Received 23 February 2023, accepted 24 April 2023, date of publication 1 May 2023, date of current version 1 June 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3271992

METHODS

READSUM: Retrieval-Augmented Adaptive Transformer for Source Code Summarization

YUNSEOK CHOI[1], **CHEOLWON NA**[2], **HYOJUN KIM**[2], **AND JEE-HYONG LEE**[2]
[1]Department of Platform Software, Sungkyunkwan University, Suwon 16419, South Korea
[2]Department of Artificial Intelligence, Sungkyunkwan University, Suwon 16419, South Korea

Corresponding author: Jee-Hyong Lee (john@skku.edu)

This work was supported in part by the Institute of Information & Communications Technology Planning & Evaluation (IITP) funded by the Korean Government [Ministry of Science and ICT (MSIT)] through the Artificial Intelligence Graduate School Program (Sungkyunkwan University) under Grant 2019-0-00421, in part by the Information and Communication Technology (ICT) Creative Consilience Program under Grant IITP-2021-2020-0-01821, and in part by the Self-Directed Multi-Modal Intelligence for Solving Unknown Open Domain Problems under Grant 2022-0-01045.

ABSTRACT Code summarization is the process of automatically generating brief and informative summaries of source code to aid in software comprehension and maintenance. In this paper, we propose a novel model called READSUM, REtrieval-augmented ADaptive transformer for source code SUMmarization, that combines both abstractive and extractive approaches. Our proposed model generates code summaries in an abstractive manner, taking into account both the structural and sequential information of the input code, while also utilizing an extractive approach that leverages a retrieved summary of similar code to increase the frequency of important keywords. To effectively blend the original code and the retrieved similar code at the embedding layer stage, we obtain the augmented representation of the original code and the retrieved code through multi-head self-attention. In addition, we develop a self-attention network that adaptively learns the structural and sequential information for the representations in the encoder stage. Furthermore, we design a fusion network to capture the relation between the original code and the retrieved summary at the decoder stage. The fusion network effectively guides summary generation based on the retrieved summary. Finally, READSUM extracts important keywords using an extractive approach and generates high-quality summaries using an abstractive approach that considers both the structural and sequential information of the source code. We demonstrate the superiority of READSUM through various experiments and an ablation study. Additionally, we perform a human evaluation to assess the quality of the generated summary.

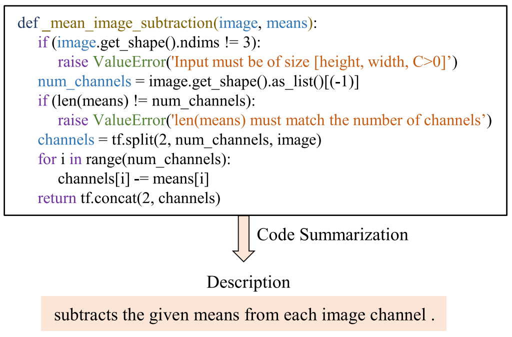
INDEX TERMS Abstract syntax tree, adaptive transformer, source code summarization, fusion network, shortest path.

I. INTRODUCTION

Code summarization is the process of automatically generating a concise and human-readable description of a code snippet's functionality. It aims to provide developers with an understanding of the code without having to read the entire implementation. Well-written summaries make the code easy for developers to understand as shown in Fig. 1. In order to generate a good summary about a source code, it is necessary to understand the structural and sequential meanings in the code (abstractive approaches), and write the summary in an easy-to-understand form (extractive approaches).

Previous approaches on automatic source code summarization can be categorized into sequence-based, structure-based, and hybrid approaches. Sequence-based approaches generated summaries by capturing the sequential information of source code [1], [2], [3], [4], [5], [6], [7]. They tokenized source code into a sequence of code tokens, and encoded them using seq2seq models. Meanwhile, structure-based approaches used Abstract Syntax Tree (AST) to capture the structural information of code [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19]. They parsed the source code into the AST and utilized graph models such as Graph Neural Networks (GNNs). Some works flattened the AST into the pre-order traversal sequence [18], [20], [21], [22], [23]. Hybrid approaches utilized both the token

The associate editor coordinating the review of this manuscript and approving it for publication was Jolanta Mizera-Pietraszko.



```
def _mean_image_subtraction(image, means):
    if (image.get_shape().ndims != 3):
        raise ValueError('Input must be of size [height, width, C>0]')
    num_channels = image.get_shape().as_list()[(-1)]
    if (len(means) != num_channels):
        raise ValueError('len(means) must match the number of channels')
    channels = tf.split(2, num_channels, image)
    for i in range(num_channels):
        channels[i] -= means[i]
    return tf.concat(2, channels)
```

subtracts the given means from each image channel .

FIGURE 1. **An example of code summarization. To write a good code summary, it is important to focus on the key aspects of the code.**

sequences and the ASTs of codes [24], [25], [26]. They parallelly processed token sequences and ASTs with independent encoders, and tried to merge them in the decoder.

A. MOTIVATION

In order to generate good summaries, we need to consider both structural and sequential information of code. However, most existing methods modeled code from either a sequential point of view or a graph (AST) point of view. Some have attempted to consider both sequential and structural information together, but they independently processed both types of information and simply combined them.

The AST, structural information, are usually processed by GNNs, and code tokens by transformers. Since GNNs and transformers are attention-based networks, we can unify them into a model. The graph attention network learns the structural information using neighboring nodes in the AST through message passing to a specific node as shown in Fig. 2(a). It locally gives attentions to neighboring nodes as shown in Fig. 2(d), and has difficulty in learning the global context information of code. Fig. 2(e) shows the self-attention in the vanilla transformer model that calculates attention values between all tokens. It can well learn the global contextual information, but it is difficult to learn the dependency between structurally related nodes because it equivalently gives attentions to all tokens in the source code as shown in Fig. 2(e).

We may say that the local scope of attention is related to structural information processing and the global scope is related to sequential information processing. By controlling the scope of attention or adopting the grey scope of attention as shown in Fig. 2(c) and Fig. 2(f), we can effectively merge the structural and sequential information processing. We propose a method of adaptive attention which combines the advantages of two attention networks: graph attention network reflecting structural information and self-attention network reflecting sequential information. If we set the attention scope narrow, the model may learn with high attention values to near nodes and low attention values to distant nodes. If we set the attention scope wide, all the nodes will have equivalent attentions. We control the scope of attentions by adding biases which are learnable parameters. Since the scope is adjusted during training, each layer of the transformer encoder can adjust it roles: extracting structural information or sequential information. In our model, the lower layers of the transformer model have narrow scopes to learn the relationships on nodes that are structurally close, as if they have the role of graph attention network. The middle layers have mid-size scopes to learn both of structural and sequential information. They may learn about blocks in a code and the sequential information of them. Finally, the higher layers learn global contextual information about the entire code with large scopes of attention.

Also, some recent works have attempted to combine a generative model with a retrieval method, which searched for similar codes and summaries in the databases [15], [26], [27]. These approaches aimed to improve the quality of the summary by using extractive information which is additional codes and summaries with high similarity to the original code. However, they separately processed and simply concatenated them. They did not consider the relation between the original code and the similar code, or the relation between the original code and the summary of the similar code. When learning the code representation from the code encoder, it is necessary to capture the features of the original code as well as those of the similar code. Also, when generating the summary in the decoder, it is necessary to have an organic fusion between the two representations, not just using the code and the summary of the similar code, and to properly reflect the keywords of the similar code's summary. We design an attention-based augmentation to reflect the relevance between the original code and the retrieved code, and a fusion network to extract the important keywords of similar summary from the original code. Finally, we improve the performance of summarization by using dual copy mechanism from the original code tokens and the retrieved similar summary tokens.

B. CONTRIBUTION

Our focuses are not only l̂ocal and global attentionb̂ut also ŝequential and structural processingîn a unified manner in a single Transformer. We consider both at the same time, not separately.

First, we modify AST for effectively capturing the structure of code. It is hard to learn the AST structure effectively because AST is a very sparse graph, a tree only connected between parents and children. So we propose a dense AST with the shortest path distance as edge weights between all nodes. Then, we flatten the dense AST keeping edge information. The flattened AST is a sequence with edges. In a sequential viewpoint, the flattened AST has more sequential information than code, because not only tokens in code but also abstract grammar nodes in AST are also presented. In a structural viewpoint, the flattened AST effectively represent the structure of code because the edge weights between two nodes are proportional to the distance between them in AST.

Since the flattened AST is a sequence with structural information, we modify self-attention to process sequences considering structural information in a single Transformer.

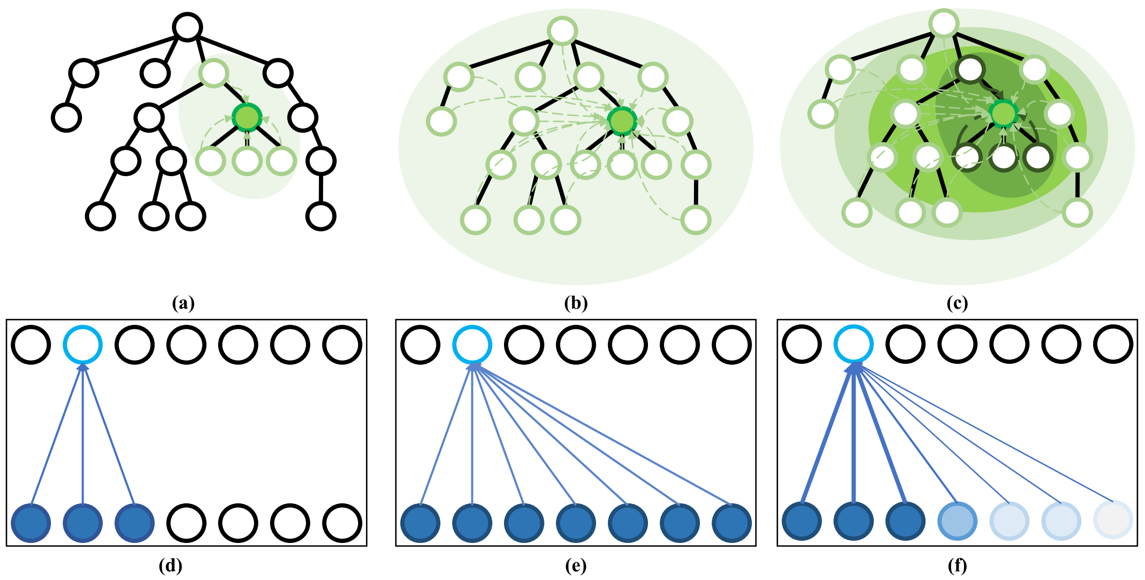


FIGURE 2. Comparison of graph attention network, self-attention network and adaptive attention network. (a), (b) and (c) are the process of message passing for one node in an AST by graph attention network, self-attention network and adaptive attention network (our proposed method), respectively. (d) graph attention network learns local and structural information about neighboring tokens, (e) self-attention network learns global and sequential information about all tokens, but (f) adaptive attention network learns adaptively both structural and sequential information using distance information between tokens.

We propose the adaptive Transformer, controlling the range of self-attention by the distance between two nodes. If two nodes are close in AST, each needs to pay more attention to the other. If far, less attention. Since more attention will be paid to neighboring nodes, the adaptive attention will help Transformer easily capture structural information.

Based on this idea, we calculate the self-attention based on the distance of two nodes and a bias, a trainable parameter. The bias in the adaptive attention controls the scope of neighborhood of each layer in Transformer. If the bias is small, the range of neighborhood will be broadened, and tokens will give similar attention to all the other nodes. Thus, a layer with a small bias tends to capture global structure. Conversely, if the bias is large, the range of neighborhood will be narrowed, and a layer tends to capture local structure.

Our Contributions of this paper are as below.

- We effectively unify the sequential and structural information into a structural sequence by flattening dense AST with shortest path distance.
- We effectively process sequential and structural information in a unified manner by a single Transformer.
- By adaptive self-attention, each layer can adaptively capture local and global structural information.
- To the best of our knowledge, we are the first to process the sequential and structural information of code in a unified way.

II. RELATED WORK

A. CODE SUMMARIZATION

Many works on source code summarization were modeled as sequence-based approaches. Iyer et al. [1] proposed Code-NN, a Long Short Term Memory (LSTM) networks with attention for code summarization and code retrieval task. Allamanis et al. [2] used a neural convolutional attentional model to consider highly-structured source code text. Hu et al. [4] proposed TL-CodeSum to summarize the source code with the API Knowledge. Also, Chen et al. [28] proposed a novel multi-task approach to use API knowledge for generating summaries from code. Wan et al. [24] used a deep reinforcement learning framework to consider an AST structure and code snippets. Liu et al. [29] proposed a encoder-decoder model for automatically generating descriptions for pull requests. Wei et al. [5] used a dual training framework by training code summarization and code generation tasks. Also, Ye et al. [6] considered the probabilistic correlation between the two tasks. Ahmad et al. [7] proposed a Transformer model using a relative position. These approaches focused on the sequential information of the source code, so the structural information was little considered about the relation between code tokens.

Also, some works tried to capture the structural information in the AST of code for source code summarization. Hu et al. [8] proposed an RNN-based model using structural-based traversal sequence as input, and Liang et al. [3] applied a tree-based recursive neural network for representing the syntax tree of code. Shi et al. [11] adopted Tree-LSTM, which is designed to capture both the syntactic and semantic structure of the source code. Harer et al. [12] proposed Tree-Transformer to handle Tree-structured data. Bui et al. [30] adopted self-supervised learning mechanism to build source code model by predicting subtrees from the context of the ASTs. Alon et al. [21] leveraged the unique syntactic structure of programming languages by sampling paths in the AST of a code snippet. Leclair et al. [14]

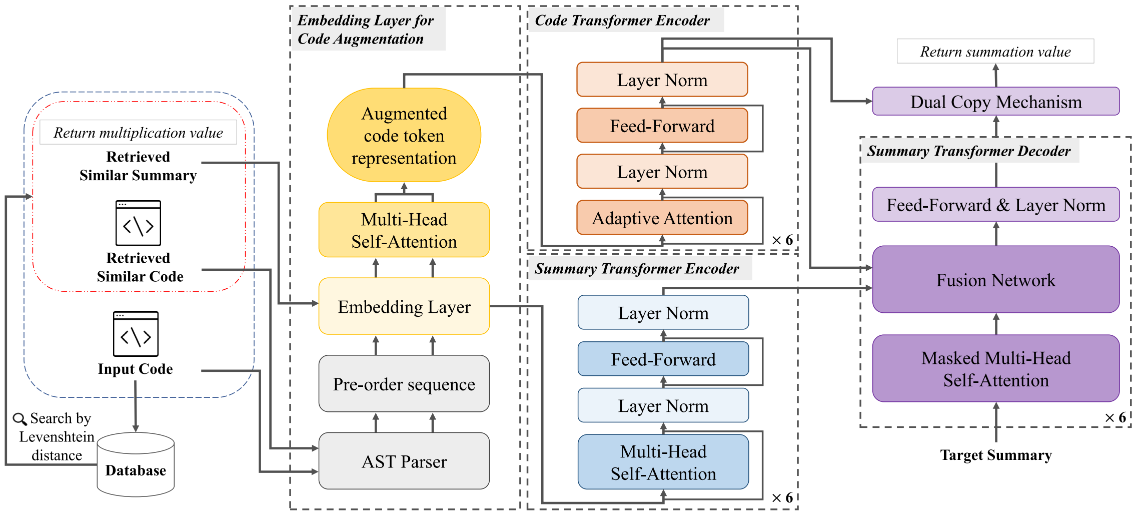

Return multiplication value
Retrieved Similar Summary
Retrieved Similar Code
Input Code
Search by Levenshtein distance
Database
Embedding Layer for Code Augmentation
Augmented code token representation
Multi-Head Self-Attention
Embedding Layer
Pre-order sequence
AST Parser
Code Transformer Encoder
Layer Norm
Feed-Forward
Layer Norm
Adaptive Attention
× 6
Summary Transformer Encoder
Layer Norm
Feed-Forward
Layer Norm
Multi-Head Self-Attention
× 6
Return summation value
Dual Copy Mechanism
Summary Transformer Decoder
Feed-Forward & Layer Norm
Fusion Network
Masked Multi-Head Self-Attention
× 6
Target Summary

FIGURE 3. **Overview of our proposed model. READSUM consists of four phases: embedding layer for code augmentation, code & summary transformer encoders, summary transformer decoder and dual copy mechanism.**

proposed encoded AST using graph neural networks and trained LSTM. Choi et al. [23] proposed a model that combines a GNN model and a Transformer model using modified AST. Shi et al. [16] tried to hierarchically split and reconstruct ASTs using Recursive Neural Network for learning the representation of the complete AST. Wu et al. [19] proposed a Transformer model which incorporated multi-view structure into attention mechanism. Guo et al. [31] proposed a Transformer model with two encoder architectures that considered the structural embedding of the AST using both types of source code and AST.

The retrieval-based method was used in Neural Machine Translation (NMT) Zhang et al. [32] Xia et al. [33], but recent works relied on both retrieved-based and generation-based approaches for the source code summarization. Zhang et al. [26] proposed retrieval-based approach using syntactic and semantic similarity for source code summarization, and Liu et al. [15] proposed a hybrid GNN using a retrieval augmented graph method. Li et al. [27] leveraged the retrieve-and-edit framework to improve the performance for code summarization.

B. LARGE LANGUAGE MODEL FOR PROGRAM LANGUAGE

The success of pre-trained models based on the Transformer architecture in natural language generation has led to the development of methods for extending these techniques to programming language tasks. CodeBERT, a pre-trained language model based on BERT [34], was proposed by [35], and incorporates both programming language and natural language representations in the pre-training stage. To incorporate code structure into CodeBERT, guo et al [36] proposed GraphCodeBERT, but these models have limited effectiveness in programming language tasks since they rely solely on the transformer encoder for PL-NL representation. In response, PLBART, a unified encoder-decoder model based on BART [37], was proposed by [38] to support both code understanding and generation tasks.

III. READSUM

We propose a novel model, **READSUM**, **RE**trieval-augmented **AD**aptive transformer for source code **SUM**marization. Fig. 3 shows the overall architecture of our proposed model, READSUM. Our model consists of four phases that are embedding layer for code augmentation, code & summary transformer encoders, summary transformer decoder and dual copy mechanism.

To learn the relation between the original code and retrieval code, we obtain augmented code representation using multi-head self-attention between the original code's AST sequence and the retrieved similar code's AST sequence at the embedding stage. Then the augmented representation is adaptively trained on structural and sequential information through adaptive attention network in code transformer encoder, and the retrieved similar summary representation is learned in summary transformer encoder, respectively. To capture the relevance between the augmented code representation and the similar summary representation at the transformer decoder, they are trained by combining two representations for fusion. Finally, the dual copy mechanism is applied to directly reflect the word of the original code and the retrieved similar summary in the process of generating the summary. The following subsections describe the problem definition of code summarization and each phase of the proposed method in detail.

A. PROBLEM DEFINITION

Suppose that we have a dataset of code and summary pairs. We retrieve the most similar code C' and summary S' pair for each input code C based on the Levenshtein distance. Then, we parse the code C as AST N to learn structural information as well as sequential information of the source code. In order to adaptively learn both structural and sequential information at the embedding stage, we obtain the shortest path distance $d_{ij} = \phi(n_i, n_j)$ between all nodes in the AST $N = \{n_1, n_2, \ldots\}$. Finally, we flatten the AST into a pre-order first sequence to train the transformer model.

B. EMBEDDING LAYER FOR CODE AUGMENTATION

We embed information on the similar code representation to the original code token representation. To obtain the relevance of the similar code to the original code, we use multi-head self-attention.

Let the original code token representations and the retrieved similar code token representations obtained from the word embedding layer be E_n and $E_{n'}$, respectively.

$$A^{rel} = MultiAtt(E_n, E_{n'}, E_{n'}) \tag{1}$$

$$E_{n^{aug}} = E_n + zA^{rel} \tag{2}$$

where *MultiAttn* is multi-head self-attention and z is the similarity score between the original code and the retrieved similar code. A^{rel} is the representation that means the relevance between the original code tokens and the similar code tokens. The augmented code representation $E_{n^{aug}}$ is obtained by Equation 2 that combines the original code token representation with the similar code token representation and the similarity score. The obtained augmented code representations are used as the input on the code transformer encoder.

C. CODE & SUMMARY TRANSFORMER ENCODERS

READSUM has two transformer encoders: code transformer encoder and summary transformer encoder. The structural and sequential information are learned to represent the relation between the original and similar code in the code transformer encoder, and the sequential information of the similar summary is learned in summary transformer encoder.

Our code transformer encoder consists of 6 modified transformer encoder layers. Each layer of the code transformer encoder is composed of two layers: adaptive attention network and feed-forward network. We design a modified multi-head self-attention, adaptive attention network, which adds a learnable bias term to self-attention to adaptively capture structural and sequential information in the code. The adaptive attention network is calculated as follows:

$$AdapAtt(Q, K, V, d) = \text{softmax}(QK^{\intercal} + g(d))V \tag{3}$$

$$g(x) = \frac{1}{\sqrt{2\pi}a}\exp(-\frac{x^2}{2b^2}) + c \tag{4}$$

where *AdapAtt* is adaptive attention network, and d is the shortest path matrix between the query Q and key K, the bias term $g(x)$ as a modified form of gaussian function, a,b and c are learnable embedding scalars. Here, the query and key is the same as the original code's AST nodes in the code transformer encoder.

Our summary transformer encoder also consists of 6 transformer encoder layers [39]. In the summary transformer encoder, the retrieved similar summary representations are learned reflecting the similar summary context to be used in the transformer decoder.

D. SUMMARY TRANSFORMER DECODER

The transformer decoder aims to predict the target summary of the original code by fusion of the augmented code representation (code transformer encoder) and the retrieved similar summary representation (summary transformer encoder). The summary transformer decoder consists of 6 modified transformer decoder layers [39]. Each layer of the code transformer decoder is composed of three layers: masked multi-head self-attention, fusion network and feed-forward network. We modify multi-head self-attention, fusion network, to learn the context by adding the similarity score between the original code and the retrieved code. First, we obtain each representation: the representation between code transformer encoder and summary transformer decoder, and the representation between summary transformer encoder and summary transformer decoder, respectively. Then, the two representations are fused via the following equation:

$$\alpha_1 = \text{softmax}(QK_1^{\intercal}) \tag{5}$$

$$\alpha_2 = \text{softmax}(QK_2^{\intercal}) \tag{6}$$

$$Att_1(Q, K_1, V_2) = \alpha_1 V_1 \tag{7}$$

$$Att_2(Q, K_2, V_2) = \alpha_2 V_2 \tag{8}$$

$$FusAtt = \frac{Att_1 + zAtt_2}{1 + z} \tag{9}$$

where α_1 is the attention score between the code transformer encoder and the summary transformer decoder, α_2 is the attention score between the summary transformer encoder and the summary transformer decoder, *FusAtt* is fusion network, and z is the similarity score. The fusion network of each transformer decoder layer learns that the code encoder representations Att_1 and summary encoder representation Att_2 are fused by reflecting the similarity between the original code and the retrieved similar code.

E. DUAL COPY MECHANISM

Finally, the summary transformer decoder predicts the t-th word using the extractive information as well as the abstract information about the similar summary information. We apply the modified copy mechanism [40], dual copy mechanism, to directly generate the word both from the original code tokens and from retrieved similar summary tokens for the extractive information. The probability of the word to be predicted at the t-th step is the sum of three scores: the attention score of the code transformer encoder α_1, the attention score of the summary transformer encoder α_2, and the t-th word prediction probability p_t from the final representation of the summary transformer decoder h_t. The dual copy mechanism is as follows:

$$P_{dual}(w) = \lambda_1\alpha_1 + \lambda_2\alpha_2 + \lambda_3 p_t \tag{10}$$

$$\lambda_1 = \sigma(W_1 h_t) \tag{11}$$

$$\lambda_2 = \sigma(W_2 h_t) \tag{12}$$

$$\lambda_3 = 1 - (\lambda_1 + \lambda_2) \tag{13}$$

where σ is a sigmoid function, W_1, W_2 are linear layer weight matrices for the code and summary copy probabilities. READSUM predicts the words of target summary by using context representations of the original code and its similar code (abstractive information), and selecting directly original code tokens and its retrieved similar summary tokens (extractive information).

IV. EXPERIMENT RESULTS

A. SETUP

We evaluate using the benchmarks of the Java dataset [4] and the Python dataset [24]. However, as mentioned in [41], there are duplication issues in the Java dataset, and the baselines used different BLEU variants as the evaluation metric. To ensure a fair comparison, we re-implement all the baselines and conduct experiments on the same benchmark datasets (including the deduplicated Java dataset). We also evaluate the baselines with the same evaluation metrics (including BLEU).

1) DATASET

We evaluate our proposed model on three datasets: the duplicated Java dataset, the deduplicated Java dataset, and the Python dataset. The duplicated Java dataset is split into 69,708/8,714/8,714 for train/valid/test. Since there are duplicate codes in the test set of the Java dataset, we removed them to create the deduplicated Java dataset split into 6,449 for test set. The Python dataset is split into 55,538/18,505/18,502 for the train/valid/test.

For obtaining ASTs of the Java and Python dataset, we use the *javalang*[1] and *ast*[2] library, respectively. Also, we tokenize the source code and the AST to subtokens as the form *CamelCase* and *snake-case*.

For a more detailed description of the datasets, please refer to Table 1.

2) HYPERPARAMETER

We limit the maximum AST and summary length to 200 and 50, and we set the training epoch to 100. The vocabulary sizes of code and summary are 50,000 and 30,000, respectively. For training the model, we use Adam optimizer [42].

We set the environment for training READSUM as follows: 4 NVIDIA 2080 Ti GPUs, Ubuntu 16.04, Python 3.9 and CUDA 10.2 version. The average training and inference time for READSUM takes about 40 and 0.5 hours, respectively. READSUM has about 87 million parameters.

The description of the dataset and hyper-parameter for the experiment is shown in Table 2.

3) EVALUATION METRICS

We use 4 evaluation metrics, BLEU [43], METEOR [44], ROUGE-L [45], and CIDEr [46] to measure the quality of the generated summaries. We conduct all experiments with BLEU with smoothing 4 based on the version of NLTK (3.6.7) The details regarding the evaluation metrics is as follows.

[1]https://github.com/c2nes/javalang
[2]https://github.com/python/cpython/blob/master/Lib/ast.py

TABLE 1. Statistics of Java dataset [4] and Python dataset [24]. For obtaining their corresponding ASTs, we use the *javalang* and *ast* library, respectively.

Dataset	Java	Python
Train	69,708	55,538
Valid	8,714	18,505
Test	8,714	18,502
Unique non-leaf nodes in ASTs	106	54
Unique leaf nodes in ASTs	57,372	101,229
Unique tokens in summaries	46,895	56,189
Avg. nodes in AST	131.72	104.11
Avg. tokens in summary	17.73	9.48

TABLE 2. Hyper-parameter of READSUM.

Hyper-parameter	Values
embedding size of AST token	512
embedding size of summary token	512
the layers of code transformer encoder	6
the layers of summary transformer encoder	6
the layers of summary transformer decoder	6
the number of multi-head	8
learning rate	0.0005
learning decay	0.99
batch-size	80
beam-size	5
early stop epoch	5

BLEU [43] is Bilingual Evaluation Understudy to evaluate the quality of generated code summaries. The formula of computing BLEU is as follows:

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

where p_n is the geometric average of the modified n-gram precisions, ω_n is uniform weights $1/N$ and BP is the brevity penalty.

METEOR [44] is used to measure the correlation between the metric scores and human judgments of translation quality. So unigram precision (P) and unigram recall (R) are computed and combined via a harmonic-mean. The METEOR score is computed as follows:

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

where *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 [45] is used to apply Longest Common Subsequence in sumarization evaluation. ROUGE-L used LCS-based F-measure to estimate the similarity between two summaries X of length m and Y of length n, assuming X is a reference summary sentence and Y is a candidate summary sentence, as follows:

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

TABLE 3. Comparison of READSUM with the baseline models on the duplicated Java datasets [4]. The improvements of READSUM over all baselines are statistically significant with $p < 0.05$.

Method	Java (Duplicated)						
	BLEU-1	BLEU-2	BLEU-3	BLEU-4	METEOR	ROUGE-L	CIDEr
Code-NN [1]	40.84	36.27	34.17	33.07	19.52	43.00	2.75
DeepCom [8]	24.74	20.88	18.73	17.54	10.78	26.81	1.20
TL-CodeSUM [4]	35.43	31.42	29.45	28.46	15.80	39.63	2.16
Astattgru [22]	37.86	34.21	32.68	31.98	18.89	41.03	2.61
NCS (code) [7]	51.84	48.09	46.11	45.05	27.10	55.21	3.95
NCS (AST)	50.36	46.85	45.03	44.05	26.68	53.36	3.84
Rencos [26]	50.31	46.92	45.09	44.04	25.14	53.73	3.74
CAST [16]	52.25	48.30	46.28	45.19	27.95	55.08	3.95
CodeBERT [35]	55.24	50.70	48.18	46.80	31.43	59.23	4.25
PLBART [38]	55.23	50.47	48.08	46.45	31.50	59.15	4.24
READSUM	56.18	51.63	49.17	47.75	31.85	59.34	4.36

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

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

CIDEr [46] is used to consider the frequency of n-grams for automatically computing consensus. $CIDEr_n$ score for n-grams of length n is computed using the average cosine similarity between the candidate sentence and the reference sentences, which accounts for both precision and recall:

$$CIDEr_n(c_i, S_i) = \frac{1}{m}\sum_j \frac{g^n(c_i)\cdot g^n(s_{ij})}{||g^n(c_i)||||g^n(s_{ij})||}$$

$$CIDEr(c_i, S_i) = \sum_{n=1}^{N} \omega_n CIDEr_n(c_i, S_i)$$

where $g^n(c_i)$ is a vector formed by $g_k(c_i)$ corresponding to all n-grams of length n, $||g^n(c_i)||$ is the magnitude of the vector $g^n(c_i)$, and ω_n is uniform weights $1/N$.

4) BASELINES

We experiment with reproducible code summarization models as our baselines:

- Code-NN [1] is a code summarization model that uses LSTM networks with an attention mechanism to generate summaries from code.
- RL+Hybrid2Seq [24] is a reinforcement learning-based code summarization model that combines an LSTM encoder with an AST-based LSTM to improve code summarization.
- DeepCom [8] is a neural code generation model that uses a structure-based traversal method to obtain a sequence of tokens from an AST.
- TL-CodeSUM [4] is a transfer learning-based code summarization model that introduces API information during comment generation, even without prior knowledge of the API.
- Astattgru [22] is an attention-based neural model for code summarization that utilizes an AST structure to generate summaries.
- NCS (code) [7] is the first model to use the Transformer architecture and includes a copying mechanism for generating words from a vocabulary.
- NCS (AST) is an alternative version of the NCS model that utilizes abstract syntax trees as input instead of code.
- Rencos [26] is a neural model based on a retrieval method that uses a sequence-to-sequence approach with a combination of syntax and semantic embeddings.
- CAST [16] is a neural model for code generation that uses a graph-based attention mechanism to generate code from natural language descriptions.
- CodeBERT [35] is a pre-trained language model for code that uses a transformer-based neural network architecture similar to BERT [34], optimized for programming tasks.
- PLBART [38] is a pre-trained language model for programming tasks that combines BART [37] with an encoder-decoder architecture and is optimized for both code understanding and generation.

We tried to re-implement and report on all available baseline models for code summarization, but we were unable to do so due to some unfairness issues and the absence of open-source codes for some works. For example, [16] and [24] removed code samples that were not parsed with *Antlr* parser, so they cannot be evaluated with the same datasets as other baselines. Some works performed preprocessing specialized to a specific program language, and conducted experiments on either the Java or the Python dataset. So, the baselines for the Java and the Python are different. We evaluate all reproducible baselines with the same evaluation metrics on the same datasets to ensure fair comparisons across all baselines as much as possible.

B. QUANTITATIVE RESULT

1) OVERALL RESULT

Table 3 shows the overall performance of the models on both the duplicated Java benchmark dataset. The READSUM model achieved state-of-the-art performance on all evaluation metrics for the Java dataset. Compared to other models such as Code-NN, DeepCom, TL-CodeSUM, and ASTattgru, the NCS model based on the Transformer architecture

TABLE 4. Comparison of READSUM with the baseline models on the deduplicated Java datasets. The improvements of READSUM over all baselines are statistically significant with $p < 0.05$.

Method	Java (Deduplicated)						
	BLEU-1	BLEU-2	BLEU-3	BLEU-4	METEOR	ROUGE-L	CIDEr
Code-NN [1]	23.92	18.63	16.44	15.44	10.25	26.67	0.91
DeepCom [8]	13.80	10.8	9.41	8.77	5.84	15.88	0.37
TL-CodeSUM [4]	23.49	19.03	16.98	16.05	9.28	27.67	0.89
Astattgru [22]	20.38	15.78	13.91	13.12	8.98	24.01	0.65
NCS (code) [7]	36.68	31.74	29.17	27.83	16.77	40.87	2.12
NCS (AST)	34.42	29.79	27.42	26.17	15.89	38.16	1.94
Rencos [26]	35.94	31.50	29.13	27.79	15.92	39.75	2.06
CodeBERT [35]	40.63	34.61	31.29	29.49	20.62	45.81	2.43
PLBART [38]	41.32	35.07	31.56	29.53	21.30	45.79	2.46
READSUM	41.90	35.94	32.76	30.98	21.52	46.03	2.58

TABLE 5. Comparison of READSUM with the baseline models on Python dataset [24]. The improvements of READSUM over all baselines are statistically significant with $p < 0.05$ except BLEU-4 of CodeBERT.

Method	Python						
	BLEU-1	BLEU-2	BLEU-3	BLEU-4	METEOR	ROUGE-L	CIDEr
CODE-NN [1]	38.74	29.79	26.41	24.98	17.97	37.16	2.01
RL+Hybrid2Seq [24]	25.46	20.90	19.30	18.91	9.39	30.19	0.80
NCS (code) [7]	43.11	36.82	33.94	32.75	20.17	47.20	2.35
NCS (AST)	40.51	34.47	31.84	30.77	18.85	44.29	2.11
Rencos [26]	41.42	36.31	33.97	33.04	19.50	45.45	2.25
CodeBERT [35]	45.94	38.95	35.65	34.17	22.26	49.70	2.54
PLBART [38]	44.53	37.55	34.31	32.91	21.36	49.12	2.44
READSUM	46.54	39.38	35.86	34.01	22.86	50.49	2.71

demonstrated much better performance, indicating the advantage of using a transformer-based model to capture sequence information for code summarization. Furthermore, the Rencos model, which utilizes a retrieval method, showed better performance than other baseline models. These results highlight the effectiveness of transformer-based models and retrieval methods in improving code summarization. Compared to CAST [16], which used two types of code and AST as input data, READSUM shows good performance using AST tokens and retrieved summary tokens. Furthermore, READSUM is compared with Rencos [26], which is one of the retrieval methods. Lastly, our approach is compared with CodeBERT and PLBART, strong pre-trained program language models. READSUM performs better than the pre-trained models trained on large code data.

Table 4 shows the overall performance of the models on the deduplicated Java dataset. The READSUM model achieved state-of-the-art performance on all evaluation metrics for the Java dataset. Similar to the previous result, READSUM achieved state-of-the-art performance on the deduplicated Java dataset. Evaluating a test dataset that has not been seen in the training dataset is a challenging task, but READSUM demonstrated superior performance compared to other sequence-to-sequence models. Furthermore, READSUM outperformed large language models such as CodeBERT and PLBART. The improvements of READSUM over all baselines are statistically significant with $p < 0.05$. READSUM demonstrates its ability to effectively capture both the structural and sequence information of code to generate high-quality summaries.

TABLE 6. Ablation study on each component of READSUM.

Component	BLEU-4	METEOR	ROUGE-L
	Java Dataset		
READSUM	47.85	31.80	59.41
w/o AdapAtt	47.63	31.48	59.20
w/o FusAtt	47.55	31.51	59.14
w/o DualCopy	47.46	31.37	59.16
	Python Dataset		
READSUM	34.05	23.21	50.49
w/o AdapAtt	33.71	22.46	50.07
w/o FusAtt	33.70	22.59	50.02
w/o DualCopy	33.54	22.51	50.24

Table 5 shows the performance of the models on the Python dataset. READSUM outperforms all the baselines except for CodeBERT's BLEU-4. We believe that CodeBERT has good performance of BLEU score because it resembles the data used for fine-tuning after pre-training. However, READSUM had better overall performance than simply having a high BLEU-4 score. Our READSUM showed that focusing on the retrieved similar summary rather than the similar code as retrieval information helps to predict more important and similar words in generating the summary.

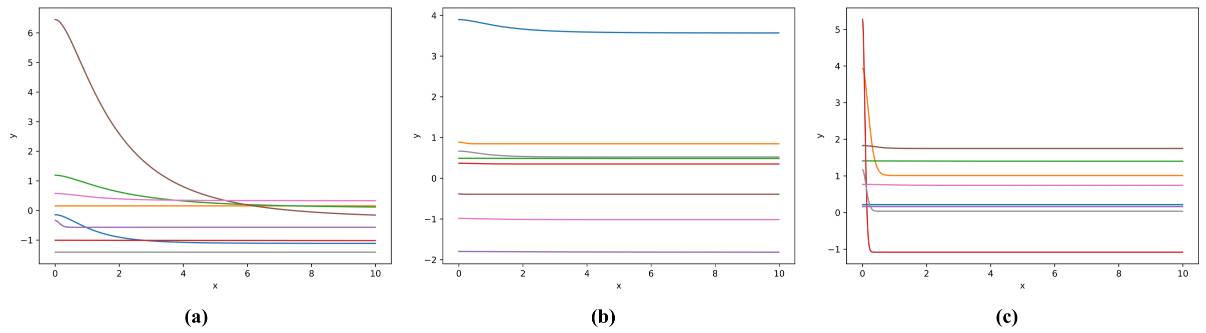


FIGURE 4. Analysis of adaptive attention network. (a), (b) and (c) show the learned bias terms in 8 multi-heads for each of 2nd layer, 4th layer, and 6th layer in code transformer encoder. The results show that the adaptive attention network is adaptively trained on the tokens to focus within each multi-head self-attention.

TABLE 7. An qualitative example on the Java and Python dataset.

Code Language	Java	Python

Original Code	public void createClusterAsync(final String projectId,
	final ClusterCreateSpec clusterCreateSpec,
	final FutureCallback <Task> responseCallback)
	throws IOException {
	String path = String.format ("%s/%s/clusters",
	getBasePath(),
	projectId);
	createObjectAsync(path, serializeObjectAsJson(
	clusterCreateSpec),
	responseCallback);}
		def move(source, destination, use_sudo = False):
		func = ((use sudo and run a root) or run)
		func('/bin/mv {0}{1}'.format(quote(source),
		quote(destination)))

Retrieved Code	public void createVmAsync(final String projectId,
	final VmCreateSpec vmCreateSpec,
	final FutureCallback <Task> responseCallback)
	throws IOException {
	String path = String.format ("%s/%s/clusters",
	getBasePath(),
	projectId);
	createObjectAsync(path, serializeObjectAsJson(
	clusterCreateSpec),
	responseCallback);}
		def copy(source, destination, recursive=False,
		use_sudo=False):
		func = ((use_sudo and run_as_root) or run)
		options = ("-r " if recursive else "")
		func("/bin/cp 01 2".format(options, quote(source)
		, quote(destination)))

Retrieved Sum.	Create a vm in the specified project.	Copy a file or directory.
NCS	Create project within the specified project.	Move a source file or directory from a file.
Rencos	Create project within the specify tenant.	Copy a file or directory.
CodeBERT	Create a cluster in the project.	Move a source file or directory.
PLBART	Create a cluster definition for the specify project.	Move a file or directory cli example:.
READSUM	Create a cluster in the specify project.	Move a file or directory.
Ground Truth	Create a cluster in the specify project.	Move a file or directory.

2) ABLATION STUDY

We perform an ablation study by validating the effectiveness of each component of READSUM. The *w/o AdapAtt* is our method that transformer encoder consists of vanilla multi-head self-attention, the *w/o FusAtt* is our method that the similarity score z is 1 in fusion network, and the *w/o DualCopy* is our method with a copy mechanism that does only use AST code tokens.

Table 6 shows the experiment results of the ablation study for each of the three cases. Compared with the *w/o AdapAtt*, READSUM slightly increases BLEU, METEOR, and ROUGE-L scores for Java and Python datasets, respectively. Since the general multi-head self-attention method calculates the same attention score between all tokens, it can be seen that the performance is low due to insufficient learning about the AST structure. We confirm that the AST representation is learned both sequential and structural information by adaptive attention. Next, in the *w/o FusAtt*, the scores of BLEU/METEOR/ROUGE-L on Java are slightly increased, but the BLEU/METEOR/ROUGE-L scores on Python dataset are significantly increased by 0.35/0.62/0.40, respectively. This is because the code representation learned from the code transformer encoder and the retrieved similar summary representation learned from the summary transformer encoder are fused in the decoder phase, and the similarity between the original code and retrieved similar summary is important to reflect on the decoder representation. Finally, in the *w/o DualCopy*, the scores of BLEU/METEOR/ROUGE-L are increased by 0.39/0.43/0.25 in the Java dataset and 0.51/0.7/0.18 in the Python dataset. The hidden state of transformer summary decoder is obtained by fusion of augmented code representation from code transformer encoder and similar summary representation from summary transformer encoder. So the dual copy mechanism generates a summary with high quality by considering both the abstractive method in which the relation between the augmented code representations and the similar summary representations are fused, and the extractive method in which original code tokens and retrieved similar summary tokens are directly extracted.

C. QUALITATIVE RESULT

1) ANALYSIS ON ADAPTIVE ATTENTION

We analyze how adaptive attention learns structural and sequential information in our proposed model. Fig. IV-B1 shows the learned Gaussian function in 8 multi-heads in 2nd layer, 4th layer and 6th layer of code transformer encoder. In the 2nd layer of the transformer encoder, the Gaussian function value becomes larger for tokens with relatively close shortest paths, so that the attention value becomes larger as shown in Fig. IV-B1(a). In 4th layer, the attention values in all multi-heads are calculated equally for all shortest paths, so adaptive attention network learns to focus on the sequential information of the AST as shown in Fig. IV-B1(b). In Fig. IV-B1(c), some bias terms in multi-heads (Red and Orange) learn information about itself by increasing attention to its own token whose shortest path is 0. In other multi-heads (Other colors), it learns to have the same attention value for all shortest paths. Therefore, we show that adaptive attention network is adaptively trained to focus on which information within each multi-head self-attention.

2) COMPARISON WITH THE BASELINES

We show one example generated from READSUM, NCS, Rencos, CodeBERT and PLBART on the Java dataset and Python dataset as shown in Table 7. In an example of the Java dataset, NCS and Rencos generate sentences of the form *"create project within"*. Furthermore, CodeBERT and PLBART generate summaries that do not necessarily follow the format of the Retrieved Summary, instead producing very general sentence forms. These models, having learned many different types of sentences during pre-training, can generate natural summaries, but may not reflect important keywords or other specific information found in the source code. However, READSUM generates the sentence similar to the ground truth, because it copies the word *"cluster"* in original code tokens and the words *"in"* and *"project"* in the retrieved similar summary tokens through the dual copy mechanism.

V. CONCLUSION

We proposed a novel model for code summarization, **READSUM**, **RE**trieval-augmented **AD**aptive transformer for source code **SUM**marization. READSUM effectively combined an abstractive approach learning both the structural and sequential information of the source code, and an extractive approach for increasing the frequency of important keywords. For adaptively learning structural and sequential information, we modified the self-attention network in the transformer encoder by adding a bias term as a learnable Gaussian function with the distance between tokens. Also, We demonstrated through analysis of the adaptive attention network that our self-attention is learned adaptively for each layer. Finally, we improved the performance of summarization by using dual copy mechanism from the original code tokens and the retrieved similar summary words. We showed the superiority of READSUM for source code summarization through various experiments and human evaluation.

REFERENCES

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

[2] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proc. 33rd Int. Conf. Mach. Learn.* B. M. Florina and K. Q. Weinberger, Eds. New York, NY, USA, Jun. 2016, pp. 2091–2100.

[3] Y. Liang and K. Q. Zhu, "Automatic generation of text descriptive comments for code blocks," in *Proc. 32nd AAAI Conf. Artif. Intell. (AAAI), 30th Innov. Appl. Artif. Intell. (IAAI), 8th AAAI Symp. Educ. Adv. Artif. Intell.*, A. S. McIlraith and Q. K. Weinberger, Eds. New Orleans, LA, USA, Feb. 2018, pp. 5229–5236.

[4] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred API knowledge," in *Proc. 27th Int. Joint Conf. Artif. Intell.*, Stockholm, Sweden, Jul. 2018, pp. 2269–2275.

[5] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," in *Proc. Adv. Neural Inf. Process. Syst.*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. A. Buc, E. B. Fox, and R. Garnett, Eds. Vancouver, BC, Canada, 2019, pp. 6559–6569.

[6] W. Ye, R. Xie, J. Zhang, T. Hu, X. Wang, and S. Zhang, "Leveraging code generation to improve code retrieval and summarization via dual learning," in *Proc. Web Conf.*, Y. Huang, I. King, T. Liu, and M. van Steen. Taipei, Taiwan, Apr. 2020, pp. 2309–2319.

[7] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics*, 2020, pp. 4998–5007.

[8] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proc. 26th Conf. Program Comprehension*, May 2018, pp. 200–210.

[9] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization," in *Proc. 7th Int. Conf. Learn. Represent.*, New Orleans, LA, USA, May 2019, pp. 1–18.

[10] K. Zhang, W. Wang, H. Zhang, G. Li, and Z. Jin, "Learning to represent programs with heterogeneous graphs," in *Proc. 30th IEEE/ACM Int. Conf. Program Comprehension*, Vancouver, BC, Canada, May 2022, pp. 378–389.

[11] Y. Shido, Y. Kobayashi, A. Yamamoto, A. Miyamoto, and T. Matsumura, "Automatic source code summarization with extended tree-LSTM," 2019, *arXiv:1906.08094*.

[12] J. Harer, C. Reale, and P. Chin, "Tree-transformer: A transformer-based method for correction of tree-structured data," 2019, *arXiv:1908.00449*.

[13] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 783–794.

[14] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," 2020, *arXiv:2004.02843*.

[15] S. Liu, Y. Chen, X. Xie, J. K. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid GNN," in *Proc. 9th Int. Conf. Learn. Represent.*, 2021, pp. 1–16.

[16] E. Shi, Y. Wang, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, "CAST: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, Punta Cana, Dominican Republic, 2021, pp. 4053–4062.

[17] C. Lin, Z. Ouyang, J. Zhuang, J. Chen, H. Li, and R. Wu, "Improving code summarization with block-wise abstract syntax tree splitting," in *Proc. IEEE/ACM 29th Int. Conf. Program Comprehension (ICPC)*, May 2021, pp. 184–195.

[18] Y. Wang and H. Li, "Code completion by modeling flattened abstract syntax trees as graphs," in *Proc. AAAI*, 2021, pp. 14015–14023.

[19] H. Wu, H. Zhao, and M. Zhang, "Code summarization with structure-induced transformer," in *Proc. Findings Assoc. Comput. Linguistics*, 2021, pp. 1078–1090.

[20] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," 2018, *arXiv:1803.09473*.

[21] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *Proc. 7th Int. Conf. Learn. Represent.*, New Orleans, LA, USA, May 2019, pp. 1–22.

[22] A. LeClair, S. Jiang, and C. Mcmillan, "A neural model for generating natural language summaries of program subroutines," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 795–806.

[23] Y. Choi, J. Bak, C. Na, and J.-H. Lee, "Learning sequential and structural information for source code summarization," in *Proc. Findings Assoc. Comput. Linguistics*, 2021, pp. 2842–2851.

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

[25] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, "Retrieve and refine: Exemplar-based neural comment generation," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 349–360.

[26] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, Jun. 2020, pp. 1385–1397.

[27] J. Li, Y. Li, G. Li, X. Hu, X. Xia, and Z. Jin, "EditSum: A retrieve-and-edit framework for source code summarization," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2021, pp. 155–166.

[28] F. Chen, M. Kim, and J. Choo, "Novel natural language summarization of program code via leveraging multiple input representations," in *Proc. Findings Assoc. Comput. Linguistics*, Punta Cana, Dominican Republic, 2021, pp. 2510–2520.

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

[30] N. D. Q. Bui, Y. Yu, and L. Jiang, "InferCode: Self-supervised learning of code representations by predicting subtrees," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, May 2021, pp. 1186–1197.

[31] J. Guo, J. Liu, Y. Wan, L. Li, and P. Zhou, "Modeling hierarchical syntax structure with triplet position for source code summarization," in *Proc. 60th Annu. Meeting Assoc. Comput. Linguistics*, Dublin, Ireland, 2022, pp. 486–500.

[32] J. Zhang, M. Utiyama, E. Sumita, G. Neubig, and S. Nakamura, "Guiding neural machine translation with retrieved translation pieces," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics*, New Orleans, Louisiana, 2018, pp. 1325–1335.

[33] M. Xia, G. Huang, L. Liu, and S. Shi, "Graph based translation memory for neural machine translation," in *Proc. 33rd AAAI Conf. Artif. Intell. (AAAI), 31st Innov. Appl. Artif. Intell. Conf. (IAAI), 9th AAAI Symp. Educ. Adv. Artif. Intell.*, Honolulu, HI, USA, 2019, pp. 7297–7304.

[34] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Lang. Technol.*, vol. 1, Jun. 2019, pp. 4171–4186.

[35] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. Findings Assoc. Comput. Linguistics*, 2020, pp. 1536–1547.

[36] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "GraphCodeBERT: Pre-training code representations with data flow," 2020, *arXiv:2009.08366*.

[37] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics*, Jul. 2020, pp. 7871–7880.

[38] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Human Lang. Technol.*, 2021, pp. 2655–2668.

[39] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, N. Aidan Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, R. Garnett, Eds. Long Beach, CA, USA, 2017, pp. 5998–6008.

[40] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," in *Proc. 55th Annu. Meeting Assoc. Comput. Linguistics*, vol. 1, Vancouver, BC, Canada, Jul. 2017, pp. 1073–1083.

[41] E. Shi, Y. Wang, L. Du, J. Chen, S. Han, H. Zhang, D. Zhang, and H. Sun, "On the evaluation of neural code summarization," in *Proc. 44th Int. Conf. Softw. Eng.*, New York, NY, USA, May 2022, pp. 1597–1608.

[42] P. D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. 3rd Int. Conf. Learn. Represent.*, Y. Bengio and Y. LeCun, Eds. San Diego, CA, USA, May 2015, pp. 1–15.

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

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

[45] C.-Y. Lin and E. Hovy, "Manual and automatic evaluation of summaries," in *Proc. Workshop Autom. Summarization*, Barcelona, Spain, 2004, pp. 74–81.

[46] R. Vedantam, C. L. Zitnick, and D. Parikh, "CIDEr: Consensus-based image description evaluation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 4566–4575.

YUNSEOK CHOI received the B.S. degree in computer engineering from Sungkyunkwan University, Suwon, South Korea, in 2015, where he is currently pursuing the Ph.D. degree supervised by Prof. Jee-Hyong Lee. His research interests include natural language processing and code intelligence.

CHEOLWON NA received the B.S. degree in computer engineering from Kunsan National University, Kunsan, South Korea, in 2020. He is currently pursuing the Ph.D. degree with Sungkyunkwan University, Suwon, South Korea, supervised by Prof. Jee-Hyong Lee. His research interests include natural language processing and code intelligence.

HYOJUN KIM received the B.S. degree in computer engineering from Kyunghee University, Suwon, South Korea, in 2020. He is currently pursuing the master's degree with Sungkyunkwan University, Suwon, supervised by Prof. Jee-Hyong Lee. His research interests include natural language processing and code intelligence.

JEE-HYONG LEE received the B.S., M.S., and Ph.D. degrees in computer science from the Korea Advanced Institute of Science and Technology, Daejeon, South Korea, in 1993, 1995, and 1999, respectively. From 2000 to 2002, he was an International Fellow with SRI International, USA. In 2002, he joined Sungkyunkwan University, Suwon, South Korea, as a Faculty Member. His research interests include fuzzy theory and applications, intelligent systems, and machine learning.