EDITSUM: A Retrieve-and-Edit Framework for Source Code Summarization

Jia Allen Li
Key Lab of High Confidence Software
Technology, MoE (Peking University)
Beijing, China
lijia@stu.pku.edu.cn

Yongmin Li
Key Lab of High Confidence Software
Technology, MoE (Peking University)
Beijing, China
liyongmin@pku.edu.cn

Ge Li*
Key Lab of High Confidence Software
Technology, MoE (Peking University)
Beijing, China
lige@pku.edu.cn

Xing Hu
School of Software Technology
Zhejiang University, Ningbo, China
xinghu@zju.edu.cn

Xin Xia
Faculty of Information Technology
Monash University, Melbourne, Australia
Xin.Xia@monash.edu

Zhi Jin*
Key Lab of High Confidence Software
Technology, MoE (Peking University)
Beijing, China
zhijin@pku.edu.cn

***Abstract*—Existing studies show that code summaries help developers understand and maintain source code. Unfortunately, these summaries are often missing or outdated in software projects. Code summarization aims to generate natural language descriptions automatically for source code. According to Gros et al., code summaries are highly structured and have repetitive patterns (e.g. "*return true if...*"). Besides the patternized words, a code summary also contains important keywords, which are the key to reflecting the functionality of the code. However, the state-of-the-art approaches perform poorly on predicting the keywords, which leads to the generated summaries suffer a loss in informativeness. To alleviate this problem, this paper proposes a novel retrieve-and-edit approach named EDITSUM for code summarization. Specifically, EDITSUM first retrieves a similar code snippet from a pre-defined corpus and treats its summary as a prototype summary to learn the pattern. Then, EDITSUM edits the prototype automatically to combine the pattern in the prototype with the semantic information of input code. Our motivation is that the retrieved prototype provides a good start-point for post-generation because the summaries of similar code snippets often have the same pattern. The post-editing process further reuses the patternized words in prototype and generates keywords based on the semantic information of input code. We conduct experiments on a large-scale Java corpus (2M) and experimental results demonstrate that EDITSUM outperforms the state-of-the-art approaches by a substantial margin. The human evaluation also proves the summaries generated by EDITSUM are more informative and useful. We also verify that EDITSUM performs well on predicting the patternized words and keywords.**

***Index Terms*—Code summarization, Information retrieval, Deep learning**

I. INTRODUCTION

During software development and maintenance, developers spend around 59% of their time on program comprehension activities [1]–[3]. A code summary provides a concise natural language description for a code snippet, which can help developers understand the program quickly and correctly [4]. Unfortunately, the code summaries are often mismatched, missing or outdated in the software projects [5]. Additionally, manually writing summaries during the development is time-consuming for developers. Therefore, it is important to explore automatic code summarization approaches.

Traditional approaches generate code summaries based on the template-based approaches and information retrieval (IR) based approaches. Template-based approaches [4], [6] firstly extract the keywords from the source code, and then fill the keywords into the predefined templates to generate a code summary. The IR-based approaches use code summaries of similar code snippets as outputs directly. Among these IR-based approaches, they retrieve the similar code snippets by various similarity metrics [7], [8] from open-source software repositories in GitHub or software Q&A sites [9], [10]. Although the traditional approaches are simple, they have achieved good results. This is because code summaries are highly structured and contain many repetitive patterns, e.g., "*return true if...*" and "*create a new...*" [11]. The manually-crafted templates and retrieved summaries provide a lot of reusable patternized words, which play an key role in the code summaries. However, for template-based approaches, manually defining templates is time-consuming and laborious, and requires a lot of expert experience. For IR-based approaches, there may be semantic inconsistencies between the retrieved summary and the input code.

With the development of deep learning, there is an emerging interest in applying neural networks for automatic code summarization. Previous studies [12]–[14] often adopt the encoder-decoder architecture [15] to learn the mapping between words and even the grammatical structure from source code to natural language based on the large-scale corpus. By virtue of the naturalness of the source code [16], [17], these neural models can mine patterns for generating code summaries from a large corpus. Besides the patternized words, a code summary also contains important keywords, which have a low frequency in training data, but are the key to reflecting the functionality of source code (more details can be found in Section II). However, the state-of-the-art nerual models [12]–

* Corresponding authors

[14] perform poorly on predicting keywords. For example, LeClair et al. [14] found 21% summaries written by humans in the test set contain words with the frequency of less than 100, but only 7% summaries generated by their proposed approach contain these words. The lack of keywords leads to the generated summaries suffer a loss in informativeness, which have a negative impact on program comprehension.

Recently, Wei et al. [18] and Zhang et al. [19] proposed two retrieval-based neural models to address the problem of keywords. They used the IR techniques to get the similar code and its summary, and then input the retrieved results and the input code into the encoder. With the assistance of the retrieved summary, their models can accurately generate patternized words. However, their models only treated the retrieved results as auxiliary information and don't solve the problem of keywords.

In this paper, we propose a novel retrieve-and-edit approach EditSum for code summarization. The improvement by template-based approaches proves that the importance of the patterns in code summaries. The improvement by IR-based approaches shows that the summaries of similar code snippets often have the same pattern. So, we treat the summary of similar code as a prototype and extract the pattern from the prototype. Considering the inconsistencies between the prototype and input code, we design a neural network to further edit the prototype automatically based on the semantic information of input code. Our motivation is that the pattern in a prototype tells the neural model "how to say" and the semantic information of input code tells the neural model "what to say".

EditSum consists of two modules: a Retrieve module and an Edit module. In the Retrieve module, given an input code snippet, we use IR techniques to retrieve the similar code snippet from a large parallel corpus and treat the summary of the similar code snippet as a prototype. Then, the Edit module generates a summary by fusing the pattern in prototype and semantic information of input code. Specifically, we propose a sequence-to-sequence (seq2seq) neural network to learn to revise the prototype based on the semantic differences of the input code and the similar code. To represent the semantic differences, we calculate an edit vector by concatenating the weighted sums of insertion word embeddings (words in input code but not in similar code) and deletion word embeddings (words in similar code but not in input code). After that, we revise the prototype summary conditioning on the edit vector to obtain a new summary.

TABLE I: The patterns of summaries in dataset.

Real Samples	**write** a test finish **to** the mesa logger
	write this tilemap **to** an xml file
	write the buffer **to** the output stream
	write grid data **to** the geotiff file
	write cdl representation **to** output stream
Pattern	write____to____
Real Samples	**this method sets** the help button visible
	this method sets the vaule of field
	this method sets a search argument for list
	this method sets the client id
	this method sets the range as a double
Pattern	this method sets________
Real Samples	**convert** an image **to** an array of integer
	convert this ip packet **to** a readable string
	convert a jingle description **to** xml
	convert the specified string **to** a url
	convert the date **to** the given timezone
Pattern	convert____to____

To evaluate our approach, we conduct experiments on a real-world Java dataset. The dataset comes from the Sourcerer repository[1] and has been processed by LeClair et al. [14], including removing duplicates and dividing into training, validation, and test sets by projects. We employ the mainstream evaluation metric BLEU [20], METEOR [21] and ROUGE [22] score that are widely used in summary generation task to evaluate the generated summaries. Experimental results show that EditSum performs substantially better than the IR-based baselines and outperforms the state-of-the-art neural baselines. The human evaluation and qualitative analysis prove the summaries generated by EditSum are informative and useful for developers to understand programs. Besides, we verify that EditSum not only accurately generates patternized words, but also generates more keywords.

Our main contributions are outlined as follows:

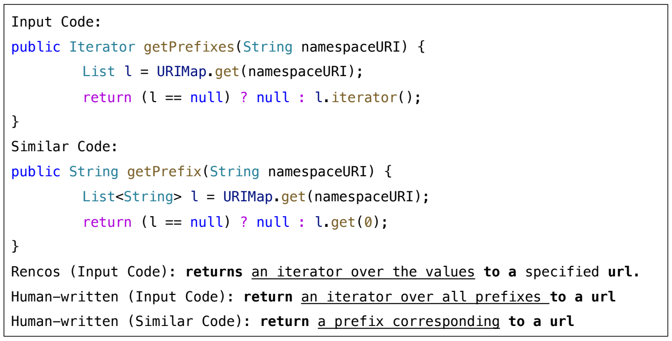
- We propose a novel retrieve-and-edit approach, namely EditSum, for code summarization. We use the summaries of similar code snippets as prototypes to assist in generating summaries.
- We design an effective editing module for summary generation, which can combine the pattern in prototype with the semantic information of code.
- We conduct extensive experiments to evaluate our approach on a large-scale Java dataset. The experimental results show that EditSum substantially outperforms the state-of-the-art approaches.

Paper Organization. The rest of this paper is organized as follows. Section II describes motivating examples. Section III presents our proposed approach. Section IV and Section V describe the experimental setup and results. Section VI and Section VII discuss some results and describe the related work, respectively. Finally, Section VIII concludes the paper and points out future directions.

II. Motivating Examples

A closer look at the code summarization dataset shows that patterns such as "*creates a new*", "*returns true if*", "*load into*", "*convert into*" are very frequent [11]. Table I shows some samples from the dataset provided by LeClair et al. [14]. The bold words are patternized words, and the dashed words denote the keywords. Such a code summary can be regarded as composed of patternized words and keywords. The pattern ensures the readability of the summary, and the keywords reflect the functionality of the source code. A good code summary should contains suitable patternized words and meaningful keywords.

[1] https://www.ics.uci.edu/lopes/datasets/



```
Input Code:
public Iterator getPrefixes(String namespaceURI) {
        List l = URIMap.get(namespaceURI);
        return (l == null) ? null : l.iterator();
}
Similar Code:
public String getPrefix(String namespaceURI) {
        List<String> l = URIMap.get(namespaceURI);
        return (l == null) ? null : l.get(0);
}
Rencos (Input Code): returns an iterator over the values to a specified url.
Human-written (Input Code): return an iterator over all prefixes to a url
Human-written (Similar Code): return a prefix corresponding to a url
```

Fig. 1: An example of the input code and similar code.

However, previous models perform well on predicting the patternized word, ignoring the importance of keywords. As Figure 1 shows, for the input code, we use the open-source search engine *Lucene*[2] to retrieve the most similar code snippet from the training corpus. The retrieval metric is based on the lexical level similarity of the source code.

In Figure 1, the summaries of input code and similar code have the same pattern (*return...to a url*), but there are semantic differences between the similar code and input code. Although the two Java methods are lexically similar, the input code returns all prefixes, while the similar code returns a certain prefix. In Figure 1, the state-of-the-art neural model Rencos [19] can correctly predict the patternized words (e.g., return, to), but it performs poorly on keywords (e.g., prefixes). The code summaries generated by Rencos achieve high scores on the patternized words, but they do not clearly express the purposes of the programs.

In this paper, we address that both pattern and keywords are important for a code summary. Inspired by previous studies, we propose a retrieve-and-edit approach by combining the pattern in existing summaries and the semantic information of input code to generate informative summaries with suitable patterns.

III. Proposed Approach

In this paper, we propose a retrieve-and-edit approach named EditSum for source code summarization, which can combine the strengths of traditional approaches and neural models. The overall framework of our model is shown in Figure 2. Our approach EditSum consists of a Retrieve module and an Edit module and generates a summary in three steps:

Step 1: Selecting a suitable prototype summary. We use a massive training set as the retrieval corpus. Given an input code, the Retrieve module uses the search engine to search for the similar code-summary pair from the corpus. The retrieval process is explained in Section III-A.

Step 2: Extracting the semantic information of the input code. In Figure 2, we mark the lexical differences between the two Java methods. We find that the different words between the two methods reflect their semantic differences to a certain extent, such as “Iteration” vs “String”, and “Prefixes” vs “Prefix”. Therefore, we calculate an edit vector based on the lexical differences between similar code and input code to represent their semantic differences. The details of this part is described in Section III-B.

Step 3: Combining the pattern in prototype with semantic information of input code. To this end, we design a neural edit module to revise the prototype based on the semantic differences between the input code and similar code. The details is presented in Section III-B.

A. Retrieve Module

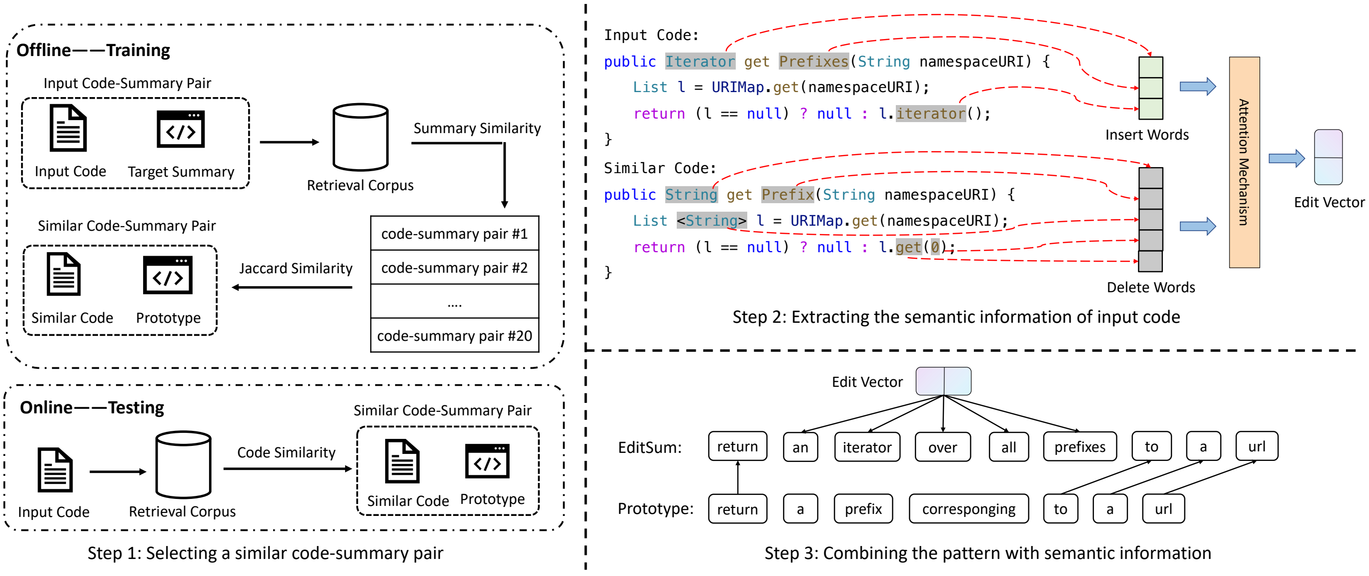
In our approach, the Retrieve module aims to retrieve the similar code-summary pair from a corpus given the input code. Inspired by previous studies [18], [19], we choose the lexical-level similarity as retrieval metric. Specifically, we adopt $BM25$ [23] as the similarity evaluation metric, which is a bag-of-words retrieval function to estimate the relevance of documents to a given query. Given a query and a document, based on TF-IDF [24], the $BM25$ function calculates the term frequency in the document of each keyword in the query and multiplies it by the inverse document frequency of this term. The more relevant two documents have, the higher the value of $BM25$ score. We leverage the open-source search engine $Lucene$ to build the Retrieve module. Since the size of the training set is quite large (over 1.9M), we use it as the retrieval corpus. We first tokenize the source code and summaries and process each code and summary pair into a document, add it to the index library, and store it on disk.

As shown in Figure 2, we use different strategies to select prototypes for training and testing. In testing, we search for the most similar code from the training set and treat its summary as the prototype. During the training phase, as we already know the targrt summary, we first retrieve top-20 code-summary pairs based on the summary similarity. Then, we reserve the retrieved summaries as prototypes whose $Jaccard$ similarity [25] to target summary in the range of [0.3, 0.7]. The $Jaccard$ similarity measures text similarity from a bag-of-words view, that is formulated as

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \quad (1)$$

where A and B are two bags of words and $|\cdot|$ denotes the number of elements in a collection. The motivation behind filtering out summaries with $Jaccard$ similarity < 0.3 is the edit module performs well only if a prototype is lexically similar to its target summary [26]. Besides, we hope the edit module does not copy the prototype so we discard summaries where the prototype and target summary are nearly identical (i.e. $Jaccard$ similarity > 0.7). We do not use code similarity to construct training data, because similar code snippets may correspond to totally different summaries. This is not conducive to our model learning how to revise a prototype. The preliminary experiments also show that constructing training data based on code similarity may cause the model to fail to converge.

[2] https://lucene.apache.org/



```
Input Code:
public Iterator getPrefixes(String namespaceURI) {
    List l = URIMap.get(namespaceURI);
    return (l == null) ? null : l.iterator();
}
Similar Code:
public String getPrefix(String namespaceURI) {
    List <String> l = URIMap.get(namespaceURI);
    return (l == null) ? null : l.get(0);
}
```

Fig. 2: The overall framework of our approach.

B. Edit Module

After that, the key step is to combine the pattern in the prototype and the semantic information of input code to generate a new summary. The structure of the Edit module is shown in Figure 3. Firstly, we utilize the prototype encoder to get the vector representation of prototype. Secondly, we compute the edit vector based on the lexical differences of two code snippets. The edit vector represents the semantic differences between the similar code and input code. Lastly, the summary decoder is used to generate a new summary conditioning on the prototype representation and edit vector.

1) Prototype Encoder: The prototype encoder takes the prototype Y' as input. We first map the one-hot vector of a token w'_i into a word embedding y'_i:

$$y'_i = W_e^\top w'_i, i \in [1, n] \tag{2}$$

where n is the length of prototype, W_e is a trainable word embedding matrix. To leverage the contextual information, we use a bidirectional long short-term memory (Bi-LSTM) [27] unit to process the sequence of word embeddings. At i-th time step, the hidden state h_i of the Bi-LSTM can be represented by:

$$\overrightarrow{h}_i = \mathrm{LSTM}\left(\overrightarrow{h}_{i-1}, y'_i\right); \overleftarrow{h}_i = \mathrm{LSTM}\left(\overleftarrow{h}_{i+1}, y'_i\right) \tag{3}$$

$$h_i = \left[\overrightarrow{h}_i \oplus \overleftarrow{h}_i\right] \tag{4}$$

where $\oplus$ is a concatenation operation. Finally, the prototype Y' is transformed to vector representation $\{h_i\}_{i=1}^n$.

2) Edit Vector: The edit vector z aims to reflect the semantic differences between the input code X and similar code X'. Suppose that X and X' only differ by a single word w. Then one might think that the edit vector z should be equal to the word embedding for w. Generalizing this intuition to multi-word edits, the multi-word insertions can be represented as the sum of the inserted word vectors, and similarly for multi-word deletions [26].

As shown in Figure 3, we define $I = \{w \mid w \in X \wedge w \notin X'\}$ as a insertion word set, and $D = \{w' \mid w' \notin X \wedge w' \in X'\}$ as a deletion word set. Because different words influence the editing process unequally, we represent the differences between X and X' using the weighted sum of word embeddings:

$$f_{diff}\left(X, X'\right) = \sum_{w \in I} \alpha_w \Phi(w) \oplus \sum_{w' \in D} \beta_{w'} \Phi(w') \tag{5}$$

where $\Phi(w)$ is the word embedding of word w and $\oplus$ denotes a concatenation operation. α_w is the weight of a insertion word w, that is computed by the attention mechanism:

$$\alpha_w = \frac{\exp\left(e_w\right)}{\sum_{w \in I} \exp\left(e_w\right)} \tag{6}$$

$$e_w = \mathbf{v}_\alpha^\top \tanh\left(\mathbf{W}_\alpha \left[\Psi(w) \oplus h_n\right]\right) \tag{7}$$

where $\mathbf{v}_\alpha$ and $\mathbf{W}_\alpha$ are trainable parameters, and h_n is the final hidden state of prototype encoder. $\beta_{w'}$ is obtained with a similar process. Then we compute the edit vector z by following linear projection, which can be regarded as a mapping from code differences to summary differences.

$$z = \tanh\left(\mathbf{W} \cdot f_{diff} + \mathbf{b}\right) \tag{8}$$

where $\mathbf{W}$ and $\mathbf{b}$ are two trainable parameters.

3) Summary Decoder: After that, we revise the prototype based on the edit vector to get a new summary. The purpose of the summary decoder is to generate a new summary.

As shown in Figure 3, the decoder takes the prototype representation $\{h_i\}_{i=1}^n$ and the edit vector z as inputs and generate a summary by a LSTM unit with attention. The hidden state of the decoder is compute by

$$s_i = \mathrm{LSTM}\left(s_{i-1}, y_{i-1} \oplus z\right) \tag{9}$$

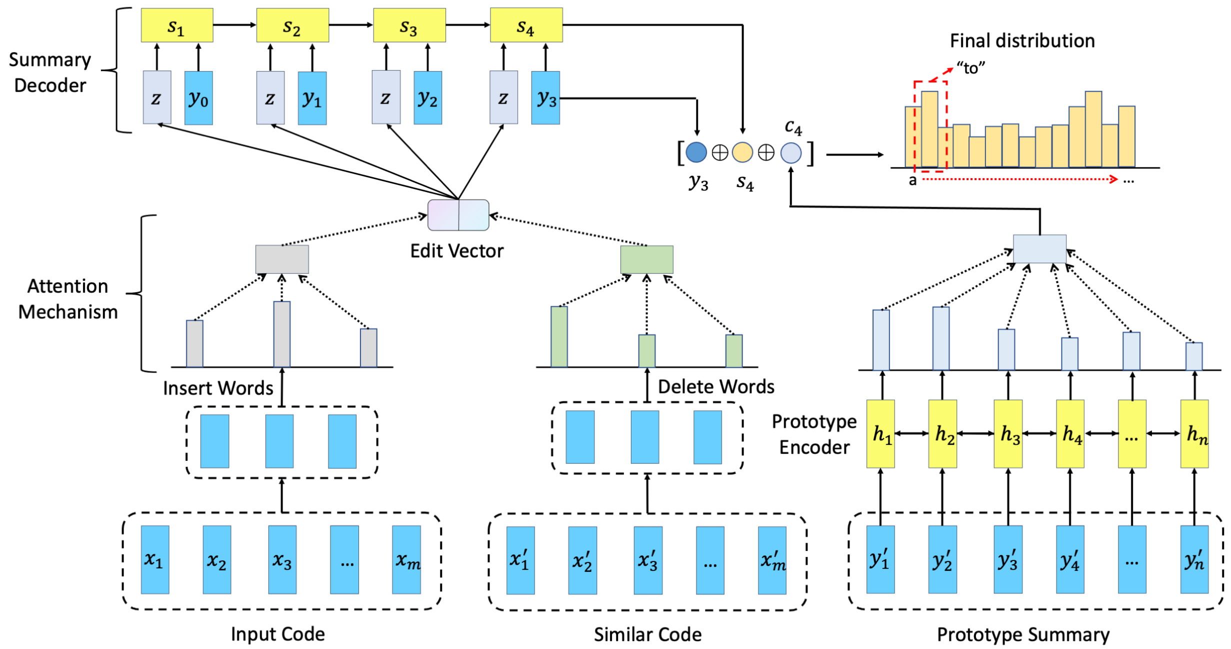


Fig. 3: The structure of the Edit module.

where s_{i-1} means the previous hidden state of the decoder, y_{i-1} is the $(i-1)$-th word embedding of ground-truth summary. We concatenate the edit vector to every input embedding of the decoder, so the edit information can be utilized in the entire generation process.

To introduce the information of the prototype, we then compute a context vector c_i by attention mechanism, which is a weighted sum of prototype representation $\{h_i\}_{i=1}^{n}$:

$$c_i = \sum_{j=1}^{n} \eta_{i,j} h_j \tag{10}$$

where attention weights are obtained by

$$\eta_{i,j} = \frac{\exp(e_{i,j})}{\sum_{k=1}^{n} \exp(e_{i,k})} \tag{11}$$

$$e_{i,j} = \mathbf{v}_\eta^\top \tanh(\mathbf{W}_\eta [h_j \oplus s_i]) \tag{12}$$

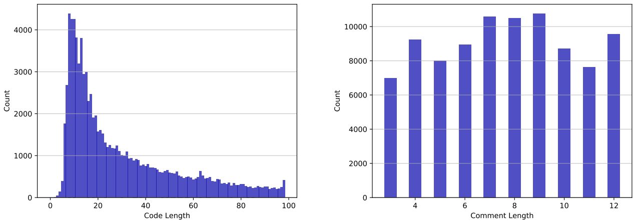
where $\mathbf{v}_\eta$ and $\mathbf{W}_\eta$ are two trainable parameters. Based on the previous word y_{i-1}, hidden state of the decoder s_i and the context vector c_i from prototype, our model compute the probability of i-th token y_i:

$$p(y_i) = \mathrm{softmax}(\mathbf{W_p}[y_{i-1} \oplus s_i \oplus c_i] + \mathbf{b_p}) \tag{13}$$

where $\mathbf{W_p}$ and $\mathbf{b_p}$ are two trainable parameters.

TABLE II: The statistics of datasets.

Dataset	Train	Valid	Test
Count	1,954,807	104,273	90,908
Avg. tokens in code	29.67	29.68	30.17
Avg. tokens in summary	7.594	7.710	7.654



(a) Code length distribution. (b) Summary length distribution.

Fig. 4: Length distribution of test data.

C. Loss Function

During training, EditSum takes a token sequence of the input code X, a summary of the input code Y, a token sequence of the similar code X', and the prototype Y' as inputs, respectively. We optimize parameters of EditSum by maximizing the probability of $p(Y|z, Y')$. The overall objective function of our model is to minimize the following loss function:

$$\mathcal{L}(\theta) = -\sum_{i=1}^{N}\sum_{t=1}^{L} \log P\left(y_t^i \mid z_i, Y_i', y_{<t}^i\right) \tag{14}$$

where θ is all trainable parameters. N is the total number of training instances and L is the length of each ground-truth summary.

During testing, we utilize the prototype encoder to represent prototypes and compute edit vectors. Then, the summary decoder is used to generate directly a summary conditioning on the prototype representation and edit vector in Equation (13).

IV. Experimental Setup

A. Dataset

Following previous studies [14], [18], we conduct experiments on a public large-scale Java dataset[3] provided by LeClair et al. [14]. The raw dataset contains 5.1 million Java methods, which is collected by Lopes et al. [28] from the Sourcerer repository. Because the raw dataset contains a large number of samples (such as repeated and auto-generated code) that are not suitable for evaluating neural models, LeClair et al. cleaned and pre-processed the dataset.

Specifically, they first extracted Java methods and Javadocs from the source code repository. Assuming the first sentence of the Javadoc describes the method's behavior [29], they extracted the first sentence of the Javadoc as the summary of a method and filtered out non-English samples. Considering the auto-generated and duplicate code might affect the evaluation, they removed these samples using heuristic rules [30] and added unique, auto-generated code to the training set. After that, they split camel case and underscore tokens and set them to lower case. Finally, they divided the dataset by project into training, validation, and test set, meaning that all methods in one project are grouped into one set. They argued that the pre-processing of the dataset is necessary for evaluating the performance of neural models. The statistical results of the dataset are shown in Table II. Figure 4 shows the length distribution of source code and summary on the test set. Based on this processed dataset, we construct new instances with the strategies described in Section III-A for our Edit module. Finally, we can obtain 19,714,828 instances for training, 104,273 instances for validation, and 90,908 instances for testing.

B. Implementation Details

Our model is implemented based on the Pytorch[4] framework. We set word embedding and LSTM hidden states to 300 dimensions and 512 dimensions, respectively. We set the batch size to 512 and train the model using Adam [31] with the initial learning rate of 0.001. The learning rate is decayed with a factor of 0.95 every epoch. To mitigate overfitting, we use dropout with 0.5. To prevent exploding gradient, we clip the gradients norm by 5. According to the statistics of the dataset in Table II and Figure 4, the maximum lengths of code and summary are set to 100 and 15, respectively. The vocabulary sizes of the code and summary are 50,000 and 50,000, respectively. The out-of-vocabulary tokens are replaced by UNK. We train the model for a maximum of 30 epochs and perform an early stop if the validation performance does not improve for 5 consecutive iterations. During the testing phase, we use a beam search and set the beam size to 10. We conduct all experiments on one Nvidia V100S GPU with 32 GB memory. Each experiment is run three times and the average results are reported.

C. Evaluation Metrics

Following the previous studies [14], [18], [19], we evaluate all approaches using the metric BLEU [20], METEOR [21], ROUGE-L [22] and ROUGE-W [22]. We regard a generated summary $\hat{Y}$ as a candidate and a huamn-written summary Y as a reference.

BLEU calculates the n-gram similarity between the generated sequence and reference sequence. The BLEU score ranges from 1 to 100 as a percentage value. The higher the BLEU, the closer the candidate is to the reference. It computes the n-gram precision of a candidate sequence to the reference:

$$BLEU-N = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{15}$$

where p_n is the ratio of length n sub-sequences in the candidate that are also in the reference. In this paper, we report the BLEU1-BLEU4 scores. BP is brevity penalty.

METEOR calculates the similarity scores between a pair of sentences by:

$$METEOR = \left(1 - \gamma \cdot frag^{\beta}\right) \cdot \frac{P \cdot R}{\alpha \cdot P + (1-\alpha) \cdot R} \tag{16}$$

where P and R are the unigram precision and recall, $frag$ is the fragmentation fraction. α, β and γ are three penalty parameters whose default values are 0.9, 3.0 and 0.5, respectively.

ROUGE-L computes F-score based on Longest Common Subsequence (LCS). Suppose the lengths of $\hat{Y}$ and Y are m and n, then:

$$P_{lcs} = \frac{LCS(X,Y)}{m}, R_{lcs} = \frac{LCS(X,Y)}{n} \tag{17}$$

$$F_{lcs} = \frac{\left(1+\beta^2\right) P_{lcs} R_{lcs}}{R_{lcs} + \beta^2 P_{lcs}} \tag{18}$$

where $\beta = P_{lcs}/R_{lcs}$ and F_{lcs} is the value of ROUGE-L. ROUGE-W [29] is an improved version of ROUGE-L, which is based on Weighted Longest Common Subsequence (WLCS).

V. Experimental Results

To evaluate our approach, in this section, we aim to answer the following three research questions:

- RQ1: How does the EditSum perform compared to the state-of-the-art neural baselines?
- RQ2: How does the EditSum perform compared to the IR-based baselines?
- RQ3: Does EditSum perform better than previous approaches for tackling the keywords problem?

A. RQ1: EditSum vs. Neural Baselines

1) Baselines: To answer this research question, we compare our approach EditSum to six state-of-the-art neural models.

- **CODE-NN** [12] is the first neural network-based model for code summarization task. It maps the source code sequence into word embeddings, then uses an LSTM unit

[3] http://leclair.tech/data/funcom/

[4] https://pytorch.org/

TABLE III: The performance of our model compared with baselines.

Approaches	Params	BLEU1	BLEU2	BLEU3	BLEU4	METEOR	ROUGE-L	ROUGE-W
Retrieve module	-	32.06	17.83	14.39	12.87	28.62	36.82	25.31
LSI	-	31.38	17.05	13.48	12.07	27.71	35.09	24.02
VSM	-	31.91	17.52	14.02	12.70	28.26	36.21	24.81
NNGen	-	33.48	18.86	14.99	13.44	29.97	38.57	26.07
CODE-NN	36.3M	32.23	14.71	8.558	6.090	29.35	37.64	25.85
DeepCom	37.9M	31.88	16.02	10.10	7.491	31.79	42.45	28.51
attendgru	37.7M	39.00	22.02	14.87	11.27	36.42	48.95	27.96
ast-attendgru	39.7M	39.32	22.19	14.98	11.42	36.99	49.40	33.58
Rencos	57.3M	34.40	19.82	14.34	11.74	34.53	46.35	31.64
Re^{2}Com	28.4M	41.69	25.78	19.70	16.79	38.04	47.65	33.22
EditSum	26.4M	**45.83**	**28.37**	**21.17**	**16.88**	**42.93**	**53.17**	**37.19**

as a decoder to generate summaries, and employs the attention mechanism to introduce information from the word embeddings.

- **DeepCom** [13] is a seq2seq model with an attention mechanism that uses LSTM units as the encoder and decoder. It proposed a SBT algorithm to convert the AST into a token sequence. It is the first model to introduce structural information of source code into code summarization.
- **attendgru** [14] is an encoder-decoder model with an attention mechanism, which takes the code sequence as input and the summary as output.
- **ast-attendgru** [14] is also a seq2seq model with an attention mechanism. Different from attendgru, it introduces the structural information of the source code by using an encoder to process the traversal sequence of AST. It concatenates the information from the two encoders as input to the decoder and generates code summaries.
- **Rencos** [19] is a retrieval-based neural model that augments an attentional encoder-decoder model with the retrieved two most similar code snippets for better source code summarization.
- **Re^{2}Com** [18] is a retrieval-based neural model that uses the summary of the similar code snippet as an exemplar to generate a code summary.

For a fair comparison, we re-implement the attendgru and ast-attendgru based on LSTM units. The embedding size and LSTM states of all baselines are set to 256 dimensions.

2) Results: We calculate the BLEU, METEOR, and ROUGE scores between the summaries generated by different approaches and human-written summaries. The experimental results are shown in Table III. We notice that CODE-NN performs the worst among all approaches. This is because CODE-NN directly uses word embeddings as the input of decoder, and does not further extract the semantic information from the source code. This shows that whether the semantic information of the code can be effectively mined has a greater impact on the performance of the code summarization model. Both DeepCom and attendgru use the encoder-decoder framework, but DeepCom performs worse. This is because the traversal sequence of the AST input by DeepCom is about 7 times longer than the token sequence of code input by attendgru. This also verifies the weakness of LSTM in processing long sequences [32]. The difference between ast-attendgru and attendgru is that the former introduces the structural information in the AST, but the improvement is limited. This is because custom identifiers are removed from the AST used in ast-attendgru, which limits the structural information in the AST. Both Rencos and Re^{2}Com combine the information retrieval technology with neural networks, but the former is less effective. Rencos retrieved two similar code snippets from the corpus and directly used them as input to the model. Re^{2}Com retrieved a similar code from the corpus, and then sent the summary of the similar code into the model as an exemplar. The experimental results show that the summary of similar code contains more valuable and reusable information than similar code that may contain noise. This also proves that it is reasonable for us to use the summaries of similar code as the prototypes.

From Table III, we can see that EditSum performs the best among all neural models, which improves the state-of-the-art Re^{2}Com by 9.93% in BLEU1, 12,85% in METEOR and 11.58% in ROUGE-L. In particular, compared with Rencos and Re^{2}Com, EditSum performs much better on all metrics. This is because Rencos and Re^{2}Com are the ensemble neural models, and they directly enter the retrieved results and the input code into the model. While EditSum regards the prototype summary as an initial draft for post-generation, which provides many reusable patternized words. So, EditSum focuses on learning how to revise the prototype based on the differences between the input code and the similar code. Besides, the number of parameters of EditSum is the smallest among all baselines. It also shows EditSum is efficient.

Compared to other metrics, we find that EditSum has a small improvement on BLEU4. This is because the improvement by EditSum mainly comes from predicting more keywords. However, the average length of the summaries in the test set is 7, and these keywords are mainly 1-3 words. Therefore, EditSum has a great improvement on BLEU1-BLEU3, and a relatively small improvement on BLEU4.

B. RQ2: EditSum *vs. IR Baselines*

1) Baselines: To answer this research question, we compare our approach EditSum to four IR-based baselines.

- **Retrieve module** is a component of our approach, whose details are described in Section III-A. We use the summary of similar code as output directly.
- **Latent Semantic Indexing** (LSI) [8] is an IR technique to analyze the semantic relationship between terms in documents. Given a code snippet, we use LSI to retrieve the similar code from the training set and use its summary as output. The retrieval metric is the cosine distance of the 500-dimensional LSI vector of the source code.
- **Vector Space Model** (VSM) [8] represents the code as a vector using Term Frequency-Inverse Document Frequency (TF-IDF). It uses cosine similarity to retrieve the summary of the similar code from the training set.
- **NNGen** [33] is an approach for generating commit messages based on nearest neighbors algorithm. It first encodes code changes into the form of a ”bag of words”, then uses the cosine distance to select the nearest k code changes. Finally, it chooses the message of the code change with the highest BLEU score as the final result. In this paper, we set k as 5.

2) Results: We calculate the BLEU, METEOR, and ROUGE scores between the summaries generated by different IR-based approaches and human-written summaries. The experimental results are shown in Table III. From Table III, the Retrieve module performs better compared with other IR-based approaches. This shows that it is effective for our Retrieve module to retrieve similar code based on the lexical similarity. LSI and VSM use different ways (LSI vectors and TF-IDF) to map source code into a vector, and their performance is similar. Compared with LSI and VSM, NNGen directly uses BLEU score as the retrieval metric, so it gets a higher BLEU score. It is worth noting that the BLEU3 and BLEU4 score of the IR-based baselines even exceeds that of some neural models (i.e, CODE-NN and DeepCom). This shows that the summaries output by the IR-based approaches have better precision scores of the 3-gram phrase and 4-gram phrase. This proves that the retrieved summaries contains a lot of valuable words, which can be used to generate the new summaries. However, there is still a gap between the summaries output by the IR-based approaches and the human-written summaries due to the differences between the similar code and the input code.

Our model significantly outperforms IR-based baselines in terms of all metrics, which proves the effectiveness of the our Edit module. Compared to the IR-based baselines. our approach EditSum treats the retrieved summary as a prototype, and then revise the prototype conditioning on the semantic differences between similar code and input code. By combining the advantages of neural networks and IR-based approaches, EditSum achieves the best performance.

C. RQ3: Tackling keywords problem

1) Metrics: According to the information retrieve technologies [24], the keywords in the summaries often are informative and are more likely to be low-frequency words. The statistics show 94.8% of tokens in the summary vocabulary of the dataset have a frequency of less than 100. However, as we described in Section I and II, previous neural network models perform poorly on predicting low-frequency words. To measure the ability of our approach on generating keywords, we collect all correctly predicted words on the test set, calculate the frequency of these words on the training set, and count the words with frequencies less than 10, 20, 50, and 100. The correctly predicted words refers to the overlap between the generated summary and the reference summary. For the words with frequencies less than 10 and 20, we manually counted the rate of keywords among these words.

TABLE IV: The number of correctly generated low-frequency words (the rate of keywords in parentheses)

Approaches	≤10	≤20	≤50	≤100
ast-attendgru	262	624	1,575	2,801
Rencos	410	948	2,254	3,791
Re^{2}Com	422 (64.69%)	1,093 (75.21%)	2,808	4,886
EditSum	**476 (74.58%)**	**1270 (86.38%)**	**3066**	**5260**

TABLE V: The results (standard deviation in parentheses) of human evaluation.

Approaches	Naturalness	Informativeness	Usefulness
Retrieve module	1.790 (0.68)	0.778 (0.59)	0.612 (0.12)
ast-attendgru	1.713 (0.76)	1.288 (0.79)	1.108 (0.89)
Rencos	1.822 (0.73)	1.320 (0.36)	1.140 (0.29)
Re^{2}Com	1.860 (0.64)	1.465 (0.52)	1.341 (0.23)
EditSum	**1.933** (0.31)	**1.802** (0.348)	**1.790** (0.29)

2) Results: The statistical results are shown in Table IV. Compared with ast-attendgru, Rencos and Re^{2}Com perform better on predicting the low-frequency words. This shows that the summaries of similar code snippets contain a lot of reusable information. We also can see that EditSum can predict more low-frequency words and more keywords than other baselines, which means that EditSum alleviates the problem of predicting keywords. The goal of learning how to revise a prototype makes our model focuses to generate more keywords.

D. Human Evaluation

1) Metrics: Although the metrics in Section IV-C can calculate the lexical similarity between the generated summaries and the reference summaries, they can not reflect the similarity at the semantic level. Moreover, the ultimate goal of the automatic code summarization model is to help developers understand the functionality of the program. Therefore, we conduct a human evaluation to measure the quality of summaries generated by different baselines on the test set. Following the previous work [18], we measure three aspects, including the *naturalness* (grammaticality and fluency of the generated summaries), *informativeness* (the amount of content carried over from the input code to the generated summaries, ignoring fluency of the text), and *usefulness* (what extent the generated summary is useful for developers to understand code). All three scores are integers, ranging from 0 to 2 (from bad to good). We invite 10 volunteers with 3-5 years of Java development experience and excellent English ability for 1 hour each to evaluate the generated summaries in the form of a questionnaire. The 10 volunteers are computer science Ph.D. students and are not co-authors of this paper. We randomly select 500 samples generated by five models (100 from Retrieve module, 100 from ast-attendgru, 100 from Re^{2}Com, 100 from Rencos, and 100 from EDITSUM). The 500 samples are divided into five groups, with each questionnaire containing one group. We randomly list the summary pairs and the corresponding input code on the questionnaire and remove their labels. Each group is evaluated by two volunteers, and the final result of a pair of summaries is the average of two volunteers. Volunteers are allowed to search the Internet for related information and unfamiliar concepts.

2) Results: The evaluation results are shown in Table V. The standard deviations of all approaches are small, indicating that their scores are about the same degree of concentration. Our model is better than all baselines in three aspects. The Retrieve module can generate more fluent summaries than the ast-attendgru because its outputs are directly retrieved from the training set. We also notice that the scores on *informativeness* of five models are higher than those on *usefulness*. This indicates that the generated summaries really contain information about the code, but this information may be redundant or not completely correct, so they only provide limited help for developers to understand the code.

VI. DISCUSSION

A. Qualitative Analysis

We present three examples generated by different approaches from the test set, as shown in Table VI. These examples show that the summaries generated by EDITSUM have a very high semantic similarity with human-written summaries. From Table VI, previous models cannot generate keywords accurately, and the generated summaries cannot reflect the intention of the programs concisely. For example, in case 1, the aim of this program is to set the color to a darker shade. However, Re^{2}Com simply describes it as setting the selected color to the specified color, which is useless for developers to understand the program. While our model EDITSUM performs well in both patternized words (e.g. set, to) and keywords (e.g. darker shade). Besides, compared with Retrieve module, we can find that our Edit module can make good use of the pattern in the prototype and revise it based on the semantics of the input code.

B. Performance for Different Lengths

We also analyze the performance of different models on different code and summary lengths (number of tokens). We calculate the BLEU score for each sample on the test set and average the scores by length. The experimental results are shown in Figure 5 and Figure 6. From these figures, we can observe that EDITSUM outperforms the Re^{2}Com with different code and summary lengths. As the lengths of the code and summary increase, EDITSUM keeps a stable improvement over Re^{2}Com. The performance of our model is always better than the baseline on the complicated code snippets with a relatively large length. This also shows the robustness of our model.

TABLE VI: Examples of generated summaries.

Case ID	Example

1	`public void drawSelected(){`
	`    if(unselectedColor instanceof Color){`
	`      setPaint(((Color)unselectedColor).darker());`
	`    }else{`
	`      setPaint(java.awt.Color.yellow);`
	`    }`
	`}`
	Retrieve Module: set the series colors to the chart
	ast-attendgru: draws the selected set of the selected color
	Rencos: p method description p
	Re^{2}Com: set the selected color to the specified color
	EDITSUM: set the color to a darker shade
	Human-written: set the color to a darker shade

2	`public void close() throws IOException {`
	`    this.servletInputStream.close();`
	`}`
	Retrieve Module: close the resources used by the work factory
	ast-attendgru: close the underlying servlet
	Rencos: close the server
	Re^{2}Com: close the resources used by the work factory
	EDITSUM: close the underlying stream
	Human-written: close the underlying stream

3	`public int read() throws IOException{`
	`    if(chunkSize==-1){`
	`        return -1;`
	`    }`
	`    if(pos<chunkSize){`
	`        pos++;`
	`        return in.read();`
	`    }else{`
	`        readChunksize();`
	`        pos=0;`
	`        if(chunkSize<=0){`
	`            return -1;`
	`        }`
	`        pos=1;`
	`        return in.read();`
	`    }`
	`}`
	Retrieve Module: read some bytes from the stream
	ast-attendgru: reads the next byte
	Rencos: reads the next byte
	Re^{2}Com: read some bytes from the stream
	EDITSUM: read the next byte of data from this input stream
	Human-written: read the next byte of data from this input stream

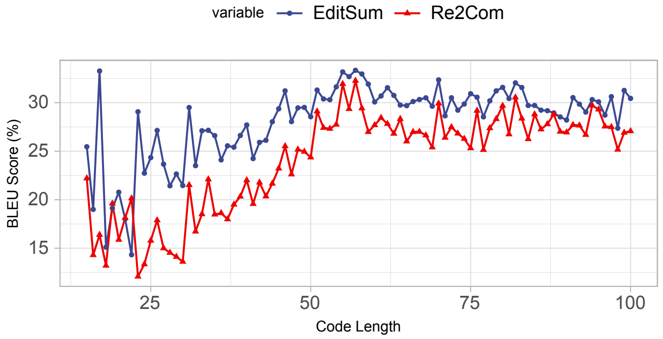


Fig. 5: BLEU scores for different code lengths.

C. Threats to Validity

There are three main threats to the validity of our model. Firstly, we only conducted experiments on a Java dataset. Although Java may not be representative of all programming

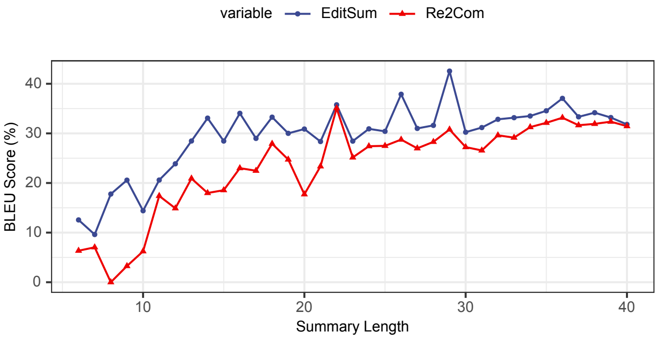


Fig. 6: BLEU scores for different summary lengths.

languages, the experimental dataset is large and safe enough to show the effectiveness of our model. Previous studies [18], [19] also only conducted experiments on this Java dataset. Besides, our model uses only language-agnostic features and can be applied in a drop-in fashion to other programming languages. Secondly, we cannot guarantee that the scores of human evaluation are fair. To mitigate this threat, we evaluate every code-summary pair by two evaluators and use the average score of the two evaluators as the final result. Finally, the Retrieve module retrieves similar code based on lexical similarity. This may result in retrieved code and input code being similar only at the lexical level, but their summaries are quite different. To address this threat, we use a large-scale Java dataset (2M) to increase the scale and diversity of retrieval corpus. The experimental results in Table III prove that the performance of our retrieval module is comparable to the performance of some neural network models (CODE-NN, DeepCom). We also propose an Edit module to alleviate this threat through revising the prototype according to the semantic differences between input code and retrieved code.

VII. Related Work

As an integral part of software development, code summaries describe the functionalities of source code. A concise and clear summary can help developers quickly understand the purpose of the program. However, it is very time-consuming and labor-intensive to write a summary manually. Therefore, more and more researchers are exploring automatic code summarization technology. Automatic code summarization approaches vary from manually-crafted templates [**?**], [6], [34], [35], information retrieval [7]–[10] and neural networks [12]–[14], [18], [19].

A. Template-based Approaches

Early studies generated code summaries based on template-based approaches. Given the signature and body of a method, Sridhara et al. [**?**] identified the content for the summary and generated natural language text that summarizes the method's actions. Moreno et al. [6] determined the class and method stereotypes and used them, in conjunction with heuristics, to select the information to be included in the summaries. Then they generated the summaries using existing lexicalization tools. McBurney et al. [34] utilized the PageRank algorithm [36] to select the important methods in the given method's context and used a template-based system to generate English descriptions of Java methods. Generating summaries based on templates can improve the readability of summaries, but defining templates is a time-consuming task and requires extensive domain knowledge. Besides, templates of different projects cannot be migrated to each other.

B. IR-based Approaches

Information retrieval technologies are also widely used in automatic code summarization. Haiduc et al. [8] represented the source code as a vector using two algorithms (VSM and LSI) and retrieved relevant terms from a code corpus. These relevant terms were integrated into a code summary. Eddy et al. [7] proposed a hierarchical probabilistic model that retrieved relevant terms from the code corpus and fused them into the summaries. Wong et al. [10] applied a token-based code clone detection tool to retrieve similar code snippets in large-scale software repositories. Although promising, IR-based approaches have two main limitations: first, they fail to extract accurate keywords used to identify similar code snippets when identifiers and methods are poorly named. Second, they rely on the size and diversity of the retrieval corpus.

C. Neural Network-based Approaches

Recently, more and more neural networks are applied to generate code summaries. Iyer et al. [12] used a Recurrent Neural Network (RNN) [37] with an attention mechanism as a decoder to generate code summaries and achieved good results on C# and SQL summaries. Because source code contains rich structural information, Hu et al. [13] proposed a neural model named DeepCom to utilize the structural information of code. They proposed a SBT algorithm to convert the AST into a token sequence, then designed a seq2seq model to generate summaries for Java methods based on the AST sequence. LeClair et al. [14] proposed two neural models (attendgru and ast-attendgru) to generate the summaries by combining the sequence information and structure information of the code. Wei et al. [18] proposed an exemplar-based summary generation framework that used the summary of the similar code snippet as an exemplar to assist in generating a target summary. Zhang et al. [19] proposed a retrieval-based neural model that augments an attentional seq2seq model with the retrieved two most similar code snippets for better source code summarization.

Different from the retrieval-based neural models [18], [19], we regard the retrieved summary as a prototype and combine the pattern in prototype with semantic information of input code. While previous models formulate it as a multi-source seq2seq task, in which the input code, prototype, and similar code are all fed to the decoder. The experimental results also prove the superiority of our approach.

VIII. Conclusion and Future Work

In this paper, we argue that code sumaries are composed of patternized words and keywords, and emphasize the shortcomings of previous models in predicting keywords. To alleviate this problem, we propose a retrieve-and-edit approach named EditSum for code summarization. EditSum contains two modules. A Retrieve module for retrieving the similar code snippet. An Edit module treats the summary of similar code as a prototype, and combine the pattern in prototype and semantic information of input code to generate a target summary. We conducted extensive experiments on a large-scale Java dataset. The experimental results show that EditSum substantially outperforms the state-of-the-art neural baselines and the IR-based baselines. Human evaluation and case analysis prove that EditSum can generate concise and informative summaries, which can effectively help developers understand the intent of the programs. The analysis of the experimental results shows that EditSum can generate more keywords and performs well on code and summaries of different lengths. In the future, we will explore how to generate standard and meaningful code summaries for various software projects.

ACKNOWLEDGMENTS

This research is supported by the National Key R&D Program of China under Grant No. 2020AAA0109400, and the National Natural Science Foundation of China under Grant Nos. 62072007, 61832009, 61620106007.

References

[1] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2017.

[2] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, 2005, pp. 68–75.

[3] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on software engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[4] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 43–52.

[5] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, 2005, pp. 68–75.

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

[7] B. P. Eddy, J. A. Robinson, N. A. Kraft, and J. C. Carver, "Evaluating source code summarization techniques: Replication and expansion," in *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 13–22.

[8] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 35–44.

[9] E. Wong, T. Liu, and L. Tan, "Clocom: Mining existing source code for automatic comment generation," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 380–389.

[10] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 562–567.

[11] D. Gros, H. Sezhiyan, P. Devanbu, and Z. Yu, "Code to comment "translation": Data, metrics, baselining & evaluation," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 746–757.

[12] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 2073–2083.

[13] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 200–20 010.

[14] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 795–806.

[15] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in Neural Information Processing Systems*, vol. 27, pp. 3104–3112, 2014.

[16] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[17] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[18] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, "Retrieve and refine: exemplar-based neural comment generation," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 349–360.

[19] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1385–1397.

[20] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[21] S. Banerjee and A. Lavie, "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*, 2005, pp. 65–72.

[22] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.

[23] S. Robertson and H. Zaragoza, *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc, 2009.

[24] S. Qaiser and R. Ali, "Text mining: use of tf-idf to examine the relevance of words to documents," *International Journal of Computer Applications*, vol. 181, no. 1, pp. 25–29, 2018.

[25] P. Jaccard, "The distribution of the flora in the alpine zone. 1," *New phytologist*, vol. 11, no. 2, pp. 37–50, 1912.

[26] K. Guu, T. B. Hashimoto, Y. Oren, and P. Liang, "Generating sentences by editing prototypes," *Transactions of the Association for Computational Linguistics*, vol. 6, pp. 437–450, 2018.

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

[28] C. Lopes, "Uci source code data sets," *http://www. ics. uci. edu/-lopes/datasets/*, 2010.

[29] D. Kramer, "Api documentation from source code comments: a case study of javadoc," in *Proceedings of the 17th annual international conference on Computer documentation*, 1999, pp. 147–153.

[30] K. Shimonaka, S. Sumi, Y. Higo, and S. Kusumoto, "Identifying auto-generated code by using machine learning techniques," in *2016 7th International Workshop on Empirical Software Engineering in Practice (IWESEP)*. IEEE, 2016, pp. 18–23.

[31] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[32] P. Koehn and R. Knowles, "Six challenges for neural machine translation," in *Proceedings of the First Workshop on Neural Machine Translation*, 2017, pp. 28–39.

[33] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: how far are we?"

in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 373–384.

[34] P. W. McBurney and C. McMillan, "Automatic source code summarization of context for java methods," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 103–119, 2015.

[35] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 574–584.

[36] S. Kamvar, T. Haveliwala, and G. Golub, "Adaptive methods for the computation of pagerank," *Linear Algebra and its Applications*, vol. 386, pp. 51–65, 2004.

[37] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Eleventh annual conference of the international speech communication association*, 2010.