FCSO: Source Code Summarization by Fusing Multiple Code Features and Ensuring Self-consistency Output

Donghua Zhang[1], Gang Lei[2], Jianmao Xiao[2,4](✉), Zhipeng Xu[1], Guodong Fan[3], Shizhan Chen[3], and Yuanlong Cao[2]

[1] School of Digital Industry, Jiangxi Normal University, Shangrao 334000, China
{Dong_hua,zp_xu}@jxnu.edu.cn

[2] School of Software, Jiangxi Normal University, Nanchang 330022, China
{leigang,jm_xiao,ylcao}@jxnu.edu.cn

[3] College of Intelligence and Computing, Tianjin University, Tianjin 300350, China
{Guodongfan,shizhan}@tju.edu.cn

[4] Jiangxi Provincial Engineering Research Center of Blockchain Data Security and Governance, Nanchang 330022, China

Abstract. Source code summarization is the process of generating a concise and generalized natural language summary from a given source code, which can facilitate software developers to comprehend and use the code better. Currently, most research on source code summarization generation focuses on either converting the source code into abstract syntax tree (AST) sequences or directly converting it into code segments and then feeding these representations into deep learning models. However, these single representation approaches ignore the semantic features of source code and destroy the structure of the abstract syntax tree, which affects the quality of the generated source code summarization. In this paper, we propose a novel source code summarization approach that fuses multiple code features into self-consistency output (FCSO). Our approach is based on a graph neural network encoder and a CodeBERT encoder with a self-attention mechanism. It extracts the sentence feature attention vector and the AST feature attention vector of the source code for feature fusion. Then, it inputs them into the Transformer decoder. Furthermore, to generate more accurate source code summaries, we adopt a new decoding strategy called self-consistency. It samples different inference paths, uses a penalty mechanism to calculate their similarity scores, and ultimately selects the most consistent answer. Our experimental results demonstrate that our proposed approach outperforms standard baseline approaches. On the Python dataset, the BLEU score, METEOR score, and ROUGE_L score increase by 11.13%, 9.12%, and 7.88%, respectively. These results show that our approach provides a promising direction for future research on source code summarization.

Keywords: Source code summarization · Code feature Fusion · Self-consistency · Transformer

1 Introduction

In the current era, the Internet is growing rapidly, expanding the size of software systems for companies. Unfortunately, every software development and maintenance operation requires developers to re-familiarize themselves with the source code, ultimately reducing operational efficiency [1]. To address this concern, high-quality source code summarization is essential as it enables programmers to swiftly comprehend and use the source code, thereby enhancing their work efficiency. High-quality code summaries improve software development and maintenance efficiency by providing accurate information on the function of the module's code [2], promoting rapid industry growth.

Research work in the area of Source Code Summarization generally falls into three categories: artificial templates, information retrieval, and deep learning models. The first approach, based on artificial templates for generating source code summaries, is the most traditional approach. Sridhara et al. [3] utilized the Software Word Usage Model (SWUM) to produce descriptive summaries of Java approaches, while Merono et al. [4] employed heuristics and natural language processing to generate Java code summarization. The second approach, based on information retrieval, involves extracting code semantic information by marking code feature information and applying information retrieval techniques to generate code summaries. Wong E et al. [5] proposed a probabilistic and statistical AutoComment model based on a large dataset and fed the mapping relationship of a vast amount of data into the AutoComment model to generate a code summary. However, the first two approaches have limitations due to their poor reusability and low accuracy of the generated code summaries. Researchers have gradually shifted to new models to carry out their work.

The most promising approach in current research on Source Code Summarization is the third category - deep learning-based models. In earlier studies, deep learning networks were commonly used for code summarization tasks. Iyer et al. [6] employed LSTM (long short-term memory network), a widely-used deep learning model, to build CODE-NN, an automatic code summary generation model capable of creating code summaries for SQL query statements. Hu et al. [7] proposed TL-CodeSum, which utilizes API information to enhance the quality of code summary generation. This approach uses two encoders to process API information and source code vocabulary information separately to improve the accuracy of generated summaries. To capture the code's semantic information more comprehensively, researchers have started focusing on generating code summaries by improving the abstract syntax tree of the code. Hu et al. [8] converted the source code into AST using an attention mechanism and presented the DeepCom approach, which inputs the AST sequence into the encoder for encoding. Wang et al. [9] fine-tuned the Transformer model and introduced the TranS approach, which leverages the Actor-Critic network to encode code vocabulary and indentation structure. The results indicate that this technique generates better summaries corresponding to source code fragments.

Although the research mentioned above has achieved the goal of code summary generation, there are still some limitations. First of all, the single use of code sequence or AST path in the task ignores the structural characteristics of AST, which will lose part of the code information. The second problem is that the previous decoder output uses the traditional Beam Search [10] strategy for path reasoning and finally selects the sequence with the highest score from all candidate sequences in the termination state as the output. However, the code summary candidate sequence obtained in this way only considers the candidate with the highest local score and cannot guarantee the global optimal solution, so there may be some repetitions or unreasonable situations in the output sequence.

To solve the above problems, we propose a source code summarization approach (FCSO) that fuses code features into self-consistent output to solve it. We found that CodeBERT [11], as a Transformer-based pre-training model, learned the semantic representation of code. It provides a robust feature extraction function, which can better capture the semantic information of the code. We have also seen that the graph neural network (GCN) can aggregate the information of AST neighbor nodes to help the model learn code structure information and context dependencies. Therefore, as an inspiration, we take whether they can integrate the semantic information and structural information of the code as a challenge to get the answer to the problem. For the first question, we use CodeBERT encoder and GCN encoder to extract sequence and AST features and then perform feature fusion so that the fused code feature self-attention vector can be input into the Transformer decoder to preserve the source code to the greatest extent-semantic and syntactic information to improve the accuracy of code summary generation. For the second question, since the current code summary generation task requires higher and higher accuracy and consistency, we abandoned the previous greedy random output strategy. To achieve this goal, we introduce a new decoding approach, Self-consistency in the Transformer decoder. After Beam search calculates the probability distribution of the final time step, it randomly samples the output inference path. Then, it obtains the most consistent answer by judging the similarity score.

In the following experiments, we used Java code and Python database as the corpus to train the model. After comparing the standard baseline method, we found that the BLEU and METEOR indicators have been improved accordingly, and the ablation experiment proved the feasibility of the FCSO approach. The main contributions of this paper are as follows:

- A source code summarization approach that fuses code features into self-consistency output (FCSO) is proposed. This approach can extract and fuse code sequence and AST features, improving code summarization generation quality.
- We break the traditional decoding strategy and add a new decoding strategy, Self-consistency. By defining a penalty mechanism, calculating the similarity score of multiple output sequences ensures the consistent output of the code summary and improves the generation accuracy.
- We compare the standard baseline approach in the experiment, and the BLEU score, METEOR score, and ROUGE_L score on the Python dataset increase by 11.13%, 9.12%, and 7.88%, respectively. It proves that our approach is effective and provides a good idea for future research.

2 Related Work

At present, the field of code summarization is mainly based on deep learning research, which is generated by improving AST traversal approaches, GCN embedding approaches, commonly used LSTM networks, encoder-decoder architectures, and Transformer models. Huo et al. [12] used the LSTM and CNN networks to learn a control flow graph (CFG) representation so that valuable information can be focused on a graphical representation. LeClair et al. [13] feed the AST as a sequence into the encoder to generate Java code annotations. Shi et al. [14] built a neural network encoder to recursively decompose the subtree of AST and then encode the processed data to generate a code summary. Hu et al. [8] proposed a structure-based traversal (SBT) approach by improving AST into a flattened sequence to solve the problem of the traditional AST sequence losing the global information of the code. LeClair et al. [15] used the SBT approach to conduct experiments and found that decoupling the code structure and code tags can better generate code summaries.

The above research mostly starts with improving code structure and AST structure. In recent years, researchers have gradually begun to use deep learning networks to solve the problem of code summarization generation. Due to the inability of traditional RNN models [16] or LSTM networks with attention mechanisms [17] to capture long-term dependency relationships, researchers have found that the Transformer model [18] utilizes self-attention mechanisms to solve this problem. Ahmad et al. [19] used a relatively encoded Transformer model to ensure the dependency of code information, and experiments found that using only an unimproved model resulted in much higher performance in code summarization generation than common deep learning networks such as RNN, indicating that using an encoder-decoder is a good approach.

In addition, some researchers have begun to start with code feature fusion, providing a new idea for code summary generation. The online learning of social performance (DeepWalk) proposed by Bryan et al. [20] is applied to CFG in a learning manner. Then it connects nodes through a convolutional neural network to achieve the goal of error positioning. Wang et al. [21] applied Self-consistency to the language model, allowing a complex reasoning problem to allow many different ways of thinking, and finally selected the only correct answer, which improved the reasoning ability of the thinking chain. Cheng et al. [22] proposed the GN-Transformer approach, which combines sequence and graph learning representations to improve the quality of code summarization generation. Wang et al. [23] constructed two encoders to fuse code-informed attention weights by learning mixture representations of codes. Similarly, Gao et al. [24] innovatively proposed a multi-modal and multi-scale approach to fuse the feature information of the code and input the code feature into the modified Transformer model decoder to improve the code summary generation performance. The above research mainly focuses on how to decompose each feature of the code. Their research more or less ignores the attention weight of the code feature or all randomly generates summary results in the traditional Beam Search method. The accuracy of the code generated is insufficient, so further research is needed.

3 The Architecture of Approach

The FCSO approach proposed in our paper consists of four main parts: data preprocessing, feature extraction, feature fusion, and self-consistency output. Firstly, the input source code is preprocessed and parsed into an abstract syntax tree and a token sequence. Next, these two parts of data are embedded into the GCN encoder and CodeBERT encoder we set up for feature extraction. The token feature vector and AST feature vector generated by the encoder are then input into the Transformer decoder for source code summarization. Finally, the self-consistency penalty mechanism is utilized in the decoder to calculate the similarity score, and the sequence with the highest similarity is retained by comparison to derive a consistent summary answer. The overall framework of the FCSO is illustrated in Fig. 1.

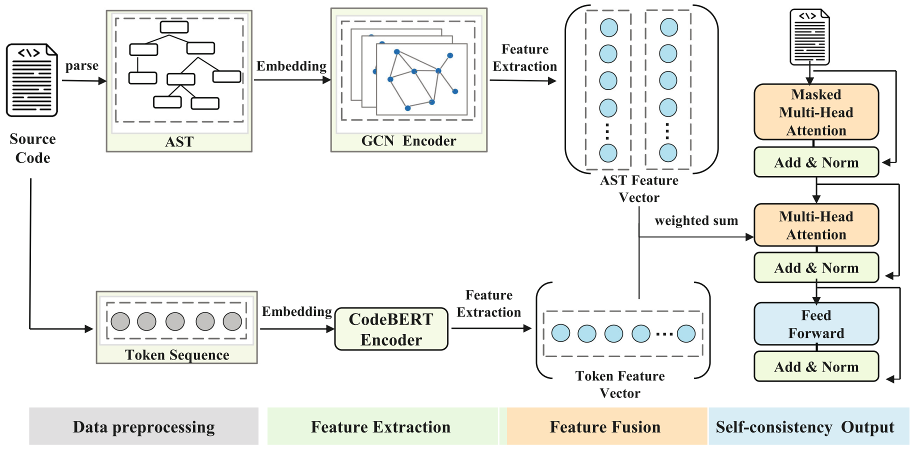


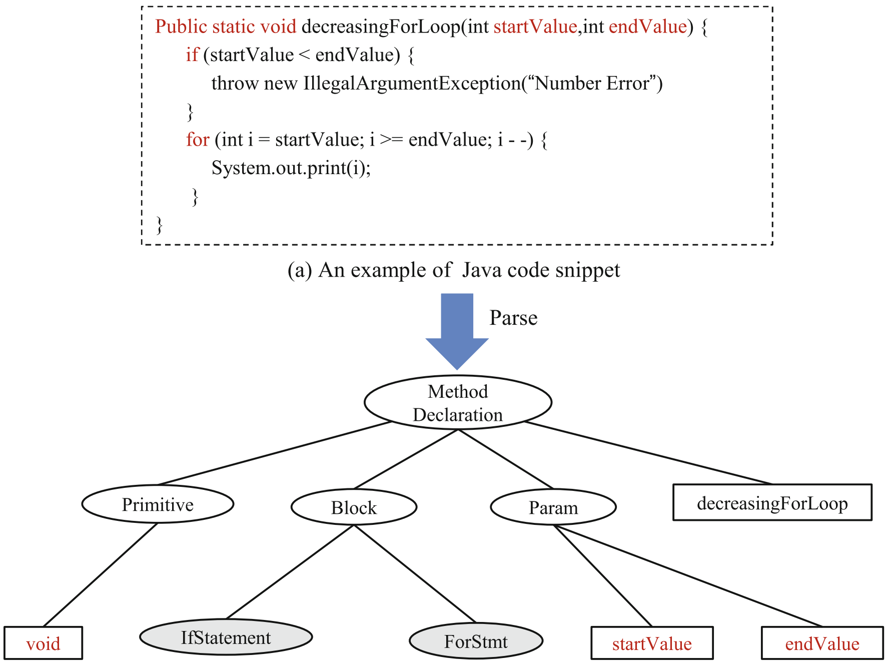
Fig. 1. The overall framework of our approach.

3.1 Data Preprocessing

In order to preserve the information of the source code more completely, we divide the code of the Java dataset and the Python dataset into two parts for data preprocessing. Part of it is processed as code sequence information, and part is converted into AST to save code structure information. In the former, we use the word segmentation toolkit to convert each piece of input code into a sequence and save it as an original suffix file as the input for the next stage. When the latter is converted into an abstract syntax tree since our experimental training is Java code and Python code, the javalang3 toolkit [23] is used to parse the Java code into AST, and the asttokens toolkit [24] is used to parse the Python code into AST.

At the same time, for the consistency of the data sets during training, we set the same length limit for the two data sets. This includes setting code tags, the average length of natural language, maximum node tree, maximum depth, and other parameters. For tag sequences that are less than or greater than the maximum length in the dataset, we pad and truncate them, respectively. In addition, we also divided the data set. Java data set and Python data set are set into the training set, validation set, and test set, which are divided into 6:2:2 and 8:1:1, respectively, to ensure that it has the same segmentation ratio as the standard baseline [4].

As shown in Fig. 2, it is an example of implementing the Java code of the decrement function and converting it to AST. In Fig. 2 (b), the gray color refers to nodes and branches and does not represent a complete tree structure. As can be seen from the figure, the AST branches from the MethodDeclaration node, and the type attribute indicates the return type of the method, which is void as a leaf node. The name attribute represents the method's name, which is also decreasingForLoop. The Param attribute represents the method's parameter list, which contains two FormalParameter nodes, representing the startValue and endValue parameters, respectively.



```
Public static void decreasingForLoop(int startValue,int endValue) {
    if (startValue < endValue) {
        throw new IllegalArgumentException("Number Error")
    }
    for (int i = startValue; i >= endValue; i - -) {
        System.out.print(i);
    }
}
```

(a) An example of Java code snippet

(b) Abstract syntax tree corresponding to Java code

Fig. 2. Example of converting Java code to AST.

3.2 Feature Extraction

After the data preprocessing in the previous step, to obtain the code's local and global information, we need to continue processing the obtained sequence and AST. Based on the Transformer model, we retained the original self-attention mechanism, improved the embedding method of the encoder, and constructed two encoders: GCN Encoder and CodeBERT Encoder.

GCN Encoder: In this encoder, we treat each node in the AST as a node in the graph data. Each node is a code block, and each edge connects two adjacent code blocks, and the normalized adjacency matrix is constructed using the connection relationship between them [25]. Initially, each node has an eigenvector and then uses the weighted average of the eigenvectors of neighboring nodes to update the eigenvector of the node. In each layer, GCN uses this method to update the feature vector of each node until the desired number of layers or feature convergence is reached [26]. Specifically, graph convolution is applied to each node, and two-layer graph convolution is used to capture AST structure information and dependencies between nodes. The computed neighbor node features are then combined with the initial features of the nodes to update the features of each node. Finally, this embedding vector is input into the encoder, and the AST feature attention vector is obtained through the self-attention mechanism. The formula for GCN propagation between layers and GCN calculation of node eigenvectors is as follows:

$$\widetilde{D}_{ii} = \sum j \widetilde{A}_{ij} \tag{1}$$

$$H^{(l+1)} = \sigma(\widetilde{D}^{-\frac{1}{2}} \widetilde{A} \widetilde{D}^{-\frac{1}{2}} H^{(l)} W^{(l)}) \tag{2}$$

$$h_i^{(l+1)} = \sigma(\sum_{j \in N(i)} \frac{1}{c_{ij}} W^{(l)} h_j^{(l)} + \theta^{(l)} h_i^{(l)}) \tag{3}$$

where the H is the characteristics of these nodes to form an $N \times D$ dimensional matrix, the relationship between each node will also form an $N \times N$ dimensional matrix A, also known as the adjacency matrix. In (1) and (2) formulas, each H is the feature of each layer, $\widetilde{A}$ is the sum of the A matrix and I identity matrix, $\widetilde{D}$ is the degree matrix of $\widetilde{A}$, and σ is the nonlinear activation function. Formula (3) can calculate the eigenvector of the GCN node. In the neighbor set $N(i)$ of node i, the eigenvector $h_i^{(l)}$ of node i in the l layer is multiplied and summed by the weight matrix $W^{(l)}$ in the l layer, and then the bias item $\theta^{(l)}$ is added. c_{ij} is a normalization factor, which is used to alleviate the problem of different degrees of different nodes.

CodeBERT Encoder: There are two parts: CodeBERT and Encoder. CodeBERT is used to extract code features, and the encoder continues to process feature vectors to generate self-attention vectors. We first tokenize the code and divide each word or symbol into tokens. For the CodeBERT model to distinguish different types of text sequences, it is also necessary to add the "[CLS]" tag at the beginning of the code and the "[SEP]" tag at the end of the sentence. Appropriate tags also need to be added at the start and end of the abstract. Then concatenate the code fragment and its abstract to form an input sequence and map the word-segmented sequence to BERT's vocabulary to obtain the digital ID representation of each token. Finally, input the preprocessed code dataset, load the trained BERT model, and extract the corresponding features. Among them, the feature vector of each code fragment can be obtained through the output of the last layer of BERT. Specifically, the vector corresponding to the last layer "[CLS]" tag can be used as the feature vector of the code fragment; similarly, the vector corresponding to the last layer "[SEP]" tag can be used as the feature vector of the summary [27]. The formula for extracting code features by the CodeBERT model is as follows:

$$h_{code} = [BERT([p_1, p_2, p_3, ..., p_n])] \tag{4}$$

where $[p_1, p_2, p_3, ..., p_n]$ is the token sequence obtained after the code sequence is mapped through the vocabulary, and the $BERT$ function converts it into the corresponding feature vector matrix.

Continuing to process the feature vectors generated by CodeBERT in the encoder can further improve the efficiency of the model. First of all, the massive text data used in CodeBERT pre-training has carried out unsupervised learning on the model so that it has a stronger semantic understanding ability; secondly, when extracting code features, CodeBERT can effectively capture the critical information in code fragments to improve the representation ability and expression efficiency of the encoder. Therefore, combining CodeBERT features with Transformer encoders can lead to better code generation results [28]. At the model structure level, both CodeBERT and Transformer use the same attention mechanism and multi-layer perceptron structure, and there are similar network structures and parameter settings between them, so they are compatible with each other. Therefore, we use the CodeBERT pre-trained model to take the output of the last layer as the input of the Transformer encoder and proceed to the next step. The sequence representation output in this encoder is the extracted sequence feature vector. The formula for the encoder using the self-attention mechanism is as follows:

$$Q_i = XW_i^q, K_i = XW_i^k, V_i = XW_i^v \tag{5}$$

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{6}$$

where Q_i(Query), K_i(Key), and V_i(Value) is obtained by multiplying the weights W_i^q, W_i^k, and W_i^v, respectively, and finally, the $softmax$ activation function is calculated.

3.3 Feature Fusion

After obtaining the sequence feature attention vector and AST feature attention vector of the source code in the previous two steps, we need to fuse these two vectors before entering the multi-head attention mechanism in the Transformer decoder. The specific way of fusion is described below. First, the similarity between two vectors is calculated by the dot product operation, and a scalar value can be obtained. Then, we use the Softmax function to normalize the similarity and convert it into an attention weight to indicate the importance of the two features in the fusion process. Finally, the attention weight is weighted and summed with the corresponding feature vector to obtain the fusion of the subsequent feature representation. This method determines the importance of features by calculating the similarity and normalization weights and then weights and sums the features according to the attention weights. This enables the adaptive fusion of different features to better capture the correlation and importance between features. At the same time, in the fusion process, we minimize the cross-entropy loss function and use the gradient descent algorithm and dropout to optimize the parameters in the model. The negative log-likelihood loss function is as follows:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{Z}\log[p(y_t^i)] \tag{7}$$

where X_i is a source code segment given in the formula, $Y_i = [y_1^i, y_2^i, y_3^i, ..., y_N^i]$ is the target digest segment prepared, N is the number of data points in the training decoder, and Z is the maximum length of the target digest given to training.

3.4 Self-consistency Output

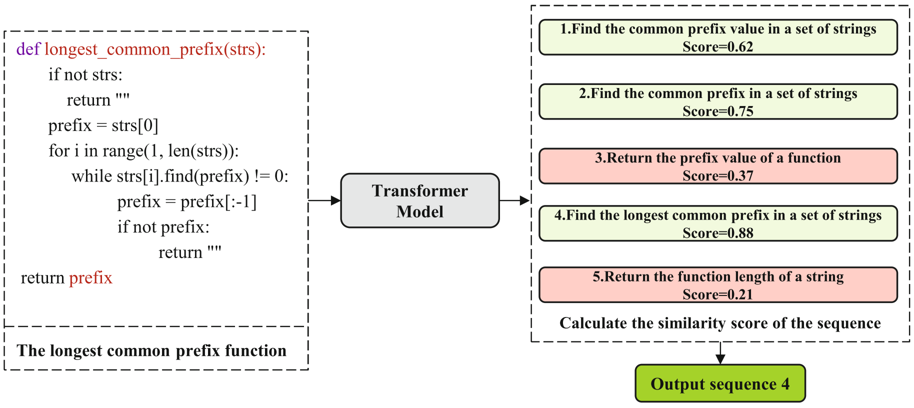
In the inference process on the decoder side, we use parameter weights after training and fusing multiple code features to load. Then Beam Search calculates the probability distribution of each time step and stores the updated candidate path ranking in a heap for subsequent summary output. For the self-consistency output, we use first samples these candidate inference paths, use the cosine similarity method to calculate the similarity score, compare the scores through the penalty mechanism of the Self-consistency scoring function, and finally retain the sequence with the highest similarity to obtain more accurate and consistent answers [21]. The calculation formula for cosine similarity and penalty mechanism is as follows:

$$cos(x, y) = \frac{x \cdot y}{||x|| \cdot ||y||} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}} \tag{8}$$

$$f(s) = g(s) - w\sum_{j=1}^{K} D_{i,j} \tag{9}$$

where D is the penalty score, K is the number of candidate sequences obtained by the Beam Search algorithm, and $D_{i,j}$ represents the penalty score between the i and j candidate sequences. Formula (8) uses the cosine similarity method to measure the similarity between n dimensional vectors x and y. At the same time, using this similarity formula, we define a new scoring function $f(s)$ for scoring each sequence and analyzing the optimal solution. Where $g(s)$ represents the initial score of the sequence s, and w is a non-negative weight coefficient used to control the degree of similarity penalty. The penalty mechanism is to subtract the penalty degree between the sequence s and other candidate sequences based on the original score $g(s)$ to save the sequences with higher similarity and filter out the sequences with lower similarity.

Figure 3 below is a Python code example of a function to find the longest common prefix of a string, showing the principle of self-consistency output. First, the source code is decoded by the trained Transformer model, and then the similarity calculation is performed on the candidate sequences in Beam Search. Because the value set by our beam size is 5, after processing the Self-consistency scoring function among the five random output sequences, sequence 4, with the highest similarity score, is reserved for output.



```
def longest_common_prefix(strs):
    if not strs:
      return ""
    prefix = strs[0]
    for i in range(1, len(strs)):
        while strs[i].find(prefix) != 0:
                prefix = prefix[:-1]
                if not prefix:
                        return ""
 return prefix
```

The longest common prefix function
Transformer Model
1.Find the common prefix value in a set of strings Score=0.62
2.Find the common prefix in a set of strings Score=0.75
3.Return the prefix value of a function Score=0.37
4.Find the longest common prefix in a set of strings Score=0.88
5.Return the function length of a string Score=0.21
Calculate the similarity score of the sequence
Output sequence 4

Fig. 3. Self-consistency output example.

4 Experiment

In the experiment, we first set up the database, evaluation indicators, model parameters, benchmark methods, etc., required for the experiment. Then, we performed model training to compare the evaluation indicators with the benchmark method. To demonstrate the effectiveness of our approach, we conduct qualitative ablation experiments. Finally, compare the summary generated by the benchmark method and the summary generated by our approach as an example.

4.1 Experiment Settings

Experimental Datasets: Our experiments are based on two databases well-recognized in source code summarization research, the Java database [7] and the Python database [29]. There are as many as 80,000 training sets, verification sets, test sets, and millions of Tokens which can fully train the parameters of the model. At the same time, we only use words with high frequency, and other words will be replaced by marks. For words beyond the maximum length, we perform a data truncation. Table 1 shows the number of training sets, AST nodes, and total number of tokens for Java and Python datasets.

Table 1. Statistical analysis of the Java and Python dataset.

Dataset	Java	Python
Train	69708	55538
Validation	8714	18505
Test	69708	18502
Unique nodes in ASTs	57478	101283
Unique tokens in code	66650	307596
Unique tokens in summary	46895	56189
Avg.node in AST	131.72	104.11
Avg.tokens in code	120.16	47.98
Avg.tokens in summary	17.73	9.48

Evaluation Metrics: To qualitatively compare the effect of the experimental generation, this paper adopts three indicators recognized in the field of machine translation and code summarization, BLEU [30], METEOR [31], and ROUGE_L [32].

The BLEU (Bilingual Evaluation Understudy) indicator is a standard machine translation indicator that measures the quality of translation by comparing the n-gram coincidence between the translation result and the reference translation.

The METEOR (Metric for Evaluation of Translation with Explicit ORdering) indicator combines various linguistics and machine learning techniques to obtain a comprehensive score by comparing the similarity in vocabulary, grammar, and semantics between the translation output and the reference translation.

The ROUGE_L (Recall-Oriented Understudy for Gisting Evaluation) indicator is often used to evaluate tasks such as text summarization and machine translation. It is more comprehensive, using the longest common subsequence (LCS) algorithm to measure the matching between the translation output and the reference translation. This enables a more comprehensive translation of the resulting code summary score.

Baselines: In order to prove that our research work is effective, the representative baseline models and methods in recent years are selected below. Metrics are generated by summarizing code replicating these methods and compared to our scores.

- CODE-NN [4]. The basic architecture of the model is LSTM with attention, which is an entirely driven generative model. The code is embedded through the encoder and decoder framework, resulting in the final C and SQL code digest.

- Code2Seq [33]. By inputting the processed AST path into the LSTM model as a sequence and outputting a fixed-length vector, the code summary is finally output under the calculation of the attention mechanism.
- Tree2Seq [34]. The model is based on end-to-end syntax, which extends from tree structure to sequence structure for input. At the same time, the model decoder also has a code summary corresponding to the output of the attention mechanism.
- DeepCom [6]. This method proposes a reversible AST traversal sequence method (SBT) for code comment generation, which provides a suitable method for future research under the same conditions as a reference.

Hyper-Parameters Setting: To better train the encoder and decoder architecture, we set appropriate parameters, as shown in Table 2. We set the size d_e in the embedding layer to 768 and the number of embeddings l_g in the GCN encoder to 300. Note that the number of $head_g$ is 8, and the number of L_g layers is 4. Similarly, the number of self-attention layers L_b in the CodeBERT encoder is 12, the d_k and d_v values are 64, and finally, the output size of the feed-forward network d_{ff} is 2048. At the same time, the length l_s in training in the decoder will be truncated to 100. During training, we set the embedding layer dropout to 0.2, the learning rate to 0.0001, and the batch size to 32.

Table 2. Hyper-Parameters Setting.

	Hyper-Parameters	Value
Embedding	d_e	768
GCN-Encoder	l_g	300
	h_g	768
	$head_g$	8
	L_g	4
CodeBERT-Encoder	l_b	400
	L_b	12
	$head_b$	12
	d_{model}	768
	d_k,d_v	64
	d_{ff}	2048
Decoder	l_s	100
	l_d	6
Training	dropout	0.2
	optimizer	Adam
	learning rate	0.0001
	batch size	32
Testing	beam size	5

4.2 Comparison Experiment and Ablation Experiment

After preparing the experimental setup, we conduct our experiments. First, we look for more excellent methods and models in recent years as the baseline and obtain the corresponding evaluation index data by reproducing their methods. Then, on the two data sets, Java code and Python code, we trained our model for 56 h before and after and compared the calculated evaluation index data with the baseline. Finally, we performed ablation experiments by removing each module to demonstrate that our various code feature attention vectors can help improve code summarization performance.

For comparative experiments, we compare the obtained baseline evaluation index data with the FCSO data, as shown in Table 3 below. The approach column in the table is each method, and the BLEU column, METEOR column, and ROUGE_L column are the corresponding method indicator percentages. It can be seen from the table that the standard baseline method is DeepCom [6], which has a BLEU score of 39.25%, a METEOR score, and a ROUGE_L score of 23.06% and 52.67% on the Java dataset. And our approach to FCSO is in bold in the table. It can be observed that with the baseline method DeepCom, the BLEU score, METEOR score, and ROUGE_L score on the Java dataset are increased by 5.45%, 3.80%, and 1.84%, respectively. On the Python dataset, the BLEU score, METEOR score, and ROUGE_L score increased by 11.13%, 9.12%, and 7.88%, respectively. From these data, it can be seen that our approach has a corresponding improvement compared with the baseline method on the Java dataset and the Python dataset, and it also shows that the FCSO approach we proposed is beneficial to the research work of source code summarization. At the same time, it can also be found that the evaluation index data on the Python dataset is lower than that of the corresponding Java dataset. This is because the Python code itself is lower than the semantic information contained in the Java code, so when converting the sequence to a semantic map such as AST, When it contains less information, there is a gap between the final generated summary and the reference value matching. Future research seems to improve the accuracy of Python dataset generation by modifying the nodes of the AST semantic tree to increase the semantic information on the Python dataset.

For the ablation experiments, we split the FCSO approach and removed each module. Then, increase quantitatively one by one and observe the change of the corresponding evaluation index data for each module added. The specific data is shown in Table 3 below. The first is to add a separate CodeBERT encoder and GCN encoder module and put the generated code feature attention vector into the Transformer decoder for scoring output. Their respective evaluation index data in the Java dataset are as expected, with BLEU scores of 32.68% and 38.55%. This data shows that a single code sequence feature and AST feature are not enough to reflect the semantic information of the code. The second is our multiple code feature attention fusion, using GCN-Encoder and CodeBERT-Encoder on the Java dataset, to score BLEU, METEOR score and ROUGE_L score 42.64%, 26.13%, and 53.12% have corresponding improvements. Finally, the Self-consistency module is added based on the above experiment. Compared with the previous experiment, the BLEU score, METEOR score and ROUGE_L score on the Java dataset increased by 1.31%, 1.74%, and 2.20%, respectively. The ablation experiments show that each module in our approach is indispensable, and the fusion of multiple features and Self-consistency output can improve the evaluation index of code summarization. For the experimental device, the experiments in this paper are conducted on an Ubuntu GPU server, with two GPUs of Tesla P40 and a graphics memory of 24 GB.

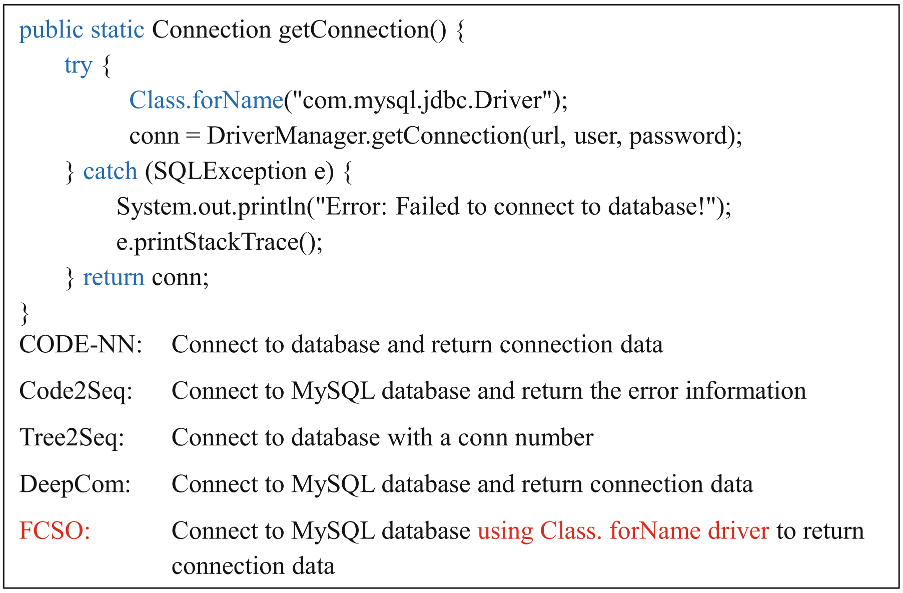
Table 3. Comparison of our proposed approach with the baseline approaches.

Approach	Java			Python		
	BLEU	METEOR	ROUGE_L	BLEU	METEOR	ROUGE_L
CODE-NN	27.51%	12.59%	40.30%	17.28%	9.16%	37.68%
Code2Seq	37.12%	20.14%	51.37%	19.88%	10.33%	37.80%
Tree2Seq	37.75%	21.95%	51.49%	20.07%	8.96%	35.64%
DeepCom	39.25%	23.06%	52.67%	20.78%	9.98%	37.35%
FCSO	**44.70%**	**26.86%**	**54.51%**	**31.91%**	**19.10%**	**45.23%**
Ablation Study						
CodeBERT-Encoder	32.68%	18.71%	43.59%	19.50%	8.64%	36.57%
GCN-Encoder	38.55%	21.43%	49.74%	20.86%	8.92%	37.31%
(GCN+BERT) Encoder	42.62%	25.13%	52.12%	30.29%	17.98%	43.67%
Self-consistency	43.93%	26.87%	54.32%	31.41%	18.38%	44.58%

4.3 Code Summarization Examples

At the end of the experiment, we compared the summary generated by the previous benchmark model with the summary generated by our approach, as shown in Fig. 4 and Fig. 5. In the examples of Java and Python codes, we use a section of the MYSQL database connection method code to observe the differences in abstracts generated by various models and methods. The blue color in the figure indicates the key information of the code, while the red color indicates the information that our approach has not lost compared to the benchmark method. In Java functions, the first few methods, such as CODE-NN and Code2Seq, are only summaries and do not involve the Class. forName method used to connect to the database. In Python functions, the same benchmark method is a single grab header function, and our approach diagram is highlighted in blue, where we found the connect method and the judgment statement.

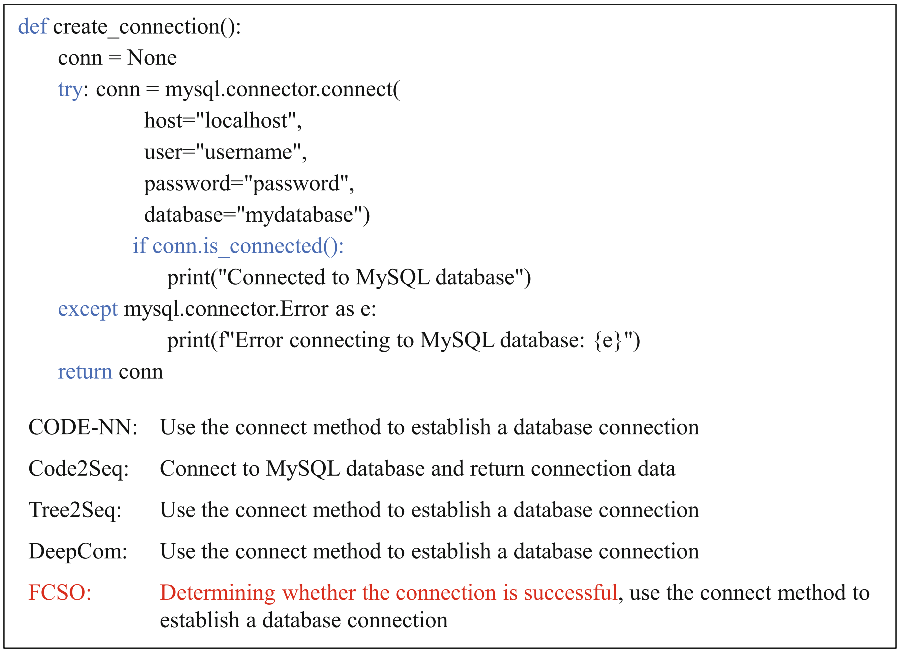
Specifically, it can be found from the figure that in the CODE-NN method, because the LSTM structure is used, it is not easy to process the semantic information of long sequences, and the key information is lost in the summary. At the same time, the Code2Seq method adds self-attention weights to the LSTM framework, and the summary output can briefly explain the connection method of the database. The SBT method implemented by DeepCom also processes AST sequences in a single way, and the summary cannot retain important judgment statements. The approach FCSO we proposed can retain the connect and if statements, and express the summary more completely, which is beneficial to the programmer's code development.



```java
public static Connection getConnection() {
    try {
        Class.forName("com.mysql.jdbc.Driver");
        conn = DriverManager.getConnection(url, user, password);
    } catch (SQLException e) {
        System.out.println("Error: Failed to connect to database!");
        e.printStackTrace();
    } return conn;
}
```

CODE-NN:	Connect to database and return connection data
Code2Seq:	Connect to MySQL database and return the error information
Tree2Seq:	Connect to database with a conn number
DeepCom:	Connect to MySQL database and return connection data
FCSO:	Connect to MySQL database using Class. forName driver to return connection data

Fig. 4. Comparing Summary Results with Baseline on Java Code Methods. (Color figure online)



```python
def create_connection():
    conn = None
    try: conn = mysql.connector.connect(
            host="localhost",
            user="username",
            password="password",
            database="mydatabase")
        if conn.is_connected():
            print("Connected to MySQL database")
    except mysql.connector.Error as e:
            print(f"Error connecting to MySQL database: {e}")
    return conn
```

CODE-NN:	Use the connect method to establish a database connection
Code2Seq:	Connect to MySQL database and return connection data
Tree2Seq:	Use the connect method to establish a database connection
DeepCom:	Use the connect method to establish a database connection
FCSO:	Determining whether the connection is successful, use the connect method to establish a database connection

Fig. 5. Comparing Summary Results with Baseline on Python Code Methods. (Color figure online)

5 Conclusion

In this paper, we propose a source code summarization approach that fuses code features into self-consistency output (FCSO). This approach first constructs a GCN encoder and a CodeBERT encoder. It outputs the AST feature attention vector and sequence feature attention vector of the source code, respectively. Then, input these two feature vectors into the Transformer model decoder for feature fusion to ensure subsequent model training. Finally, we use the self-consistency decoding strategy to calculate the similarity score of the sequence output by beam search and output a consistent code summary answer, which improves the accuracy of the generated summary. In particular, we conducted comparative benchmark experiments and ablation experiments on two data sets, Java and Python, and the indicators of the experimental results have been significantly improved. It proves that our approach can learn and express the global and local features of the code more accurately and retain the semantic information of the code more completely to help programmers better understand and use the code and improve the development efficiency of software code.

In the future, our research will pay more attention to modifying the structure in the decoder and think about how to realize the fusion of the front encoder information and the back decoder information to improve the performance of source code summarization. Or it may also study code summary generation in other fields, such as helping the summary generation of the current popular blockchain smart contract code, which can promote the rapid development of this field.

Acknowledgements. This work is supported by Jiangxi Provincial Natural Science Foundation under Grant No. 20224BAB212015, the Foundation of Jiangxi Educational Committee Under Grant No. GJJ210338, the National Natural Science Foundation of China (NSFC) under Grant No. 62363015, 61962026 and the National Natural Science Key Foundation of China Grant No. 61832014.

References

1. Ko, A.J., Myers, B.A., Aung, H.H.: Six learning barriers in end-user programming systems. In: 2004 IEEE Symposium on Visual Languages-Human Centric Computing, pp. 199–206. IEEE, September 2004
2. Eddy, B.P., Robinson, J.A., Kraft, N.A., Carver, J.C.: Evaluating source code summarization techniques: replication and expansion. In: 2013 21st International Conference on Program Comprehension (ICPC), pp. 13–22. IEEE, May 2013
3. Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., Vijay-Shanker, K.: Towards automatically generating summary comments for Java methods. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 43–52, September 2010
4. Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L., Vijay-Shanker, K.: Automatic generation of natural language summaries for Java classes. In: 2013 21st International Conference on Program Comprehension (ICPC), pp. 23–32. IEEE, May 2013

5. Wong, E., Yang, J., Tan, L.: AutoComment: mining question and answer sites for automatic comment generation. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 562–567. IEEE, November 2013
6. Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, vol. 1: Long Papers, pp. 2073–2083, August 2016
7. Hu, X., Li, G., Xia, X., Lo, D., Lu, S., Jin, Z.: Summarizing source code with transferred API knowledge (2018)
8. Hu, X., Li, G., Xia, X., Lo, D., Jin, Z.: Deep code comment generation. In: Proceedings of the 26th Conference on Program Comprehension, pp. 200–210, May 2018
9. Wang, W., Zhang, Y., Zeng, Z., Xu, G.: TranS3: a transformer-based framework for unifying code summarization and code search. arXiv preprint arXiv:2003.03238 (2020)
10. Freitag, M., Al-Onaizan, Y.: Beam search strategies for neural machine translation. arXiv preprint arXiv:1702.01806 (2017)
11. Feng, Z., et al.: CodeBERT: a pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020)
12. Huo, X., Li, M., Zhou, Z.H.: Control flow graph embedding based on multi-instance decomposition for bug localization. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, no. 04, pp. 4223–4230, April 2020
13. LeClair, A., Haque, S., Wu, L., McMillan, C.: Improved code summarization via a graph neural network. In: Proceedings of the 28th International Conference on Program Comprehension, pp. 184–195, July 2020
14. Shi, E., et al.: Cast: enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. arXiv preprint arXiv:2108.12987 (2021)
15. LeClair, A., Jiang, S., McMillan, C.: A neural model for generating natural language summaries of program subroutines. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 795–806. IEEE, May 2019
16. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems, vol. 27 (2014)
17. Luong, M.T., Pham, H., Manning, C.D.: Effective approaches to attention-based neural machine translation. arXiv preprint arXiv:1508.04025 (2015)
18. Vaswani, A., et al.: Attention is all you need. In: Advances in Neural Information Processing Systems, vol. 30 (2017)
19. Ahmad, W.U., Chakraborty, S., Ray, B., Chang, K.W.: A transformer-based approach for source code summarization. arXiv preprint arXiv:2005.00653 (2020)
20. Perozzi, B., Al-Rfou, R., Skiena, S.: DeepWalk: online learning of social representations. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 701–710, August 2014
21. Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Zhou, D.: Self-consistency improves chain of thought reasoning in language models. arXiv preprint arXiv:2203.11171 (2022)
22. Cheng, J., Fostiropoulos, I., Boehm, B.: GN-Transformer: fusing sequence and graph representation for improved code summarization. arXiv preprint arXiv:2111.08874 (2021)
23. Wang, Y., Dong, Y., Lu, X., Zhou, A.: GypSum: learning hybrid representations for code summarization. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, pp. 12–23, May 2022

24. Gao, Y., Lyu, C.: M2TS: multi-scale multi-modal approach based on transformer for source code summarization. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, pp. 24–35, May 2022
25. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y.: Graph attention networks. arXiv preprint arXiv:1710.10903 (2017)
26. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 (2016)
27. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
28. Sun, Z., Zhu, Q., Xiong, Y., Sun, Y., Mou, L., Zhang, L.: TreeGen: a tree-based transformer architecture for code generation. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, no. 05, pp. 8984–8991, April 2020
29. Barone, A.V.M., Sennrich, R.: A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. arXiv preprint arXiv:1707.02275 (2017)
30. Papineni, K., Roukos, S., Ward, T., Zhu, W.J.: BLEU: a method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, pp. 311–318, July 2002
31. Banerjee, S., Lavie, A.: METEOR: an automatic metric for MT evaluation with improved correlation with human judgments. In: Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization, pp. 65–72, June 2005
32. Lin, C.Y.: ROUGE: a package for automatic evaluation of summaries. In: Text Summarization Branches Out, pp. 74–81, July 2004
33. Alon, U., Brody, S., Levy, O., Yahav, E.: code2seq: generating sequences from structured representations of code. arXiv preprint arXiv:1808.01400 (2018)
34. Tai, K.S., Socher, R., Manning, C.D.: Improved semantic representations from tree-structured long short-term memory networks. arXiv preprint arXiv:1503.00075 (2015)