Received May 30, 2020, accepted July 21, 2020, date of publication July 24, 2020, date of current version August 4, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3011744

Fret: Functional Reinforced Transformer With BERT for Code Summarization

RUYUN WANG[1], (Student Member, IEEE), HANWEN ZHANG[1], GUOLIANG LU[2], (Member, IEEE), LEI LYU[1], AND CHEN LYU[1]
[1]School of Information Science and Engineering, Shandong Normal University, Jinan 250014, China
[2]School of Mechanical Engineering, Shandong University, Jinan 250061, 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 summarization has long been viewed as a challenge in software engineering because of the difficulties of understanding source code and generating natural language. Some mainstream methods combine abstract syntax trees with language models to capture the structural information of the source code and generate relatively satisfactory comments. However, these methods are still deficient in code understanding and limited by the long dependency problem. In this paper, we propose a novel model called **Fret**, which stands for **F**unctional **RE**inforced **T**ransformer with BERT. The model provides a new way to generate code comments by learning code functionalities and deepening code understanding while alleviating the problem of long dependency. For this purpose, a novel reinforcer is proposed for learning the functional contents of code so that more accurate summaries to describe the code functionalities can be generated. In addition, a more efficient algorithm is newly designed to capture the source code structure. The experimental results show that the effectiveness of our model is remarkable. Fret significantly outperforms all the state-of-the-art methods we examine. It pushes the BLEU-4 score to 24.32 for Java code summarization (14.23% absolute improvement) and the ROUGE-L score to 40.12 for Python. An ablation test is also conducted to further explore the impact of each component of our method.

INDEX TERMS BERT language representation model, software engineering, source code summarization, transformer network.

I. INTRODUCTION

Programmers are keyboard pianists who write beautiful pieces of code by striking keys. They create great software through efficient collaboration and program optimisation. In most cases, development does not happen overnight. More precisely, programmers need to update and maintain code over a long period. Iterations of functionality are often based on existing code. A high-quality code comment can point out key functions directly and enable programmers to innovate efficiently, whereas a piece of messy code with a useless comment may force our pianists to improvise. For fast code comment generation, researchers have begun to explore methods of source code summarization.

The associate editor coordinating the review of this manuscript and approving it for publication was Xiaobing Sun.

Deep neural networks are actively used in various fields, such as recurrent neural networks (RNNs) [1] and long short-term memory (LSTM) [2]. In 2014, Sutskever *et al.* [3] proposed the sequence-to-sequence (Seq2Seq) model, which pushed the field of neural machine translation (NMT) to a new height. In 2017, Transformer [4] became the new state of the art in the area of NMT and solved one of the most challenging problems: long-term dependency. Bidirectional encoder representations from Transformers (BERT) [5] used the mask mechanism to address the problem of polysemy and deepen the understanding of natural language (NL).

A. MOTIVATION

Recent studies show that by combining and innovating existing NMT technologies, code comments can be generated based on extensive training. Due to their strong relevance to machine translation, Seq2Seq and the attention mechanism are extensively applied to code summarization tasks, such as those discussed in [6], [7], and [8]. However, these authors found that a "flat" neural architecture cannot capture structural information well, and researchers have therefore tried to introduce other mechanisms for the task of code summarization. Allamanis *et al.* [9] used a convolutional attention network to generate code comments, which addressed the problem of multi-threaded computation. Moore *et al.* [10] also used convolutional neural networks (CNNs) to summarize source code. Some studies have indicated that pure NL models cannot understand the structural information of code, so researchers have introduced abstract syntax trees (ASTs) into the NMT model to capture code structure and generate comments; examples of this strategy include [11], [14]–[18], and [19]. Furthermore, Hu *et al.* [16] regarded the application program interface (API) as another factor worth considering, and they parsed an API with a tree as well. Wei *et al.* [20] proposed the dual model, considering code summarization and code generation as mutually compatible goals, so that both tasks could be performed by one model. After repeated improvements by researchers, the performance of machine code summarization models has dramatically improved.

Nevertheless, the existing models still face three challenges, which create bottlenecks in improving the quality of code comments:

1) **Limitation by the long dependency problem.** The length of code is uncertain. A piece of code can easily be hundreds or even thousands of words long, and a code element may depend on another element many tokens away.
2) **Lack of effective guidance.** It is inefficient to train models aimlessly on large amounts of data. Existing methods aim to improve the effectiveness of code summarization in various ways, such as by using CNNs rather than LSTM, embedding ASTs, or adding attention mechanisms. However, these methods do not aim directly at improvement.
3) **Lack of code understanding.** Deep understanding of programs has always been a challenge in the field of code summarization. The current mainstream approach obtains word maps mechanically, but performance still encounters bottlenecks due to a lack of code understanding.

The Transformer model [4] is skilled in solving the problem of long dependency. However, the original architecture of Transformer was not designed for programs, which means that it cannot capture the structural information of programs, not to mention understanding the code.

B. CONTRIBUTION

In this paper, we propose a code summarization model called **Fret**, which stands for **F**unctional **RE**inforced **T**ransformer with BERT. The overall workflow is depicted in Fig. I-A. For this model, we developed a novel reinforcer for learning the code functionalities, through which Fret can generate more

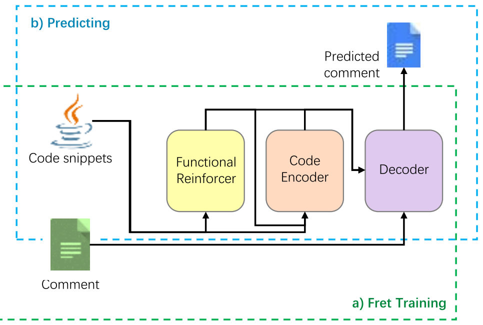


FIGURE 1. **The overall workflow of our code summarization approach.**

functionally clear, accurate, and comprehensive descriptions of the code. The code is input into the functional reinforcer and the encoder, and then a comment is generated by the decoder. To be more specific, the architecture we propose can be described as a "reinforcer-encoder-decoder" form. Fret combines the reinforcer with Transformer and BERT to bridge the gap between source code and NL. In this way, our model can tackle the three challenges previously mentioned, namely, alleviating the problem of long dependency, gaining effective guidance, and deepening code understanding. The reinforcer integrates a token-segmented functional emphasis mechanism to strengthen the functional understanding of code. The main functionalities of code snippets are already extracted and condensed into the names of the snippets by programmers (e.g., "`render_to_file()`" and "`test_story_three_response()`," which have a high probability of performing the task that their names represent.) Additionally, using segmented tokens can reduce the dictionary size and the probability that uncommon tokens are replaced by "⟨UNK⟩" as discussed in [17]. In addition, to capture the code's structural information, we designed a new sentence-level structure encoding algorithm that can embed structural information in a more efficient way.

In summary, the contributions of this paper are as described below.

- We propose Fret, a novel model using a functionally reinforced Transformer with BERT, which introduces functional guidance into code summarization, alleviating the problem of long dependencies while also deepening the understanding of the code. The most essential idea of this approach is that, through functional guidance, Fret can generate more functionally clear, accurate, and comprehensive descriptions of the code.
- We propose a more comprehensive functionality-reinforced source code representation method, which reinforces the functionality into code embedding by using the newly proposed functional reinforcer and structure encoding.
- We conducted a series of experiments including quantitative and qualitative comparisons that demonstrate the superiority of Fret compared with other state-of-the-art methods.

C. ORGANISATION

The remainder of this paper is organised as follows: Section II covers related work in code summarization. Section III describes the detailed design of our model. Sections IV and V show the experiments and the results of our method on the Java and Python datasets compared to those of other state-of-the-art methods. Finally, the discussion and conclusion are given in Sections VI and VII.

II. RELATED WORK

A. CODE REPRESENTATION

The first step of understanding code in depth is to represent it appropriately. In recent years, researchers have made continuous efforts in this area. They have achieved excellent results and put forward many impressive algorithms. Researchers have borrowed approaches from the field of natural language processing (NLP) [21]–[23], ranging from statistical machine translation [24], [25] to deep neural network learning [3], to serve the purpose of summarizing the language in programs. Tung *et al.* [26] proposed a statistical semantic language model for source code that merges semantic information into code tags. It also merges local contexts in semantics and global technical capabilities into an n-gram topic model. Recent studies have preferred the application of neural network learning. Gu *et al.* [27] used an RNN encoder-decoder model [3] that encodes a user query sequence of words into a fixed-length context vector to generate a context-based API sequence. In the work of Piech *et al.* [28], a method of embedding preconditions and postconditions simultaneously in a shared Euclidean space was proposed to encode programs as a linear mapping. Mou *et al.* [29] took code structure information into account by incorporating a syntax structure tree into the code representation. Because a program contains rich, explicit, and complex structural information, a convolution kernel was designed on top of the program's AST to capture structural information. In the field of code searching, researchers have proposed new ways to tackle the problem of code representation. Yao *et al.* [30] proposed a model called MMAN to address the problem of code retrieval. They developed a comprehensive multimodal representation, considering ASTs and CFGs to represent the unstructured and structured features of the source code. Gu *et al.* [31] proposed a model called CODEnn, which creatively embeds code and NL descriptions jointly into a high-dimensional vector space. They divided code into three parts, which are the method name, API sequence, and token, and then obtained the code vector by inputting these parts into an RNN and MLP for fusion. Lv *et al.* [32] proposed CodeHow, a code search model that could recognise potential APIs by applying the extended Boolean model. This model searches code based on both code text and name matching and then identifies and returns relevant APIs after ranking. Allamanis *et al.* [34] proposed a model that tackles the retrieval problem between NL and source code by combining the techniques of source code statistical modelling with bimodal modelling of images and natural language. A probabilistic context-free grammar and neuroprobabilistic language-based model was proposed by Maddison and Tarlow [33], which aims to generate a structure for natural source code. Both Allamanis *et al.* [34] and Maddison and Tarlow [33] considered source code as a tree. Our proposed approach differs significantly from the abovementioned model in terms of code representation: beyond taking code features into account, Fret also incorporates the extraction of functional information and implements an efficient algorithm to extract structural information. Furthermore, we present a new technique, using the Hadamard product, to integrate these new features that can contribute to improved performance.

B. CODE SUMMARIZATION

As a branch of software engineering, source code summarization has long been viewed as a challenge. There are many different methods for automatically summarizing code, including manually crafted templates [35], [36], information retrieval [37], [38], and deep neural network learning [6], [14], [17]–[19]. Manual template matching is one method of code summarization. McBurney and Mcmillan [35] located the most important context-based methods with PageRank and collected relevant keywords with SWUM. Wong *et al.* [36] proposed a general approach that used a context-sensitive text similarity technique to find similar code snippets and annotated them to describe other similar snippets. Methods based on information retrieval and code matching are also applied in this task. Haiduc *et al.* [37] proposed a text retrieval (TR) technique for lexical and structural information in code to solve the code summarization problem. Haiduc *et al.* [38] used the combination of a term's location in a program and the TR technique to capture the meaning of methods and classes. In other words, they used automatic text summarization techniques to generate source code comments. Iyer *et al.* [6] proposed a CODE-NN model that uses LSTM networks and attention mechanisms to summarize the code from noisy online programming websites. Hu *et al.* [17], [18] proposed a model called DeepCom that uses NLP technology to analyse the structural information of Java methods through a deep neural network. It generates comments from learned features. Wan *et al.* [19] presented the actor-critic model, in which a mixed attention layer, an AST-based LSTM layer and other LSTM layers integrate the structure and sequential content of the code. Shido *et al.* [14] proposed an extension of Tree-LSTM based on the work of Tai *et al.* [11]. The model constructed a distributed representation of ordered trees so that Tree-LSTM could handle any number of ordered children. Shen *et al.* [12] proposed an automatic summarization model with the aim of adapting to source code changes. Liu *et al.* [13] used latent semantic indexing and clustering to group source artifacts with similar vocabularies, extracted topics composed of the vectors of independent words based on latent semantic indexing, and then used Minipar to generate summaries.

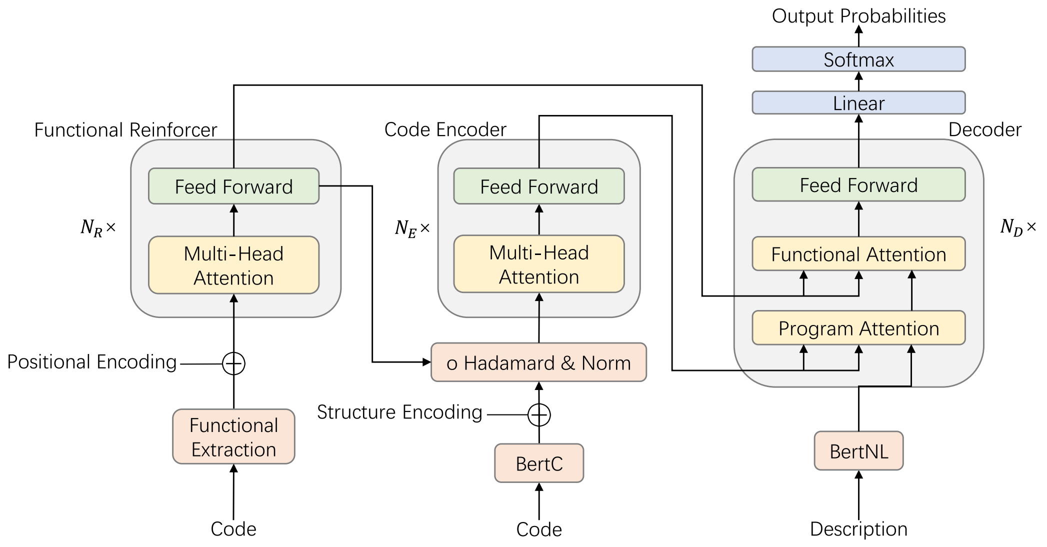


FIGURE 2. An overview of our proposed model, Fret, for code summarization.

Unlike the above models, Fret uses Transformer to bridge the gap between code and NL. The code embedding generated by BERT and the structural information are utilised to perform code summarization.

III. MODEL ARCHITECTURE

Fret is composed of three parts: the functional reinforcer, code encoder, and decoder. An overview of our model is shown in Fig. 2. The reinforcer emphasises the functional keywords, and it guides the summarization process with a focus on the code's functionality. Similar to many NMT models, Fret also relies on an encoder-decoder architecture to bridge the gap between code and NL. For clarity, we specify δ, x, and y as the data carriers for the reinforcer, encoder, and decoder, respectively. The subscripts $_R(_r)$, $_E(_e)$, and $_D(_d)$ symbolise these three parts.

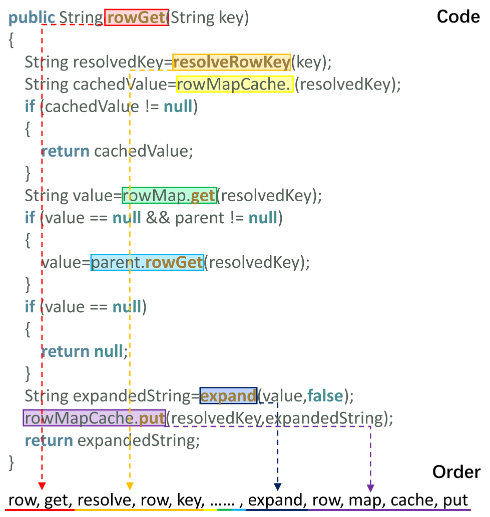
A. FUNCTIONAL REINFORCER

The functional reinforcer is used to extract the most critical function-indicated tokens in the code and, after integration, input them into the encoder. We first extract nr keywords $(keyword_1, keyword_2, \cdots, keyword_{nr})$ from the code snippets and embed them into the vector $(k_1, k_2, \cdots, k_{nr})$ as input. The output of the reinforcer Z_R is transmitted to the encoder to await further calculation; this output contains the functional information of the code snippet.

1) FUNCTIONAL EXTRACTION

Functional extraction is an operation that extracts functional keywords from a code snippet and concatenates them into a vector called the functional order: $order = (keyword_1, keyword_2, \cdots, keyword_{nr})$.

An example of the extraction process is shown in Fig. 3. The keyword "`resolveRowKey`" is a word combination; we first separate it into single words, which yields



```
public String rowGet(String key)
{
   String resolvedKey=resolveRowKey(key);
   String cachedValue=rowMapCache. (resolvedKey);
   if (cachedValue != null)
   {
      return cachedValue;
   }
   String value=rowMap.get(resolvedKey);
   if (value == null && parent != null)
   {
      value=parent.rowGet(resolvedKey);
   }
   if (value == null)
   {
      return null;
   }
   String expandedString=expand(value,false);
   rowMapCache.put(resolvedKey,expandedString);
   return expandedString;
}
```

row, get, resolve, row, key, ... , expand, row, map, cache, put

FIGURE 3. One example of the functional extraction process.

the sequence "`resolve, row, key`," and then splice this sequence onto the entire *order*. This is the functional extraction operation. Later, each keyword $keyword_i$ will be embedded into a vector k_i of d dimensions, that is, $(k_1, k_2, \cdots, k_{nr}) \in \mathbb{R}^{nr \times d}$.

2) POSITIONAL ENCODING

The positional encoding can incorporate the order information PE_{pos} into the keyword embedding $(k_1, k_2, \cdots, k_{nr})$ with:

$$\begin{cases} PE_{(pos,2i)} = sin(\frac{pos}{10000^{2i/d}}) \\ PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{2i/d}}), \end{cases} \tag{1}$$

where PE is the positional encoding, pos is the position of k_i, and d is the embedding size. Since the position information is ignored in attention, we add the positional encoding at the bottom of the reinforcer as is done in [4]. The actual input of the reinforcer is $\boldsymbol{\delta}$, and each character δ_i of the input is calculated according to:

$$\delta_i = k_i + PE_{pos}, \tag{2}$$

where the input of the reinforcer is $\boldsymbol{\delta} = (\delta_1, \delta_2, \cdots, \delta_{nr})$, $\boldsymbol{\delta} \in \mathbb{R}^{nr \times d}$, nr indicates the token number, and d denotes the embedding size. Here, $\boldsymbol{\delta}$ retains both functional and positional information, and it is inputted into the first layer of the reinforcer; that is, we use multi-head attention.

3) MULTI-HEAD ATTENTION

Multi-head attention provides multiple representation subspaces for the attention layer; hence, it is skilled in learning word distances. Specifically, it uses three matrices as hidden layers: the query matrix Q, key matrix K, and value matrix V. Each of them is calculated from a dot product of the learnable weight matrix $W_{(\cdot)} \in \mathbb{R}^{d \times d_k}$:

$$[Q, K, V] = [\delta_1, \delta_2, \cdots, \delta_{nr}][W_Q, W_K, W_V]. \tag{3}$$

We first calculate the self-attention as follows:

$$Attention(Q_R, K_R, V_R) = softmax(\frac{Q_R \cdot K_R^T}{\sqrt{d_k}}) \cdot V_R, \tag{4}$$

where d_k indicates the dimension of K_R. The generated output tensor is named Z. To counteract the problem of the increasing size of the dot product, the value of $(Q_R \cdot K_R^T)$ is scaled by $\frac{1}{\sqrt{d_K}}$. Next, each head $head_i$ applied in attention is computed as:

$$head_i = Attention(QW_{Ri}^Q, KW_{Ri}^K, VW_{Ri}^V), \tag{5}$$

where $W_{Ri}^Q \in \mathbb{R}^{d \times d_k}$, $W_{Ri}^K \in \mathbb{R}^{d \times d_k}$, and $W_{Ri}^V \in \mathbb{R}^{d \times d_v}$. The weight matrix $W_{(\cdot)}^{(\cdot)}$ is learnable and is utilised to generate multi-head matrices $Q_{(\cdot)}^{(\cdot)}$, $K_{(\cdot)}^{(\cdot)}$, and $V_{(\cdot)}^{(\cdot)}$ with the dot product. Then, the output of multi-head attention can be computed by:

$$\begin{aligned} Z_R &= MultiHead(Q_R, K_R, V_R) \\ &= Concat(head_1, head_2, \cdots, head_h) \cdot W_R^O, \end{aligned} \tag{6}$$

where the output concatenated weight $W_R^O \in \mathbb{R}^{hd_v \times d_k}$. In this work, we set $h = 12$ and $d_k = d_v = d/h = 64$. To avoid the problem of dispersion, residual connection $(Z_R + \boldsymbol{\delta})$ is applied before layer normalisation [39]:

$$\widetilde{Z_R} = LayerNorm(\boldsymbol{\delta} + Z_R). \tag{7}$$

Here, *LayerNorm* can be calculated by:

$$\mu_i = \frac{1}{m}\sum_{i=1}^{m} x_{ij}, \tag{8}$$

$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_{ij} - \mu_j)^2, \tag{9}$$

$$LayerNorm(x) = \alpha \odot \frac{x_{ij} - \mu_j}{\sqrt{\sigma^2 + \epsilon}} + \beta, \tag{10}$$

where α and β prevent excessive loss; ϵ prevents division by zero.

4) FEED-FORWARD NETWORKS

Fully connected feed-forward networks consist of a ReLU activation with linear transformations on both sides.

This network is applied to each position separately by the same method as in [4]. The dimensions of its input and output are set to $d = 768$, and the dimension of the inner layer is $d_{ff} = 3072$. The final output of the reinforcer is $\widetilde{\widetilde{Z_R}} \in \mathbb{R}^{nr \times d}$ (hereafter referred to as Z_R).

B. CODE ENCODER

The code encoder takes two inputs: the output of the reinforcer Z_R and the sentence-level code embedding vector $\boldsymbol{x}$. However, before the encoder operates on them, they undergo some preprocessing. Considering that BERT excels in sentence understanding and embedding, we implement the input code at the sentence level.

1) BertC

BertC stands for "BERT for Code." BERT uses Transformer's encoder as its kernel to capture the bidirectional information of a sentence. Two tasks are adapted to increase understanding: a) masked language model, to mask 15% of the words in the document (80% of which are replaced with "`[mask]`," 10% with the original word and 10% with a random word); b) next-sentence prediction.

BERT needs to be pre-trained. The architecture of ELMo [40] replaces the bidirectional language model with multi-head attention. In brief, the embedding of a token depends on its context. Analogously, BERT also needs to be preprocessed to embed the location information before serving:

$$Input_i = E_{token} + E_{seg} + E_{pos} \tag{11}$$

where E_{token}, E_{seg}, and E_{pos} represent token embeddings, segment embeddings and position embeddings, respectively. We set the embedding size $d_{BERT} = 768$ as recommended in [5]. The input is processed as follows:

Pre-training We embed the code sentence token by token and form it into a token-embedding sequence:

$[E_{[CLS]}, E_{t1}, \cdots, E_{tn}, E_{[SEP]}, E_{tn+1}, \cdots, E_{tn+m}, E_{[SEP]}]$
where E indicates the embedding vector with dimension d; `[CLS]` is the symbol of the first word; `[SEP]` is a signal of clauses; t denotes a token, the first sentence has n tokens, and the second sentence has m tokens.

Embedding We first segment a code snippet at the sentence level. Each sentence is further embedded by tokens $[E_{t1}, E_{t2}, \cdots, E_{tn}]$. It is input into BERT, and its sentence embedding xs_i, $xs_i \in \mathbb{R}^{d_{BERT}}$ is obtained. As a result, we obtain a code embedding at the sentence level: $\boldsymbol{xs} = (xs_1, xs_2, \cdots, xs_{ne})$, where $\boldsymbol{xs} \in \mathbb{R}^{ne \times d_{BERT}}$.

2) STRUCTURE ENCODING

Structure encoding can capture the hierarchical relationships within code. The details of structure encoding are presented in Algorithm 1.

Algorithm 1 Structure Encoding Algorithm

```
Require: Code snippet $code$; weight $w = 1$; BERT dimension $d_{BERT}$;
Ensure: Structure encoding $SE$;
  $SE \leftarrow \varnothing$;
  for each sentence $s_i$ in $code$ do
    if $s_i$ is a sublayer then
      $w \leftarrow w/2$;
    else if $s_i$ is a parent layer then
      $w \leftarrow w * 2$
    else
      $w \leftarrow w$;
    end if
    $SE_i \leftarrow SE_{i-1}$.append([$w$]*$d_{BERT}$)
  end for
  return $SE$
```

After Algorithm 1, we can obtain the structure encoding SE. Since the input sequence $\boldsymbol{xs} = (xs_1, xs_2, \cdots, xs_{ne})$ embedded by BertC is missing structural information, structure encoding incorporates this essential information into $\boldsymbol{xs}$ before it is inputted. The matrix $\boldsymbol{xs}$ adds the structure encoding SE with:

$$\boldsymbol{xs}_{str} = xs + SE, \tag{12}$$

where $\boldsymbol{xs}_{str}$ denotes the structure-encoded code sentence and SE denotes the structure-encoding matrices returned by Algorithm 1. Thus, here, $\boldsymbol{xs}_{str} \in \mathbb{R}^{ne \times d_{BERT}}$, and we assign $d = d_{BERT} = 768$.

3) HADAMARD & NORM

The process by which we perform information fusion is described in this section. We can obtain the functional order output $Z_R \in \mathbb{R}^{nr \times d}$ from the final stack of the reinforcer according to (6):

$$Z_R = MultiHead(Q_R, K_R, V_R).$$

For brevity, we introduce an intermediate variable C_R to represent the reinforcer coefficient:

$$C_R = Z_R^T \cdot Z_R, \tag{13}$$

where $Z_R \in \mathbb{R}^{nr \times d}$ and thus $C_R \in \mathbb{R}^{d \times d}$; the structure-encoded sequence $\boldsymbol{xs}_{str} \in \mathbb{R}^{ne \times d_{BERT}}$; nr is the token number of $\boldsymbol{\delta}$ in the reinforcer; and ne is the sentence number of $\boldsymbol{xs}_{str}$ in the encoder.

To strengthen the influence of the function code, we multiply each element in $(\boldsymbol{sx}_{srt} \cdot C_R)$ by its corresponding positional element in $\boldsymbol{xs}_{str}$; specifically, we take the Hadamard product. Then, we norm it before inputting it into the encoder:

$$\boldsymbol{x} = Norm(\boldsymbol{xs}_{str} \circ (\boldsymbol{sx}_{srt} \cdot C_R)), \tag{14}$$

where $\boldsymbol{x} = (x_1, x_2, \cdots, x_{ne})$, $\boldsymbol{x} \in \mathbb{R}^{ne \times d_{BERT}}$, is the input of the code encoder. Then, the code encoder receives the input $\boldsymbol{x}$ to calculate matrices K_E, V_E, and Q_E following (4), (5), and (6).

C. DECODER

Based on the consideration of both the program and functional guidance, similar to the models proposed by Iida *et al.* [43] and Liu *et al.* [44], the decoder aims to generate the target sequence $\boldsymbol{y}$ by sequentially predicting the probability of a word y_t conditioned on the previously generated words $y_1, y_2, \cdots, y_{t-1}$. First, the target sequence $(y_1, y_2, \cdots, y_{nd})$ is formatted at the token level and fed into the decoder sequentially for training. During prediction, we apply two attention layers to integrate the outputs of the functional reinforcer and code encoder. Token y_t at time t is input into the decoder to generate token y_{t+1} at the next time step.

1) BertNL

BertNL stands for "BERT for Natural Language," which is distinct from but similar to BertC, described in section III-B1. The only difference is that the training of BertC uses the code extracted from datasets, whereas BertNL is based on a model pre-trained by Google.[1] BertNL can transform a comment into a token-level vector $\boldsymbol{y} = (y_1, y_2, \cdots, y_{nd})$, where nd is the token number of $\boldsymbol{y}$, and its embedding size is set to $d = 768$, which is the same as that of the encoder.

2) PROGRAM ATTENTION

Program attention integrates the information contained in a code snippet. The essence of program attention is multi-head attention. From the code encoder, the attention module inherits K_E and V_E, which are computed from the code snippet $\boldsymbol{xs} = (xs_1, xs_2, \cdots, xs_{ne})$. Q is computed from $\boldsymbol{y} = (y_1, y_2, \cdots, y_{nd})$ (for the first stack in the decoder) or transmitted by the forward layer (for other stacks of blocks). According to (6) and (7), the output of program attention Z_p can be calculated as follows:

$$Z_p = MultiHead(Q, K_E, V_E), \tag{15}$$
$$\widetilde{Z_p} = LayerNorm(Z_p). \tag{16}$$

Simultaneously, the output value is also conveyed to the next layer, which is functional attention.

3) FUNCTIONAL ATTENTION

Functional attention integrates the information contained in man-made names. Similar to the way program attention operates, functional attention receives the matrices K_R and V_R from the functional reinforcer. Its Q^* values are also calculated (for the first stack of a block) or given by the previous layer. Here, this kind of attention inherits the importance information integrated by the functional reinforcer. Similar to program attention, the output of functional attention Z_f is calculated according to (6) and (7):

$$Z_f = MultiHead(Q^*, K_R, V_R), \tag{17}$$
$$\widetilde{Z_f} = LayerNorm(Z_f), \tag{18}$$

[1]The code of BERT and its pre-trained models are available at https://github.com/google-research/bert

where Q^* is the query of functional attention and can be calculated by:

$$Q^* = Z_p * W^Q. \tag{19}$$

Finally, two fully connected layers follow, which extract features for prediction. Therefore, the output of the decoder is:

$$Z_D = FFN(\widetilde{Z_f}), \tag{20}$$

where *FFN* is a widely used feed-forward neural network; here, we use ReLU as its activation function.

4) TRAINING AND INFERENCE

The target sequence $\boldsymbol{y}$ can be generated by sequentially predicting the probability of a token y_t conditioned on its previously generated words $y_1, y_2, \cdots, y_{t-1}$ as well as its corresponding code snippet input $\boldsymbol{xs}$ and keywords $\boldsymbol{\delta}$; i.e.,

$$p(y_1, y_2, \cdots, y_{nd}|\boldsymbol{\delta}, \boldsymbol{xs}) = \prod_{t=1}^{nd} p(y_t|\boldsymbol{\delta}, \boldsymbol{xs}), \tag{21}$$

where $p(y_t|\boldsymbol{\delta}, \boldsymbol{xs})$ can be calculated by:

$$p(y_t|\boldsymbol{\delta}, \boldsymbol{xs}) = p(y_t|y_{<t}, \boldsymbol{\delta}, \boldsymbol{xs}). \tag{22}$$

Among all possible candidates, the token of the code comment is predicted by softmax based on the output from the decoder. The output Z_{D_t} is first input into two fully connected neural network layers, denoted as linear layers, and then the probabilities $p(y_t|\boldsymbol{\delta}, \boldsymbol{xs})$ of the code description token are calculated by softmax. The match is determined as follows:

$$p(y_t|y_{<t}, \boldsymbol{\delta}, \boldsymbol{xs}) = Softmax(Linear(Z_{D_t})), \tag{23}$$

where Z_{D_t} is the output of the last stack of the decoder at the current time step and $p(y_t|y_{<t}, \boldsymbol{\delta}, \boldsymbol{xs})$ is the probability of the comment token at the current time step.

In model prediction, we use the output of the decoder at the previous time step $y_t = Z_{D_{t-1}}$ as the input of the current time step, i.e., the query. In contrast, during training, we use the token of the ground-truth sequence $y_1, y_2, \cdots, y_{nd}$ at the previous time step as the input of the decoder at the current time step. In addition, the output of the t-query can use all the previous key-value pairs.

To obtain the real code comment, we need to calculate the probability $p(y_t|y_{<t}, \boldsymbol{\delta}, \boldsymbol{xs})$ by inputting it into softmax, from which we can obtain its corresponding token by indexing. The output probabilities are calculated by the softmax layer as follows:

$$\sigma_i(z) = \frac{e^{z_i}}{\sum_{j=1}^{T} e^{z_j}}, \tag{24}$$

where z_i is the ith value of $Linear(Z_{D_t})$, $\sum_{i=1}^{T} \sigma_i(z) = 1$; T is the vocabulary size. Our model is trained by the cross-entropy loss function, which is defined by the following formula:

$$loss = -\frac{1}{T} \sum_{i=1}^{T} y_i log(\sigma_i), \tag{25}$$

where σ_i is the ith value of the output vector σ obtained from softmax. $y_i \in \mathbb{R}^T$ and $\sum_{i=1}^{T} y_i = 1$.

After the end of the code is reached (the end symbol "⟨EOS⟩" is generated or the maximum length is reached), the output sequence $\boldsymbol{y} = (y_1, y_2, \cdots, y_{nd})$ constitutes the comment on its corresponding input source code.

IV. EXPERIMENTS

To evaluate the performance of our model, we conducted comparative experiments on Java and Python datasets with various baselines.

A. DATASETS

1) JAVA

The Java dataset we use was collected by Hu *et al.* [17]; it contains Java code extracted from GitHub[2] at a high quality. It is composed of code snippets and their corresponding Javadoc documentation comments. From the statistics of the dataset, shown in Fig. 4, we can draw two conclusions: a) most of the code length is concentrated in the same interval, as (0, 100) contains 90.71% of the items in the Java dataset formed at the token level; b) the code formed at the token level is much longer than that at the sentence level. At the sentence level, the length of the input vector can be reduced by roughly a third.

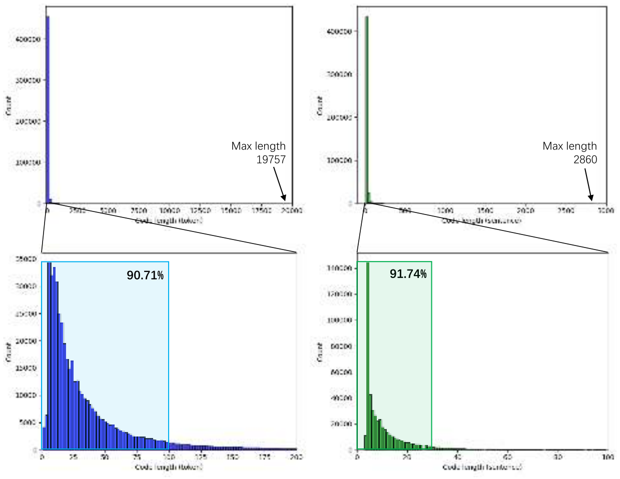


FIGURE 4. Statistics of the Java dataset. Although the longest code in the dataset is very long, most of the code is short and concentrated.

For the sake of fairness, we limit the length of the token-level method to 100 (90.71%) and the length of the sentence-level method to 35 (91.74%). A "token" includes all words and special characters that appear in the code. A "sentence" includes all sentences and special characters that occupy a line (e.g., "{" or "}"). In training, we replace tokens in both code and descriptions that occur fewer than ten times with special characters "⟨UNK⟩." In our approach, to give BERT a better understanding of code, we feed in the input sentence by sentence (even the overlong parts).

[2]https://github.com/xing-hu/EMSE-DeepCom

2) PYTHON

To assess the universality of our model, we selected a Python dataset for comparative tests. The dataset we used was collected by Miceli-Barone *et al.* [41] and was extracted from Python projects on GitHub. The Python dataset contains 108,726 code-comment pairs. We performed the same operations as for the Java dataset, dividing the dataset into three parts: training (60%), validation (20%), and testing (20%). The distribution trend is also very similar to that of the Java dataset (see Fig. 4), so we do not repeat it here.

B. BASELINES

We compared our model with two NMT models and five other code summarization models:

- The Seq2Seq [3] model consisting of an encoder-decoder is capable of dealing with the problem of many-to-many translations.
- Transformer [4] achieves state-of-the-art performance in the area of NMT, and it is also an encoder-decoder-based model. The attention layer improves LSTM's defects in gathering bidirectional information from a context.
- CODE-NN [6] uses LSTM networks with an attention mechanism to summarize code.
- Tree2Seq [42] is an end-to-end syntactic NMT model that uses a tree-based Seq2Seq model with an attention mechanism.
- DeepCom [17] is an attention-based Seq2Seq model that utilises deep neural networks to analyse structural information.
- Multi-way Tree-LSTM [14] is an AST extension of the Tree-LSTM model.
- The actor-critic model [19] is an AST structure-based model that encodes code snippets into a deep reinforcement learning framework.

C. TRAINING DETAILS

BertNL and BertC are set according to $BERT_{BASE}$'s recommendations [5], with $L = 12$, $H = 768$, and $A = 12$. Specifically, BertNL is a model with fine-tuning based on the pre-trained "`BERT-Base, Uncased`" model (English Wikipedia) provided by Google. For Transformer, we set the learning rate $lr = 0.01$, the mini-batch size $batch\_size = 64$, the number of reinforcer units $N_R = 6$, the encoder unit $N_E = 6$, and the decoder unit $N_D = 6$. We also use the Adam optimiser [45] with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$.

The experiments are performed with Python 3.6, TensorFlow 1.13.0, and Cuda 10.0. We train our models on a machine with a 5.0 GHz Intel Core i9-9900K CPU, 64 GB RAM, and 2*NVIDIA RTX-2080ti GPUs, running OS Ubuntu 16.04.

D. METRICS

To evaluate the performance of our model, we conducted experiments with four metrics that are widely used in the area of NMT: BLEU (BLEU-N) [46], METEOR [47], ROUGE-L [48], and RIBES [49].

BLEU (BLEU-N) is a metric that correlates highly with human judgements. It calculates n-gram precision between generated and target sentences. METEOR computes a combined score of unigram precision and unigram recall, which can evaluate how well-ordered a generated sentence is. ROUGE-L is based on the length of the longest average sequence that computes recall scores between two summaries. RIBES is proposed for distant-language translation based on rank correlation coefficients.

V. RESULTS

A. COMPARED TO BASELINES

To verify the effectiveness of our model in code summarization tasks, Fret was compared with other state-of-the-art models with experiments that used the Java and Python datasets. The experimental results are shown in Tables 1 and 2. It can be seen from the experimental results that our model outperforms the state-of-the-art models on all metrics: BLEU, METEOR, ROUGE-L, and RIBES. Our results were 14.23% higher than the previous best BLEU-4 score, achieved by the Multi-way Tree-LSTM model, as Table 1 shows. Under the mainstream metric BLEU-4, which is more in line with human judgment standards, we achieve better results than the compared methods, which shows that our model is the new state-of-the-art method. At the same time, the new model we create based on Transformer performs dramatically better across all metrics.

The excellent performance of our model on both the Java and Python datasets demonstrates that our model has broad generality and can accommodate the characteristics of different programming languages. Its performance on different metrics shows that our model has distinct advantages in terms of the order of generated language, word accuracy, and the rationality of long sentences.

B. ABLATION TEST

We conducted an ablation test on the Java dataset to determine which mechanism contributes more to our model: the reinforcer, structure encoding, BertC, and BertNL. The experimental results are shown in Table 3. With the addition of BERT, Transformer is greatly improved compared with the original version (see lines 2 to 3). More notably, compared with the token level, BertC at the sentence level shows a small but non-negligible improvement (see line 3). This positive performance confirms that sentence-level code embedding has a better presentation effect than token-level code embedding. We also note that with the functional reinforcer, the model's improvement is compelling (see line 6); the BLEU-4 score of Transformer is increased by 16.6%. Finally, the Transformer model combined with the functional reinforcer, structure encoding, and BertC&NL—i.e., Fret—achieves the best results (see the last line). In addition, the experimental results also support our hypothesis that structure encoding is parallel to AST in terms of functional effect (see lines 4 to 5), since both of them can support

TABLE 1. The experimental results of Fret compared to other state-of-the-art approaches with the Java dataset.

	BLEU-1	BLEU-2	BLEU-3	BLEU-4	METEOR	ROUGE-L	RIBES
Seq2Seq	0.1006	0.0328	0.0195	0.0035	0.0601	0.1709	0.0908
Transformer	0.1190	0.0630	0.0403	0.0265	0.0798	0.1642	0.0973
CODE-NN	0.3032	0.2296	0.2169	0.1338	0.1769	0.3312	0.2669
Tree2Seq	0.2852	0.2177	0.1960	0.1566	0.1666	0.3097	0.2664
DeepCom	0.2992	0.2438	0.2253	0.2109	0.1726	0.3319	0.2854
Multi-way	0.3102	0.2577	0.2207	0.2129	0.1723	0.3390	0.3048
Actor-critic	0.2424	0.1986	0.1669	0.1496	0.1481	0.3203	0.2734
Fret (ours)	**0.3153**	**0.2652**	**0.2473**	**0.2432**	**0.1730**	**0.3615**	**0.3241**

TABLE 2. The experimental results of Fret compared to other state-of-the-art approaches with the Python dataset.

	BLEU-1	BLEU-2	BLEU-3	BLEU-4	METEOR	ROUGE-L	RIBES
Seq2Seq	0.1660	0.0651	0.0100	0.0056	0.0945	0.2838	0.2429
Transformer	0.1745	0.1015	0.0506	0.0152	0.0856	0.3105	0.2694
CODE-NN	0.1846	0.1525	0.1076	0.0710	0.0910	0.3534	0.3113
Tree2Seq	0.2068	0.1713	0.0915	0.0615	0.0715	0.3316	0.2909
DeepCom	0.2290	0.1880	0.1544	0.1355	0.0915	0.3638	0.3232
Multi-way	0.2375	0.2079	0.1705	0.1413	0.1003	0.3876	0.3440
Actor-critic	0.2427	0.1033	0.1526	0.1266	0.0929	0.3813	0.3496
Fret (ours)	**0.2481**	**0.2081**	**0.1813**	**0.1548**	**0.1046**	**0.4012**	**0.3573**

TABLE 3. The results of ablation tests conducted with the Java dataset indicate the effectiveness of our proposed components and mechanisms.

	BLEU-4	METEOR	ROUGE-L	RIBES
Transformer	0.0265	0.0798	0.1642	0.0973
+BERT(token)	0.1059	0.1042	0.1952	0.1646
+BERT(sentence)	0.1232	0.1059	0.2117	0.2042
+AST	0.1565	0.1290	0.2419	0.2447
+Structure encoding	0.1577	0.1353	0.2405	0.2401
+Reinforcer	0.1829	0.1495	0.3154	0.2799
+Reinforcer+ Structure encoding	0.2043	0.1572	0.3302	0.3012
+Reinforcer+BERT+ Structure encoding	**0.2432**	**0.1730**	**0.3615**	**0.3241**

the prediction of code comments based on program structure. However, structure encoding is more concise and excels at saving computing resources. The pre-trained language model, BERT, has a larger training scale; it has a good ability to embed words but faces difficulties in training. From the experimental results (see line 7 in Table 3), we can conclude that the high performance of Fret does not depend simply on a larger training scale but on the collaboration among all modules.

Throughout the experiment, we found that the functional reinforcer provided tremendous support. For example, BertC could understand every sentence of the programming language but lacked the capacity to grasp the critical point, namely, functional guidance; the reinforcer remedied this defect. The reinforcer forces the decoder to produce the correct output by emphasising the parts of the code that are most essential and functional.

C. PARAMETER ANALYSIS

Different lengths of code contain more or less information. Likewise, the length of the code description affects the difficulty of prediction. To explore the effects of code length and comment length on Java summarization results, we conducted two sets of tests. The results are shown in Figs. 5 and 6. The other methods use token-level input, while Fret uses sentence-level code as input. To ensure the fairness of the comparison experiment, we take the number of sentences as the measurement index and then separate sub-datasets according to their code length.

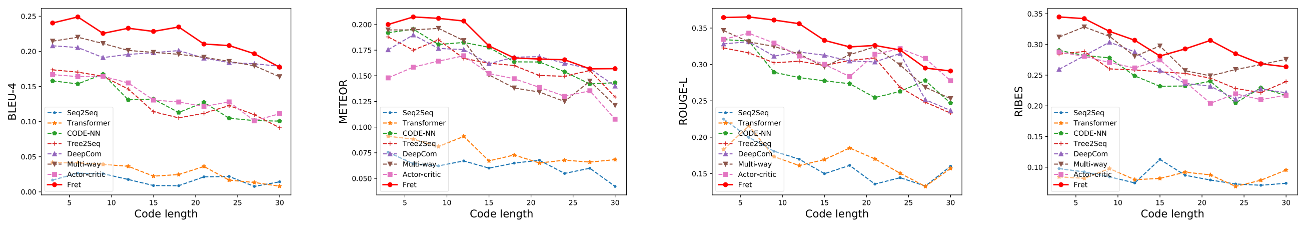
From the experimental results, the code length has relatively little influence on the summary effect, whereas the comment length has a significant impact. As shown in Fig. 5, with the increase of the code length, the score of each indicator shows a slight fluctuation, although the overall trend is downward. Fig. 6 shows quite clearly that as the comment length increases, the value of each metric drops dramatically.

We looked more closely at the data from the Java dataset and found that as the code length grows, there is no clear increasing tendency in the comment length. Therefore, as shown in Fig. 5(a), when the length of the code grows from 15 to 18, the prediction result of Fret is slightly better. In Fig. 5, the downward trend is relatively imperceptible.

When faced with long descriptions, the models seem to be powerless. We observe that paragraphs vary in text, and different nuances and different wording can convey the same meaning. Therefore, it may not be possible to judge the predictive power of a model from the results alone. Of course, we cannot rule out the possibility that the model will start to deviate for various reasons when predicting a long description.

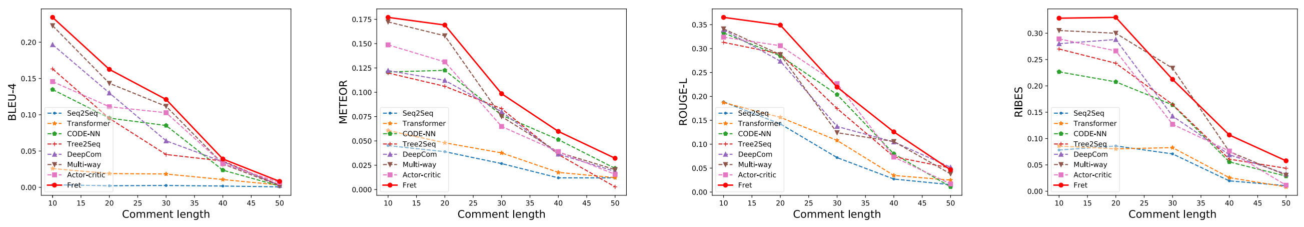
D. CASE STUDY

Statistics cannot fully show the predictive power of a model, so we choose examples with relatively good results on both Java (see Case 1) and Python (see Case 2) datasets



(a) BLEU (b) METEOR (c) ROUGE-L (d) RIBES

FIGURE 5. The results of experiments conducted with different metrics and varying code lengths.



(a) BLEU (b) METEOR (c) ROUGE-L (d) RIBES

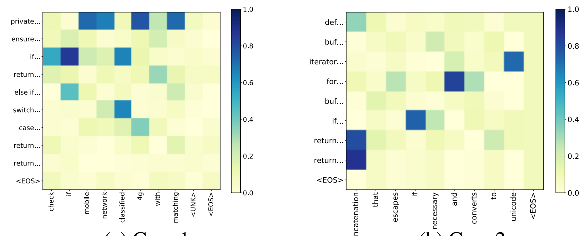
FIGURE 6. The results of experiments with different metrics and varying comment lengths.

TABLE 4. Two summarized examples generated by Fret on the Java and Python datasets.

	Case 1 (Java)	Case 2 (Python)

Code	private boolean matchesMobile4g(NetworkIdentity ident){
	    ensureSubtypeAvailable();
	    if (ident.mType == TYPE_WIMAX) {
	        return true;
	    }
	    else if (matchesMobile(ident)){
	        switch (getNetworkClass(ident.mSubType)) {
	            case NETWORK_CLASS_4_G:
	                return true;
	        }
	    }
	    return false;
	}
		def markup_join(seq):
		    buf = []
		    iterator = imap(soft_unicode, seq)
		    for arg in iterator:
		        buf.append(arg)
		        if hasattr(arg, '__html__'):
		            return Markup(u'').join(chain(buf, iterator))
		    return concat(buf)

Ground truth	Check if mobile network is classified 4G with matching IMSI.	Concatenation escapes if necessary and converts to unicode.
DeepCom	return the mobile work function with bool.	chain the sequence.
Multi-way	ensure the network match mobile network and return a bool result.	jointly concatenate the input sequence.
Fret sum	check if the mobile network class is 4g and return match.	concatenate the sequence if necessary and markup unicode.



(a) Case 1 (b) Case 2

FIGURE 7. The visualised attention thermodynamic chart for Cases 1 and 2.

for in-depth analysis. The experimental results are shown in Table 4 and Fig. 7. The code comments generated in both examples describe the functionality of the code to some extent. As shown in Table 4, the representation of Case 1 on the left is very consistent with the code comment given in the dataset. Case 2, on the right, does not fit so well, but thanks to the reinforcer, it still generalises the functionality of code in a way that seems somewhat acceptable. We also introduce another two state-of-the-art methods, DeepCom and Multi-way, as comparison, from which we can see that the results generated by Fret can not only describe the functionality of the code (e.g., "`concatenate the sequence`" in case 2) but also capture more critical information, such as "`markup unicode`", which was not generated by the other two methods. Another example is that in case 1, both the words "`classified`" and "`matching`" are the most important functional descriptions in ground truth, which is successfully generated by Fret. However, Multi-way generated only the second one, and DeepCom did not capture any of them.

A heatmap of code attention in the decoder is shown in Fig. 7. We used BERT at the sentence level to embed the input. As seen from the charts, some sentences have a great influence on the output. For example, the code in Case 1, "`private boolean matchesMobile4g (NetworkIdentity ident)`," is strongly related to the words "`mobile`," "`network`," "`4g`," and "`matching`." This demonstrates the advantages of sentence-level code embedding: more output can be obtained at less cost (e.g., computing resources, time). Concerning the performance on the Python dataset, Fret also produced accurate results. A program sentence such as "`iterator = imap(soft_unicode, seq)`" has a high relevance to the word "`unicode.`" From any of the above analyses, we can conclude that Fret has a strong operational capability in code summarization tasks.

E. HUMAN EVALUATION

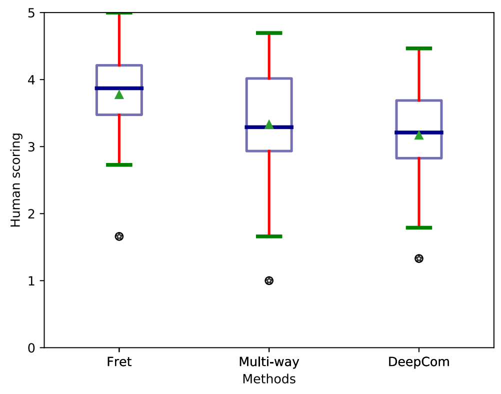
We included the human evaluation to qualitatively analyse the experimental results. As a task of software engineering, whether the experimental results can be applied in practise is one of the criteria for evaluating how good the approach is. Hence, human evaluation scoring has an irreplaceable role as one of the indicators in evaluating results.

1) SURVEY PROCEDURE

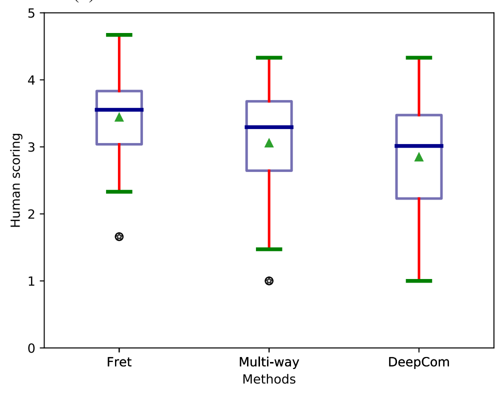
We invited 6 evaluators to score the generated results; all of them were postgraduates or PhD students from Shandong Normal University who majored in computer science with 2-5 years of experience in Java programming and at least one year of experience in Python. We randomly selected 50 generated results from each of the two datasets and divided them into 2 groups, each group with 3 individuals independently involved in scoring. In addition, the allowable scores ranged from 0 to 5, with the highest score (5) indicating that the generated results can accurately describe the code function and the lowest score (0) indicating that the generated results are totally irrelevant to the code function. During the evaluation, ground truth and generated results from DeepCom and Multi-way are provided at the same time.

2) RESULTS

We obtained 300 scores from the evaluators, where 150 are scores of the Java dataset, and another 150 are scores of the Python dataset. The Kappa statistic [50] is used to ensure the consistency of the scores across the groups of evaluators, and the average Kappa statistic is calculated to be 0.72 and 0.67 for the two groups, respectively. Based on the ranking classification of the Kappa statistic, the consistency of the scores across the groups was confirmed and can be used for experimental analysis. The scoring results are shown in Fig. 8. Not only has Fret done well in quantitative analysis, but qualitative analysis also confirms Fret has strong ability in code summarization. Figs. 8 (a) and (b) show that Fret scored significantly higher than Multi-way and DeepCom. At all score levels, including high (4-5), average (2-3) and low (0-1),



(a) Human evaluation score on Java dataset.



(b) Human evaluation score on Python dataset.

FIGURE 8. Human evaluation score on the Java and Python datasets. The score for each code comment is calculated by averaging the scores of three evaluators from the same group.

Fret achieves better results. In addition, Fret's human scoring is more concentrated than Multi-way and DeepCom, which verifies the better robustness of our approach. Moreover, the Mann-Whitney U test also confirmed that the improvements of Fret are statistically significant over the other two methods, with p-values not exceeding 0.002 on both datasets.

VI. DISCUSSION

A. USAGE LIMITATIONS

The reinforcer module can be applied to the great majority of high-level programming languages, depending on a programmer's compliance with the code naming convention. Camel-case naming (e.g., camelCase) and underscore naming (e.g., under_score) are well-known and widely used naming conventions in programming. Because these two naming methods are human-friendly for word segmentation, as presented previously [51], their use is increasingly promoted. Moreover, PyCharm[3] uses PEP8 (Python Enhancement Proposal) by default [52], and it provides soft tips (yellow wavy underlines) for naming during the programming process. The official Python website also has detailed instructions for

[3]https://www.jetbrains.com/pycharm/

using PEP8.[4] This evidence shows that normative naming is supported, encouraged, and popularised in the field of software development.

However, some codes may not comply with such norms. For instance, code written by beginners, code restricted by programming languages (such as assembly language), and code that complies with special development requirements may not be suitable for use with the proposed method. To fill this gap, we plan to add a learnable module in follow-up work to adjust the weight ratio of the reinforcer dynamically; that is, code with formal naming (respectively, non-formal naming) will strengthen (respectively, weaken) the guiding role of the reinforcer. In addition, training for Fret, especially for BERT, is time consuming, and we plan to incorporate active learning and incremental learning [53] into future improvements to reduce the training scale. Overall, we believe that the current solution has apparent advantages, is feasible, and can be applied to most code.

B. ERROR ANALYSIS

Unsatisfactory generation results can be approximately classified into the following three categories: a) vague or inaccurate functional descriptions; b) incorrect variable descriptions; and c) target missing descriptions. A generated example of the first defect is the ``return check result if a file path exist,'' while the ground truth is the ``Checks device for SuperUser permission.'' We speculate that this may be due to the use of abbreviations in the code function names ``checkSu()'', where Fret cannot parse it into the form of ``check'' ``S'' ``U'' at the participle stage, nor can it associate ``S'' ``U'' with ``SuperUser'', resulting in unsatisfactory generation. We plan to improve the accuracy of the generation by including such common programming abbreviations into the dictionary before embedding. Another generated example is ``return 0 as pad string,'', but the ground truth is ``Return n as padded string.'' This may be due to the ternary operator (i.e., b ? x: y) in the code ``return n < 10 ? ``0'' + n: n + ``'';'' being hard to understand, so that Fret gave a result different from expectations. Since the essence of the ternary operator is a simplified "if-else" structure, we plan to recode the ternary operator back to the form of if-else with the help of structure encoding. A generated example of the third classification is ``create an event source impl log.'' However, the ground truth code comment is ``EventSource provides a text-based stream abstraction for Java'', which describes the purpose of applying this method. This may be because the invoking subject for code ``public EventSourceImpl() { LOG.entering(CLASS_NAME, ``<init>''); }'' is not invisible to Fret, so Fret cannot conjecture where and why it is being used. We may address this problem in future work by taking the invoking subject into account.

[4]https://www.python.org/dev/peps/pep-0008/

C. THREATS TO VALIDITY

Our proposed model may suffer two threats to validity, as we identified as follows.

The first is the non-standard or exceptional naming conventions. As we discussed above, the functional extraction of the reinforcer relies on standard naming, so non-standard naming or exceptional naming may affect the accuracy of the reinforcer. We plan to introduce a learnable impact factor to control the decision weights of the reinforcer to circumvent some potentially bad influences in the follow-up work.

The second threat to validity is external validity. There is inevitably some noise in the datasets we use, although we have tried to discard this noise as much as possible in the preprocessing. Additionally, in this paper, only Java and Python programming languages are tested to see how the model performs, so performance on other programming languages is not predictable. In the follow-up work, we plan to test the performance of the model by using more programming languages while collecting more high-quality code from GitHub and making our own dataset to minimise noise in the dataset.

VII. CONCLUSION

In this paper, we propose a novel code summarization model called **Fret**, which stands for **F**unctional **RE**inforced **T**ransformer with BERT. Fret, based on a reinforcer-transformer architecture, is an effective tool for code summarization by emphasising the function of the code, through which Fret can generate more functionally clear, accurate, and comprehensive descriptions of the code. In addition, with the help of Transformer and BERT, Fret alleviates the problem of code understanding and long dependency. Experiments with Java and Python datasets show that the code comments of Fret are significantly better than those of other state-of-the-art methods. In addition, we conduct an ablation test to further verify the effectiveness of the mechanisms we propose. An in-depth discussion of the effects according to the length of the code and comments also highlights Fret's extraordinary ability. Two examples are given to demonstrate the performance of our model in practise. Overall, our model is novel. We obtain improvement in terms of dealing with long dependencies and understanding code, and we also adopt a reinforcer to guide generated comments in a more dynamic functional direction. Generally, Fret achieves remarkable performance in code summarization tasks.

REFERENCES

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986.

[2] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[3] 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.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.

[5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Lang. Technol., (Long Short Papers)*, vol. 1, 2019, pp. 1–16.

[6] 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.

[7] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, "A neural architecture for generating natural language descriptions from source code changes," in *Proc. 55th Annu. Meeting Assoc. Comput. Linguistics (Short Papers)*, vol. 2, 2017, pp. 1–6.

[8] C. Psarras, T. Diamantopoulos, and A. Symeonidis, "A mechanism for automatically summarizing software functionality from source code," in *Proc. IEEE 19th Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Jul. 2019, pp. 121–130.

[9] 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.

[10] J. Moore, B. Gelman, and D. Slater, "A convolutional neural network for language-agnostic source code summarization," in *Proc. 14th Int. Conf. Eval. Novel Approaches Softw. Eng.* Setúbal, Portugal: SCITEPRESS, 2019, pp. 1–12.

[11] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," in *Proc. 53rd Annu. Meeting Assoc. Comput. Linguistics 7th Int. Joint Conf. Natural Lang. Process. (Long Papers)*, vol. 1, 2015, pp. 1–11.

[12] J. Shen, X. Sun, B. Li, H. Yang, and J. Hu, "On automatic summarization of what and why information in source code changes," in *Proc. IEEE 40th Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 1, Jun. 2016, pp. 103–112.

[13] Y. Liu, X. Sun, X. Liu, and Y. Li, "Supporting program comprehension with program summarization," in *Proc. IEEE/ACIS 13th Int. Conf. Comput. Inf. Sci. (ICIS)*, Jun. 2014, pp. 363–368.

[14] Y. Shido, Y. Kobayashi, A. Yamamoto, A. Miyamoto, and T. Matsumura, "Automatic source code summarization with extended tree-LSTM," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2019, pp. 1–8.

[15] Q. Chen, H. Hu, and Z. Liu, "Code summarization with abstract syntax tree," in *Proc. Int. Conf. Neural Inf. Process.* Cham, Switzerland: Springer, 2019, pp. 652–660.

[16] 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. 2269–2275.

[17] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proc. 26th Conf. Program Comprehension (ICPC)*, May/Jun. 2018, pp. 200–20010.

[18] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Softw. Eng.*, vol. 25, no. 3, pp. 2179–2217, May 2020.

[19] 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. (ASE)*, Sep. 2018, pp. 397–407.

[20] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 6563–6573.

[21] Z. Li, Z. Peng, S. Tang, C. Zhang, and H. Ma, "Text summarization method based on double attention pointer network," *IEEE Access*, vol. 8, pp. 11279–11288, 2020.

[22] J. Gong, H. Ma, Z. Teng, Q. Teng, H. Zhang, L. Du, S. Chen, M. Z. A. Bhuiyan, J. Li, and M. Liu, "Hierarchical graph transformer-based deep learning model for large-scale multi-label text classification," *IEEE Access*, vol. 8, pp. 30885–30896, 2020.

[23] Y. Dong, P. Liu, Z. Zhu, Q. Wang, and Q. Zhang, "A fusion model-based label embedding and self-interaction attention for text classification," *IEEE Access*, vol. 8, pp. 30548–30559, 2020.

[24] P. Brown, J. Cocke, S. D. Pietra, V. D. Pietra, F. Jelinek, R. Mercer, and P. Roossin, "A statistical approach to language translation," in *Proc. 12th Conf. Comput. Linguistics*, vol. 1. Stroudsburg, PA, USA: Association for Computational Linguistics, 1988, pp. 71–76.

[25] P. F. Brown, V. J. D. Pietra, S. A. D. Pietra, and R. L. Mercer, "The mathematics of statistical machine translation: Parameter estimation," *Comput. Linguistics*, vol. 19, no. 2, pp. 263–311, Jun. 1993.

[26] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proc. 9th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, Aug. 2013, pp. 532–542.

[27] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 631–642.

[28] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas, "Learning program embeddings to propagate feedback on student code," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 1–10.

[29] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proc. 30th AAAI Conf. Artif. Intell.*, 2016, pp. 1–7.

[30] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. Yu, "Multi-modal attention network learning for semantic source code retrieval," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 13–25.

[31] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proc. 40th Int. Conf. Softw. Eng.*, May 2018, pp. 933–944.

[32] F. Lv, H. Zhang, J.-G. Lou, S. Wang, D. Zhang, and J. Zhao, "CodeHow: Effective code search based on API understanding and extended Boolean model (E)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 260–270.

[33] C. Maddison and D. Tarlow, "Structured generative models of natural source code," in *Proc. Int. Conf. Mach. Learn.*, 2014, pp. 649–657.

[34] M. Allamanis, D. Tarlow, A. Gordon, and Y. Wei, "Bimodal modelling of source code and natural language," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 2123–2132.

[35] 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.

[36] E. Wong, T. Liu, and L. Tan, "CloCom: Mining existing source code for automatic comment generation," in *Proc. IEEE 22nd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, Mar. 2015, pp. 380–389.

[37] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng. (ICSE)*, May 2010, pp. 223–226.

[38] 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.

[39] J. Lei Ba, J. Ryan Kiros, and G. E. Hinton, "Layer normalization," 2016, *arXiv:1607.06450*. [Online]. Available: http://arxiv.org/abs/1607.06450

[40] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing," *IEEE Comput. Intell. Mag.*, vol. 13, no. 3, pp. 55–75, Aug. 2018.

[41] A. V. Miceli-Barone and R. Sennrich, "A parallel corpus of Python functions and documentation strings for automated code documentation and code generation," in *Proc. 8th Int. Joint Conf. Natural Lang. Process. (Short Papers)*, vol. 2, 2017, pp. 1–5.

[42] A. Eriguchi, K. Hashimoto, and Y. Tsuruoka, "Tree-to-sequence attentional neural machine translation," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics (Long Papers)*, vol. 1, 2016, pp. 1–11.

[43] R. Iida, C. Kruengkrai, R. Ishida, K. Torisawa, J.-H. Oh, and J. Kloetzer, "Exploiting background knowledge in compact answer generation for why-questions," in *Proc. AAAI Conf. Artif. Intell.*, vol. 33, 2019, pp. 142–151.

[44] C. Liu, S. He, K. Liu, and J. Zhao, "Curriculum learning for natural answer generation," in *Proc. 27th Int. Joint Conf. Artif. Intell.*, Jul. 2018, pp. 4223–4229.

[45] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*. [Online]. Available: http://arxiv.org/abs/1412.6980

[46] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. 40th Annu. Meeting Assoc. Comput. Linguistics (ACL)*, 2001, pp. 311–318.

[47] S. Banerjee and A. Lavie, "METEOR: An automatic metric for MT evaluation with improved correlation with human judgments," in *Proc. ACL Workshop Intrinsic Extrinsic Eval. Measures Mach. Transl. Summarization*, 2005, pp. 65–72.

[48] C.-Y. Lin, G. Cao, J. Gao, and J.-Y. Nie, "An information-theoretic approach to automatic evaluation of summaries," in *Proc. Main Conf. Hum. Lang. Technol. Conf. North Amer. Chapter Assoc. Comput. Linguistics*, Jun. 2006, pp. 463–470.

[49] H. Isozaki, T. Hirao, K. Duh, K. Sudoh, and H. Tsukada, "Automatic evaluation of translation quality for distant language pairs," in *Proc. Conf. Empirical Methods Natural Lang. Process*. Stroudsburg, PA, USA: Association for Computational Linguistics, 2010, pp. 944–952.

[50] J. Sim and C. C. Wright, "The kappa statistic in reliability studies: Use, interpretation, and sample size requirements," *Phys. Therapy*, vol. 85, no. 3, pp. 257–268, Mar. 2005.

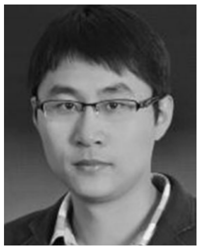
[51] B. Sharif and J. I. Maletic, "An eye tracking study on camelCase and under_score identifier styles," in *Proc. IEEE 18th Int. Conf. Program Comprehension*, Jun. 2010, pp. 196–205.

[52] A. Tosti, N. Cameli, B. M. Piraccini, P. A. Fanti, and J. P. Ortonne, "Characterization of nail matrix melanocytes with anti-PEP1, anti-PEP8, TMH-1, and HMB-45 antibodies," *J. Amer. Acad. Dermatol.*, vol. 31, no. 2, pp. 193–196, Aug. 1994.

[53] Z. Zhou, J. Shin, L. Zhang, S. Gurudu, M. Gotway, and J. Liang, "Fine-tuning convolutional neural networks for biomedical image analysis: Actively and incrementally," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 7340–7351.

RUYUN WANG (Student Member, IEEE) is currently pursuing the bachelor's degree with the School of Information Science and Engineering, Shandong Normal University, Jinan, China, supervised by Chen Lyu. Her research interests include program comprehension, automatic program summarization, and component-based software development.

HANWEN ZHANG received the bachelor's degree in computer science and technology from Shandong Normal University, Jinan, China, in 2018, where she is currently pursuing the master's degree in software engineering, supervised by Chen Lyu. Her current research interests include software reuse, natural language processing, and automatic code comment generation.



GUOLIANG LU (Member, IEEE) received the bachelor's and master's degrees in mechatronic engineering from Shandong University, Jinan, China, in 2006 and 2009, respectively, and the Ph.D. degree in computer science from the Graduate School of Information Science and Technology, Hokkaido University, Sapporo, Japan, in 2013. He is currently an Associate Professor with the School of Mechanical Engineering, Shandong University. His research interests mainly include time series analysis, natural language processing, and intelligent software engineering.

LEI LYU received the Ph.D. degree in computer application technology from the University of Chinese Academy of Sciences, 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.