Retrieval-based Neural Source Code Summarization

Jian Zhang*
SKLSDE Lab, Beihang University, China
zhangj@act.buaa.edu.cn

Xu Wang*†
SKLSDE Lab, Beihang University, China
wangxu@act.buaa.edu.cn

Hongyu Zhang
The University of Newcastle, Australia
hongyu.zhang@newcastle.edu.au

Hailong Sun*
SKLSDE Lab, Beihang University, China
sunhl@act.buaa.edu.cn

Xudong Liu*
SKLSDE Lab, Beihang University, China
liuxd@act.buaa.edu.cn

ABSTRACT

Source code summarization aims to automatically generate concise summaries of source code in natural language texts, in order to help developers better understand and maintain source code. Traditional work generates a source code summary by utilizing information retrieval techniques, which select terms from original source code or adapt summaries of similar code snippets. Recent studies adopt Neural Machine Translation techniques and generate summaries from code snippets using encoder-decoder neural networks. The neural-based approaches prefer the high-frequency words in the corpus and have trouble with the low-frequency ones. In this paper, we propose a retrieval-based neural source code summarization approach where we enhance the neural model with the most similar code snippets retrieved from the training set. Our approach can take advantages of both neural and retrieval-based techniques. Specifically, we first train an attentional encoder-decoder model based on the code snippets and the summaries in the training set; Second, given one input code snippet for testing, we retrieve its two most similar code snippets in the training set from the aspects of syntax and semantics, respectively; Third, we encode the input and two retrieved code snippets, and predict the summary by fusing them during decoding. We conduct extensive experiments to evaluate our approach and the experimental results show that our proposed approach can improve the state-of-the-art methods.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Source code summarization, Information retrieval, Deep neural network

*Also with Beijing Advanced Innovation Center for Big Data and Brain Computing, Beijing, China.
†Corresponding author: Xu Wang, wangxu@act.buaa.edu.cn.

ACM Reference Format:
Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based Neural Source Code Summarization. In *42nd International Conference on Software Engineering (ICSE '20), May23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380383

1 INTRODUCTION

Source code summarization aims to generate *summaries*, which are concise descriptions of source code and are often presented in the form of code comments. Summaries are important for understanding and maintaining source code. Developers often spend a lot of time on reading and comprehending programs [11, 32, 33, 51] when there is no good software documentation [31, 53]. However, well-commented projects are few [12, 26] and manually writing source code comments is tedious and time-consuming. Also, comments should evolve with the evolution of source code [14], which could incur more maintenance cost. Therefore, it is important to explore automatic source code summarization techniques [42].

Information Retrieval (IR) has been widely used in automatic source code summarization [13, 18, 19, 46, 60, 61]. The IR techniques are used to select appropriate terms from the original code snippets for producing term-based summaries. For example, Haiduc et al. [18, 19] adopted Latent Semantic Indexing (LSI) and Vector Space Model (VSM) to choose good terms from source code and produce source code summaries. Eddy et al. [13] further improved this method through topic modeling. Rodeghero et al. [46] modified the term weights of VSM through eye-tracking and obtained better summaries. Besides the term-based summaries, summaries can be also generated from the comments of similar code. Since code duplication [27, 28, 35] is common in large-scale code repositories, code clone techniques are adopted to retrieve similar code snippets from existing code repositories or Q&A sites. The comments of the similar code can be adapted for generating a new summary [60, 61].

Recent work on Neural Machine Translation (NMT) [4] shows that neural source code summarization is promising [21, 23, 24, 34, 55]. These neural models usually follow an encode-decoder framework. For instance, Iyer et al. [24] proposed an end-to-end neural network, which can encode code snippets with an embedding layer and decode them into natural language summaries through a Long Short Term Memory (LSTM) [20] model with attention mechanism. To incorporate different aspects of a code snippet, many approaches encode the code snippet by extracting its API sequence [23] or Abstract Syntax Tree (AST) [21, 34, 55].

As described above, IR-based methods can effectively leverage the existing terms of the original code or summaries of similar code snippets [13, 18, 19, 46, 60, 61], while NMT-based methods generate summaries word-by-word from the whole corpus by directly maximizing the likelihood of the next word given the previous words [21, 23, 24, 34]. The NMT-based methods generally prefer high-frequency words in the corpus and may have trouble with low-frequency words [3, 64]. That is, if the ground-truth summaries contain words that are rare in the corpus, the NMT-based methods may ignore these words and produce wrong results. A more recent study on commit message generation [37] found that a very simple retrieval-based method can outperform a carefully designed NMT model. What is more, in the community of natural language processing (NLP), Zhang et al. [64] revealed that NMT model is comparatively weak in generating infrequent *n*-grams because it tends to generate more frequent *n*-grams for natural language translation. While in practice, as our experiments show (Section 3), 80.6%/77.3% summaries contain low-frequency words (frequency≤100) for 108,726 Python and 69,708 Java code snippets, respectively. Simply increasing data size cannot help mitigate such problem since the frequencies of common words will increase more quickly than rare words and new low-frequency words may be exposed.

Since the summaries of similar code snippets from IR-based methods are reusable [60, 61], the words in the expected summaries (including the low-frequency ones) are also highly probable to appear in them. For example, for the code snippet in Figure 1, we retrieve two most similar code snippets and their summaries, where one is similar with respect to syntax and the other is similar with respect to semantics. We can see that the low-frequency word "iis" (which is a web server developed by Microsoft) also appears in the retrieved summaries, but is ignored by NMT. In contrast, the high-frequency word "create" can be captured by NMT. Clearly, it is desirable to have a hybrid approach that combines the NMT-based and IR-based methods and simultaneously incorporates both the high-frequency words in the corpus and the low-frequency words in the summaries of the similar code snippets.

In this paper, we propose a novel neural architecture namely **Re**trieval-based **N**eural Source **Co**de **S**ummarizer (*Rencos*), which can take advantages of both the NMT-based methods and the retrieval-based methods. More specifically, we first train an attentional encoder-decoder model to obtain an encoder for all code samples and a decoder for generating natural language summaries; Second, given an input source code snippet, we retrieve the most similar code snippets from the training set. In this work, we obtain two most similar code snippets based on the syntax-level and semantics-level information of the source code, respectively. For syntactic level, we parse code snippets into ASTs and calculate their similarities based on ASTs. For semantic level, we reuse the trained encoder to embed code snippets with semantic vectors and compute the similarities based on these vectors. Finally, during the testing, *Rencos* uses the trained model to encode the input and the retrieved two code snippets as context vectors. It then decodes them simultaneously, and at each time step it adjusts the conditional probability of the next word using the similarity values and

```
def create_app(name, site, sourcepath, apppool=None):
    pscmd = list()
    pscmd.append("New-WebApplication -Name '{0}' -
                    Site'{1}'".format(name,site))
    pscmd.append(" -PhysicalPath '{0}'".format(sourcepath))
    if apppool:
       pscmd.append(" -applicationPool '{0}'".format(apppool))
    cmd_ret = _srvmgr(str().join(pscmd))
    if  cmd_ret['retcode'] == 0:
        if name in list_apps(site): return True
    return False
```

Ground truth: create an iis application .
Syntactic retrieval: remove an iis application .
Semantic retrieval: create an iis virtual directory .
NMT: create the new app .

Figure 1: An example of NMT-based and IR-based source code summarization, where the correct words are marked in red

the conditional probabilities from the retrieved two code snippets. In this way, it trains a classical NMT model and incorporates the retrieved information to enhance the prediction results of the NMT model.

We conduct extensive experiments on two real-world datasets, and the results demonstrate that our approach is better than those using only IR-based methods or NMT-based methods. The proposed approach also outperforms the state-of-the-art work with respect to four widely-used quantitative metrics (BLEU, ROUGE-L, METEOR, and CIDER). Furthermore, we also perform a human evaluation by posting 900 micro-tasks and hiring 129 workers through Amazon Mechanical Turk (AMT). The results further confirm the correctness of the summaries generated by our approach.

Our main contributions are outlined as follows:

- We propose a novel retrieval-based neural architecture to enhance the NMT model for summarizing source code with the help of most similar code snippets. To the best of our knowledge, this is the first work that combines retrieval-based and NMT-based methods in source code summarization;
- We conduct extensive experiments to evaluate our approach on two real-world datasets. We also perform a human evaluation through Amazon Mechanical Turk (AMT). All results confirm that the proposed approach is effective and outperforms the state-of-the-art methods.

The remainder of this paper is organized as follows. Section 2 describes the details of our approach. We evaluate our approach in Section 3. The threats to validity and related work are presented in Section 4 and Section 5, respectively. Finally, we conclude our work in Section 6.

2 OUR APPROACH

2.1 Overview

In this work, we propose one retrieval-based neural source code summarization approach (*Rencos*), which can combine the strengths of NMT model and retrieval-based methods for better source code summarization. Unlike retrieval-based neural models in NLP [15,

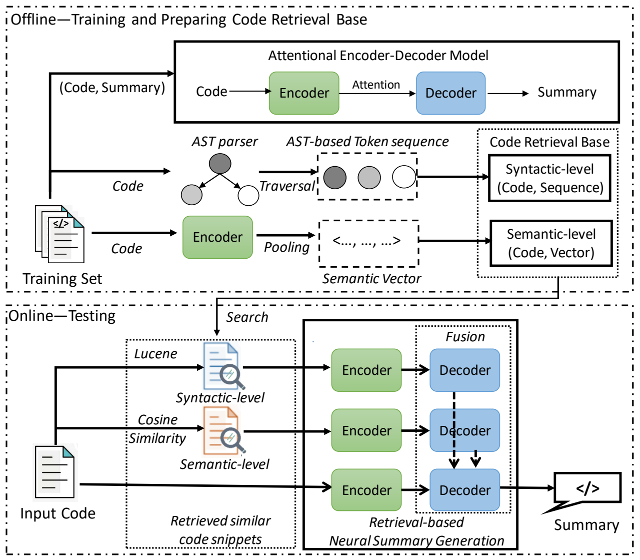

Figure 2: The overall framework of our approach

64], our approach does not need additional encoders for the retrieved code with joint training [15], or the translation pieces (*n*-grams in the retrieved target sentences) [64]. Our approach includes an attentional encoder-decoder model, two similarity-based code snippet retrieval components: syntactic-level and semantic-level, and the retrieval-based neural summary generation. The online and offline workflow is shown in Figure 2.

Specifically, during training, we first collect a large corpus containing source code snippets and their natural language summaries to train an attentional encoder-decoder model (Section 2.2). Such an encoder-decoder model can be directly used for all inputs and retrieved code snippets during the testing phase. In order to accelerate the code retrieval efficiency during testing, we offline parse all the code snippets from the training set into ASTs and turn them into syntactic-level token sequences by tree traversal. We also use our trained encoder to embed code into semantic vectors by pooling. All the code sequences and semantic vectors are stored as the code retrieval base.

When testing online, given one input code snippet, we search for two most similar ones from the code retrieval base (Section 2.3). One is obtained by the commonly-used retrieval engine Lucene[1] on the syntactic-level token sequences (i.e. syntactic-level), and the other is selected with the highest cosine similarity based on the semantic vectors (i.e. semantic-level). Then we produce the final summary by fusing the results of the two retrieved code snippets in our retrieval-based neural summary generation (Section 2.4).

2.2 Attentional Encoder-Decoder Model

Similar to the existing neural source code summarization methods [21, 23, 24, 34, 55], we build and train an attentional encoder-decoder model, which includes only one encoder for avoiding the high overhead of jointly training additional encoders for the retrieved code [15]. This model is trained only once, but can be used in three places: encoding and decoding the input code snippets of the testing set and the retrieved code snippets from the training set, and helping retrieve the most similar code snippets at the semantic level (as shown in Figure 2). In this way, our approach is simple and efficient without extra training.

For the encoder, suppose there is one code snippet c consisting of a sequence of words $w_1, \cdots, w_n$, an embedding layer is first used to initialize these words by vectors:

$$x_i = W_e{}^\top w_i, i \in [1, n], \tag{1}$$

where n is the length of code snippet c, W_e is the embedding matrix. Then we use LSTM units [63] to encode the sequence of vectors $x = x_1, \cdots, x_n$ into hidden states $h_1, \cdots, h_n$. For simplification, we denote the LSTM unit as:

$$h_t = LSTM(x_t, h_{t-1}). \tag{2}$$

We further employ Bidirectional LSTM (Bi-LSTM) [49] to capture semantics in front and behind of the current position.

When decoding the code snippet and generating the i-th summary word, an attentional decoder first computes the context vector v_i over the sequence of hidden states $h_1, \cdots, h_t, \cdots, h_n$ according to the following formula:

$$v_i = \sum_{j=1}^{n} a_{ij} h_j. \tag{3}$$

Here a_{ij} is the attention weight of h_j and computed by:

$$a_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{n} exp(e_{ik})}, \tag{4}$$

$$e_{ij} = a(s_{i-1}, h_j), \tag{5}$$

where s_{i-1} means the last hidden state of the decoder. In Equation 5, we use a Multi-Layer Perception (MLP) unit [44] as an alignment model a [4]. At time i, the hidden state s_i of the decoder is updated by:

$$s_i = LSTM(s_{i-1}, y_{i-1}), \tag{6}$$

where y_{i-1} is the previous input vector. To jointly take into account past alignment information, we exploit the input-feed approach proposed by [40] which concatenates v_{i-1} with inputs y_{i-1}.

$$p(y_i | y_1, \ldots, y_{i-1}, c) = g(y_{i-1}, s_i, v_i), \tag{7}$$

where g is the generator function, which applies a MLP layer along with $softmax$. We also adopt the beam search algorithm [59], that is, at each time step t, we keep top-B best hypotheses where B is the beam size.

Training such an attentional encoder-decoder model is to minimize the loss function:

$$\mathcal{L}(\theta) = -\sum_{i=1}^{N} \sum_{t=1}^{L} \log P(y_t^i | y_{<t}^i, c), \tag{8}$$

where θ is the trainable parameters, N is the total number of training instances and L is the length of each target sequence. After training with the pairs of code snippets and corresponding summaries, we obtain an encoder which can represent code snippets, and a decoder which is able to predict summaries word-by-word by maximizing the conditional probabilities of the next words in Equation 7. As a result, if we only use such an attentional encoder-decoder model for source code summarization like the existing

[1]http://lucene.apache.org

work [21, 23, 24, 34, 55], the model will prefer the high-frequency words and ignore the low-frequency words [3, 64] such as "iis" in the example of Figure 1.

2.3 Similar Code Retrieval

Because of the existence of code duplication [27, 28, 35], previous retrieval-based source code summarization work reused the summaries of similar code snippets [60, 61]. Their experience suggests that the words in expected summaries (including the low-frequency ones) are also highly probable to appear in the summaries of similar code. Based on the encoder-decoder model above, our approach can incorporate the knowledge of similar code snippets and their summaries from the training set for better prediction of low-frequency words. For inclusion of as many useful low-frequency words as possible, we want to retrieve such similar code snippets from different aspects. In this work, since the training set is very large, we aims to design two efficient source code retrieval components based on the syntax structure and the embedding semantics of source code. More code retrieval methods can be easily integrated to our approach as well.

2.3.1 *Syntactic-level Source Code Retrieval.*

Unlike plain texts, source code has its own syntax structure and the syntactic information is important for finding similar source code.

Such syntactic knowledge has been explored by previous work on code clone detection [7, 25] based on costly tree-based matching methods. Recent studies on the source code summarization [21, 34, 55] also considered the syntax structure of source code by adding the AST-based RNN encoder for joint supervised training. In our work, since the training set is very large and the unsupervised method is preferred, we efficiently measure the syntactic similarities of code snippets based on the token sequences of ASTs without training.

Specifically, given one input code snippet from the testing set and any one from the training set, we parse them to two ASTs and then calculate the similarity between the two ASTs. However, if the training set is very large (For example, the size N is more than 50k in our experiment), it will incur much overhead to directly compute the similarities based on tree matching algorithms such as Tree Edit Distance [9], since the computational complexity is $O(N^3)$. A suitable compromise is to convert ASTs into token sequences through tree traversal. Therefore, we parse the input code snippet and all of the training set to ASTs, and further obtain their sequence representations using the preorder traversal, as depicted in Figure 2. Based on these sequences, we use an off-the-shelf and widely-used search engine Lucene to efficiently retrieve the most similar code snippet from the training set at the syntactic level.

2.3.2 *Semantic-level Source Code Retrieval.*

Recently the neural network based methods [56, 57, 65, 67], which encode source code into semantic vectors, have shown their superiority in capturing code semantics. But they need training with additional computation burden. In contrast, we reuse the trained encoder mentioned in Section 2.2. As described above, the Bi-LSTM based encoder is capable of capturing the sequential information of source code and embedding the semantics into hidden state vectors.

Given a code snippet c, we encode it by the Bi-LSTM encoder and get a sequence of hidden state vectors $[h_1, \ldots, h_n] \in \mathbb{R}^{n\times 2k}$, where k is the vector dimension and n is the length of code snippet c. Similar to [16], we apply a global max pooling operation over the vector sequence to obtain the semantic representation $r_c \in \mathbb{R}^{1\times 2k}$ as follows:

$$r_c = [max({h_i}^1), \cdots, max({h_i}^{2k})], i = 1, \cdots, n. \tag{9}$$

For a testing code snippet c_{test} and any code snippet c_i from the training set, we compute their cosine similarity:

$$cosine(\overrightarrow{r_{test}}, \overrightarrow{r_i}) = \frac{\overrightarrow{r_{test}} \cdot \overrightarrow{r_i}}{\|\overrightarrow{r_{test}}\| \|\overrightarrow{r_i}\|} \in [-1, 1], i \in [1, N]. \tag{10}$$

where N is the size of the training set. At last, the code snippet with the highest score is selected as the most similar one in the training set.

Based on both source code retrieval components, since the training set is usually large, we can offline prepare a code retrieval base which stores the pairs of code and AST-based token sequence and the pairs of code and semantic vector before testing, as depicted in Figure 2. Given one new code snippet for testing online, we retrieve the syntactic-level similar code snippet based on the efficient Lucene search engine, and search for the semantic-level similar one by computing the simple cosine similarity of vectors that is quite fast.

It is not trivial to choose the most similar code snippet (i.e. top-1) at the syntactic or semantic level. On the one hand, more top-k ($k > 1$) similar candidates may be noisy especially when their similarities are not high. On the other hand, if a similarity threshold T is set to filter code snippets with relatively low similarities, it may eliminate the useful low-frequency words as well. Because it is hard to find such a static threshold to distinguish whether one code snippet includes useful low-frequency words or not. We find that selecting the most similar code snippet without the threshold (i.e. $k = 1$ and $T = 0$) has a good trade-off between the inclusion of useful low-frequency words and the noisy information, which will be discussed in Section 3.

2.4 Retrieval-based Neural Summary Generation

After training our attentional encoder-decoder model and retrieving the two most similar code snippets from the training set, we will predict and generate summary sentences online for the code snippets from the testing set. Intuitively, we augment the attentional encoder-decoder model (for high-frequency words) with the retrieved similar code snippets (for low-frequency words). One naive solution is to enhance the probabilities of all the words in the similar code snippets by the similarity degrees, but it may include noisy words such as "remove" in the example (Figure 1). Zhang et al. [64] tried to filter these noisy words with the translation pieces based on the source and target word alignment. However, unlike the neural machine translation, the code snippets and their summaries usually miss the property of word alignment, which will be discussed in Section 3. In contrast, besides the similarities of code snippets, we also consider the conditional probability of each word during decoding to help eliminate the possible noisy words that usually have low conditional probabilities. As Figure 3 shows, we take the

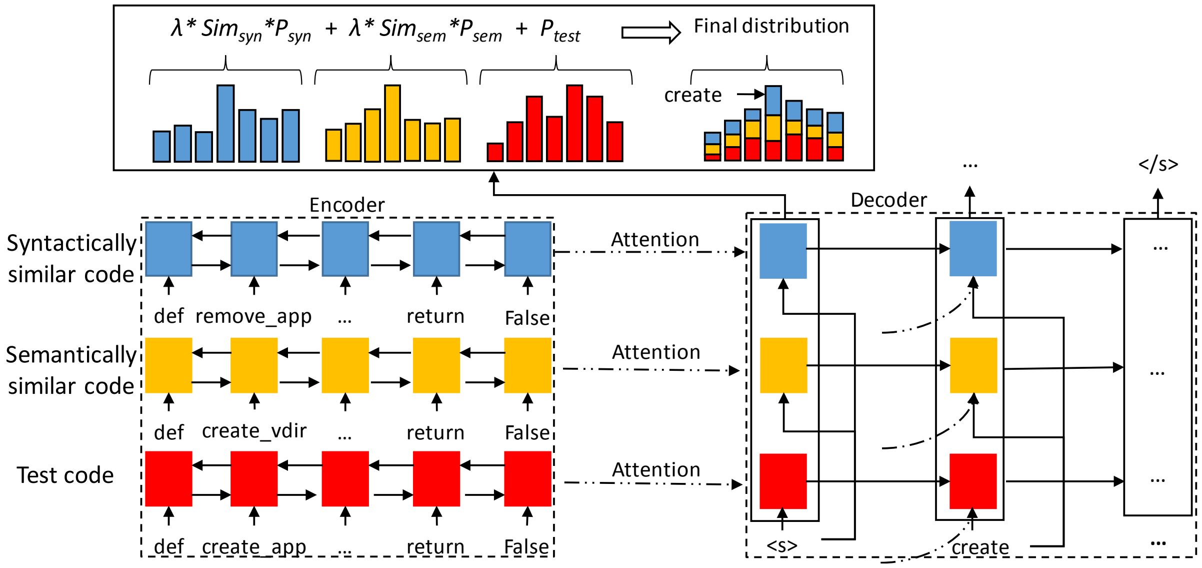
λ* Sim_syn*P_syn + λ* Sim_sem*P_sem + P_test
Final distribution
create
Encoder
Decoder
Syntactically similar code
Semantically similar code
Test code
def remove_app ... return False
def create_vdir ... return False
def create_app ... return False
Attention
Attention
Attention
<s>
create
...
</s>

Figure 3: Retrieval-based Neural Summary Generation. $\langle s\rangle$ **and** $\langle /s\rangle$ **represent the begin and end symbols, respectively**

example in Figure 1 as illustration. Based on the encoder and decoder of our trained model in Section 2.2, our retrieval-based neural summary generator encodes each testing code snippet and its two most similar ones simultaneously, gets their context vectors with attentional mechanism and decodes them to predict the summary words by fusing the conditional probabilities and similarities.

Specifically, for one testing code snippet c_{test}, we search and retrieve its two most syntactically and semantically similar code snippets c_{syn} and c_{sem} from the training set mentioned in Section 2.3. Then we encode these three code snippets by our trained attentional encoder-decoder model (Section 2.2) in parallel. Based on the obtained three sequences of hidden states H_{test}, H_{syn} and H_{sem} for c_t, c_{syn} and c_{sem}, at each time step t during decoding, we can compute the attention weights to produce context vectors according to Equation 3, and then calculate the conditional probabilities to predict the next word by Equation 7. For simplification, we denote these conditional probabilities by $P_{test}(y_t|y_{<t})$, $P_{syn}(y_t|y_{<t})$ and $P_{sem}(y_t|y_{<t})$, respectively.

In order to enhance the prediction performance of the original attentional encoder-decoder model, we leverage the retrieved code snippets and fuse all these three conditional probabilities. A straightforward way is to simply add them together. However, when the similarity between c_{test} and c_{syn} (or c_{sem}) is too low (even c_{syn} is the retrieved most similar one in the training set), $P_{syn}(y_t|y_{<t})$ may have a negative impact on predicting correct summary words. Thus we should consider the values of similarities for the fusion as well. Directly reusing the similarities in the code retrieval (such as results in Equation 10) is not reasonable, because we calculate these similarities with different methods and from different aspects: one at the syntactic level and the other at the semantic level. To solve this problem, we normalize the similarities to make them comparable based on the text edit distances $d(c_{test}, c_{syn})$ and $d(c_{test}, c_{sem})$ using dynamic programming [8], which is also adopted in [64]:

$$Sim(c_{test}, c_{ret}) = 1 - \frac{d(c_{test}, c_{ret})}{max(|c_{test}|, |c_{ret}|)}. \tag{11}$$

where c_{ret} denotes any retrieved code snippet. With these normalized similarities, we combine the conditional probabilities to get the final conditional distributions as follows:

$$\begin{aligned} P_{final}(y_t|y_{<t}) =& P_{test}(y_t|y_{<t}) + \\ & \lambda \cdot Sim(c_{test}, c_{syn}) P_{syn}(y_t|y_{<t}) + \\ & \lambda \cdot Sim(c_{test}, c_{sem}) P_{sem}(y_t|y_{<t}))). \end{aligned} \tag{12}$$

where λ is a hyper-parameter that can be manually tuned.

Based on the final conditional distributions above, we can predict the next word one-by-one and finally generate the whole summary sentence, which incorporates the knowledge from the retrieved code snippets.

3 EXPERIMENTS

In this section, we conduct experiments to evaluate the effectiveness of our proposed approach and compare it with several state-of-the-art methods from Software Engineering (SE) and Natural Language Processing (NLP) communities.

3.1 Experimental Setup

We conduct experiments on two public large-scale datasets in Python and Java, respectively. For simplification, we call them PCSD (Python Code Summarization Dataset) and JCSD (Java Code Summarization Dataset). PCSD is provided by Barone et al. [6], which contains Python functions and their comments from open source repositories in GitHub[2]. It uses docstrings (document strings) as natural language descriptions, which we call comments. This dataset includes 108,726 code-comment pairs and has been used for training and evaluation in [55]. JCSD is a dataset consisting

[2]https://github.com

of Java methods and comments collected by Hu et al. [23] from popular repositories in GitHub. The comment of JCSD is the first sentence extracted from its Javadoc, which is similar to the practice in [17]. The original dataset includes two parts for API sequence summarization and code summarization, and we choose the latter part with 69,708 code and comment pairs as our JCSD dataset. For fair comparison, we split PCSD into the training set, validation set and testing set with fractions of 60%, 20% and 20%, respectively, and split JCSD in proportion of 8 : 1 : 1 to keep the same split settings as baselines [23, 55]. To make the training and testing sets disjoint, we remove the duplicated samples from the testing set. In common with [23, 55], we set the length limits (in terms of #words) of code snippets and summaries (i,e., 100 and 50 for PCSD, 300 and 30 for JCSD), since such settings can cover most of their original lengths. The statistics of these two datasets are described in Table 1, where MaxL, AvgL and UniT are the maximum length, the average length, and the total number of unique tokens, respectively. We also consider the low-frequency words in the summaries whose frequencies are no more than 10 and 100. NumW is the number of low-frequency words and NumS is the number of summaries which contain at least one low-frequency word. The percentages of low-frequency words in all unique words of summaries, and the percentages of summaries containing at least one low-frequency word are also shown in the parentheses.

For tokenizing source code, we use the libraries *tokenize*[3] and *javalang*[4] to get tokens of Python and Java code snippets, respectively. Like [1], we further split code tokens into subtokens by snake case or camel case to reduce data sparsity. Meanwhile, we use the *ast*[5] library and *javalang* to obtain their corresponding ASTs. To tokenize summaries, we utilize the *tokenize* module of NLTK toolkit [39]. Besides, we limit the maximum vocabulary size of source code and summary to 50k since too large vocabulary may lead to worse performance. The out-of-vocabulary words are replaced by *UNK*. For better comparison, we also apply such tokenization to other approaches to avoid the potential influence.

We implement our approach based on the open-source system OpenNMT[6] [30]. During training, we set the embedding size to 256 and the dimensions of hidden states in LSTM to 512. The batch size is set to 32 and the maximum iterations is 100k. We adopt the widely-used Adam [29] as the optimizer with learning rate 0.001 for training our model. The beam size is set to 5, and the hyper-parameter λ is set to 3. All the above hyper-parameters are determined based on the validation set by selecting the best ones among some alternatives. All the experiments are conducted on one Ubuntu 16.04 server with 16 cores of 2.4GHz CPU, 128GB RAM and a Titan Xp GPU with 12GB memory.

3.2 Evaluation Metrics

Similar to existing work [21, 23, 24, 55], we evaluate the performance of different approaches using common metrics including BLEU [45], METEOR [5], ROUGE-L [36] and CIDER [54]. These metrics are also popular in machine translation, text summarization, and image captioning.

Given the generated summary X and the ground-truth Y, BLEU measures the n-gram precision between X and Y by computing the overlap ratios of n-grams and applying brevity penalty on short translation hypotheses. BLEU-1/2/3/4 correspond to the scores of unigram, 2-grams, 3-grams and 4-grams, respectively. The formula to compute BLEU-$N (N = 1, 2, 3, 4)$ is:

$$BLEU\text{-}N = BP \cdot \exp \sum_{n=1}^{N} \omega_n \log p_n,$$

where p_n is the precision score of the n-gram matches between candidate and reference sentences. BP is the brevity penalty and ω_n is the uniform weight $1/N$.

For a pair of sentences to be compared, METEOR creates a word alignment between them and calculates the similarity scores by

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

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 is widely used in text summarization [43, 50] and provides F-score based on Longest Common Subsequence (LCS). Suppose the lengths of X and Y are m and n, then:

$$P_{lcs} = \frac{LCS(X, Y)}{m}, R_{lcs} = \frac{LCS(X, Y)}{n}, F_{lcs} = \frac{(1 + \beta^2) P_{lcs} R_{lcs}}{R_{lcs} + \beta^2 P_{lcs}},$$

where $\beta = P_{lcs}/R_{lcs}$ and F_{lcs} is the value of ROUGE-L.

CIDER is usually used for measuring the quality of image captions, which considers the frequency of n-grams in the reference sentences by computing the TF-IDF weighting for each n-gram. $CIDER_n$ score for n-gram is computed using the average cosine similarity between the candidate sentence and the reference sentences. The final result is calculated by combining the scores for different n-grams (up to 4).

Note that the scores of BLEU, METEOR and ROUGE-L are in the range of [0,1] and usually reported in percentages. But CIDER is not between 0 and 1, and thus it is reported in real values.

3.3 Baselines

We compare our approach with existing work on source code summarization. They can be divided into two groups: Retrieval-based and NMT-based approaches. In addition, we also compare with a state-of-the-art algorithm that combines the above two kinds of approaches although it is originally proposed for natural language translation. We use the default settings of these approaches unless otherwise stated.

3.3.1 *Retrieval-based approaches.*

- **LSI** is a text retrieval technique for analyzing the latent meaning or concepts of documents. It is used in [18] to choose the most important terms of code snippets as term-based summaries. In order to generate human-readable summary sentences, for any testing code snippet, we use LSI to retrieve the most similar one from the training set and take its summary as the result. The similarity is computed based on the LSI-reduced vectors and cosine distance, and we set the vector dimension to be 500.

[3]https://docs.python.org/2/library/tokenize.html
[4]https://pypi.org/project/javalang
[5]https://docs.python.org/2/library/ast.html
[6]https://github.com/OpenNMT/OpenNMT-py

Table 1: The statistics of two datasets

Dataset	Source Code Length (#words)			Summary Length (#words)			Word Frequency ≤10		Word Frequency ≤100	
	MaxL	AvgL	UniT	MaxL	AvgL	UniT	NumW	NumS	NumW	NumS
PSCD	157,116	133.1	481,756	333	9.9	37,111	32,093(86.5%)	46,481(42.8%)	36,003(97.0%)	87,626(80.6%)
JSCD	4842	99.9	230,336	670	17.1	35,535	30,342(85.4%)	34,207(41.4%)	34,223(96.3%)	63,954(77.3%)

Table 2: Method comparison for source code summarization

Methods	PCSD							JCSD						
	BLEU-1/2/3/4(%)				METEOR(%)	ROUGE-L(%)	CIDER	BLEU-1/2/3/4(%)				METEOR(%)	ROUGE-L(%)	CIDER
LSI	36.3	23.6	20.1	17.6	17.2	40.0	1.982	31.4	22.5	19.3	17.3	14.4	34.8	1.803
VSM	38.9	26.1	22.1	19.3	19.0	42.7	2.216	33.3	24.4	21.1	19.0	15.4	36.6	1.983
NNGen	36.5	23.8	20.1	17.4	17.1	40.2	1.967	33.0	24.4	20.9	18.7	15.0	36.3	1.933
CODE-NN	30.8	15.4	10.7	8.1	13.4	35.1	1.229	23.9	12.8	8.6	6.3	9.1	28.9	0.978
TL-CodeSum	31.1	16.5	12.5	10.4	13.6	35.3	1.335	29.9	21.3	18.1	16.1	13.7	33.2	1.66
Hybrid-DRL	41.1	26.2	19.5	15.0	17.9	42.2	2.042	32.4	22.6	16.3	13.3	13.5	36.5	1.656
GRNMT	38.6	24.0	18.8	15.8	18.5	43.4	1.978	32.6	22.6	17.9	15.5	15.0	37.6	1.732
Rencos	**43.1**	**29.5**	**24.2**	**20.7**	**21.1**	**47.5**	**2.449**	**37.5**	**27.9**	**23.4**	**20.6**	**17.3**	**42.0**	**2.209**

- **VSM** is the abbreviated form of Vector Space Model [48]. A classic example of VSM is Term Frequency-Inverse Document Frequency (TF-IDF), which is adopted by some automatic source code summarization work [19, 46]. Different from LSI, it indexes code snippets into weight vectors based on term frequency and document frequency. Once obtaining the vector representations, we retrieve the summary of the most similar code snippet by the cosine distance.
- **NNGen** is a simple but effective Nearest Neighbor based algorithm that is proposed to produce commit messages for code changes [37]. After building the vectors of code changes based on bag of words and the term frequency, it retrieves the nearest neighbors of code changes by the cosine similarity of vectors and the BLEU-4 score. Then NNGen directly reuses the commit message of the nearest neighbor. We reproduce such an algorithm in our task by replacing code changes with code snippets.

3.3.2 *NMT-based approaches.*

- **CODE-NN**[7] is the first neural approach that learns to generate summaries of source code [24]. It is an LSTM encoder-decoder neural network that encodes code snippets to context vectors with attention mechanism and produces summaries.
- **TL-CodeSum**[8] is a multi-encoder neural model that encodes API sequences along with code token sequences and generates summaries from source code with transferred API knowledge [23]. It first trains an API sequence encoder using an external dataset. The learned API representations are then applied to source code summarization task to assist the summary generation.
- **Hybrid-DRL**[9] is an advanced neural approach with hybrid code representations and deep reinforcement learning [55]. The basic architecture is also a multi-encoder NMT to learn structural and sequential information by encoding ASTs and tokens of source code. Similar work on incorporating ASTs is also done by [21, 34], but Hybrid-DRL further uses reinforcement learning to solve the exposure bias problem during decoding and obtains better performance.

Furthermore, as we mentioned, we also include an approach that guides NMT with retrieved translation pieces in natural language translation [64] (referred to as **GRNMT**). The translation pieces are *n*-grams of the retrieved target sentences that also match the common words in both the input and the retrieved source sentences by word alignment. During decoding, GRNMT uses the translation pieces to enhance the word prediction accuracy.

3.4 Results and Discussion

We present the experimental results and analysis through the following research questions.

RQ1: How does our proposed approach perform compared to the baselines? Table 2 shows the performances of different methods to generate code summaries in terms of our evaluation metrics. We compute the values of these metrics using the same scripts provided by Hybrid-DRL [55]. For these metrics, the bigger is better, and the best one of each metric is marked in bold.

From the table, we can see that retrieval-based methods LSI, VSM and NNGen yield good results. In particular, VSM achieves the best performance among these three techniques. It is a little confusing that LSI seems to be poorer than VSM, since LSI considers the term association and is usually better than VSM (TF-IDF) in capturing semantics of plain texts[66]. But as mentioned in [66], if the dimension of TF-IDF (i.e., the vocabulary size) is much larger than that of LSI (i.e., 500), it may produce better performance. In our case, the source code corpus has a huge vocabulary size of more than 50k in TF-IDF since the unique tokens of source code are much more than those in plain texts due to the arbitrary identifiers. Such a phenomenon is also observed in Haiduc et al.'s work [19]. Comparing VSM with NNGen, VSM is better since it considers the document frequency but NNGen does not.

Among all three NMT-based methods of CODE-NN, TL-CodeSum and Hybrid-DRL, CODE-NN performs worse than the other two

[7]https://github.com/sriniiyer/codenn
[8]https://github.com/xing-hu/TL-CodeSum
[9]https://github.com/wanyao1992/code_summarization_public

neural models because it only depends on the embeddings of tokens (i.e., at the lexical level) to understand the semantics of code snippets. In contrast, TL-CodeSum is better since it captures more semantics of source code with learned API sequence knowledge. We also use the external dataset from the original JCSD provided in [23] to pre-train the API sequence encoder for JCSD, but PCSD does not have such external dataset. Thus TL-CodeSum is more effective on JCSD than on PCSD. For Hybrid-DRL, it has better performance than TL-CodeSum on PCSD since the AST-based structural information of source code is incorporated and the exposure bias problem is solved. But on JCSD, for the metrics including BLEU-3, BLEU-4, METEOR and CIDER, Hybrid-DRL is worse than TL-CodeSum probably because the API sequence knowledge from the external dataset of the original JCSD dominates the performance. In addition, we find that CODE-NN and TL-CodeSum have an overall worse performance than the retrieval-based ones, which is not surprising due to the low-frequency word problem as described in Section 1.

As a combination of retrieval-based and NMT-based methods, GRNMT guides one simple encoder-decoder model with the retrieved translation pieces. It can outperform CODE-NN in all metrics, TL-CodeSum and Hybrid-DRL in some metrics such as METEOR and ROUGE-L, which means that the retrieval information actually helps. In addition, GRNMT is better than all retrieval-based methods for ROUGE-L, which indicates that the encoder-decoder neural model can also contribute to the performance.

Finally, from the table, we can see that our approach achieves the best performance for all evaluation metrics. The reason is that we retrieve the most similar code snippets at both syntactic level and semantic levels as additional contexts to our attentional encoder-decoder model. We also enhance the summary generation by the fusion of the similarities and the conditional probabilities for the next word prediction, as described in Section 2. Compared with our approach, unlike it in the task of natural language translation, GRNMT is worse since it is difficult to precisely match the *n*-grams of summaries with the corresponding elements in code snippets and get high-quality translation pieces.

Due to the large scale training set and the online code retrieval during testing, our approach may cost much more time to generate summaries than a single NMT model. However, we only need 89 ms in average to generate the summary for each testing code snippet in the two datasets of our experiment because of the efficient search engine Lucene and the fast computation of the cosine similarity.

RQ2: How effective are the main components of our approach? In our approach, we design two code retrieval components from different aspects including the syntactic level and semantic level, we want to know whether they are effective. In this experiment, at first we do not use our attentional encoder-decoder model, but each time adopt only one of these two retrieval components (Only Syntactic Retrieval and Only Semantic Retrieval) and directly take the summary of the most similar code snippet from the training set as the final generated one. Then we use the attentional encoder-decoder model (NMT) and consider the impact of adding the retrieved code snippets in four different scenarios: without any retrieval code (NMT); adding the most similar code snippet retrieved at the syntactic level (NMT+Syntactic Retrieval); adding the most similar code snippet retrieved at the semantic level (NMT+Semantic Retrieval); adding both the two most similar code snippets above (NMT+Both Retrieval, which is also our final approach).

We present the experimental results in Table 3. Both of our syntactic-level and semantic-level retrieval components are more effective than existing retrieval-based methods LSI, VSM and NNGen, which are described in Table 2. This is because we capture more syntactic or semantic information mentioned in Subsection 2.3. In addition, our syntactic-level and semantic-level retrieval components have slightly different but comparable results, which indicates that the syntactic and semantic information from two different aspects of source code are both useful in code retrieval. Based on the two retrieved code snippets, we add any one or both to our original attentional encoder-decoder model. Compared with the case without any retrieval code, we can see that the retrieved information can indeed enhance the performance of the neural model, even when we add just one most similar code snippet retrieved at either the syntactic or semantic level. When these two most similar code snippets are both utilized, we can obtain the best performance.

RQ3: Does our approach perform better than NMT-based methods for tackling the low-frequency word problem? As mentioned in Section 1 and 2, our retrieval-based neural model can tackle the low-frequency word problem better than NMT-based methods, which may correctly generate more low-frequency words. To illustrate it, we perform a statistical analysis about the low-frequency words in generated summaries. For the testing set, our generated summaries include 10,350 and 5,729 unique words for PCSD and JCSD, respectively, and each summary has 8.4 (PCSD) and 12.0 (JCSD) words in average. We first collect all the correctly generated words according to the ground-truth summaries. Then we count the frequencies of all these correct words in the training set, and record the number of the correct and low-frequency words (frequency = 1, 2, 5, 10, 20, 50, and 100). In this experiment, we compare *Rencos* with our original attentional encoder-decoder model (NMT) and show how they deal with the low-frequency words. The experiment results are shown in Table 4, where *Ratio* is the quotient of $Rencos/NMT$, which indicates the degree of improvement our approach achieves on the low-frequency words.

From the table, we can see that our approach can correctly predict more low-frequency words than NMT when the word frequency is small (≤ 100). For example, for the words that occur only once in the training set, the number correctly predicted by our approach is 1.77 and 1.93 times that of NMT on the datasets of PCSD and JCSD, respectively. Obviously these additional low-frequency words come from the summaries of our retrieved code snippets, which validates our claim that our retrieval-based neural source code summarization can more effectively deal with the low-frequency word problem than the NMT-based methods. Moreover, when the word frequency increases, we can see that NMT has a trend to perform better as the ratio decreases. This indicates that for high-frequency words NMT can easily capture them and the benefit brought by the retrieved source code becomes small.

RQ4: Will our approach perform better if we retrieve top-*k* (*k*>1) similar code snippets and filter them according to a similarity threshold *T*? In our approach, our two syntactic-level and semantic-level retrieval components both select the most similar code snippet ($k = 1$) without threshold ($T = 0$) by default. In this RQ, we study whether the most $k(k > 1)$ similar code snippets and

Table 3: Effectiveness of each component of the proposed approach

Descriptions	PCSD							JCSD						
	BLEU-1/2/3/4(%)				METEOR(%)	ROUGE-L(%)	CIDER	BLEU-1/2/3/4(%)				METEOR(%)	ROUGE-L(%)	CIDER
Only Syntactic Retrieval	39.8	27.4	23.3	20.2	19.5	43.5	2.296	33.9	25.2	21.7	19.5	15.9	37.4	2.020
Only Semantic Retrieval	39.5	27.1	23.1	20.1	19.1	43.1	2.270	33.7	25.3	22.1	19.9	15.4	37.0	2.049
NMT	37.5	22.5	17.1	14.2	17.3	42.3	1.871	31.1	20.7	16.0	13.8	13.8	36.3	1.633
NMT+Syntactic Retrieval	41.9	28.2	22.8	19.5	20.4	46.5	2.344	36.3	26.7	22.1	19.5	16.7	40.9	2.106
NMT+Semantic Retrieval	42.2	28.4	23.2	19.8	20.6	46.6	2.362	36.8	27.2	22.6	19.9	17.0	41.3	2.164
NMT+Both Retrieval	**43.1**	**29.5**	**24.2**	**20.7**	**21.1**	**47.5**	**2.449**	**37.5**	**27.9**	**23.4**	**20.6**	**17.3**	**42.0**	**2.209**

Table 4: Number of correctly generated low-frequency words

Word Frequency		1	2	5	10	20	50	100
PCSD	NMT	452	376	272	176	145	84	82
	Rencos	799	515	344	223	184	88	109
	Ratio	1.77	1.37	1.26	1.27	1.27	1.05	1.33
JCSD	NMT	126	75	45	27	38	28	16
	Rencos	243	138	73	38	49	37	18
	Ratio	1.93	1.84	1.62	1.41	1.29	1.32	1.11

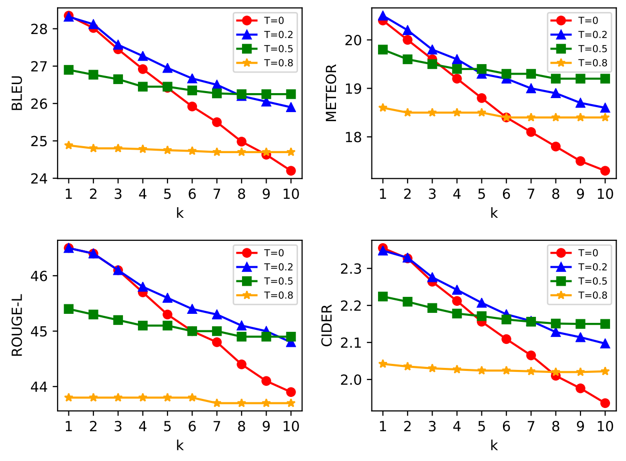

Figure 4: Performance of top-k retrieved similar code snippets with different thresholds T

similarity threshold $T(T > 0)$ can provide more useful knowledge for generating a good summary. Thus we use the semantic-level retrieval component and conduct experiments on PCSD by obtaining $k = 1, 2, ..., 10$ code snippets with the k highest similarities by Equation 10. We also consider to filter them by the similarity threshold $T = 0.2, 0.5, 0.8$. For simplification, we calculate the average value of BLEU-1/2/3/4 and mark it as BLEU.

The experimental results are shown in Figure 4. We can see that when k increases, the performance becomes worse with any threshold. For example, the BLEU score decreases nearly 4% when we set $k = 10$ and $T = 0$. We further calculate the distribution of similarities in Equation 11 between top-$k(k = 1, 2, ..., 10)$ similar code snippets and the corresponding input. As shown in Figure 5, we find that the overall similarity decreases greatly when k increases. This means that when k is bigger, the k-th similar code snippet may be completely irrelevant to the testing one and provide more

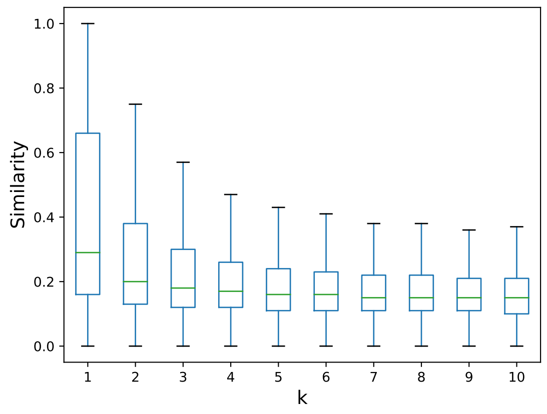

Figure 5: Similarity distributions of code snippets

noisy words. The threshold T can effectively filter these irrelevant code snippets, but may eliminate potential useful low-frequency words as well and make the performance worse, for example, when $T = 0.8$. When k is small (e.g. k=1), such elimination dominates the performance since the retrieved similar code snippets are almost all useful but filtered by the threshold. Also, combining more code snippets during testing is much time-consuming. Therefore, we choose the most similar code snippet ($k = 1$) without threshold ($T = 0$) in this work for the best performance and high efficiency. In addition, when the similarities of retrieved code snippets are low (even for the most similar one), our approach can automatically tolerate them because we take the similarity as one factor in Equation 12 , which tends to ignore the words produced by low-similarity code snippets.

3.5 Examples

For qualitative analysis of our approach, we present two examples of summaries generated by different methods from the testing sets of Python and Java datasets, respectively. A simple Python example is shown in Figure 1. Our approach generates the summary "create an iis application", which is exactly the same as the ground-truth, indicating that our approach can effectively combine the high-frequency word "create" from NMT and the low-frequency word "iis" form retrieval-based methods. The other more difficult Java example is depicted in Table 5, where the reference is the ground-truth summary written by developers, and others are automatically generated summaries by different methods. We can see that the high-frequency word "remove" is captured by all the NMT-based methods, GRNMT and our approach, while all the retrieval-based

Table 5: A Java code snippet with generated summaries

```java
public void removeColumn(final String columnName){
  if(columnName == null) {  return; }
  List<String> cols = Arrays.asList(getInfo().headers);
  final int colIndex = cols.indexOf(columnName);
  removeColumn(colIndex);
}
```

Reference: remove the column represented by its name
LSI: get index of this column name
VSM: adds the given column to this table
NNGen: get index of this column name
CODE-NN: remove a column from the table .
TL-CodeSum: remove column at specified index .
Hybrid-DRL: removes a column from the column .
GRNMT: remove a column from the table .
Our approach: remove the column represented by the index

methods miss it. In addition, the phrase "represented by" is correctly predicted only by our approach since it appears in the summaries of our two retrieved code snippets and is effectively captured by the retrieval-based summary generation. Hence our approach generates the closest summary to the reference.

3.6 Human Evaluation

The above four metrics BLEU, ROUGE-L, METEOR, and CIDER can be used to compare summaries generated by different methods. However, they mainly calculate the textual similarity between the reference and the generated summaries, rather than the semantic similarity. Therefore we perform human evaluation to complement the quantitative evaluation in terms of those metrics.

For human evaluation of our approach, we recruit workers from Amazon Mechanical Turk (AMT) [10], a worldwide crowdsourcing website. AMT supports micro-tasks (also known as Human Intelligence Tasks, HITs) posted by clients. Remote workers can complete HITs for money. We randomly choose 100 code snippets from the testing sets (50 from PCSD and 50 from JCSD) and their comments produced by three methods VSM, Hybrid-DRL, and *Rencos*. As depicted in Table 2, VSM is the best among the retrieval-based approaches. Hybrid-DRL is the best on PCSD among the NMT-based approaches, and is comparable with TL-CodeSum on JCSD. For each of these three methods, we obtain the natural language comment pairs consisting of the reference comments written by developers and the generated ones, and then post them as HITs to AMT. Each HIT is assigned to three different workers, and such redundancy can help obtain more consistent results. Finally, we have 900 HITs (3 methods of VSM, Hybird-DRL and *Rencos*, 100 code snippets and 3 redundant assignments). These HITs are completed by 129 unique workers. Each HIT costs 0.03US$ and all HITs spend 122 minutes in total.

In each HIT webpage, we ask "How similar is the meaning of these two source code comments? Comment1: X; Comment2: Y" where X and Y are one comment pair, and workers can select a score between 1 to 5 where 1 means "Not Similar At All" and 5

[10]https://www.mturk.com/

Table 6: The distribution of the scores of the generated comments

Score	1	2	3	4	5	Avg	≥4	≥3	≤2
VSM	24	18	31	17	10	2.71	27	58	42
Hybrid-DRL	0	26	48	23	3	3.03	26	74	26
Rencos	2	13	39	30	16	3.45	46	85	15

means "Highly Similar/Identical". We get three scores from workers for every HIT and choose the median value as the final score. Table 6 shows the score distribution of the reference and generated comments. We can see that our approach achieves the best scores and improves the average (Avg) score from 2.71 (VSM) and 3.03 (Hybrid-DRL) to 3.45. Specifically, among the randomly selected 100 code snippets, our approach can generate 16 highly similar or even identical comments with the reference ones (score = 5), 46 good comments (score ≥ 4) and 85 comments that are not bad (score ≥ 3). Our approach also receives the smallest number of negative results (score ≤ 2).

Based on the 100 final scores for each approach of *Rencos*, Hybrid-DRL and VSM, we conduct Wilcoxon signed-rank tests [58] and compute Cliff's delta effect sizes [41]. Comparing *Rencos* with Hybrid-DRL and VSM, the p-values of Wilcoxon signed-rank tests at 95% confidence level are 0.00025 and 2.111e-06, which means the improvements achieved by our approach are statistically significant. In addition, Cliff's delta effect sizes are 0.252 (small but non-negligible) and 0.3321 (medium), respectively. In summary, the results of human evaluation confirm the effectiveness of the proposed approach.

4 THREATS TO VALIDITY

There are three main threats to the validity of our evaluation.

- In the implementation of existing methods, we directly use the public code of CODE-NN, TL-CodeSum, and Hybrid-DRL provided by their authors, but the code of GRNMT is not available. We have tried our best to read the paper carefully and consult the authors about many details. We will eliminate this threat as soon as the tool is publicly available.
- The scale of datasets. As we analyzed in Section 1, larger datasets cannot help mitigate the low-frequency words problem but may result in more infrequent words. In our evaluation, we have used two public large datasets, which include 108,726 Python and 69,708 Java code snippets and their summaries. In our future work, we will experiment with even larger-scale datasets and further evaluate the effectiveness of *Rencos* in handling low-frequency words.
- The retrieved code snippets may not always have high similarities, especially when the code base is small. We suggest to increase the scale and diversity of code bases to avoid such threat. However, we should note that *Rencos* only takes the similarity as one factor in Equation 12. If the similarity is low, *Rencos* tends to choose the words predicted by the NMT model. In this way, *Rencos* can still guarantee that its performance is comparable with that of NMT.

5 RELATED WORK

5.1 Source Code Summarization

In software engineering community, Information Retrieval techniques are widely used for automatic source code summarization. Some studies extract terms from source code for generating term-based summaries [13, 18, 19, 46]. Haiduc et al. [18, 19] use IR methods including LSI and VSM to choose top-k terms from a code snippet. They treat each function of source code as a document and index on such a corpus by LSI and VSM, then the most similar terms based on their cosine distances between documents are selected as the summary. Their work is improved by topic modeling in [13]. Rodeghero et al. [46] improve the process of selecting terms by eye-tracking and modify the weights of VSM for better code summarization. Besides the term-based summaries, code clone detection techniques are used to retrieve similar code snippets from open-source GitHub projects and Stack Overflow, and reuse their sentence comments as code summaries [60, 61]. In addition, Sridhara et al. [52] design heuristics to choose statements from Java methods, and use the Software Word Usage Model (SWUM) to identify keywords from those statements and create summaries though manually-crafted templates.

Recently many neural models are proposed to generate source code summaries. Allamanis et al. [1] suggest method and class names as code summaries by the neural logbilinear context model, which is based on embeddings of code tokens. The convolutional attention model is presented to predict extreme summarization of source code such as function names [2]. The NMT-based models are also widely used to generate summaries for code snippets with encoder-decoder neural networks [21–24, 34, 55]. Iyer et al. [24] propose an attentional LSTM encoder-decoder network for automatically generating short natural language summaries of source code snippets. Hu et al. [23] use one additional encoder to leverage API sequences and improve the summary generation by learned API sequence knowledge. The abstract syntax tree structures are incorporated by [21, 34, 55]. Moreover, its performance can be further improved by deep reinforcement learning [38, 55] to solve the exposure bias problem during decoding.

Compared with the above work, our approach can take the advantages of both Information Retrieval and NMT-based methods by enhancing the NMT model with the retrieved similar code snippets from the training set, resulting in better performance than the above state-of-the-art work.

5.2 Code Clone Detection and Code Retrieval

Many studies [27, 28, 35] show that much duplicated code exists in a large code base. There are a lot of work investigating code clone and similar code retrieval techniques. Traditional code clone detection techniques such as DECKARD [25] and SourcererCC [47] detect similar code with the tree-based or token-based code comparison. Recently neural models are incorporated to learn vector representations of source code and use these vectors to compute code similarity [56, 57, 65, 67], which can achieve better performance but need additional time-consuming training and labeled data. In this work, during the online testing, we need lightweight and efficient search of similar code from a large-scale training set, thus we prefer to reuse our trained encoder to learn semantic vectors rather than training new neural models. Also, we want to find more similar code snippets from the syntax aspect. To avoid complex operations over ASTs [9], we convert ASTs to token sequences and leverage the off-the-shelf efficient search engine Lucene for code retrieval.

5.3 Retrieval-based Neural Machine Translation

Due to the low-frequency word problem in neural machine translation , some studies that combine the retrieval information with encoder-decoder neural models have been recently proposed in the NLP community [10, 15, 62, 64]. Zhang et al. [64] extract the translation pieces, which are n-grams from retrieved target sentences and their corresponding source words appearing in testing sentences. Then these translation pieces are leveraged to guide the decoding. In natural language translation, the source and target words of retrieved translation pairs can be easily matched through word alignment. However, in source code summarization, it is difficult to find out which code elements produce certain specific words in the short summaries. As a result, the similar idea is not effective in our work. Gu et al. [15] incorporate the retrieval information with new encoders and build one complex encoder-decoder neural network. In order to improve NMT with additional retrieval information, a Translation Memory (TM) which consists of source and target sentence pairs is leveraged by graph representation and self-attention mechanism over the graph [62], or augmenting the source data with retrieved fuzzy TM targets by means of concatenation [10]. Compared with their work, we do not need additional encoders, and directly combine the similarities and the conditional probabilities of retrieved similar code snippets to predict the summary words during decoding.

6 CONCLUSION

In this paper, we propose a novel retrieval-based neural approach named *Rencos* that augments an attentional encoder-decoder model with the retrieved two most similar code snippets for better source code summarization. Given one new code snippet, instead of only reusing the retrieved summaries or only generating summaries based on NMT, *Rencos* can automatically generate summary by the fusion of retrieved code snippets and itself. We also design two code retrieval methods from the aspects of syntax and semantics to accommodate more knowledge about the code. We have evaluated the effectiveness of our approach through extensive experiments and the results show that it outperforms the related approaches.

Our code, experimental data and results are publicly available at *https://github.com/zhangj111/rencos*.

ACKNOWLEDGMENTS

This work was supported partly by National Key Research and Development Program of China (No.2018YFB1004805), partly by National Natural Science Foundation of China (No.61702024, 61932007, 61972013 and 61421003) and ARC DP200102940.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 38–49.

[2] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*. 2091–2100.

[3] Philip Arthur, Graham Neubig, and Satoshi Nakamura. 2016. Incorporating Discrete Translation Lexicons into Neural Machine Translation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Austin, Texas, 1557–1567.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[5] Satanjeev Banerjee and Alon Lavie. 2005. 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*. 65–72.

[6] Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A Parallel Corpus of Python Functions and Documentation Strings for Automated Code Documentation and Code Generation. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, Vol. 2. 314–319.

[7] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 368–377.

[8] Richard Bellman. 1966. Dynamic programming. *Science* 153, 3731 (1966), 34–37.

[9] Philip Bille. 2005. A survey on tree edit distance and related problems. *Theoretical computer science* 337, 1-3 (2005), 217–239.

[10] Bram Bulté and Arda Tezcan. 2019. Neural Fuzzy Repair: Integrating Fuzzy Matches into Neural Machine Translation. In *57th Conference of the Association for Computational Linguistics (ACL)*. 1800–1809.

[11] Thomas A Corbi. 1989. Program understanding: Challenge for the 1990s. *IBM Systems Journal* 28, 2 (1989), 294–306.

[12] Sergio Cozzetti B de Souza, Nicolas Anquetil, and Káthia M de Oliveira. 2005. 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*. ACM, 68–75.

[13] Brian P Eddy, Jeffrey A Robinson, Nicholas A Kraft, and Jeffrey C Carver. 2013. Evaluating source code summarization techniques: Replication and expansion. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 13–22.

[14] Beat Fluri, Michael Wursch, and Harald C Gall. 2007. Do code and comments co-evolve? on the relation between source code and comment changes. In *14th Working Conference on Reverse Engineering (WCRE 2007)*. IEEE, 70–79.

[15] Jiatao Gu, Yong Wang, Kyunghyun Cho, and Victor O. K. Li. 2018. Search Engine Guided Neural Machine Translation. In *AAAI*.

[16] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.

[17] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 631–642.

[18] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 223–226.

[19] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*. IEEE, 35–44.

[20] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[21] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*. ACM, 200–210.

[22] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2019. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* (2019), 1–39.

[23] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred API knowledge. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. AAAI Press, 2269–2275.

[24] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. 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)*, Vol. 1. 2073–2083.

[25] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 96–105.

[26] Mira Kajko-Mattsson. 2005. A survey of documentation practice within corrective maintenance. *Empirical Software Engineering* 10, 1 (2005), 31–55.

[27] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Software Eng.* 28 (2002), 654–670.

[28] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 2005. An Empirical Study of Code Clone Genealogies. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal) *(ESEC/FSE-13)*. ACM, New York, NY, USA, 187–196.

[29] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[30] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M. Rush. 2017. OpenNMT: Open-Source Toolkit for Neural Machine Translation. In *Proc. ACL*. https://doi.org/10.18653/v1/P17-4012

[31] Andrew J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE, 199–206.

[32] Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering* 12 (2006), 971–987.

[33] Thomas D LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*. ACM, 492–501.

[34] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A Neural Model for Generating Natural Language Summaries of Program Subroutines. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 795–806. https://doi.org/10.1109/ICSE.2019.00087

[35] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. 2006. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering* 32, 3 (March 2006), 176–192.

[36] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. *Text Summarization Branches Out* (2004).

[37] Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: how far are we?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 373–384.

[38] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. 2019. Automatic Generation of Pull Request Descriptions. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 176–188.

[39] Edward Loper and Steven Bird. 2002. NLTK: The Natural Language Toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics*.

[40] Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. 1412–1421.

[41] Guillermo Macbeth, Eugenia Razumiejczyk, and Rubén Daniel Ledesma. 2011. Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica* 10, 2 (2011), 545–555.

[42] Paul W McBurney. 2016. *Improving program comprehension via automatic documentation generation*. University of Notre Dame.

[43] Ramesh Nallapati, Bowen Zhou, Cicero dos Santos, Caglar Gulcehre, and Bing Xiang. 2016. Abstractive Text Summarization using Sequence-to-sequence RNNs and Beyond. In *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*. 280–290.

[44] Sankar K Pal and Sushmita Mitra. 1992. Multilayer perceptron, fuzzy sets, and classification. *IEEE Transactions on neural networks* 3, 5 (1992), 683–697.

[45] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 311–318.

[46] Paige Rodeghero, Collin McMillan, Paul W McBurney, Nigel Bosch, and Sidney D'Mello. 2014. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 390–401.

[47] Hitesh Sajnani, Vaibhav Pratap Singh Saini, Jeffrey Svajlenko, Chanchal Kumar Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-Code. *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (2016), 1157–1168.

[48] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A vector space model for automatic indexing. *Commun. ACM* 18, 11 (1975), 613–620.

[49] Mike Schuster and Kuldip K Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing* 45, 11 (1997), 2673–2681.

[50] Abigail See, Peter J Liu, and Christopher D Manning. 2017. Get To The Point: Summarization with Pointer-Generator Networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* 1073–1083.

[51] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. 2010. An examination of software engineering work practices. In *CASCON First Decade High Impact Papers*. IBM Corp., 174–188.

[52] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering.* ACM, 43–52.

[53] Armstrong A Takang, Penny A Grubb, and Robert D Macredie. 1996. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.* 4, 3 (1996), 143–167.

[54] Ramakrishna Vedantam, C Lawrence Zitnick, and Devi Parikh. 2015. Cider: Consensus-based image description evaluation. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 4566–4575.

[55] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* ACM, 397–407.

[56] Hui-Hui Wei and Ming Li. 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence.* AAAI Press, 3034–3040.

[57] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.* ACM, 87–98.

[58] Frank Wilcoxon, SK Katti, and Roberta A Wilcox. 1970. Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test. *Selected tables in mathematical statistics* 1 (1970), 171–259.

[59] Sam Wiseman and Alexander M Rush. 2016. Sequence-to-Sequence Learning as Beam-Search Optimization. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing.* 1296–1306.

[60] Edmund Wong, Taiyue Liu, and Lin Tan. 2015. Clocom: Mining existing source code for automatic comment generation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER).* IEEE, 380–389.

[61] Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. Autocomment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 562–567.

[62] Mengzhou Xia, Guoping Huang, Lemao Liu, and Shuming Shi. 2019. Graph based translation memory for neural machine translation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 7297–7304.

[63] Wojciech Zaremba and Ilya Sutskever. 2014. Learning to execute. *arXiv preprint arXiv:1410.4615* (2014).

[64] Jingyi Zhang, Masao Utiyama, Eiichro Sumita, Graham Neubig, and Satoshi Nakamura. 2018. Guiding Neural Machine Translation with Retrieved Translation Pieces. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers).* Association for Computational Linguistics, New Orleans, Louisiana, 1325–1335. https://doi.org/10.18653/v1/N18-1120

[65] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering.* IEEE Press, 783–794.

[66] Wen Zhang, Taketoshi Yoshida, and Xijin Tang. 2011. A comparative study of TF*IDF, LSI and multi-words for text classification. *Expert Systems with Applications* 38, 3 (2011), 2758–2765.

[67] Gang Zhao and Jeff Huang. 2018. DeepSim: Deep Learning Code Functional Similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018).* 141–151.