source code.

However, not all the "commented" code statements can be used as training sets in our study as they may represent invalid commenting instances that correspond to incorrect commenting decisions. To eliminate these invalid commenting instances, a series of preprocessing rules were applied to the original "commented" code statements. The first type of invalid commenting instance concerns code statements that have been commented out, e.g., ``// if(obj.getName() == null)''. This occurs when the code line is a test code line or abandoned code changed by developers. The second type of invalid commenting instance concerns the comments automatically generated by tools, such as ``// Auto-generated catch block''.

For the auto-generated comments, we identify the keywords (such as: "Auto-generated") to eliminate them. For the commented-out code statements, we use a static analysis method to identify them. Specifically, for a comment, we first remove its comment notation (e.g., " // "), then use AST parser to parse the "comment". If the "comment" can be parsed as syntactic tokens, such as *IfStatement*, then the "comment" is identified as a commented-out code line. To evaluate the effectiveness of the method, we conducted an experiment. The result shows that, out of 2000 randomly selected comments that contain 19 commented-out code, our method detected 18 commented-out codes correctly, while the keyword-based method detected 14 correctly.

4. Learning code features

To feed the source code into the LSTM model, we need extract the features of each code line to form a sequence. To deal with the features of single code line, we extract the code syntactic and semantic features from each code line. The detailed process of feature extraction is described in the following sections.

4.1. Code syntax preserving

We assume that the developers selecting the commenting positions in the source code is closely related to the syntax of

source code. Because developers tend to add comments to the logically independent code snippets (Steidl et al., 2013b), and the code logic can be reflected by the code syntactical information. We capture the code syntactical information by analyzing the program's AST (Huang et al., 2017). An AST is a syntactical structure of source code, depicting the relationships among the components of the code in a hierarchical tree view (Huang et al., 2017).

In this study, we user Eclipse JDT to extract abstract syntax trees of codes. And the code syntax was subdivided into 96 types,[1] and each type is denoted by a syntactic token (e.g., *IfStatement*, *ForStatement*, *ReturnType*, *VariableDeclaration*, etc.). We parse the syntactic structure of each code statement in source code from its abstract syntax tree. For each type of syntactic token in the abstract syntax tree, we first get its start line and end line in the source code via parsing the abstract syntax tree, and then count the syntactic tokens involved by each code line. For example, Fig. 3 shows the process of code tokenizing, and the number pair of $\langle 18, 20\rangle$ in Step 2 represents the start line and end line of token *IfStatm* (i.e., *IfStatement*) in the source code, and Step 3 shows the syntactic tokens involved by each code line. It is worth noting that a single code line may contain multiple types of syntactic tokens. E.g., the code line ``` ``if(epu.getName() == null)'' ``` in Fig. 3 contains 4 syntactic tokens: *IfStatement*, *ConditionalExpression*, *MethodInvocation*, and *NullLiteral*.

After applying the syntax analysis method to the source code, each code line can obtain its syntax tokens. According to the natural order of the code statements, these syntax tokens consist of a syntactical corpus, which will be used to capture the relationship between commenting positions and code syntactical information in the model training phrase.

4.2. Code semantics preserving

Besides the syntactical information, the code semantics also need to be preserved when predict the commenting positions in the source code. Source code is similar to natural language, as it is composed of identifiers that contain a wealth of code semantics (Beniamini et al., 2017). Code semantics have been effectively used in the fields of software traceability recovery (Guo et al., 2017), linkable knowledge mining (Xu et al., 2016), API mining (Nguyen et al., 2017), bug reports retrieval (Yang et al., 2016). Driven by these encouraging results, we extract semantic information to characterize the commenting positions.

In this work, the code semantics are extracted from the text elements of the source code. Obviously, not all the text elements in the source code play a positive role to preserve the code semantics, some text elements may weaken the code semantic. Therefore, a series of preprocessing rules are applied: (1) split the camel-case words into single words, such as: '*getFirstName*' is divided into '*get*', '*first*' and '*name*'. (2) filter out the stopwords[2] in the source code, such as: '*and*', '*the*', '*an*', etc. (3) filter out the words that are numbers or single letters, such as: '*n*', '*i*', '0', etc. (4) To reduce the amount of vocabulary in the entire corpus, we applied the stem segmentation technique (Porter, 1980) to the textual elements. Because English verbs may appear in different tenses, such as past tense, future tense, and perfect tense, we transformed verbs of different tenses into their original forms.

[1] The detailed introductions of the 96 syntax can be found in http://help.eclipse.org/photon/index.jsp; In package: org.eclipse.jdt.core.dom.

[2] https://www.ranks.nl/stopwords.

4.3. Features fusion

Since the subsequent deep learning algorithms take numeric vectors as inputs, the syntactical and semantic features should be mapped to numbers. To achieve this, a mapping is built to link each syntax token (or textual element) to a numeric vector. Specifically, we apply word2vec (Mikolov et al., 2013) with the continuous skip-gram model to convert each syntax token (or textual element) to a vector representation.

The syntax information is expressed in syntax tokens, while the semantic information is expressed in textual elements. It is difficult to directly fuse two features expressed in different way. Inspired by the work of Li et al. (2018), we address this issue by projecting syntax tokens and textual elements as meaning vectors in a shared representation space. Li et al. build an unified corpus that contains comment-words and APIs to train a shared representation space in their APIs searching task. Similarly, we build an unified corpus that contains both syntax tokens and textual elements in this study. Specifically, for each code statement, we randomly shuffle its syntax tokens and textual elements, and add the shuffled result into the corpus, which is regarded as a shuffling strategy in Li et al. (2018). Fig. 4 shows the process of shuffling the syntax tokens and textual elements of code line ``` ``if(epu.getName() == null)'' ```, where the shuffling step is to break the fixed locations of syntax tokens and textual elements, and tries to obtain randomly collocations between them.

To obtain the vector representation of a term, we used the continuous skip-gram model to learn the word embedding of a central term (i.e., w_i) (Mikolov et al., 2013) on the entire corpus. It is well known that the word embedding is an intermediate result of the continuous skip-gram model. Continuous skip-gram is effective at predicting the surrounding terms in a context window of $2k$+1 terms (generally, k=2, and the window size is 5). The objective function of the skip-gram model aims at maximizing the sum of log probabilities of the surrounding context terms conditioned on the central term (Mikolov et al., 2013):

$$\sum_{i=1}^{n} \sum_{-k\le j\le k, j\neq 0} \log p(w_{i+j}|w_i) \tag{1}$$

where w_i and w_{i+j} denote the central term and the context term, respectively, in a context window of length $2k$+1 and n denotes the length of the term sequence. $\log p(w_{i+j}|w_i)$ is the conditional probability, defined using the softmax function:

$$\log p(w_{i+j}|w_i) = \frac{exp(v'^{T}_{w_{i+j}} v_{w_i})}{\sum_{w\in W} exp(v'^{T}_{w} v_{w_i})} \tag{2}$$

where v_w and v'_w are the input and output vectors of a term w in the underlying neural model, and W is the vocabulary of all terms. Intuitively, $p(w_{i+j}|w_i)$ estimates the normalized probability of a term w_{i+j} appearing in the context of a central term w_i over all terms in the vocabulary. Here, we employ negative sampling method (Lin et al., 2018) to compute this probability.

After applying word2vec, each syntax token (or textual element) in the corpus is associated with a vector representation and formed a word dictionary. To obtain the semantic information of a code statement, we first collect its syntax tokens and textual elements, then determined the corresponding vector representation of each syntax token and textual element from the dictionary. With word2vector model, each word is transformed into a fixed-length vector. However, since different code lines have different numbers of syntax tokens (or textual elements), it is difficult to keep a fixed-length vector for a code line if we simply connect the vectors of all the words in a code line. To address this challenge, we transform a code statement into a vector representation by averaging all the vectors (i.e., syntax token and textual element

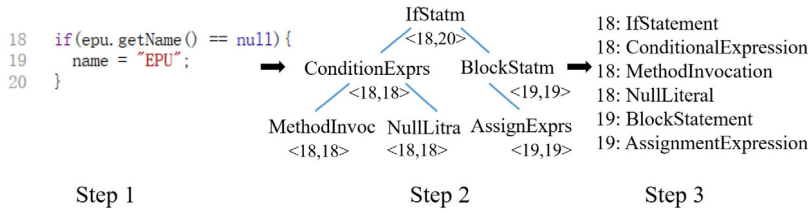


Fig. 3. Code tokenizing.

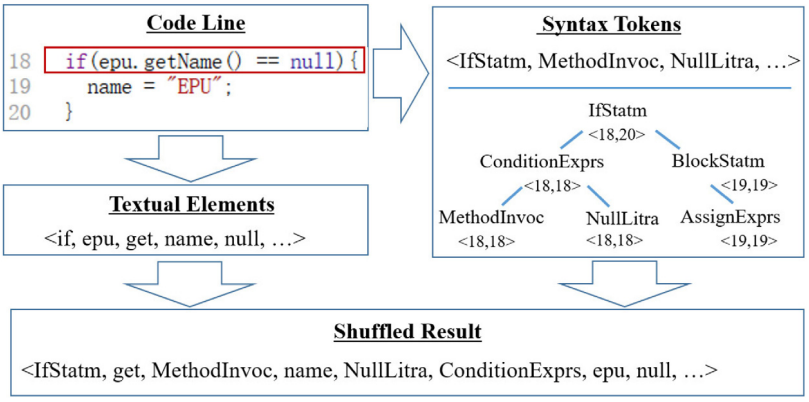


Fig. 4. Shuffling strategy for syntax tokens and textual elements.

vectors) the statement contains, which has been proven effective in the study (Yang et al., 2016). The vector that holds the structural and semantic information can then be leveraged by our proposed deep learning network for obtaining deep representations capable of distinguishing commenting positions from source code.

5. The LSTM architecture

Source code can be regarded as sequential data, hence in this study we try to use the RNN (Guo et al., 2017) network to model the commenting patterns on the source code via the code syntactic and semantic information, and further to predict the commenting position in source code. A prominent drawback of the standard RNN model is that the network degrades when long dependencies exist in the sequence due to the phenomenon of exploding or vanishing gradients during back-propagation (Bengio et al., 1994), researchers have proposed several variants with mechanics to preserve long-term dependencies, such as Long Short Term Memory (LSTM). LSTM has been repeatedly applied to solve semantic relatedness tasks and has achieved convincing performance (Rocktäschel et al., 2015; Tai et al., 2015; Lin et al., 2018). In general, the context information of the source code is useful for predicting comment positions, but the normal LSTM can only learn in a sequence in one direction. To obtain the correlations of the surrounding code statements of statement x_i, the bidirectional LSTM (Bi-LSTM) (Schuster and Paliwal, 1997) can be used to serve this purpose.

Fig. 5 illustrates a typical LSTM unit. Using retained memory cell state and the gating mechanism, the LSTM unit remembers information until it is erased by the *forget* gate; as such, LSTM handles long-term dependencies more effectively. If we let $\overrightarrow{h^n}$ and $\overleftarrow{h^n}$ be the activations of the forward and backward chains on the nth level of a stack, then:

$$\overrightarrow{h}_t^n = \phi_f(z_t^n, \overrightarrow{h}_{t-1}^n) \quad and \quad \overleftarrow{h}_t^n = \phi_b(z_t^n, \overleftarrow{h}_{t+1}^n), \tag{3}$$

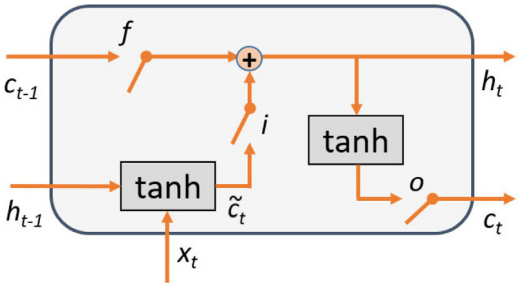


Fig. 5. A single unit of LSTM.

where, $z_t^0 = x_t$ and $z_t^n = [\overrightarrow{h}_t^{n-1}; \overleftarrow{h}_t^{n-1}]$ for $n \geq 1$. The output on the final linear-output layer is then

$$y_t = W_{\overrightarrow{h}y}\overrightarrow{h}^N + W_{\overleftarrow{h}y}\overleftarrow{h}^N + b_y, \tag{4}$$

where, $W_{\overrightarrow{h}y}$ and $W_{\overleftarrow{h}y}$ are the weights for the forward and backward activations; b_y is the bias vector.

Fig. 6 presents the architecture of our Bi-LSTM. The networks compute a commenting probability value for each code statement with considering its context code syntactic and semantic information. The training objective is to minimize the cross entropy errors of the true labels **X** and the predicted labels **Z** of the code statements:

$$Loss_H(\mathbf{X}, \mathbf{Z}) = -\sum_{k=1}^{d} x_k \log z_k + (1 - x_k) log(1 - z_k), \tag{5}$$

where d is the vector dimensions. We solve the optimization problem using Adam update algorithm (Kingma and Ba). Once we learn the networks' parameters, the networks can be used to predict the commenting positions in source code.

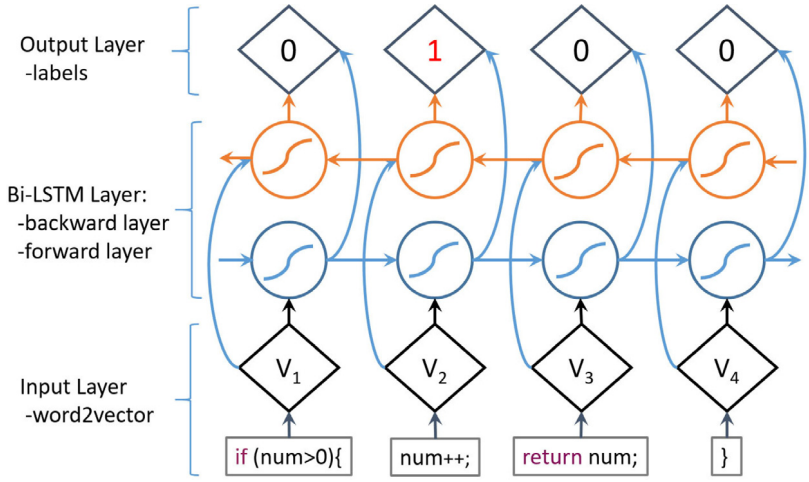


Fig. 6. Architecture of the proposed Bi-LSTM.

Table 2
The number of selected methods.

Projects	Method amount
Ant	831
ANTLR	401
ArgoUML	966
conQAT	561
JDT	6880
JHotDraw	146
JabRef	791
JFreeChart	720
Lucene	2943
Vuze	703
Total	14,942

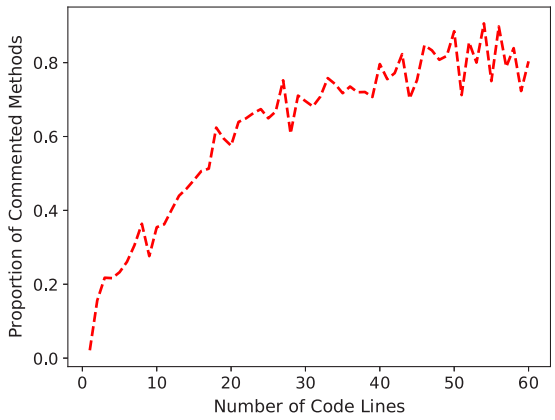


Fig. 7. Commenting rate for the methods with different number code lines.

6. Experiment setup

In this section, we evaluate the predicted commenting positions from our approach that we described in this study. Our objective is to assess the accuracy of the predicted commenting positions when comparing with the reference commenting positions in the test set. In this section we ask:

- RQ1: What is the accuracy of *CommtPst* in predicting commenting positions?
- RQ2: What are the impacts of structural and semantic features on the accuracy?
- RQ3: Does the method size (i.e., code amount) affect the accuracy of *CommtPst*?
- RQ4: Does the number of comments per method affect the accuracy of *CommtPst*?

We ask RQ1 to evaluate the Bi-LSTM model compared with other deep learning models (e.g., standard RNN and LSTM) and other word representation (e.g., BoW), which we describe in the following subsection. We ask RQ2 in order to evaluate the impacts of code structural and semantic features on the prediction accuracy, respectively. For RQ3 and RQ4, we want to evaluate whether the methods with different code lines and comments in the dataset can affect the accuracy of *CommtPst*.

6.1. Data selection and hyper-parameters setup

Intuitively, shorter methods contain fewer internal comments. We conduct a quantitative analysis to analyze the relationship between method length and the commenting rate. As Fig. 7 shown, the *X*-axis is the number of code lines in the methods, and the *Y*-axis is the proportion of the methods involving internal comments, i.e., commented methods. As the length of method increases, the percentage of commented methods increases significantly, meaning that the shorter the method, the less likely it is to be commented. When the number of lines of code is 5, less than a quarter of methods that require comments, so we chose 5 as a threshold to select the methods. After filtering out the methods with less than 5 code lines from the datasets, we get 14,942 methods (as shown in Table 2). In order to better learn and predict comment position, we randomly selected 2000 pieces of data with full zero tags and added them to the dataset. Then, there are a total of 16,942 methods in our data sets. In the experiment, 13,942 of them are used as training set, and 1500 of them are used as validation set, and 1500 of them are used as test set, which is approximately 8:1:1.

Because the methods in the datasets have different lengths, it requires our Bi-LSTM network be able to use the variable-length methods as input. According to the statistics, there are about 95% methods have less than 50 code lines in our datasets. Then, the time steps of our Bi-LSTM network are set as 50. If a method snippet has less than 50 code lines, we use zero vector to fill up it; if a method snippet has more than 50 code lines, we truncate the extra code lines. When training the model, the size of the batch is 60. We add a hidden layer to the input layer and the output layer, respectively, and the size of hidden layers is 256.

Table 3
Experimental results of *CommtPst* and *CommentSuggester*.

Approach	PRC	REC	F-MEASURE
CommentSuggester	0.660	0.573	0.601
CommtPst	0.792	0.602	0.684

6.2. Evaluation criteria

In order to evaluate the effectiveness of *CommtPst*, the instances labeled as positive by *CommtPst* are compared against the instances that are real positive. The precision (*PRC*), recall (*REC*), F-measure (*F-MEASURE*) are computed, respectively.

$$PRC = \frac{TP}{TP + FP} \tag{6}$$

$$REC = \frac{TP}{TP + FN} \tag{7}$$

$$F - MEASURE = 2 * \frac{PRC * REC}{PRC + REC} \tag{8}$$

Where *TP* is the number of true positives (positive instances correctly categorized), *FP* is the number of false positives (positive instances incorrectly categorized), *TN* is the number of true negatives (negative instances correctly categorized), and *FN* is the number of false negatives (negative instances incorrectly categorized). Therefore, the precision is the percentage of positive instances identified by *CommtPst* that are actually positive instances. The recall is the percentage of true positive instances that are successfully retrieved by *CommtPst*. The F-measure is the weighted harmonic mean of the precision and the recall and can be used as a comprehensive indicator of the combined precision and recall values.

7. Results analysis

Before discussing the experimental results, it is necessary to discuss whether either precision or recall matter most in our work. In the case of comment generation, it is more important to find comment position as correct as possible rather than as many as possible. Because incorrect comment position information leads to unnecessary comment, which degrades the quality of code comments and wastes computational resources. In the case of remind programmers adding comments, too many incorrect comment position prompts can interfere with the programmer's normal work and increase the workload, which is contrary to our original intention. Therefore, accuracy is more important than recall in our research problem.

7.1. RQ1: Prediction accuracy

In our previous work (Huang et al., 2019), we proposed a traditional machine learning method named *CommentSuggester* to predict comment position. Because commenting is closely related to the context information of source code, the traditional method predicts the comment position by extracting the code context features that the authors manually designed, which includes structural context features, syntactic context features, and semantic context features. However, manually selected features may not be sufficient, which leads to inaccurate modeling. Therefore, *CommentSuggester* have a good performance on the within-project scenario, while perform not well enough on the cross-project scenario. This may be because developers on different projects have different comment styles, which makes it difficult to learn a general comment position prediction model. Compared with traditional machine learning methods, deep learning methods can automatically mine and model the relationship between code context information and comment positions. The comparison of this method with our RNN-based method is shown in Table 3. As we can see, our method has a better accuracy rate, recall rate, and F-MEASURE value, suggesting that it does provide better prediction than traditional method.

Table 4
Evaluation results.

No.	Combinations	PRC	REC	F-MEASURE
1	RNN+BoW	0.685	0.557	0.614
2	LSTM+BoW	0.663	0.568	0.612
3	Bi-LSTM+BoW	0.722	0.603	0.657
4	RNN+word2vector	0.648	0.616	0.631
5	LSTM+word2vector	0.694	0.600	0.644
6	Bi-LSTM+word2vector	0.792	0.602	0.684

To understand the importance of word2vector, we compare the performance of word2vector with other word representation technique (i.e., BoW). The process of applying BoW to source code has two steps. First, the source code must be preprocessed to build a corpus; Second, the method snippet is denoted by a vector that reflects the term weight of the method snippet (such as: *tf–idf*). To understand the importance of Bi-LSTM, we compare the performance of other deep learning models, i.e., RNN and LSTM. RNN is suited for processing sequential data such as source code. Because RNN uses the same unit (with the same parameters) across all time steps, they are able to process sequential data of arbitrary length (Guo et al., 2017).

Since our main concern is whether *CommtPst* can accurately identify the commenting positions from the source code, we focus on the performance of *CommtPst* on the positive instances. Table 4 shows a summary of the *PRC*, *REC*, and *F-MEASURE* values for positive instances using different combinations of deep learning algorithms and word representation techniques. We can observe that RNN+BoW and LSTM+BoW have almost the same performance. Meanwhile, RNN+word2vector has better recall but worse precision than the LSTM+ word2vector, and their F-MEASURE values have no significant difference. Overall, the performance of LSTM algorithm gets equivalent as that of RNN algorithm (see *F-MEASURE* values of the 1st vs. 2nd and 4th vs. 5th rows in Table 4).

On the other hand, Bi-LSTM+BoW has better precision, recall and F-MEASURE values than LSTM+BoW. Meanwhile, Bi-LSTM+word2vector has better recall and precision than LSTM+word2vector, and Bi-LSTM+ word2vector outperforms LSTM+word2vector in terms of *F-MEASURE* value. Overall, Bi-LSTM algorithm outperforms LSTM algorithm significantly (see *F-MEASURE* values of the 2nd vs. 3th and 5th vs. 6th rows in Table 4). This suggests that Bi-LSTM algorithm can improve the performance of commenting position prediction compared with the traditional RNN and LSTM algorithm. This is due to the backward layer of Bi-LSTM, which reinforces the association relationship between code lines and comments, and further to identify more true commenting positions in source code.

RNN+BoW has better precision but worse recall and *F-MEASURE* than RNN+ word2vector; and the performance of the LSTM+word2vector is always better than that of LSTM+BoW in term of precision, recall and *F-MEASURE* values; Bi-LSTM+BoW has worse precision, recall and *F-MEASURE* than Bi-LSTM+word2 vector; In summary, word2vector outperforms BoW by a small margin (see *F-MEASURE* values of the 1st vs. 4th, 2nd vs. 5th and 3rd vs. 6th rows in Table 4). This suggests that word-level semantics encoded by bag of word is not the best way to

Table 5
Results of multiple random partition data sets.

No.	PRC	REC	F-MEASURE
1	0.792	0.602	0.684
2	0.798	0.592	0.680
3	0.763	0.615	0.681
4	0.764	0.609	0.678
5	0.786	0.600	0.681

Table 6
Features choice effect.

Features	PRC	REC	F-MEASURE
Syntactic	0.737	0.557	0.634
Semantic	0.763	0.587	0.663
Syntactic + Semantic	0.792	0.602	0.684

determine possible commenting positions in the source code. In contrast, word2vector maps individual words to a dense real-valued low-dimensional vector space. Each dimension represents a latent semantic or syntactic feature of the word. As a result, word2vector is more suitable for the commenting position prediction.

To further prove the effectiveness of our method, we conducted multiple experiments with 5 different random seed partition datasets, the results are shown in Table 5. It can be seen that our method can maintain a good performance in these experiments, and the random partition datasets affects little on the performance of our method.

In summary, Bi-LSTM combining with word2vector can get the best accuracy in commenting position prediction. For consistency, we use Bi-LSTM+ word2vector as the default combination in the experiments.

7.2. RQ2: Features choice effect

Furthermore, we evaluated the effect of different code features (i.e., syntactic features and semantic features) on the prediction accuracy, as presented in Table 6. Each result in Table 6 represents the value of a metric obtained with the Bi-LSTM algorithm combining with word2vector. We use the same training set and test set as showing in RQ1. The difference is that we only generate the syntactic feature vectors for the dataset when we evaluate the effect of syntactic features; similarly, to evaluate the effect of semantic features, we only generate the semantic feature vectors of the dataset.

Generally, every type of feature is useful. We can see that when only applying syntactic features to train the learning model, *CommtPst* achieves 0.737 of precision, 0.557 of recall, and 0.634 of F-measure. When only applying semantic features, the average values of *PRC* and *F-MEASURE* exhibited obvious improvement, increasing to 0.763 and 0.667, respectively. The growth of *PRC* and *F-MEASURE* values reveal that the syntactic context features extracted from the source code provide a good indication of the commenting patterns in the source code. Lastly, when combining syntactic and semantic features to the learning model, the average values of *PRC*, *REC* and *F-MEASURE* have increased considerably. In general, the effective fusion of the two features enables our model to learn more commenting patterns from the source code.

We can observe that the prediction accuracy presents a continuous increase when adding new types of feature into the learning model. This continuous accuracy increase can also confirm that the two types of features are useful in characterizing the logical dependencies of code statements from different perspectives and in determining whether a code line requires commenting.

Table 7
Results of different Vector Fusion Approach.

Approach	PRC	REC	F-MEASURE
Summation	0.793	0.576	0.667
Maximum	0.756	0.559	0.643
Average	0.792	0.602	0.684

To keep a fixed-length vector for each code statement, we transform a code statement into a vector representation by averaging all the vectors (i.e., syntax token and textual element vectors) the statement contains. We also considered summation method (i.e., adding up all the vectors) and maximum method (i.e., max pooling, which only keeps the maximum value of each vector) as comparison, the results show in Table 7. We can see the average method performs best, while the maximum method is significantly lower than that of average and summation method in each evaluation criterion, which could be because the maximum method breaks the positional relationship between similar code statements in a higher-dimensional vector space.

7.3. RQ3: Code amount effect

In order to examine the impact of method size on *CommtPst* determining the commenting position, we divide the test set into subsets based on the number of code statements in the methods. We observe that the number of code line of the methods in the range of 5 to 50 accounts for more than 94% of the total methods, hence we mainly focus on the performance of *CommtPst* on the methods with such range of code lines.

Fig. 8(a) shows the number of methods with different number of code statements in the test set. We can observe from the statistics: the number of code statements for most of the methods range from 5 to 30. Fig. 8(b), (c), and (d) show the *PRC*, *REC* and *F-MEASURE* values when applying *CommtPst* on the datasets of methods with different code lines. The results show that the variation of code lines of the methods shows a certain affect on the performance of *CommtPst* when the number of code lines ranges from 5 to 50, but the impact is limited.

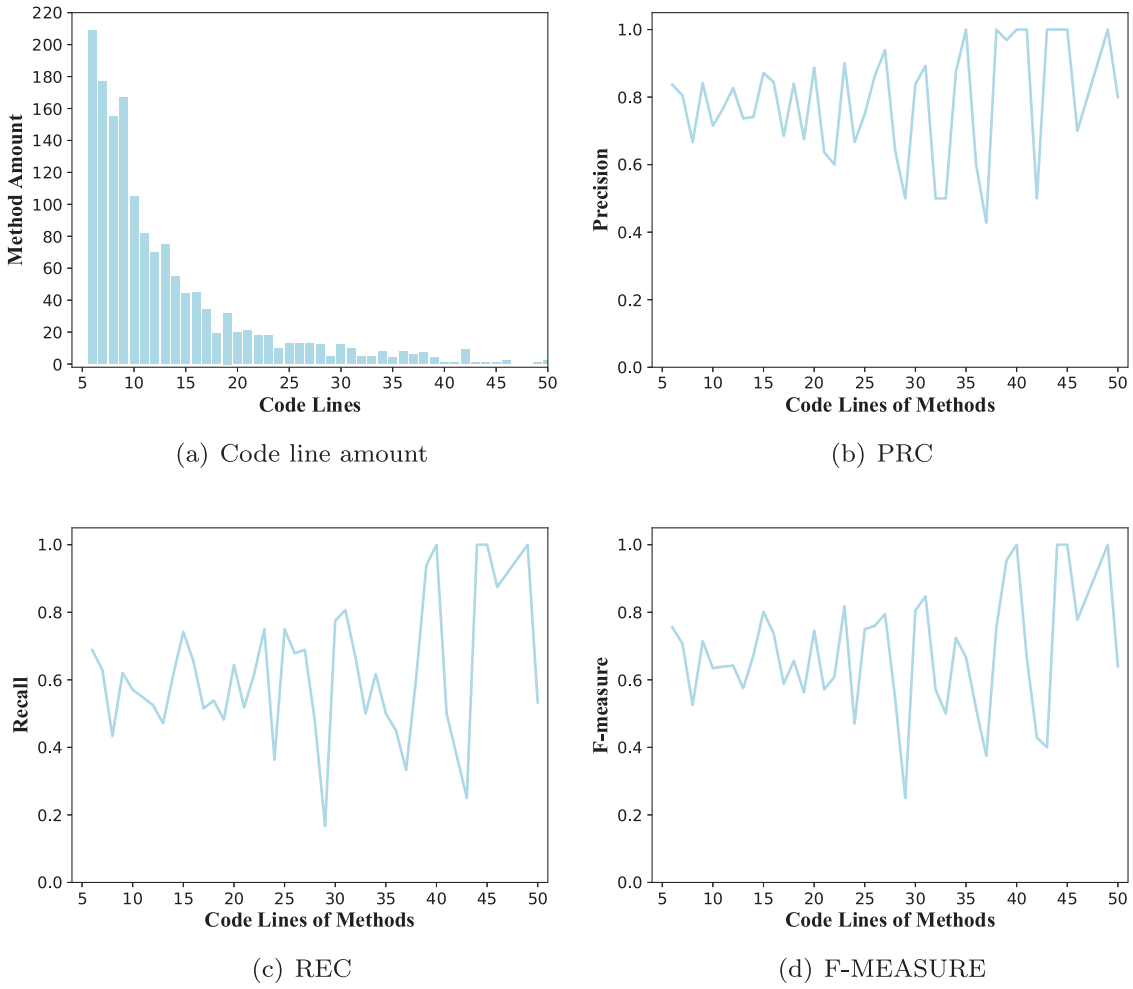
In summary, *CommtPst* shows a robustness on the datasets of the methods snippets with different number of code statements. This result also demonstrates the effectiveness of *CommtPst* applying Bi-LSTM network to model the logical relationships of code statements. This all depends on the memory ability of Bi-LSTM. With the number of code statements increases, Bi-LSTM can remember the long-term dependency of code statements via the code syntactic and semantic information, and further to identify the commenting patterns in the code sequence. As a result, *CommtPst* performs fluctuating but stable accuracy on the datasets of methods with different number of code statements.

7.4. RQ4: Comment amount effect

A method snippet may involve only one comment, or more than one comments. In order to evaluate the effect of *CommtPst* applying on the datasets of methods involving different number of comments, we classify the methods into different subsets according to their comment amount.

Fig. 9(a) shows the number of methods with different number of comments in the test set. We can observe that the methods involving less than or equal to 4 comments account for more than 81% of the total methods, hence we mainly focus on the performance of *CommtPst* on the methods with such range of comments.

Fig. 9(b), (c) and (d) show the *PRC*, *REC* and *F-MEASURE* values when applying *CommtPst* on the datasets of methods involving



(a) Code line amount (b) PRC

(c) REC (d) F-MEASURE

Fig. 8. Code lines effect on the performance of *CommtPst*.

different number of comments. As the number of comments increases, the *PRC* values achieved by *CommtPst* show an upward trend, and the *REC* values show a stability trend. As a result, the *F-MEASURE* values achieved by *CommtPst* have no significant variation when the comment amount ranges from 1 to 4. Therefore, *CommtPst* also performs a good robustness on the datasets of the methods snippets involving different number of comments if we consider the *F-MEASURE* values (with little variation).

8. Discussions: *FN* is real *FN*?

8.1. FN *is real* FN*?*

The recall shows that *CommtPst* achieved some false negative instances. That is, *CommtPst* identifies the uncommented positions that required comments. To verify whether the false negative instances really do not need comments, we qualitatively analyze some false negative instances with developers, and ask the developers that would they agree that those positions need comments?

We randomly selected 100 false negative instances identified by *CommtPst*, and asked 5 programmers to determine whether the false negative instances really need to add comments. Meanwhile, we require the programmers give us the reasons if he agrees to add comments to the false negative instances. The 5 participants are professional developers, who reported experience in software development ranging from 5 to 10 years (average 8). The 5 participants declared that they use comment frequently, mainly to comprehend the code written by others. In addition, all the participants reported they have created comment frequently.

Table 8
Estimated experimental results after FN correction.

	PRC	REC	F-MEASURE
CommtPst_FN	0.859	0.621	0.721

The 5 programmers accomplished their validations independently. To our surprise, they reached a consensus that 32 false negative instances should add comments. Namely, all the 5 programmers agree to add comments to the 32 positions. In this case, the 32 false negative instances can be considered as the comment instances that developers did not add (or forgot to add), but *CommtPst* acquires the corresponding commenting patterns from the existing dataset, and identifies the commenting positions. Presume that the observed 32% rate of fully agreed comment recommendations is consistent, then, 32% of the FN instances detected by our method should be TP instances actually, and the formula for calculating the accuracy and recall corrected is as follows:

$$precision' = \frac{TP + 0.32FN}{TP + FP + 0.32FN}, \tag{9}$$

$$recall' = \frac{TP + 0.32FN}{TP + FN}, \tag{10}$$

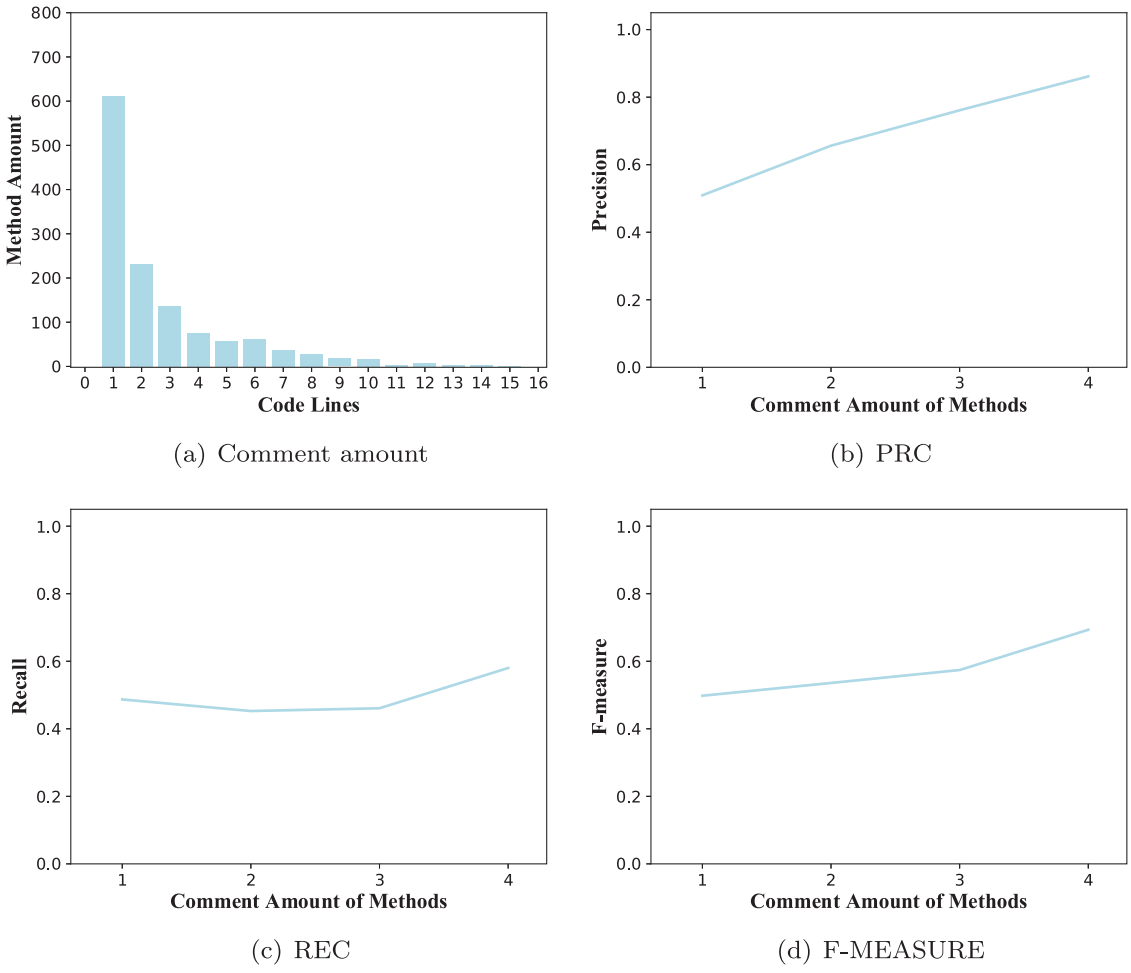


Fig. 9. Comment amount effect on the performance of *CommtPst*.

The performance of our method would be improved as shown in Table 8. There are mainly two reasons for programmers agreeing to add comments to the false negative instances. The most common reason is the complex code logic. As Fig. 10(a) shows, some programmers hold the view that the code in case 1[3] is difficult to understand, and it should be commented; Part of reason is the bad code naming style. As Fig. 10(b) shows, some programmers think that the code statements in case 2[4] have bad naming style, which cannot be self-explaining, so it should be commented.

Therefore, due to the complex code logic and bad naming style, the code may not be understood by a code reader in the future. Then, *CommtPst* can remind programmers the commenting opportunities when they are programming, and the generated comments are useful for the code readers to understand the coding decision details of the programmers.

8.2. Shuffling strategy VS. no-shuffling strategy

To prove the effectiveness of the shuffling strategy, we conducted a comparison experiment. In the shuffling strategy, we randomly shuffle the syntax tokens and textual elements for each code statement, and add the shuffled result into the corpus. In comparison, we build another corpus that firstly add the syntax tokens of a code statement in it, and then add the textual elements of the code statement in it, which is called no-shuffling strategy. Then, we conducted a comparison experiment for commenting position prediction when applying the shuffling strategy and no-shuffling strategy. Table 9 shows the results.

Table 9
Experimental results on the shuffling and no-shuffling strategies.

Strategies	PRC	REC	F-MEASURE
shuffling strategy	0.792	0.602	0.684
no-shuffling strategy	0.782	0.585	0.669

The results show that *CommtPst* with shuffling strategy performs better than that with no-shuffling strategy. Word embedding mines the relationships of syntax tokens and textual elements based on the co-occurrence of them. Then, if we did not shuffle the syntax tokens and textual elements (i.e., no-shuffling strategy), and word embedding maybe hardly mine the overall relationships of syntax tokens and textual elements in a fixed window size. As a results, the terms with the similar meaning in the syntax tokens and textual elements may not be captured. On the other hand, if we shuffle them in the corpus, word embedding is effective in revealing the overall relationship between syntax tokens and textual elements in a fixed window. Then, word embedding on the shuffling strategy is easier to express the syntax tokens and textual elements with similar meaning.

[3] coming from method dragHandle() in project ArgoUML.
[4] coming from method updateNode() in project JHotDraw.

```
...
1:  for (Fig fig : figs) {
  //##########
2:     if (fig instanceof FigMessage &&
          ((FigMessage) fig).isSelfMessage()&&
          Model.getCoreHelper().getDestination(fig.
          getOwner()).equals(workOnFig.getOwner())) {
3:          ((FigMessageSpline) ((FigMessage)
            fig).getFig()).translateFig(
            newCenterX - oldCenterX, 0);
4:     }
5: }
...
```

(a) case 1

```
...
  //##########
1: if (scaled.getY() > 2) {
2:    for (int i = 0, n = (int) Math.ceil((dh - gy0)/
      (gmydelta)); i < n; i++) {
3:       double y = gy0 + i * gmydelta;
4:       double x1 = 0;
5:       double y1 = y;
6:       double x2 = dw;
7:       double y2 = y;
...
```

(b) case 2

Fig. 10. *FN* examples (//########## denotes a false negative instance).

9. Related work

CommtPst, which identifies the commenting positions by modeling the code logical relationships via deep learning algorithms, is related to two lines of research: source code commenting practices and applying deep learning to source code analysis.

Source code commenting practices: High quality code comments contain a wealth of information that is useful in program comprehension and software maintenance (Wong et al., 2015; Wong et al.). Current code commenting practices mainly focus on comment quality evaluation and code comment generation.

Recently, many researchers have focused their attention on code comment quality evaluation and have proposed relevant evaluation metrics and tools. To the best of our knowledge, research on the problem of evaluating code comment quality was first reported in the 1990s (Oman and Hagemeister, 1992; García and Alvarez, 1996). Oman and Hagemeister (1992) and García and Alvarez (1996) evaluated code comment quality by assessing the proportion of code comments in a software system. However, this evaluation metric is crude, as some redundant comments (e.g., copyright comments) were also considered. Steidl et al. (2013a) employed decision tree to categorize code comments and assess the quality of code comments through four metrics, such as usefulness and completeness. Khamis et al. (2010) proposed a tool called JavadocMiner to evaluate the quality of code comments. JavadocMiner assessed code comment quality through a series of simple heuristic algorithms from both the association between code and comments and the language used for comments. Comment quality evaluation analyze existing codes and comments to evaluate its quality. Although comment quality evaluation can monitor the results of code comment quality, it has no guidance on comment writing and quality improvement. Therefore, we need a tool that can help improve the quality of source code comments.

Another commenting practice concerns code comment generation (Wong et al., 2015; Sridhara et al., 2010; Moreno et al., 2013; Iyer et al., 2016; Allamanis et al., 2016; Hu et al., 2018). Wong et al. proposed a novel method to automatically generate code comments by mining large-scale Q&A data from StackOverflow. Meanwhile, Wong et al. (2015) applied code clone detection techniques to discover similar code segments in software repositories and used existing comments to describe similar code segments. In addition, Sridhara et al. (2010) presented a novel technique to automatically generate descriptive comments for Java methods. Furthermore, Moreno et al. (2013) presented a technique to automatically generate readable comments for Java classes. Iyer et al. (2016) used LSTM model to design a code comment automatic generation method for C# code snippets. Allamanis et al. (2016) used convolutional neural network to generate short text descriptions for code snippets. Hu et al. took structural and semantic information from the abstract syntax tree, converted it into sequence information, and then used the machine translation model to translate the code into comments. Since then, Hu et al. (2018) took code API information into consideration, which further improves the accuracy of code comment generation. Most of the existing comment auto-generation work is a summary of a method, but when we focus on the comments inside the method, the existing works become inapplicable because we do not know where to add the comment. A tool predicting the position of the comment in the source code becomes necessary. Zhu et al. (2015) used a machine learning-based approach to automatically learn common logging practices about where to log from existing logging instances, thereby reducing the amount of work required for logging decisions. In our previous work (Huang et al., 2019), we also used a traditional machine learning-based approach, learning comment patterns from code comment instances to predict comment positions in code.

Applying deep learning to source code analysis: With the development of deep learning technology, many researchers apply deep learning to source code analysis. Mou et al. (2016) designed an abstract syntax tree-based convolutional neural network. Their network performed convolution operations on the abstract syntax trees of a program, and this operation can make the structural information contained in the program more salient. White et al. (2016) introduced deep learning algorithms to learn the syntactic and semantic information in code clone detection. In contrast to traditional tree-based methods (Jiang et al., 2007; Gabel et al., 2008), their technique transformed abstract syntax trees to full binary trees when they extracted syntactic information from programs. To integrate domain knowledge into the process traceability links recovery, Guo et al. (2017) proposed a tracing network architecture that utilized word embedding and recurrent neural network models to generate trace links. Gu et al. (2016) applied a deep learning approach, RNN Encoder–Decoder, for generating API usage sequences for a given API-related natural language query. In addition, Gu et al. proposed a deep neural network named CODEnn for code search, which learns a unifed vector representation of both source code and natural language queries so that code snippets semantically related to a query can be retrieved according to their vectors.

In addition, Xu et al. (2016) used convolutional neural network to predict semantically linkable knowledge in developer online forums. In their study, they adopted neural language model and convolutional neural network to capture word and document-level semantics of knowledge units. To detect security vulnerabilities in the software, Lin et al. (2018) used Bi-LSTM network to learn rich features that generalize across similar projects. Their approach implements vulnerable function prediction in a cross-project scenario. Meanwhile, Wang et al. (2016) employed a deep learning algorithm to automatically learn the mappings between program semantics and software defects. In their approach, they used deep belief network to automatically learn semantic features

from token vectors extracted from programs' abstract syntax trees. Zhang et al. (2019) proposed a new method of source code representation based on ASTNN (ASTNN). Unlike existing models that work on the whole AST, ASTNN breaks each large AST into a series of small statement trees and encodes the statement trees into vectors by capturing the lexical and syntax knowledge of the statements, which solves the problem of long-term dependence caused by large-size ASTs.

These works of combining deep learning with code analysis have given us a lot of inspirations, such as the way to combine the semantic information and syntactic information of codes, as well as the use of Bi-RNN to learn the context information of source codes, etc.

10. Threats to validity

In this section, we focus on the threats that could affect the results of our case studies. One of the threat to validity is the suitability of our evaluation measure. We use a conventional measure to evaluate the effectiveness of the proposed approach in this paper. Because the issue in this study can be modeled as a binary classification (i.e., commented and uncommented code statements), we introduce the precision (*PRC*), recall (*REC*), F-measure (*F-MEASURE*) evaluate the effectiveness of *CommtPst*. Meanwhile, we compare the performances of *CommtPst* under different deep learning models and word representation technique. All these metrics can evaluate the effectiveness of *CommtPst*. Thus, we believe there is little threat to suitability of our evaluation measure.

Another threat to validity is the generalizability of our results. *CommtPst* is used to identify the commenting positions in the source code. All of the source code are written by Java language. When applying our approach to the projects written by other programming languages, such as C, C++, Python, etc., some particular code syntax (e.g., pointer operation in C++) should be carefully handled when extracting the code syntax. In the future, further investigation by analyzing even more projects written by other programming languages is needed to mitigate this threat.

The last threat to validity is the quality of the comments. In the data set collection, the projects we selected have the following characteristics: a long evolutionary history, active updates, developed by well-known companies or non-profit organizations, and so on. But even with those restricted conditions, it is inevitable that these projects contain some low-quality comments (although some filter mechanisms are proposed). In the future, further investigation in the comment quality of the projects is needed to mitigate this threat.

11. Conclusion and future work

Code commenting plays an important role in program comprehension and software maintenance. This paper proposes a novel method, *CommtPst*, of identifying suitable commenting position in source code. To determine comment positions in the source code, we extracted two types of features, i.e., syntactic features and semantic features. Then, deep learning techniques were applied to learn common commenting patterns based on the two features. Experimental results demonstrated the feasibility and effectiveness of *CommtPst*. In the future, we will further consider the complementarity of *CommtPst* and the approach used to automatically comment code snippets. Such a combination can build a more smarter commenting approach, which can self-determine the commenting positions in the source code, and then generate the comments for the code snippets inside the method body. Obviously, such a combination can make *CommtPst* play a more important role in practice.

CRediT authorship contribution statement

Yuan Huang: Methodology, Validation, Writing - original draft. **Xinyu Hu:** Data curation, Software, Resources. **Nan Jia:** Methodology, Software, Resources. **Xiangping Chen:** Project administration, Writing - review & editing. **Zibin Zheng:** Investigation, Writing - review & editing, Validation. **Xiapu Luo:** Formal analysis, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research is supported by the Key-Area Research and Development Program of Guangdong Province (2020B010164002), National Natural Science Foundation of China (61902441, 61976061), Guangdong Basic and Applied Basic Research Foundation (2020A1515010973), Hong Kong RGC Projects (No. 152223/17E, 152239/18E), China Postdoctoral Science Foundation (2018M640855), the Fundamental Research Funds for the Central Universities (20wkpy06, 20lgpy129). Xiangping Chen is the corresponding author.

References

Allamanis, M., Peng, H., Sutton, C., 2016. A convolutional attention network for extreme summarization of source code. In: Proceedings of the 33rd International Conference on Machine Learning, PMLR, vol. 48. PMLR, pp. 2091–2100.

Bavota, G., Gethers, M., Oliveto, R., Poshyvanyk, D., Lucia, A.d., 2014. Improving software modularization via automated analysis of latent topics and dependencies. ACM Trans. Softw. Eng. Methodol. 23 (1), http://dx.doi.org/10.1145/2559935.

Bengio, Y., Simard, P., Frasconi, P., 1994. Learning long-term dependencies with gradient descent is difficult. IEEE Trans. Neural Netw. 5 (2), 157–166. http://dx.doi.org/10.1109/72.279181.

Beniamini, G., Gingichashvili, S., Orbach, A.K., Feitelson, D.G., 2017. Meaningful identifier names: The case of single-letter variables. In: Proceedings of the 25th International Conference on Program Comprehension. In: ICPC '17, IEEE Press, Piscataway, NJ, USA, pp. 45–54. http://dx.doi.org/10.1109/ICPC.2017.18, https://doi.org/10.1109/ICPC.2017.18.

Chen, H., Huang, Y., Liu, Z., Chen, X., Zhou, F., Luo, X., 2019. Automatically detecting the scopes of source code comments. J. Syst. Softw. 153, 45–63. http://dx.doi.org/10.1016/j.jss.2019.03.010, http://www.sciencedirect.com/science/article/pii/S016412121930055X.

Gabel, M., Jiang, L., Su, Z., 2008. Scalable detection of semantic clones. In: Proceedings of the 30th International Conference on Software Engineering. In: ICSE '08, ACM, New York, NY, USA, pp. 321–330. http://dx.doi.org/10.1145/1368088.1368132, http://doi.acm.org/10.1145/1368088.1368132.

García, M.J.B., Alvarez, J.C.G., 1996. Maintainability as a key factor in maintenance productivity: A case study. In: Proceedings of the 1996 International Conference on Software Maintenance. In: ICSM '96, IEEE Computer Society, Washington, DC, USA, 87–.

Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A., 2014. The Java Language Specification (Java SE 8 edition), Java SE 8 ed. Oracle.

Gu, X., Zhang, H., Kim, S., Deep code search, in: 2018 IEEE/ACM 40th International Conference on Software Engineering, ICSE, 2018, pp. 933–944.

Gu, X., Zhang, H., Zhang, D., Kim, S., 2016. Deep API learning. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: FSE 2016, Association for Computing Machinery, New York, NY, USA, pp. 631–642. http://dx.doi.org/10.1145/2950290.2950334, https://doi.org/10.1145/2950290.2950334.

Guo, J., Cheng, J., Cleland-Huang, J., 2017. Semantically enhanced software traceability using deep learning techniques. In: 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE. pp. 3–14. http://dx.doi.org/10.1109/ICSE.2017.9.

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., A deep code comment generation, in: Proceedings of the 26th IEEE International Conference on Program Comprehension, 2018, pp. 200–210.

Hu, X., Li, G., Xia, X., Lo, D., Lu, S., Jin, Z., 2018. Summarizing Source Code with Transferred API Knowledge. pp. 2269–2275. http://dx.doi.org/10.24963/ijcai.2018/314.

Huang, Y., Chen, X., Liu, Z., Luo, X., Zheng, Z., 2017. Using discriminative feature in software entities for relevance identification of code changes. J. Softw.: Evol. Process. 29 (7), e1859. http://dx.doi.org/10.1002/smr.1859, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1859.

Huang, Y., Hu, X., Jia, N., Chen, X., Xiong, Y., Zheng, Z., 2019. Learning code context information to predict comment locations. IEEE Trans. Reliab. 1–18. http://dx.doi.org/10.1109/TR.2019.2931725.

Huang, Y., Jia, N., Chen, X., Hong, K., Zheng, Z., 2018. Salient-class location: Help developers understand code change in code review. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2018, ACM, New York, NY, USA, pp. 770–774. http://dx.doi.org/10.1145/3236024.3264841, http://doi.acm.org/10.1145/3236024.3264841.

Huang, Y., Jia, N., Shu, J., Hu, X., Chen, X., Zhou, Q., 2020. Does your code need comment? Softw. - Pract. Exp. 50 (3), 227–245. http://dx.doi.org/10.1002/spe.2772, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2772.

Huang, Y., Zheng, Q., Chen, X., Xiong, Y., Liu, Z., Luo, X., 2017. Mining version control system for automatically generating commit comment. In: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM. pp. 414–423. http://dx.doi.org/10.1109/ESEM.2017.56.

Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L., 2016. Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational. pp. 2073–2083. http://dx.doi.org/10.18653/v1/P16-1195.

Jiang, L., Misherghi, G., Su, Z., Glondu, S., 2007. DECKARD: Scalable and accurate tree-based detection of code clones. In: Proceedings of the 29th International Conference on Software Engineering. In: ICSE '07, IEEE Computer Society, Washington, DC, USA, pp. 96–105. http://dx.doi.org/10.1109/ICSE.2007.30, http://dx.doi.org/10.1109/ICSE.2007.30.

Keyes, J., 2002. Software Engineering Handbook. CRC Press.

Khamis, N., Witte, R., Rilling, J., 2010. Automatic quality assessment of source code comments: The javadocminer. In: Proceedings of the Natural Language Processing and Information Systems, and 15th International Conference on Applications of Natural Language To Information Systems. In: NLDB'10, Springer-Verlag, Berlin, Heidelberg, pp. 68–79, http://dl.acm.org/citation.cfm?id=1894525.1894535.

Kingma, D., Ba, J., Adam: A Method for Stochastic Optimization, in: International Conference on Learning Representations, 2014.

Li, X., Jiang, H., Kamei, Y., Chen, X., 2018. Bridging semantic gaps between natural languages and APIs with word embedding. IEEE Trans. Softw. Eng. (01), http://dx.doi.org/10.1109/TSE.2018.2876006, 1–1.

Lin, G., Zhang, J., Luo, W., Pan, L., Xiang, Y., Vel, O.D., Montague, P., 2018. Cross-project transfer representation learning for vulnerable function discovery. IEEE Trans. Ind. Inf. 14 (7), 3289–3297. http://dx.doi.org/10.1109/TII.2018.2821768.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., Dean, J., 2013. Distributed representations of words and phrases and their compositionality. In: Proceedings of the 26th International Conference on Neural Information Processing Systems. In: NIPS'13, Curran Associates Inc., USA, pp. 3111–3119, http://dl.acm.org/citation.cfm?id=2999792.2999959.

Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L., Vijay-Shanker, K., 2013. Automatic generation of natural language summaries for Java classes. In: Program Comprehension, ICPC 2013 IEEE 21st International Conference on. pp. 23–32. http://dx.doi.org/10.1109/ICPC.2013.6613830.

Moreno, L., Bavota, G., Di Penta, M., Oliveto, R., Marcus, A., Canfora, G., 2014. Automatic generation of release notes. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: FSE 2014, ACM, New York, NY, USA, pp. 484–495. http://dx.doi.org/10.1145/2635868.2635870, http://doi.acm.org/10.1145/2635868.2635870.

Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z., 2016. Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence. In: AAAI'16, AAAI Press, pp. 1287–1293, http://dl.acm.org/citation.cfm?id=3015812.3016002.

Nguyen, T.D., Nguyen, A.T., Phan, H.D., Nguyen, T.N., 2017. Exploring API embedding for API usages and applications. In: 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE. pp. 438–449. http://dx.doi.org/10.1109/ICSE.2017.47.

Oman, P., Hagemeister, J., 1992. Metrics for assessing a software system's maintainability. In: Proceedings Conference on Software Maintenance 1992. pp. 337–344. http://dx.doi.org/10.1109/ICSM.1992.242525.

Porter, M., 1980. An algorithm for suffix stripping. Program: Electronic Library and Information Systems, vol. 14. pp. 130–137. http://dx.doi.org/10.1108/eb046814.

Poshyvanyk, D., Gethers, M., Marcus, A., 2013. Concept location using formal concept analysis and information retrieval. ACM Trans. Softw. Eng. Methodol. 21 (4), http://dx.doi.org/10.1145/2377656.2377660, https://doi.org/10.1145/2377656.2377660.

Rocktäschel, T., Grefenstette, E., Hermann, K.M., Kociský, T., Blunsom, P., 2015. Reasoning about entailment with neural attention. CoRR abs/1509.06664, arXiv:1509.06664.

Schuster, M., Paliwal, K.K., 1997. Bidirectional recurrent neural networks. IEEE Trans. Signal Process. 45 (11), 2673–2681. http://dx.doi.org/10.1109/78.650093.

Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., Vijay-Shanker, K., 2010. Towards automatically generating summary comments for java methods. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. In: ASE '10, ACM, New York, NY, USA, pp. 43–52. http://dx.doi.org/10.1145/1858996.1859006, http://doi.acm.org/10.1145/1858996.1859006.

Steidl, D., Hummel, B., Juergens, E., 2013a. Quality analysis of source code comments. In: 2013 21st International Conference on Program Comprehension, ICPC. pp. 83–92. http://dx.doi.org/10.1109/ICPC.2013.6613836.

Steidl, D., Hummel, B., Juergens, E., 2013b. Quality analysis of source code comments. In: 2013 21st International Conference on Program Comprehension, ICPC. pp. 83–92. http://dx.doi.org/10.1109/ICPC.2013.6613836.

Tai, K.S., Socher, R., Manning, C.D., 2015. Improved semantic representations from tree-structured long short-term memory networks. Comput. Sci. 5 (1), 36.

Tao, Y., Dang, Y., Xie, T., Zhang, D., Kim, S., 2012. How do software engineers understand code changes?: An exploratory study in industry. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. In: FSE '12, ACM, New York, NY, USA, pp. 51:1–51:11. http://dx.doi.org/10.1145/2393596.2393656, http://doi.acm.org/10.1145/2393596.2393656.

Thongtanunam, P., McIntosh, S., Hassan, A.E., Iida, H., 2016. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In: Proceedings of the 38th International Conference on Software Engineering. In: ICSE '16, ACM, New York, NY, USA, pp. 1039–1050. http://dx.doi.org/10.1145/2884781.2884852, http://doi.acm.org/10.1145/2884781.2884852.

Wang, S., Liu, T., Tan, L., 2016. Automatically learning semantic features for defect prediction. In: Proceedings of the 38th International Conference on Software Engineering. In: ICSE '16, ACM, New York, NY, USA, pp. 297–308. http://dx.doi.org/10.1145/2884781.2884804, http://doi.acm.org/10.1145/2884781.2884804.

White, M., Tufano, M., Vendome, C., Poshyvanyk, D., 2016. Deep learning code fragments for code clone detection. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. In: ASE 2016, ACM, New York, NY, USA, pp. 87–98. http://dx.doi.org/10.1145/2970276.2970326, http://doi.acm.org/10.1145/2970276.2970326.

Wong, E., Liu, T., Tan, L., 2015. Clocom: Mining existing source code for automatic comment generation. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER. IEEE, pp. 380–389.

Wong, E., Yang, J., Tan, L., AutoComment: Mining question and answer sites for automatic comment generation, in: Ieee/Acm International Conference on Automated Software Engineering, 2014, pp. 562–567.

Xia, X., Lo, D., Pan, S.J., Nagappan, N., Wang, X., 2016. HYDRA: Massively compositional model for cross-project defect prediction. IEEE Trans. Softw. Eng. 42 (10), 977–998.

Xu, B., Ye, D., Xing, Z., Xia, X., Chen, G., Li, S., 2016. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering, ASE. pp. 51–62.

Yan, M., Xia, X., Shihab, E., Lo, D., Yin, J., Yang, X., 2019. Automating change-level self-admitted technical debt determination. IEEE Trans. Softw. Eng. 45 (12), 1211–1229.

Yang, X., Lo, D., Xia, X., Bao, L., Sun, J., 2016. Combining word embedding with information retrieval to recommend similar bug reports. In: 2016 IEEE 27th International Symposium on Software Reliability Engineering, ISSRE. pp. 127–137. http://dx.doi.org/10.1109/ISSRE.2016.33.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X., 2019. A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st International Conference on Software Engineering, ICSE. IEEE, pp. 783–794.

Zhu, J., He, P., Fu, Q., Zhang, H., Lyu, M.R., Zhang, D., 2015. Learning to log: Helping developers make informed logging decisions. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1. IEEE, pp. 415–425.

Yuan Huang received the Ph.D. degree in computer science from Sun Yat-sen University in 2017. He is an associate research fellow in the School of Data and Computer Science, Sun Yat-sen University. He is particularly interested in software evolution and maintenance, code analysis and comprehension, and mining software repositories.

Xinyu Hu is a postgraduate student at the Sun Yat-sen University. His research interest includes software engineering, code analysis and comprehension, and mining software repositories.

Nan Jia received the Ph.D. degree in computer science from Sun Yat-sen University in 2017. She is a lecturer in the Department of Big Data Science and Technology, Hebei GEO University. She is particularly interested in data mining and software engineering.

Xiangping Chen is an associate professor at the Sun Yat-sen University. She got her Ph.D. degree from the Peking University in 2010. Her research interest includes software engineering and mining software repositories.

Zibin Zheng received the Ph.D. degree from the Chinese University of Hong Kong, in 2011. He is currently a Professor at School of Data and Computer Science with Sun Yat-sen University, China. He serves as Chairman of the Software Engineering Department. He published over 120 international journal and conference papers, including 3 ESI highly cited papers. According to Google Scholar, his papers have more than 7000 citations, with an H-index of 42. His research interests include blockchain, services computing, software engineering, and financial big data. He was a recipient of several awards, including the Top 50 Influential Papers in Blockchain of 2018, the ACM SIGSOFT Distinguished Paper Award at ICSE2010, the Best Student Paper Award at ICWS2010. He served as BlockSys'19 and CollaborateCom'16 General Co-Chair, SC2'19, ICIOT'18 and IoV'14 PC Co-Chair.

Xiapu Luo received the Ph.D. degree in computer science from The Hong Kong Polytechnic University. He was a Post-Doctoral Research Fellow with the Georgia Institute of Technology. He is currently an Assistant Professor with the Department of Computing and an Associate Researcher with the Shenzhen Research Institute, The Hong Kong Polytechnic University. His current research focuses on smart phone security and privacy, network security and privacy, and Internet measurement.