IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 5, 2021, accepted January 20, 2021, date of publication February 1, 2021, date of current version February 8, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3055955

A Multi-Module Based Method for Generating Natural Language Descriptions of Code Fragments

XUEJIAN GAO, XUE JIANG, QIONG WU, XIAO WANG, LEI LYU, AND CHEN LYU
School of Information Science and Engineering, Shandong Normal University, Jinan 250014, China

Corresponding author: Chen Lyu (lvchen@sdnu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61602286 and Grant 61976127, in part by the Shandong Key Research and Development Program under Grant 2018GGX101003, and in part by the Shandong Province Higher Educational Science and Technology Program under Grant J16LN09.

ABSTRACT Code fragment natural language description generation, also known as code summarization, refers to obtaining a natural language sequence describing a given code fragment's functionality. It is broadly agreed that applying code summarization into production can significantly improve the efficiency of software development and maintenance. In recent years, syntactic analysis (SA) technology and Latent Dirichlet Allocation (LDA) has been widely used in code summarization and has achieved good results. However, most of the existing techniques focus on core code statements, and thus their generated code summarization lacks a logical description of the code fragment's holistic information. To this end, we propose a code summarization method based on multiple modules to generate natural language for each code statement by constructing a new type of natural language template. Meanwhile, to utilize the code fragment's holistic information, we adopt the code statement partition rules and cosine similarity measure to rank and optimize the weight of the overall information of the code fragment, and finally generate the holistic natural language description of the code fragment. The experimental results demonstrate that our method can generate more concise and logical natural language descriptions than existing models.

INDEX TERMS Source code summarization, program comprehension, program description.

I. INTRODUCTION

With the vigorous development of computer technology, various software and applications are emerging in an endless stream. The amount of source code involved and its dependency libraries have also shown a blowout growth [6]. Compared with natural languages, the semantics of programming languages are more abstract. The code contains a large number of APIs with complex dependencies, which will significantly increase the difficulty for program developers to understand, modify, and maintain the code [23]. Code summarization technology can automatically analyze code syntax and semantic structure and generate corresponding program function descriptions, effectively facilitating programmers to understand the code.

There has been a lot of studies focused on code summarization tasks. Haidue *et al.* [1] divided the code summarization method into extraction summarization method and abstract summarization method, and applied text summarization technology to perform code summarization. Specifically, they adopt the Vector Space Model (VSM) and Latent Semantic Index (LSI) to extract code keywords for generating code summarization. LSI usually analyzes the vector relationship between words and documents for text mining and improves information retrieval accuracy. However, this method relies too much on the extraction of keywords. The keywords extracted in practical applications belong to code surface semantic information, which does not reflect the code function well. For this reason, Brian *et al.* [2] proposed hierarchical Pachinko Allocation Model (hPAM) to implement code summarization tasks. Pachinko Allocation Model (PAM) is used to express potential relationships between topics by constructing directed acyclic graphs (DAG). By building a 4-layer PAM using directed acyclic graph, the code themes were connected in such a graph, and code summaries are obtained using text retrieval techniques. Compared with previous methods, hPAM can generate code descriptions more accurately by overcoming the limitation of extracting only keywords. Different from the above work, Laura *et al.* [3] proposed a method to generate structured code

The associate editor coordinating the review of this manuscript and approving it for publication was Mario Luca Bernardi.

summarization automatically. This method used heuristic methods to generate program descriptions by analyzing Java class information automatically. This work is the first to consider and analyze class-related information in code summarization generation [43]. However, its drawback is that the contextual relationships of classes, i.e., the invocation relationships with other classes, are not considered in the generation of Java class summarization. At present, the traditional work lacks global consideration of the comprehensive information of the code fragment. Most of the existing methods focus on the strong migration of text summarization techniques for code interpretation, with semantic ambiguity problems and keyword weight distribution [24].

In recent years, the booming development of deep learning and its huge application potential has opened another door for code summarization [25]. Nguyen *et al.* proposed to use artificial deep neural networks (DNN) for code summarization. Specifically, Nguyen *et al.* [4] built a 4-layer neural network: the input layer, feature layer, concept layer, and output layer. They utilized identifiers, APIs, and other code features as vectorized input and abstract these vectorized code feature into concepts useful for code summarization. The research demonstrated that DNN is capable of completing code summarization tasks. Iyer *et al.* [5] proposed the CODE-NN model and applied it to the mission of code summarization. Through the long and short-term memory network (LSTM) [26] and attention mechanism [27], code features were extracted as the input, and the natural language summarization was output. Experiments on C# and SQL languages demonstrated that this model was suitable for code summarization tasks and achieved significant performance. However, the DNN-based models require a large amount of the same type of massive data for training, and the quality of the dataset directly affects the performance of the model. How to construct or select high-quality code summarization datasets is still a challenging problem to be solved in such DNN-based models.

To overcome the limitation of the traditional code summarization method that lacks global consideration. We propose a code summarization method based on multiple modules. This method does not depend on deep learning models and specific training datasets. It mainly includes three algorithm function modules: the preprocessing module, the internal processing module, and the external processing module. According to the established division rules, the preprocessing module divides the source code fragments into various source code statements, which are described in detail in Section III-A. The internal processing module processes various source code statements and combines natural language templates to generate natural language descriptions for each source code statement. The external processing module fully considers the global information of the code fragments. It uses the sorting optimization strategy to optimize and fuses each source code statement's natural language description. Finally, the global natural language description of the source code fragment can be generated. To verify the superiority of this method, we conduct experiments on two different types of datasets. The experimental results show that our method performs better than existing approaches.

The main contributions of this paper are summarized as follows.

1) We propose a code summarization method for the overall source code fragment, which divides, optimizes, and sorts the code statements through a multi-module processing mechanism to consider the code fragment's global information and reduce the redundant information in the generated results.
2) We propose a weight calculation mechanism to prioritize the generated program description sentences to ensure that the generated natural language has high logic.

II. PROBLEM STATEMENT

To improve the quality of the software, program maintainers need to regularly maintain the software and understand the meaning or usage of each identifier in code. In reality, the probability for the human to choose the same core keywords for the same concept is less than 20% [22]. If program maintainers and developers have different ways to understand the same concept, then program maintenance probably deviates from the original track [41]. Therefore, there is an urgent need to generate a short description for the code to describe the code function accurately and effectively avoid errors caused by differences in conceptual understanding between maintainers and developers [28].

Studies have shown that programmers prefer to read the concise natural language in software development and maintenance compared to source code [29]. In this section, we explore the problem of how to generate easy-to-understand natural language descriptions for source code automatically.

The fundamental goal is that the natural language description should contain the most critical information in the source code to describe the code's functionality fully. The programmer should read the code description with the same effect as reading the source code, but with different reading efficiency [30].

There are many challenges in realizing the program description of the source code: 1) Logic: it is necessary to ensure that the generated natural language description conforms to the natural language logic and human reading habits. In this way, programmers can clearly understand the meaning of the source code when reading these descriptions [31]. 2) Conciseness: the natural language description should be concise since too long natural language description will increase the reader's reading burden and reduce the effectiveness of the code summarization work [32]. 3) Completeness: part of the information is probably lost during the conversion from code to text description. The premise is that this information is irrelevant; the information that affects understanding the method cannot be lost [33].

III. APPROACH

As illustrated in Fig. 1, we propose a code summarization method based on multiple modules: 1) Preprocessing module, which divides a given code fragment into various statements according to division rules; 2) Internal processing module, which utilizes CamelCase and Software Word Usage Model (SWUM) to split identifiers of various statements to mine source code characteristics, and combines natural language templates to generate natural language descriptions of source code statements; 3) External processing module, which sorts the natural language descriptions generated in the internal processing module. According to the sentence weight value algorithm, we adopt the cosine similarity measurement method to optimize the sorted natural language description. Finally, the natural language description of the source code fragment is generated.

Note that, the goal of Module 1 is to classify each code statement. Module 2 aims to mine the information in the source code to generate a natural language description of source code statements. The objective of Module 3 is to delete redundant information and optimize the text information to generate natural language descriptions of the source code fragments that conform to human reading habits.

A. PREPROCESSING MODULE

The preprocessing module's input is a given source code fragment, and the output is the type of the recognized statement, including direct statements, indirect statements, and special statements. Fig. 2 shows the framework of the preprocessing module. The concepts and division rules of each type of statement are provided in this section.

This module quickly distinguishes statement types based on common formats defined by statement division rules. The purpose of distinguishing various types of statements is to process the source code fragments' statements hierarchically to determine their priority. Since each statement contains different contents, the focus of processing the three types of statements is also different. The analysis of direct statements gives the program maintainer a general idea of the behavior of the code fragment. Although these direct statements do not contain all the code information, they describe the main behavior of the code fragment. Compared to the other two types of statements, indirect statements are the most numerous in code fragments and contain a lot of important information, such as the context and calling logic of a code fragment. There are fewer types of special statements, but they usually have a higher weight in the Java language.

A standard Java program is used as an example to expand the discussion of various types of statements.

1. *Type variable*/ $= X$;
Meaning: (Declaration/assignment of types to variables)
2. *variable.method*;
Meaning: (Method call to a variable)
3. *execution variable*(*e.g.*, *Operation*);
Meaning: (A Series of Execution Operations on Variables)

Note that the above code includes the definition, call and execution of the variable (e.g. operation). The type of X is a numeric value or an instantiated object.

1) DIRECT STATEMENT

Definition 1: In the source code fragment, the statement that can express the operation behaviour of the source code fragment is called the direct statements. By analyzing the direct statements, we can determine the operation behaviour of the source code fragment.

We consider two different formats of code and give their direct statement definitions, respectively.

Case 1: The code that is in the standard format. It usually includes type declaration/assignment of variables, method calls to variables, and a series of execution operations on variables. In this case, the operation statement executed on the variable (e.g., output variable value) contains the direct content of the code fragment. Thus, we define such a statement as a direct statement, where the result of running the program can be obtained directly by executing a series of operation statements on the variable.

Case 2: The code that does not conform to the standard format. In this case, the type declaration/assignment part of the variable or the set of execution operations on the variable is missing from the source code fragment. At this point, we identify the operations performed on variables as direct statements, which are type declaration/assignment statements or method call statements for variables.

```
int sum=0;
for(int i=0; i<str.length; i++) {
   int myint = integer.parseInt(str[i])
   sum = sum + myint;
}
```

As listed in the code above, it contains assignments to the variables and a series of execution operations on the variable. If the above code contained only the second statement, the type declaration/assignment of the variable would be missing from the code. At this point, we consider it to belong to Case 2 and can determine that this statement is a direct statement.

2) INDIRECT STATEMENT

Definition 2: In the source code fragments, statements that assist the execution of the direct statement and serve as the primary operation of the program are defined as the indirect statement.

Indirect statements are the central part of the source code fragments and often occupy a lot of space in the entire source code fragments. Indirect statements usually contain the operation, subject, and auxiliary parameters of a code fragment, which provide the main content for the generated natural language description. Therefore, compared with direct statements, indirect statements also contain a lot of information, such as variable declaration types and method calls. The

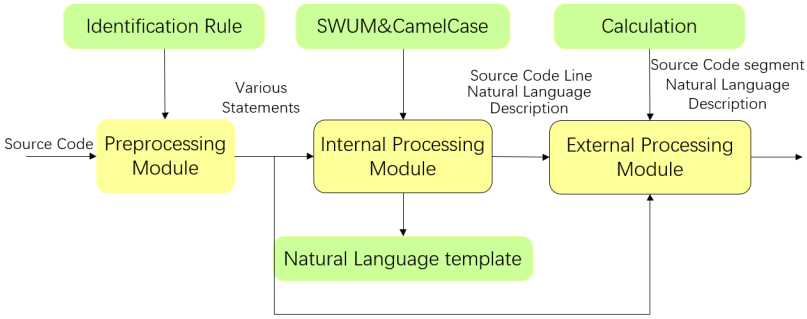


FIGURE 1. The overall architecture of our code summarization approach.

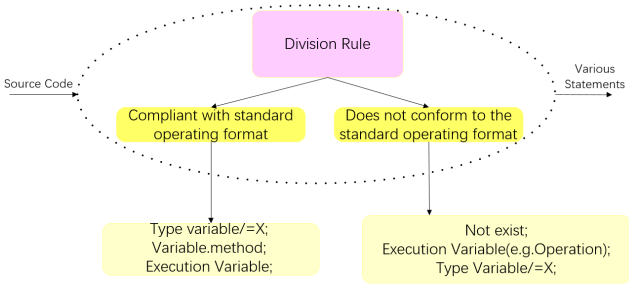


FIGURE 2. The preprocessing frame diagram of our method.

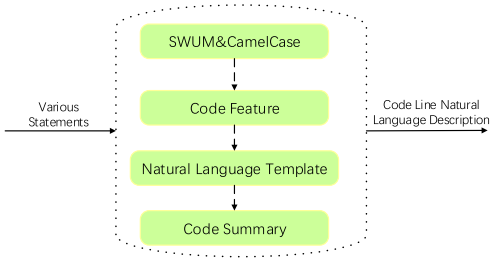


FIGURE 3. The internal processing frame diagram of our method.

information generated from indirect statements can enable program maintainers better to understand the meaning of an entire code fragment.

3) SPECIAL STATEMENT

Usually, if some special statements appear in a code fragment, they often convey more code information than direct statements, e.g., the standard output function in the Java language. Given a piece of code that contains the statement `sum = sum + myint`, the statement is judged to be a direct statement according to the format of the Java program, and the statement conveys information about the "sum" operation of this piece of program code. Suppose you add the statement `System.out.println("The sum of the array elements is:" + sum)` at the end of the program code, the result of running the code is "The sum of the array elements is: 100". Compared with the direct statement `sum = sum + myint`, the information delivered is richer and more direct. Special statements have higher priority than direct statements. We stipulate that in Java programs, statements containing the `System.out.println( )` function are special statements.

B. INTERNAL PROCESSING MODULE

Fig. 3 illustrates the overall framework of the internal processing module. Specifically, it first uses the SWUM and CamelCase to mine the identifier characteristics and then combines the natural language template to generate the description of code statements.

The input information of this module is various types of statements, and the output information is the natural language description of the code statements. Its main task is to identify the actions, themes, and auxiliary parameters in these input statements.

1) NATURAL LANGUAGE TEMPLATES

Since the location of the subject and auxiliary parameters is not fixed, we created a natural language text template in the following form:

Verb A preposition B

In this template, A is the subject and B is the auxiliary parameter. The positions of "verb" and "preposition" are fixed. We stipulate that in a specific method call if part B does not exist, the generated natural language ignores "preposition". We will introduce the conversion strategy of natural language templates from four common code forms.

Case 1: Program statements are in the form of variable declaration or assignment. The format of a statement of this type is "Type variable = number". The right side of the operator is the subject, and the left side is the auxiliary parameter. In the processing of the operator itself, if there is no unique keyword (such as "new") on the right side, "assignment" is placed after the subject. Eventually, the description of the above statement can be generated as "add number assignment to type variable" according to the natural language template.

Case 2: Program statements are in the form of object instantiation, i.e., statements contain the "new" keyword, and such statements follow the format: "Class Object = new Class( )". This module uses CamelCase and SWUM to split the identifier on the right side of the operator. We manually

added a template to the internal processing module according to this format: ''Instantiation class for the object''. The class after the keyword ''new'' is the same as the class in front of the object, so only the class name after the keyword ''new'' needs to be processed, and ''new'' itself is replaced by ''Instantiation'' in the description statement. For example, given a direct statement: ''DataOutputStream ds = new DataOutputStream(fs)'', we first use the CamelCase method and SWUM to split the identifier ''DataOutputStream'' into ''Data, Output, Stream''. Since the parameter ''fs'' is not defined, the natural language sentence ''Instantiation data output stream for fs'' is generated.

Case 3: The form of the program statements is the class name-calling method. This form is more complicated, and usually, such statements follow the format: ''class.method( )''. First, we adopt the CamelCase and SWUM to split the format into ''Class.verbW (Parameter)'' to make it satisfy the natural language template. ''W'' is the abbreviation of the word, which is used to indicate the identifier after the verb. In the actual project source code, ''W'' and ''Parameter'' may not exist. The topics may be ''Class'', ''W'', and ''Parameter''. The internal processing module judges the position of the subject according to the following situations:

1) When both ''W'' and ''Parameter'' exist, and there is no connection between the names of ''W'' and ''Parameter'', ''W'' is the subject by default. For example, in ''container.setLayout(null)'', ''Layout'' is the subject, and ''null'' is the auxiliary parameter. At this time, ''Layout'' corresponds to part ''A'' in the natural language template, and ''null'' corresponds to ''B''. According to the template, a description sentence is generated: ''set Layout for container null''.
2) When ''W'' exists and ''Parameter'' does not exist. Such as ''container.setLayout( )'', the subject is ''W''. Generate a description sentence based on the template: ''set layout for null''.
3) When ''W'' does not exist and ''Parameter'' exists, such as ''container.set(null)'' , subject is ''null''. Generate a description sentence based on the template: ''set null for container''.
4) When neither ''Parameter'' nor ''W'' exists. For example, for ''container.set( )'', ''class'' is the subject, and the description sentence is generated according to the template: ''set container''.

Case 4: For ''System.out.println( )''. If the output of the function is a natural language sentence, the content in the function is directly extracted to describe the sentence. If the output of the function contains a special form of identifier, please refer to Case 1, Case 2, and Case 3 for handling.

C. EXTERNAL PROCESSING MODULE

Since the output of the internal processing module is a natural language description of an independent source code statement, the entire source code fragment's behavior cannot be described in general. To address this problem, we design an external processing module to establish connections between independent sentences for sentence optimization. Finally, a natural language description of the source code fragment is generated.

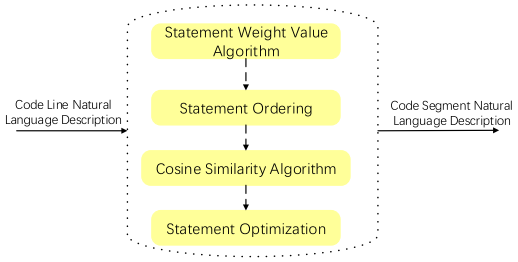


FIGURE 4. The external processing frame diagram of our method.

Fig. 4 shows the overall framework of the external processing module. The external processing module takes description sentences (collectively referred to as sentences) output by the internal processing module as input. It first sorts these sentences through the sentence weight sorting algorithm according to the subject word and sentence type weight value. Then, it adopts the cosine similarity algorithm to calculate the similarity between every two sentences. Finally, the redundant information is removed based on the sentence optimization algorithm to form a natural language description of the source code fragment.

The ultimate goal of this module is to optimize multiple independent sentences into a short natural language description text that conforms to human reading habits. A total of m sentences may contain n subjects, so it is urgent to distinguish the most important sentences and subject words, and delete duplicate words. To avoid accidental deletion of subject words, we sort them first and then delete them.

1) SENTENCES WEIGHT ALGORITHM

The final natural language description content is automatically aggregated from each line of sentences. We aim to prioritize individual sentences based on their weight values. Specifically, we first arrange the sentences with high weight values before the sentences with low weight values, and then the natural language description of the source code fragment can be generated according to the sentence with the highest weight value. Based on the above purpose, we calculate the importance of sentences from two aspects: sentence subject and sentence type. The calculation method of the weight value of statement S_d $(1 \leq d \leq n)$ in all statements set S_n $(n \geq 1)$ is as follows:

$$Wei(S_d) = Wei_S(S_d) + Wei_T(S_d), \tag{1}$$

where $Wei(S_d)$ represents the weight value of sentence S_d, $Wei_S(S_d)$ represents the weight value of the subject word in sentence S_d, S is the abbreviation of the subject word; $Wei_T(S_d)$ is expressed as the type importance of sentence S_d; ''T'' is the abbreviation of the word Type.

a: SENTENCES SUBJECT WEIGHT CALCULATION

In terms of subject word selection, many methods pay too much attention to the frequency of a certain word, that is, the higher the word frequency of a word in the text, the more critical the word maybe. However, we found that this relationship between word frequency and importance is not satisfied in all cases. For example, in a source code fragment describing ''SET DIALOG'', the word ''SET'' may appear very frequently, but it only indicates an action and cannot be used as the subject vocabulary of the code. Therefore, the frequency of occurrence of a word cannot be used as the only criterion for measuring the importance of the word. To solve the above problems, we calculate the importance of subject words based on the subject word decision strategy in the internal processing module. Specifically, when a vocabulary is determined as a subject word, the higher the frequency of its occurrence, the higher the importance of the vocabulary. Based on the above analysis, the calculation method of the weight value of a single subject word k is as follows:

$$Wei(k) = T(k) / \sum_{i=1}^{n} T(k_i), \tag{2}$$

where $T(k)$ represents the number of occurrences of the subject word k; $T(k_i)$ represents all subject words in the code statement; n is the total number of occurrences of all subject words in the code fragment. When the value of n is 1, it means that the code statement contains only one subject word k.

The calculation method of the subject words importance of sentence S_d is further derived as follows:

$$Wei_S(S_d) = \sum_{j=1}^{n} Wei(T(k_j) : k_j \in S_d), \tag{3}$$

where $T(k_j)$ represents all the subject words in sentence S_d; n is the total number of occurrences of all subject words in sentence S_d.

b: SENTENCES TYPE WEIGHT CALCULATION

According to the priority of the sentence types, the weight value of each type of sentence is a special sentence, direct sentence and indirect sentence in descending order. If the parameter value of the sentence type is set too high, it will weaken the influence of the subject word on the sentence weight, and if it is too low, it will increase the influence. According to Eq. (3), the subject word weight of a sentence is the sum of the weights of individual subject words in the sentence.

Based on the above analysis, each sentence needs to be processed hierarchically. Given two sentences S_1 and S_2 arbitrarily from the sentence set $S_n(n \geq 1)$, the hierarchical processing process is described as follows:

If sentences S_1 and S_2 belong to the same type, both sentences have the same weight. In this case, the weight values of the two sentences are determined by the weight of the sentence subject words. In the calculation of the weight value of the same type sentences, the parameter value needs to be set. According to Eq. (2), the weight of a single subject word is within the interval [0, 1]. If a sentence only contains one subject word, the difference between the weights of different sentence types is set to 1.0. Accordingly, the hierarchical statement type weight is calculated as follows:

$$Wei_T(S_d) = \begin{cases} 2.1, & special\ statement \\ 1.1, & direct\ statement \\ 0.1, & indirect\ statement \end{cases}$$

If sentences S_1 and S_2 are not of the same type. The weight values of the two sentences types are different, and the weight values of the two sentences is determined by the sentence type weight value.

c: SENTENCES SORTING ALGORITHM

In the foregoing work, we introduce that the weight value of the entire sentence is determined by the sentence type. Therefore, in the sentence sorting process, the types of sentences S_1 and S_2 are first judged. Two sentences S_1 and S_2 are arbitrarily given from the sentence set $S_n(n \geq 1)$. Specifically, if sentence S_1 and sentence S_2 are of the same type, calculate the weight of sentence subject words, When the weight of the subject word of sentence S_1 is greater than that of sentence S_2, the output order is: $S_1 \rightarrow S_2$, which means the output sentence S_1 is preferred; otherwise, the output order is: $S_2 \rightarrow S_1$, which means the output sentence S_2 is preferred. When the subject word weight of sentence S_1 is equal to the subject word weight of sentence S_2, the output order is: $S_1 \Longleftrightarrow S_2$, which means that the sentence is sorted according to the input order of the sentence. If sentence S_1 and sentence S_2 are not of the same type, the weight of the sentence subject word is ignored and the sentence type is considered. When the priority of sentence type of sentence S_1 is greater than sentence S_2, the output order is: $S_1 \rightarrow S_2$; otherwise, the output order is: $S_2 \rightarrow S_1$. The detailed process is shown in Table 1.

2) SENTENCES SIMILARITY CALCULATION

a: COSINE SIMILARITY CALCULATION

In terms of similarity calculation, we evaluate the degree of similarity between sentences by calculating the cosine of the angle between corresponding vectors. We map the sentence S_n (such as S_1, S_2) to a m-dimensional space vector, and calculates the similarity of the sentence in the form of the space vector. When the vector cosine value changes within the interval $[-1, 1]$, the similarity of sentences can be displayed intuitively.

Suppose the sentence set $S_n = \{S_1, S_2, \ldots, S_n\}$ corresponds to the vector set $V_n = \{V_1, V_2, \ldots, V_n\}$, If the angle between vectors V_i and V_j is θ, then the plane vector cosine formula $cos\theta = \frac{V_i \cdot V_j}{|V_i| \times |V_j|}$ is used to further derive the

TABLE 1. Sentence ordering algorithm process of our method.

Input sentence	Impact factor (type, subject)		Output order
	Sentence type	Subject weight	
$S_1, S_2(S_1, S_2 \in S_n, n \geq 1)$	if $S_1 > S_2$		then $S_1 \rightarrow S_2$
	if $S_2 > S_1$		then $S_2 \rightarrow S_1$
	if $S_1 = S_2$	if ①$S_1 > S_2$ if ② $S_1 < S_2$	then ①$S_1 \rightarrow S_2$ then ② $S_2 \rightarrow S_1$
	if $S_1 = S_2$	if $S_1 = S_2$	then $S_1 \Longleftrightarrow S_2$

similarity formula in the case of multidimensional vectors:

$$similarity(v_i, v_j) = \frac{\sum_{\substack{i=1 \\ j=1}}^{m}(V_i \cdot V_j)}{\sqrt{\sum_{i=1}^{m}(V_i)^2} \times \sqrt{\sum_{j=1}^{m}(V_j)^2}} \quad (4)$$

b: SENTENCES OPTIMIZATION ALGORITHM

Given any two sentences $S_i(1 \leq i \leq n)$ and $S_j(1 \leq j \leq n)$ from the set of sentences $S_n(n \geq 1)$, when calculating the similarity of sentences S_i and S_j, a threshold interval is set according to the cosine similarity calculation principle. When the similarity value changes in this interval, if the $similarity(S_i, S_j)$ value is 1, it is judged that the two sentences are completely duplicated, and one of these two sentences is randomly deleted; If the $similarity(S_i, S_j)$ value is 0, it is judged that the two sentences are completely non-repetitive. At this point, the results are output in the order in which the sentences are entered. If $0<similarity(S_i, S_j)<1$, it is judged that the two sentences have duplicate information, and the redundant information is optimized and deleted for sentences S_i and S_j. The specific process is as follows.

The input sentences S_i and S_j of any length can be regarded as vocabulary sets composed of n_1 and n_2 words $S_i = \{w_{i1}, w_{i2}, w_{i3}, \ldots, w_{in_1}\}$, $S_j = \{w_{j1}, w_{j2}, w_{j3}, \ldots, w_{jn_2}\}$, where $w_{i1}, w_{i2}, w_{i3}, \ldots, w_{in_1}$ and $w_{j1}, w_{j2}, w_{j3}, \ldots, w_{jn_2}$, respectively represent the vocabulary constituting sentences S_i and S_j. The sentence optimization operation uses double inner and outer loops to traverse the vocabulary sets in sentences S_i and S_j. If the current traversal vocabulary $w_i(w_i \in S_i)$ and $w_j(w_j \in S_j)$ are repeated in the inner loop, keep the format of sentence S_i where vocabulary w_i is located and delete vocabulary w_j in sentence S_j until the vocabulary between the two sentences is all traversed and compared; If the current traversal vocabulary $w_i(w_i \in S_i)$ and $w_j(w_j \in S_j)$ are not repeated in the inner loop, skip the current inner loop.

The pseudo code of the sentence optimization algorithm described in this section is shown in Algorithm 1.

In Algorithm 1, $Summary(S_i \rightarrow S_j)$ represents a natural language description composed of sentences S_i and S_j in the initial form; $Summary(S_i/S_j)$ represents a natural language description composed of S_i after the sentence S_j is deleted; $Summary(S_i, S_j)$ represents a natural language description composed of sentences S_i and S_j after the operation of optimizing and deleting redundant information.

Algorithm 1 Flow Chart of Sentence Optimization Algorithm

Input: $S_i, S_j \in S_n(n \geq 1)$.
Output: *Summary*.

```
Algorithm Process:
if Similarity(S_i, S_j) = 0 then
    Output:Summary(S_i → S_j);
if Similarity(S_i, S_j) = 1 then
    Output:Summary(S_i/S_j);
if 0<Similarity(S_i, S_j)<1 then
    Output:Summary(S_i, S_j).
```

3) ALGORITHM

According to the relationship between the sentences, the sorting algorithm based on sentence weight value calculation and the optimization algorithm based on cosine similarity calculation in the external processing module are summarized as follows.

Step 1 Enter the sentence of the source code statement.
Step 2 Calculate the importance of subject words in each sentence according to Eq. (1).
Step 3 Calculate the importance of each line of sentences based on subject words and sentence types according to Eq. (2) and rank the sentences in order of importance.
Step 4 Calculate the similarity between sentences according to Eq. (4) and execute Algorithm 1.

IV. EXPERIMENTAL SETTING

A. RESEARCH QUESTIONS

To evaluate the performance of this research method. This paper conducts experiments in terms of accuracy, simplicity, and smoothness to answer the following three research questions:

TABLE 2. Statistics of the datasets.

	Hmkcode	Algorithms
Files	54	156
Code Statement	2961	14303
Vocabulary	518943	1744766

1) **RQ1 (Accuracy Evaluation)**: Can the program description generated by our method accurately describe the functionality of the source code?
2) **RQ2 (Simplicity Evaluation)**: Compared with the traditional method, does the program description generated by our method use concise sentences to express the meaning of the source code?
3) **RQ3 (Smoothness Evaluation)**: Is the program description generated by our method semantically fluent and in line with the programmer's reading habits? Is it easier than viewing the source code?

B. DATASETS

We collected Github's open source project Hmkcode[1] and algorithm data[2] as experimental datasets. The statistics of the number of files, code statements, and vocabulary included in these projects are listed in Table 2.

C. METRICS

In the experiment, we use three evaluation indicators, namely accuracy, simplicity and smoothness, which are defined as follows:

Accuracy Rate. The accuracy rate is the ratio of generated program description subject words to actual program code subject words. For the program description T of a code fragment, the formula for calculating accuracy is defined as follows:

$$Accuracy = \frac{\sum_1^m K_m}{\sum_1^n C_n}, \tag{5}$$

where $K_m = \{K_1, K_2, \ldots, K_m\}$ is the set of program description subject words, and $C_n = \{C_1, C_2, \ldots, C_n\}$ is the set of program code subject words.

Simplicity. The standard of conciseness can be measured by the total number of words that constitute the description of the program. In converting the program code to its final program description, under the premise of retaining important information, the more redundant information is deleted, the more concise the program description is. Suppose that the program description sentence T is generated from the program code statement set $P = \{P_1, P_2, \ldots, P_n\}$, the simplicity of the sentence T is calculated by:

$$Simplicity = \frac{\sum_1^n P_n - \sum W_T}{\sum_1^n P_n}, \tag{6}$$

where $\sum_1^n P_n$ is the total vocabulary of the program code fragment, including repeated words; $\sum W_T$ is the number of words contained in the program description T.

[1]https://github.com/hmkcode/Java.
[2]https://github.com//TheAlgorithms/Java.

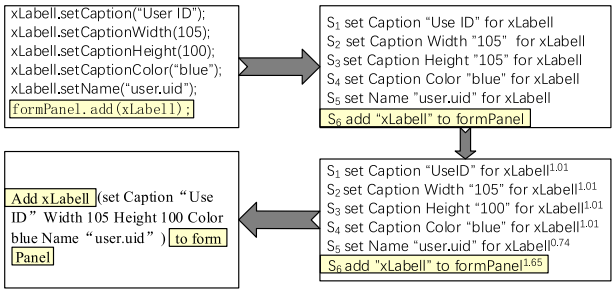


FIGURE 5. The overall workflow of our code summarization approach.

Smoothness. Smoothness refers to whether the program description is semantically fluent and whether it conforms to the programmer's reading habits. The smoothness evaluation standard adopts the manual evaluation method, which is described in detail in Section V-B.

V. EXPERIMENTAL RESULTS

A. ANSWER TO RQ1

To evaluate the generated descriptions' accuracy, we conduct the experiment by judging if the correct subject words can be generated. Compared with two automatic generation methods and two human groups, our method can extract the subject words more accurately.

The two automatic methods are the TextRank proposed by Mihalcea *et al.* [14], and the Suncode proposed by Msie *et al.* [34]. In addition to this, we compared the method with two groups of people: the first group consisted of six PhD students majoring in computer science at Shandong Normal University with more than 3 years of programming experience (referred as PhD Group); the second group consisted of six programmers working at Inspur Software Company and developing Java for more than 5 years (referred as Pro Group). In the manual extraction method, the two groups were asked to provide the corresponding descriptions of the code fragments with the following requirements: 1) No limit on the number of words in the description; 2) People in the same group are free to discuss; 3) People in different groups are not allowed to discuss with each other.

In this experiment, we used the sample program code in Fig. 5 as the experimental input, and then calculated the number of subject words contained in the generated program descriptions.

As shown in Table 3, the bold part is the three subject words. The experimental results show that the program descriptions generated by both the proposed method and Pro Group contain the entire vocabulary in the code lines, especially retaining all the subject words. The PhD Group omitted some parameter values but kept the subject words. In the program description generated by the TextRank, most the subject words are ignored, resulting in a relatively simple description with limited information about the program function. The Suncode can summarize the class and method names in the source code, and conduct information mining on the core running program. Although its generated result

TABLE 3. Comparison of keyword extraction results between various methods.

	Our method	PhD Group	Pro Group	TextRank	Suncode
xLabel1	Y	Y	Y	N	Y
Caption	Y	Y	Y	N	Y
user	Y	Y	Y	Y	N
ID	Y	Y	Y	N	N
Width	Y	Y	Y	N	Y
105	Y	N	Y	N	N
Height	Y	Y	Y	N	Y
100	Y	N	Y	N	N
Color	Y	Y	Y	N	Y
blue	Y	N	Y	N	N
Name	Y	Y	Y	N	Y
User.uid	Y	Y	Y	Y	N
formPanel	Y	Y	Y	Y	Y
	100%	76.9%	100%	23.1%	53.8%

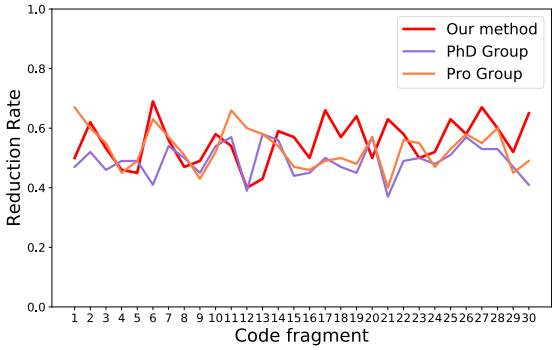


FIGURE 6. Comparison of reduction rate between our method and other methods.

retains all the subject words, some parameters are missing, affecting the ability of the description to express the detailed functions of the code.

For the other 30 codes in the dataset, we counted the number of subject words in the generated program descriptions. The results show that the natural language description generated by our method contains a total of 275 subject words, and the accuracy rate reaches 96%. Both the manual ways and the other two baseline accuracy rates are only over 70%. This experiment shows that the sorting algorithm has a high accuracy rate in retaining the subject words.

In conclusion, the above results verify that our method can extract subject words in code fragments more accurately than manual methods, Suncode and TextRank. As a result, our method's program descriptions can more accurately describe the functions of the code.

B. ANSWER TO RQ2

For the code fragment shown in Fig. 5, the simplicity of our approach is 40%. We also tested the simplicity of each method for generating results on the other 30 code fragments.

As shown in Fig. 6, the average vocabulary simplicity rates of our method, PhD group, and Pro Group are 55.4%, 49.0%, and 53.1%, respectively. This verifies that the proposed method can effectively improve the simplicity of the program description. Note that we did not conduct comparative experiments with other automatic methods in simplicity rates.This is because these methods generate only a single program description sentences, not paragraphs of program description, so they have no basis for comparison.

The experimental results show that the vocabulary simplicity rate of our method is highest, which verifies that our method can generate more concise descriptions of the code fragments.

C. ANSWER TO RQ3

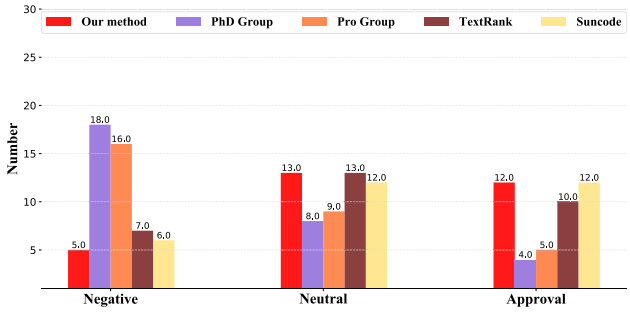
The evaluation of smoothness currently relies on human judgment. For this reason, we invited ten software engineers from Inspur as evaluators to evaluate the experimental results generated by our method, two manual ways and two baselines. We randomly select 30 pieces of source code from the dataset as experimental data.

1) EVALUATION PROCESS

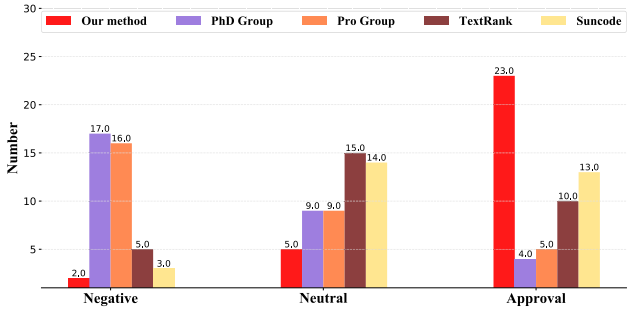
We divided the 10 evaluators equally into two groups, with one group providing only 30 pieces of source code and the other group providing both 30 pieces of source code and the corresponding comments (descriptions). At the same time, we set up a set of questionnaires for the evaluators to score the results of each method, including:

- How complete is the code summarization information?
- How well does the code summarization summarize the function of the code?
- How smooth is the code summarization?
- Does the code summarization conform to reading conventions?
- How well did the evaluators approve of the code summarization?

Evaluators are required to assign a score of 1 to 10 to each generated result for each evaluation question, and then use the average score of each problem as the final result. The higher the average score is, the higher the smoothness is. We divide the measurement results into Approval, Neutral, and Negative. Negative corresponds to 1∼3, indicating that the evaluator believes the comment is poor and cannot describe the meaning of the program code fragment at all or loses some vital information (such as the subject name); Neutral corresponds to 4∼7 points, indicating that the evaluator believes the comment is generally effective and



(a) Smoothness results of the first group of assessors.



(b) Smoothness results for the second group of assessors.

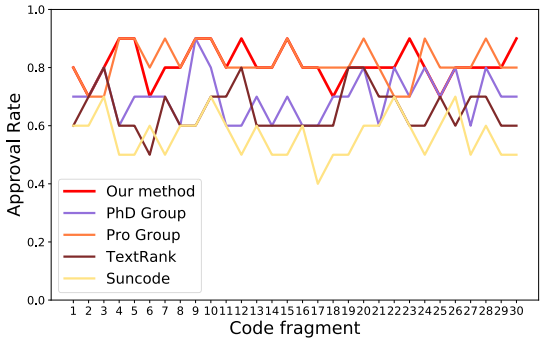
FIGURE 7. Comparison of smoothness between our method and other methods.

can basically describe the meaning of the source code, but the comment is too complicated and not concise enough; Approval corresponds to 8∼10 points, indicating that the evaluator believes that the comment is more consistent with the programmer's intent and can accurately describe the meaning of the code.

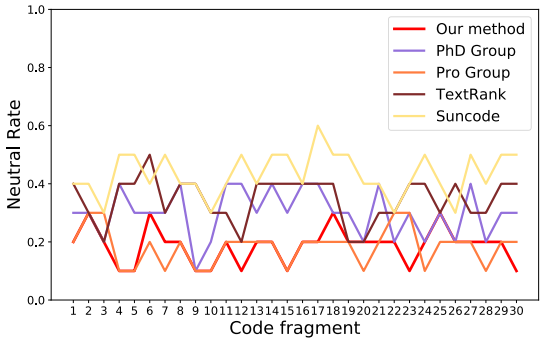
2) RESULTS

As shown in Fig. 7 (The horizontal coordinates indicate the ratings, and the vertical coordinates indicate the number of code fragments in each rating). We conducted two sets of manual evaluations for 30 source code fragments. For the first group of evaluators, the number of code fragments in the Approval and Neutral ratings of our methods is equal but much higher than those in the Negative. Both manual methods have a smaller number of Approval code fragments and a larger number of Negative code fragments. The TextRank method and the Suncode method obtain relative good Approval results but are lower than our method. For the second group of evaluators, they believe our method has the highest number of Approval code fragments. Meanwhile, the number of Negative code fragments is higher for both manual methods. The TextRank method and Suncode method are similar to the first set of results. From the two sets of manual evaluations, it can be seen that our method has the most significant advantage. In other words, our method has the highest smoothness.

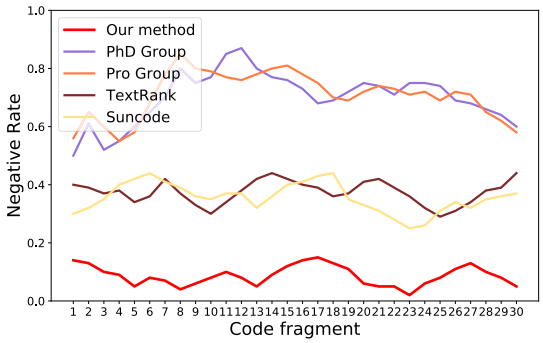
Longitudinally there is some variability in the assessment results due to the different code information given by the two groups of assessors. The first group of evaluators gave only 30 source code segments with no corresponding comments,



(a) Comparison of support rates between our method and other methods.



(b) Comparison of the neutral rate between our method and other methods.



(c) Comparison of the negative rate between our method and other methods.

FIGURE 8. Evaluation results of our method and other methods.

so it is more subjective. Therefore, there is a somewhat conservative attitude towards our method, and the number of code fragments of Neutral rating is higher. A similar phenomenon occurs with the TextRank method and the Suncode method. The second group of evaluators gave 30 source code segments and their corresponding comments, and the final score was derived by referring to the comments. From the second set of scores, we can see that the number of Neutral code fragments decreases, and the number of Approval code fragments increases. The program descriptions generated by our method have a high match with the comments and achieve better smoothness.

We also analyzed the scores of the two groups of raters together and calculated the percentage of approval, neutral, and negative for each method. As shown in Fig. 8, our method, PhD Group, Pro Group, TextRank and Suncode achieved Approval rates of 81.3%, 70.0%, 81.6%, 65.3% and 56.7%, respectively. The evaluators gave the least proportion of neutral opinions on the program description written by the Pro Group, and the rest are our method, the PhD Group and other methods in order. The evaluators have the lowest negative rate of the method in this paper, followed by the TextRank method and the Suncode method, and the manual ways have the highest negative rate. Experimental results show that the method in this paper optimizes program description based on cosine similarity, and can generate natural language description with high smoothness, which is more in line with human reading habits.

VI. THREATS TO VALIDITY

We have identified the following threats to validity.

A. INTERNAL VALIDITY

A threat to internal validity stems from the limitations of code mining tools. In this paper, we use SWUM and CamelCase to mine code features. Some important words may be lost during code mining, and although the loss rate is low, the amount of missing information may increase when applied to large projects. In the future, we will explore more effective code mining tools to extract code features.

B. EXTERNAL VALIDITY

Threats to external validity include the quality and representativeness of the datasets. We have collected only two Java datasets to verify the effectiveness of our method. We will continue to try multiple types of datasets to validate our approach. Another threat to external validity is code language limitation. Our approach is to experiment only with the Java language. Although the original design goals were not limited to the Java language, there is no direct experimental data to show that the method applies to other programming languages. In the future, we plan to generalize our approach to other programming languages. In addition, we invited several volunteers to write natural language descriptions by hand, and the sample code fragments in the comparison experiments were shorter, considering the time limit for reading the code manually. Therefore, there may exist errors when dealing with large projects.

VII. RELATED WORK

A. AUTOMATIC TEXT SUMMARIZATION

Automatic text summarization refers to the use of concise language to summarize the text's subject, which is similar to the task of code summarization. Therefore, code summarization can learn from the studies of automatic text summarization [7], [44]. According to the different output types, automatic text summarization can be divided into knowledge-based abstract summarization method [6] and statistics-based extractive summarization method [8].

The abstract text summarization method aims to use natural language understanding technology to analyze the text and generate a summary based on the analysis result. Liu *et al.* [9] proposed to use abstract meaning representation (AMR)-based methods [10] to generate text summarization. AMR adopted a directed acyclic graph with root nodes to express the semantic structure of sentences. It used JAMR to analyze the sentence structure and express each sentence as an AMR graph. The authors merged the AMR graphs of all sentences into a total AMR graph to extract the most semantic information. Finally, integer linear programming (ILP) and support vector machine (SVM) was used to constrain the whole AMR graph and learned the coefficients, respectively, to obtain the optimal subgraph that can be reduced to a summarization. Although the AMR-based graph model can effectively represent the semantic structure of the text, the current research on AMR is only at the initial stage, and more efficient graph decoding algorithms need to be explored in the future. Rush *et al.* [11] used the Encoder-Decoder framework based on deep learning and the attention mechanism to perform sentence-level text summarization. They represented each word in the sentence as a vector and inputted it to the encoder for semantic encoding to obtain a fixed-dimensional semantic vector. Then, a decoder was used to decode the semantic vector to get the text summarization output. To solve the problem of processing long sentences caused by the fixed vector dimension, the authors introduced an attention mechanism [12], which assigned different weights to the word vectors that were input into the decoder to determine its influence on the current output. In this way, its performance is improved. However, the models mentioned above are based on deep learning, which relies on the support of a large amount of training data. Besides, the input of the model is a sequence of words, lacking consideration of the semantic structure of the text. To summarize, the abstract text summarization approach has the advantages of flexibility and readability and generates higher quality summarization [36]. However, the generalizability of such methods has yet to be improved as the implementation of the models requires a large knowledge base and training data support.

The extractive text summarization method selects vital sentences from the text to generate a summarization. The earliest method of extracting text summarization was based on word frequency statistics [13]. This method divided the text into word combinations and calculated the frequency of words in the text. The authors used words with high frequency as abstract words for the text. Inspired by PageRank, Rada *et al.* [14] proposed a graph-based text processing algorithm (TextRank) and applied it to generate text summarization. This work first divided the text into a collection of individual sentences and vectorized the sentences. Then, it calculated the similarity between sentence vectors and stored the similarity score in the form of a matrix. The authors converted the matrix into a graph with sentences as nodes and similarity scores as edges' weights. After that, they used the TextRank algorithm to obtain the sentence with the highest

ranking, which was finally used as the summarization of the text [45]. The advantage of the extractive method is that 1) it can retain the original text information to the greatest extent; 2) it is faster; and 3) the summarization's subject is not easy to shift. The disadvantage is that it merely selects information in the text and is not flexible enough. Existing research shows that the extractive text summarization methods are generally better than the abstract text summarization methods [37]. However, with the continuous development of deep learning today, abstract text summarization is developing rapidly, and good results have also been obtained on specific datasets.

Techniques in text summarization can also be applied to code summarization. For example, CamelCase aims to combine two or more words into one word. From a formal point of view, compound words are uneven and hump-shaped. For instance, ''PlayStation'' is the product name of the SONY company, and the first letters of the words ''Play'' and ''Station'' are both capitalized. The spelling does not conform to English writing standards, but it is widely used in programming languages. In this paper, we use CamelCase to process identifiers. For example, we use CamelCase to split the identifier ''ABYButton234'' to extract ''Button.'' ''ABY'' and ''234'' are automatically ignored because ''ABY'' and ''234'' may have no practical meaning in this identifier. This is because using the first letter of a word to separate identifiers is more concise and more consistent with human reading habits than separating words by other characters. For example, ''DataBase'' seems easier to read than ''DATABase'' [39].

SWUM refers to the software word usage model, which captures source code text and structure information through a three-layer structure of $SWUM_{word}$, $SWUM_{program}$, and $SWUM_{core}$. In this paper, we use it to split code identifiers to obtain method information, such as actions, topics, and some parameters [38]. Specifically, given a calling method, we analyze the calling method name (signature). If the method name contains verb components (such as $Contents.add(\text{``}list\ 1''))$, SWUM directly expresses the verb ''add'' as the operation for the method call. However, not all codes contain verb components. Thus, SWUM infers words that may represent actions based on the structure of the method name (such as the position of the word in the method name and form type). For example, in *Image.savedImage*( ), SWUM uses ''save'' to represent the operation of the method.

B. AUTOMATIC CODE SUMMARIZATION

Automatic code summarization is essentially the implementation of a mapping relationship between code and natural language [42]. Maskeri *et al.* [17] proposed using information retrieval techniques for code analysis and proposed a semantic clustering-based strategy to reveal code intent. In another work, Kuhn *et al.* [15] proposed a three-level hierarchical Bayesian probabilistic topic model based on LDA to model and analyze code summarization. Specifically, this work first extracted the semantic topic information and user manual information presented in the code using LDA. Then, it calculated the correlation between the software code and the software description document. Finally, it extracted the text describing the code functionality using the LexRank algorithm. Baldi *et al.* [16] verified the effectiveness of using LDA technology to mine code topics through experiments. In general, LDA techniques are effective for the abstraction of code topics, but such techniques are not sufficiently complete for mining code features to reflect code functionality comprehensively. Therefore, there is still much room for the development of LDA-based code topic techniques.

The development of deep learning has opened up another area of code summarization methods [35]. Convolutional Neural Network (CNN) contains convolution operations that consist of three parts: convolution, activation, and pooling. Translation invariance is the most crucial feature of CNN, and the effect of feature aggregation can be obtained through convolution operation [18]. Allamanis *et al.* [19] used VSM (Vector Space Model) and CNN to study code summarization. The researchers treated the code as an image, and each code element could correspond to a point in the picture. Therefore, CNN could extract the features of the code elements like a convolutional image to capture the high-level semantic structure of the code. The author conducted experiments with 10 Java projects and achieved good results. In addition to CNN, Recurrent Neural Network (RNN) [20] has achieved great success in natural language processing (NLP), especially Seq2seq uses RNN as the basic network and is widely used in machine translation and other tasks [21]. Iyer *et al.* [5] applied Seq2seq to the field of code summarization and achieved significant results. They segmented the codes, vectorized them using word embedding techniques, and inputted them to the encoder for encoding, where the codes were converted to hidden vectors. The decoder accepted the hidden vectors as input and assigned different weights to them. Finally, the required natural language description was output by the decoder. In general, code summarization methods based on deep learning use unsupervised or semi-supervised learning. It summarizes code features by training a large amount of data, which affects the model's generalization ability. Therefore, it is a challenge to obtain high-quality labeled training data for code summarization [40].

VIII. CONCLUSION

In this paper, we proposed a new method that can automatically generate source code natural language descriptions. This method is unique because it focuses on the overall statements in the entire code fragment, rather than merely processing a line of code. This method first classifies code statements according to the division rules and performs priority processing. Then, it uses identifier processing tools to mine code characteristics and generates program descriptions for each line of code statements based on natural language templates. Finally, it adopts cosine similarity measures to sort and optimize the program descriptions of code statements to generate natural language descriptions of the source code fragment. We conducted comparative experiments and manual evaluations on two datasets. The experimental results show that the

proposed method outperforms other competitors significantly in terms of accuracy, simplicity, and smoothness.

REFERENCES

[1] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proc. 17th Work. Conf. Reverse Eng.*, Oct. 2010, pp. 35–44.

[2] B. P. Eddy, J. A. Robinson, N. A. Kraft, and J. C. Carver, "Evaluating source code summarization techniques: Replication and expansion," in *Proc. 21st Int. Conf. Program Comprehension (ICPC)*, May 2013, pp. 13–22.

[3] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *Proc. 21st Int. Conf. Program Comprehension (ICPC)*, May 2013, pp. 23–32.

[4] A. Tuan Nguyen and T. N. Nguyen, "Automatic categorization with deep neural network for open-source java projects," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion (ICSE-C)*, May 2017, pp. 164–166.

[5] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics (Long Papers)*, vol. 1, 2016, pp. 2073–2083.

[6] S. Gehrmann, Y. Deng, and A. M. Rush, "Bottom-up abstractive summarization," 2018, *arXiv:1808.10792*. [Online]. Available: http://arxiv.org/abs/1808.10792

[7] M. Gambhir and V. Gupta, "Recent automatic text summarization techniques: A survey," *Artif. Intell. Rev.*, vol. 47, no. 1, pp. 1–66, Jan. 2017.

[8] L. Luo, X. Ao, Y. Song, F. Pan, M. Yang, and Q. He, "Reading like HER: Human reading inspired extractive summarization," in *Proc. Conf. Empirical Methods Natural Lang. Process. 9th Int. Joint Conf. Natural Lang. Process. (EMNLP-IJCNLP)*, 2019, pp. 3024–3034.

[9] F. Liu, J. Flanigan, S. Thomson, N. Sadeh, and N. A. Smith, "Toward abstractive summarization using semantic representations," 2018, *arXiv:1805.10399*. [Online]. Available: http://arxiv.org/abs/1805.10399

[10] L. Banarescu, C. Bonial, S. Cai, M. Georgescu, K. Griffitt, U. Hermjakob, K. Knight, P. Koehn, M. Palmer, and N. Schneider, "Abstract meaning representation for sembanking," in *Proc. 7th Linguistic Annotation Workshop Interoperability Discourse*, 2013, pp. 178–186.

[11] A. M. Rush, S. Chopra, and J. Weston, "A neural attention model for abstractive sentence summarization," 2015, *arXiv:1509.00685*. [Online]. Available: http://arxiv.org/abs/1509.00685

[12] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2014, *arXiv:1409.0473*. [Online]. Available: http://arxiv.org/abs/1409.0473

[13] O. Tas and F. Kiyani, "A survey automatic text summarization," *Pressacademia*, vol. 5, no. 1, pp. 205–213, Jun. 2017.

[14] R. Mihalcea and P. Tarau, "Textrank: Bringing order into text," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2004, pp. 404–411.

[15] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic clustering: Identifying topics in source code," *Inf. Softw. Technol.*, vol. 49, no. 3, pp. 230–243, Mar. 2007.

[16] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya, "A theory of aspects as latent topics," *ACM SIGPLAN Notices*, vol. 43, no. 10, pp. 543–562, Oct. 2008.

[17] G. Maskeri, S. Sarkar, and K. Heafield, "Mining business topics in source code using latent Dirichlet allocation," in *Proc. 1st Conf. India Softw. Eng. Conf. ISEC*, 2008, pp. 113–120.

[18] I. Hadji and R. P. Wildes, "What do we understand about convolutional networks?" 2018, *arXiv:1803.08834*. [Online]. Available: http://arxiv.org/abs/1803.08834

[19] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 2091–2100.

[20] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," 2014, *arXiv:1409.2329*. [Online]. Available: http://arxiv.org/abs/1409.2329

[21] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2014, pp. 3104–3112.

[22] P. W. McBurney and C. Mcmillan, "Automatic documentation generation via source code summarization of method context," in *Proc. 22nd Int. Conf. Program Comprehension - ICPC*, 2014, pp. 279–290.

[23] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred API knowledge," in *Proc. 27th Int. Joint Conf. Artif. Intell.*, Jul. 2018, pp. 1–9.

[24] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization," 2018, *arXiv:1811.01824*. [Online]. Available: http://arxiv.org/abs/1811.01824

[25] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proc. 26th Conf. Program Comprehension*, May 2018, p. 200.

[26] P. Malhotra, L. Vig, G. Shroff, and P. Agarwal, "Long short term memory networks for anomaly detection in time series," in *Proc. ESANN*, vol. 89, 2015, pp. 89–94.

[27] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," 2015, *arXiv:1508.04025*. [Online]. Available: http://arxiv.org/abs/1508.04025

[28] P. W. McBurney and C. Mcmillan, "An empirical study of the textual similarity between source code and source code summaries," *Empirical Softw. Eng.*, vol. 21, no. 1, pp. 17–42, Feb. 2016.

[29] W. Li, Y. Cao, J. Zhao, Y. Zou, and B. Xie, "Toward summary extraction method for functional topic," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. Companion (QRS-C)*, Jul. 2017, pp. 16–23.

[30] R. Wang, H. Zhang, G. Lu, L. Lyu, and C. Lyu, "Fret: Functional reinforced transformer with BERT for code summarization," *IEEE Access*, vol. 8, pp. 135591–135604, 2020.

[31] P. W. McBurney and C. Mcmillan, "Automatic source code summarization of context for java methods," *IEEE Trans. Softw. Eng.*, vol. 42, no. 2, pp. 103–119, Feb. 2016.

[32] C. babu K, K. C., and S. N, "Entity based source code summarization (EBSCS)," in *Proc. 3rd Int. Conf. Adv. Comput. Commun. Syst. (ICACCS)*, Jan. 2016, pp. 1–5.

[33] X. Wang, L. Pollock, and K. Vijay-Shanker, "Automatically generating natural language descriptions for object-related statement sequences," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2017, pp. 205–216.

[34] R. Al-Msie'deen and A. H. Blasi, "Supporting software documentation with source code summarization," 2018, *arXiv:1901.01186*. [Online]. Available: http://arxiv.org/abs/1901.01186

[35] Z. Liu, X. Xia, C. Treude, D. Lo, and S. Li, "Automatic generation of pull request descriptions," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 176–188.

[36] J. Cheng and M. Lapata, "Neural summarization by extracting sentences and words," 2016, *arXiv:1603.07252*. [Online]. Available: http://arxiv.org/abs/1603.07252

[37] M. Zhong, P. Liu, Y. Chen, D. Wang, X. Qiu, and X. Huang, "Extractive summarization as text matching," 2020, *arXiv:2004.08795*. [Online]. Available: http://arxiv.org/abs/2004.08795

[38] M. Harman, N. Gold, R. Hierons, and D. Binkley, "Code extraction algorithms which unify slicing and concept assignment," in *Proc. 9th Work. Conf. Reverse Eng.*, Oct. 2000, pp. 11–20.

[39] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in *Proc. 6th IEEE Int. Work. Conf. Mining Softw. Repositories*, May 2009, pp. 71–80.

[40] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, Sep. 2018, pp. 397–407.

[41] Y. Yang, X. Chen, and J. Sun, "Improve language modeling for code completion through learning general token repetition of source code with optimized memory," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 29, no. 11n12, pp. 1801–1818, Nov. 2019.

[42] R. Kadar, S. M. Syed-Mohamad, and N. A. Rashid, "Semantic-based extraction approach for generating source code summary towards program comprehension," in *Proc. 9th Malaysian Softw. Eng. Conf. (MySEC)*, Dec. 2015, pp. 129–134.

[43] S. Rai, T. Gaikwad, S. Jain, and A. Gupta, "Method level text summarization for java code using nano-patterns," in *Proc. 24th Asia–Pacific Softw. Eng. Conf. (APSEC)*, Dec. 2017, pp. 199–208.

[44] X. Wang, L. Pollock, and K. Vijay-Shanker, "Developing a model of loop actions by mining loop characteristics from a large code corpus," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2015, pp. 51–60.

[45] H. Niu, I. Keivanloo, and Y. Zou, "Learning to rank code examples for code search engines," *Empirical Softw. Eng.*, vol. 22, no. 1, pp. 259–291, Feb. 2017.

XUEJIAN GAO is currently pursuing the master's degree with the School of Information Science and Engineering, Shandong Normal University, Jinan, China, under the guidance of C. Lyu. His research interests include natural language processing, code analysis, and artificial intelligence.

XUE JIANG is currently pursuing the bachelor's degree with the School of information Science and Engineering, Shandong Normal University, Jinan, China, under the guidance of C. Lyu. Her research interests include program comprehension, automatic program summarization, and component based software development.

QIONG WU is currently pursuing the bachelor's degree with the School of information Science and Engineering, Shandong Normal University, Jinan, China, under the guidance of C. Lyu. Her research interests include code analysis and artificial intelligence.

XIAO WANG is currently pursuing the master's degree with the School of Information Science and Engineering, Shandong Normal University, Jinan, China, under the guidance of C. Lyu. Her research interests include natural language processing, code analysis, and artificial intelligence.

LEI LYU received the Ph.D. degree in computer application technology from the University of Chinese Academy of Science, in 2013. He is currently an Associate Professor with the School of Information Science and Engineering, Shandong Normal University, Jinan, China. His current research interests include software engineering and programming languages, including automated software analysis and software evolution.

CHEN LYU received the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2015. He is currently an Associate Professor with the School of Information Science and Engineering, Shandong Normal University, Jinan, China. His research interests include program comprehension, software maintenance and evolution, and source code summarization.