IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

ALSI-Transformer: Transformer-Based Code Comment Generation with Aligned Lexical and Syntactic Information

YOUNGMI PARK[1], AHJEONG PARK[1], AND CHULYUN KIM.[1], (Member, IEEE)
[1]Department of Information Technology Engineering, Sookmyung Women's University, Seoul 04310, Korea

Corresponding author: Chulyun Kim (e-mail: cykim@sookmyung.ac.kr).

This research was supported by Culture, Sports and Tourism R&D Program through the Korea Creative Content Agency grant funded by the Ministry of Culture, Sports and Tourism in 2022 (Project Name: Development of software copyright application technology for fair trade and distribution, Project Number: R2022020041, Contribution Rate: 90%)

ABSTRACT Code comments explain the operational process of a computer program and increase the long-term productivity of programming tasks such as debugging and maintenance. Therefore, developing methods that automatically generate natural language comments from programming code is required. With the development of deep learning, various excellent models in the natural language processing domain have been applied for comment generation tasks, and recent studies have improved performance by simultaneously using the lexical information of the code token and the syntactical information obtained from the syntax tree. In this paper, to improve the accuracy of automatic comment generation, we introduce a novel syntactic sequence, Code-Aligned Type sequence (CAT), to align the order and length of lexical and syntactic information, and we propose a new neural network model, Aligned Lexical and Syntactic information-Transformer (ALSI-Transformer), based on a transformer that encodes the aligned multi-modal information with convolution and embedding aggregation layers. Through in-depth experiments, we compared ALSI-Transformer with current baseline methods using standard machine translation metrics and demonstrate that the proposed method achieves state-of-the-art performance in code comment generation.

INDEX TERMS program comprehension, comment generation, natural language processing, deep learning

Nomenclature

Acronyms

AST	Abstract Syntax Tree
CAT	Code-Aligned Type sequence
ISBT	Improved Structure-Based Traversal
SBT	Structure-Based Traversal

I. INTRODUCTION

CODE comments are the main factor that helps in understanding the working process of source codes. During software development and maintenance, developers usually take a significant amount of time to understand programs. Commented codes are easier to understand than uncommented codes, and high-quality code comments can effectively improve program comprehension [1]–[4]. However, writing comments and keeping them up to date requires substantial time and effort [5]. As the scale of open-source software grows, the need for code comment generation technology that automatically generates high-quality natural language comments increases. Code comment generation is currently an active research topic, and achievements in this work can also be adapted to other natural language processing (NLP) tasks such as code search, code translation, and code classification.

Early code comment generation techniques used template-based methods with predefined generation rules and information-retrieval-based approaches with code summarization dataset. With the development of deep learning, various excellent models in the NLP domain have been applied to programming language domains, particularly code comment generation. Hu *et al.* [6] showed a significant performance improvement in code comment generation by treating comment generation as a neural machine translation (NMT) problem of converting code comments.

State-of-the-art models [4], [7] with recurrent neural network (RNN) [8], Seq2Seq [9], and transformer [10] to gener-

ate comments generally used structural information obtained by various abstract syntax tree (AST) traversal methods of source code, such as the structure-based traversal (SBT) method. This indicates that simultaneously using the lexical and syntactic information of a programming language increases the accuracy of translation from the source code to the comment.

In this paper, we emphasize the importance of aligning lexical and syntactic information. Many studies [4], [6], [7], [11] have used SBT, where the order and number of tokens differ from those of the corresponding keywords in the original source code. In NLP, it has been commonly observed that machine translation performs better between languages with similar word order. For instance, in English-German and English-Korean translations, the former has similar word order [12], [13] while the latter does not, and machine translation naturally performs better for English-German translation. Hence, if the order and length of lexical and syntactic information differ, as in SBT, learning with these multi-modal information may have a limited accuracy improvement.

Therefore, we propose the Aligned Lexical and Syntactic information Transformer (ALSI-Transformer) to synchronize the order and length of lexical and syntactic information. ALSI-Transformer uses two types of sequence information extracted from the source code as multi-modal features: a code sequence, which represents the lexical information, and a new data type called Code-Aligned Type sequence (CAT), which represents the syntactic information. These representations are combined into one piece of information using the convolution layer and the embedding aggregation layer. By leveraging this multi-modal information, ALSI-Transformer can generate high-quality comments. We demonstrate the effectiveness of ALSI-Transformer in code comment generation through thorough experiments.

Our contributions are as follows:

- We propose a new data type, CAT; to the best of our knowledge, we are the first to align the order of syntactic information to lexical information.
- The size of ALSI-Transformer is smaller than those of related models while achieving excellent comment generation performance.
- We evaluated the performance of several methods to create a combined feature that merges two inputs. The performance of Gate Network was the best, and it was applied to ALSI-Transformer.
- We compared ALSI-Transformer with six baselines on two performance metrics, BLEU and METEOR; which achieved state-of-the-art performance.

The remainder of this paper is organized as follows. Section II presents background methods on code summarization. Section III elaborates on the details of ALSI-Transformer. Section IV presents the experimental setup, results, and discussion. Finally, Section VI concludes the paper and presents potential future works.

II. RELATED WORK

A. TEMPLATE-BASED CODE SUMMARIZATION

Generally, template-based approaches follow some custom rules and extract keywords from the source code. Sridhara *et al.* proposed an automatic comment generation technique that extracts keywords from the Java method and describes the functionality in [1] and an automatic technique that identifies code snippets and implement high-level abstractions of behavior and represents it as a natural language description in [14].

McBurney and Mcmillan [15] used context-based methods PageRank and SWUM to extract keywords from a Java source code. Moreno *et al.* [16] generated code comments on the responsibility of Java classes using class stereotype identification and heuristic rules. However, template-based approaches do not capture the intrinsic attributes of the source code.

B. RETRIEVAL-BASED CODE SUMMARIZATION

Information retrieval-based approaches return the best matching code summarization from similar code snippets with similar correlations in the code dataset. Haiduc *et al.* [17] used a text retrieval technique and vector space model for code analysis to calculate code similarity from large datasets. They investigated a program comprehension that automatically determines code descriptions based on an automated text summarization technique in [18]. Wong *et al.* [19] collected a large Q&A dataset on Stack Overflow and proposed a model that generates code comments by mapping an input code to codes in Stack Overflow via similarity calculation. However, this method depends on the source code dataset and fails to retain rich contextual information about the source code.

C. LEARNING-BASED CODE SUMMARIZATION

After the two aforementioned methods, advances in machine learning and deep learning models have led to learning-based code annotation generation approaches [20]. Hu *et al.* [6] considered the task of generating comments from the source code as a machine translation problem from programming language to natural language and applied the NMT model to this problem, outperforming the existing methods of code summarization. CODE-NN [20] achieved high performance by using a long short-term memory (LSTM) network for the first time. Initially, the source code was simply regarded as plain text, and features of various programming languages were extracted [18]. In this method, identifiers or keywords in the source code can be extracted, but intrinsic attributes, such as the operation flow of the code, may not be retained. To overcome this problem, structural information of the source code has been utilized by using various methods of traversing the AST [21]. Deepcom [6] utilizes the traditional SBT and Hybrid-DeepCom [11] created comments on Java methods through an encoder–decoder structure with attention. Yang *et al.* [7] used Sim_SBT, and SeTransformer [4] used improved structure-based traversal (ISBT) to deduplicate SBT and

improve its representation. They used the transformer [22] instead of an RNN, which has a long-term dependency on long input data.

Most AST traversal methods generate sequences for code information focusing on structural information. However Aljumah *et al.* [23] compared source code summary experiments with and without AST, and found that AST information does not improve model performance in all situations although the text of function can benefit from AST if it is not clear. Therefore, we designed a language model using CAT with a new sequence by adopting a different source code structure extraction method.

III. INPUT FORMAT AND NETWORK MODEL

In this section, the overall framework of ALSI-Transformer is explained in detail. Figure 1 illustrates the overall structure of the ALSI-Transformer model. We describe ALSI-Transformer in three essential parts: source code information, input embedding, and model.

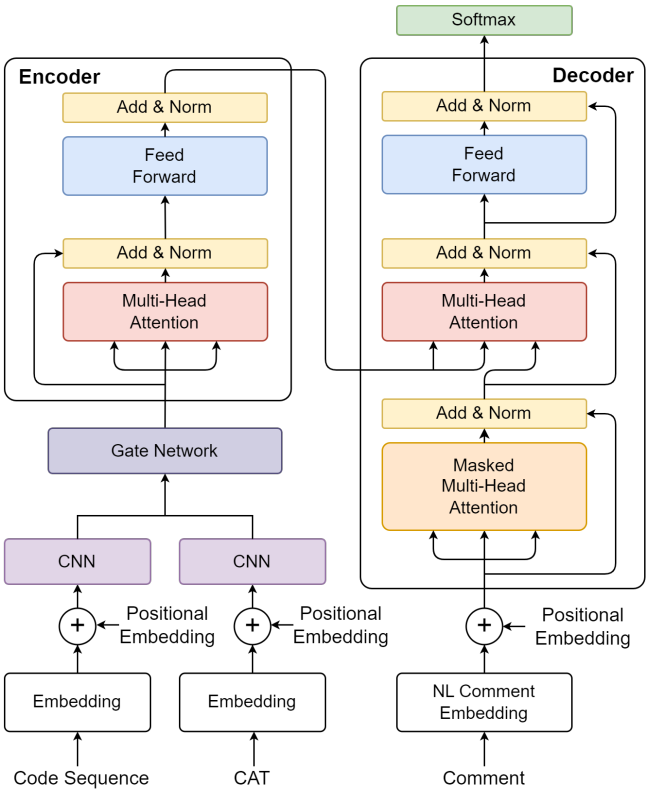


FIGURE 1. Structure of the ALSI-Transformer.

A. SOURCE CODE INFORMATION

1) Code Sequence

A code sequence is a lexical representation of the source code. Most tokens in the source code are composed of identifiers, separators, and operators. Similar to subjects and objects in natural language, the code also has lexical information.

2) Code-Aligned Type sequence (CAT)

One of the key points in ALSI-Transformer, CAT, is a syntactic representation of the source code. Haiduc *et al.* [18] used only lexical information using code tokens for comment generation. Hybrid-DeepCom [11], ComFormer [7], SeCNN [24], and SeTransformer [4] use the source code structure information as well as the source code in comment generation. These methods use lexical and syntactic information together.

We extracted the structural information of the source code to obtain the lexical and syntactic information. The biggest difference from previous studies is that when obtaining the structure information of the source code, CAT aligns the types according to the code sequence. Previous studies [4], [7], [11], [25] utilized SBT that traverses the AST extracted from the source code. Various methods for traversing the AST such as Sim SBT and ISBT have emerged; however these methods contain redundant and unnecessary information. Most importantly, the code token and SBT sequence are not aligned because some code tokens are lost in the process of converting the source code to AST. In contrast, CAT minimizes information loss when extracting the code structure from the source code. Specifically, we extract the AST generated from the source code. The AST is a tree that separates the source code written in a programming language into semantic units and change them into a structure that can be understood by a computer. Each node in this tree has source code information, such as node id, value, type, and child nodes. We tokenize the source code and extract the type information. [1]

Then, the type information of each code token is arranged according to the order of the code sequence. In this manner, a CAT with the same sequence and length as the code sequence is extracted. Therefore, we can obtain better syntactic information than the existing SBT.

Examples of SBT and CAT are shown in Figure 2. Because SBT creates a sequence according to the syntax tree order rather than the code order, the *type* information of some code tokens is lost. Additionally, there are cases where parentheses and types overlap, making the SBT sequence much longer than the original code sequence. Moreover, code type information is missing in SBT, but CAT has type information for all code sequences. Therefore, in CAT, the code sequence and the mapped code type sequence correspond, and both have the same length. In Figure 2, *Modifier*, the type of 'public', *Separator*, the type of 'parentheses, comma, semicolon', and *Keyword*, the type of 'BOOL_', are omitted in SBT, whereas the types of all code tokens are shown in CAT. In addition, the order of *ReferenceType*, which is an 'AnnotatedPrimitive Type' type, and *FormalParameter*, which is a 'type' type, is reversed in SBT, but CAT follows the code sequence order.

[1] Javalang https://pypi.org/project/javalang/

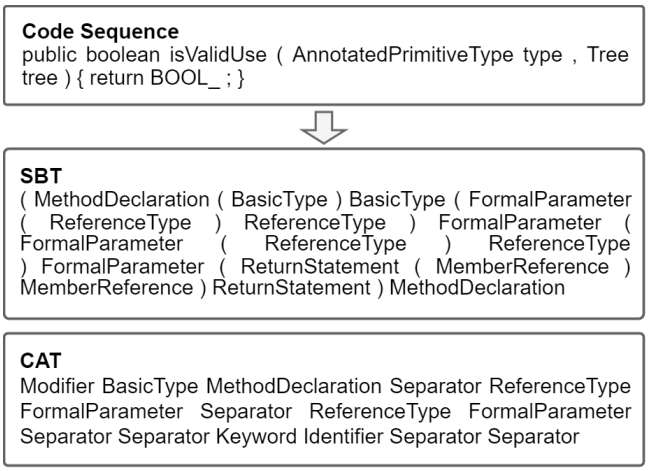


FIGURE 2. Example of SBT and CAT.

3) Out-of-Vocabulary (OOV) Problem

Code identifiers have many variations depending on the developer's naming preferences and may follow the naming convention for each programming language (e.g., camel case naming convention, snake case naming convention).

Therefore, the out-of-vocabulary (OOV) problem tends to occur in such data. To alleviate the OOV problem, we utilize the byte pair encoding (BPE) algorithm [26]. BPE is a preprocessing operation that separates and encodes tokens into smaller units of meaningful sub-tokens through subword segmentation. For example, there is the word 'representation' in the training set and 'present' in the test set; 'present' does not appear in the training set. In this case, the BPE algorithm splits the word 'representation' into 're', 'pre', 'sent', 'a', 't', 'i' and 'on'. A Subsequently, the decoder can create the word "present" from the fragmented word tokens. The vocabulary size is set to 52,000, and special tokens include <start> for the start of the input, <end> for the end of the input, <pad> for padding, and <unk> for OOV.

B. ALSI-TRANSFORMER

1) Convolution Layer

CNN can reduce the size (e.g., length and width) of images and minimize computational load [27]. Research to improve the performance in NLP tasks by combining CNN and transformer is being actively conducted [28]. Similarly, ALSI-Transformer uses CNNs to process input embeddings. In most cases, the length of the source code and the large dimensions make training computationally expensive. Therefore, CNNs can be used to compress the dimensions of the data and shorten the training time.

A previous study confirmed the effectiveness of compressing the code using CNN [4]; the local information (i.e., context and order) of the source code was preserved through CNN and reflected in learning. In addition, we aim to express the contextual meaning intensively. After this series of operations, the input data code sequence and CAT can effectively reduce the dimensions of each unique piece of information while minimizing the loss.

2) Embedding Aggregation Layer

Our proposed model utilizes the code sequence and CAT as inputs. Because ALSI-Transformer is a one-encoder-based model, we must merge two inputs. These two inputs can be merged in various ways. When inputs are fed into a single encoder, to reflect the advantage of using the same length and order of lexical and syntactic information, it is important to adopt an appropriate method to generate combined features. We experimented with six aggregation methods and finally chose Gate Network. The experiment is detailed in Section IV-B3.

The ALSI-Transformer jointly encodes the lexical and the syntactic representations through Gate Network and creates a combined feature X. Given the code sequence input $x_c = (x_{c1}, \cdots, x_{c768})$ and CAT input $x_q = (x_{q1}, \cdots, x_{q768})$, we set axis = -1 for concatenation. Then, Gate Network generates a gate context C_g as

$$C_g = \sigma(FC([x_c, x_q])), \tag{1}$$

where $\sigma(\cdot)$ is a logistic sigmoid function, and FC denotes a fully-connected layer. The combined feature is defined as

$$X = C_g \cdot x_c + (1 - C_g \cdot x_q) \tag{2}$$

and is fed to the encoder.

3) Encoder and Decoder

We constructed model components and implemented computational processes according to the standard transformer [22]. The encoder is represented in the block on the left of Figure 1, and the decoder is in the block on the right. Both the encoder and decoder are configured in a stacked form of multi-head attention and feed-forward layers. The positional embeddings were added at the bottom of the encoder and decoder to reflect the token order of the code sequence, CAT, and natural language. Detailed training settings and hyperparameters are given in Section IV-A3

Encoder: The proposed model has one encoder, which comprises $N = 3$ identical layer stacks. The encoder uses combined feature X as the input. The encoder inputs first flow through a self-attention $Attention(Q, K, V)$ layer, and the outputs are fed to feed-forward neural networks. Finally, the combined embeddings learned from the aligned lexical and syntactic information of the source code are passed to the decoder.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{3}$$

Decoder The decoder is composed of a stack of $N = 3$ identical layers. Masked multi-head attention is added, and the structure of the rest of the decoder is the same as that of the encoder. The masked multi-head attention masks certain

values so that they are not affected when the parameters are updated. The mask keeps the decoder from viewing future information. The combined embeddings are passed through the N decoder layers along with the natural language comments and target mask. Finally, an output sequence $(y_1, ..., y_n)$ is generated.

IV. EXPERIMENTS

In this section, we present the verifications performed to evaluate the performance and quality of the automatically generated code comments by our proposed method. In Section IV-A, we introduce the experimental setup. In detail, we present the statistics of the code corpus and the performance metrics for verifying ALSI-Transformer. Moreover, we describe the training details of each component in ALSI-Transformer and training environments. In Section IV-B, we explain the experimental results of ALSI-Transformer compared to baselines and present examples of how our proposed approach generates the comment.

A. SETUP

1) Code Corpus

In the experiments, we used the code corpus collected by Hu *et al.* [11], which has been widely used in previous studies on code comment generation [7], [11], [29], [30]. Each data consists of <java method-comment> pairs, and this code corpus has been refined according to a set of rules to ensure quality. The constant numbers and strings in the source code are replaced with <num_> and <str_> tokens, respectively. The train, test, and validation sets consisted of 445,812, 20,000, and 20,000 pairs, respectively. Statistical information about the code length, CAT length, and comment length is presented in Table 1.

TABLE 1. Statistics of code corpus lengths (Code, CAT, and Comment).

Statistics for Code and CAT Length					
Avg. tokens	Mode	Median	max	min	std
53.78	10.00	38.00	199.00	5.00	44.09
Statistics for Comment Length					
Avg. tokens	Mode	Median	max	min	std
10.36	8.00	9.00	34.00	0.00	4.75

2) Performance Metrics

Our study used two neural machine translation metrics, bilingual evaluation understudy (BLEU) and metric for evaluation of translation with explicit ordering (METEOR), to evaluate the quality of hypothesis comment generated by the model compared to the reference comment. The details of each metric are as follows:

BLEU is the basic metric to evaluate the performance of NMT models [31]. It is used to measure the similarity by comparing the degree of matching n-grams $(N = 1 \sim 4)$ in the hypothesis and reference. In addition, unigrams are used to measure word translation accuracy, and high-order n-gram is used to measure the fluency of sentence translation. METEOR is a method that supplements some inherent defects of the standard BLEU [32]. It is based on the weighted harmonic mean of single precision and recall of single word, and it computes word matches and concordance relationships between synonyms, roots, affixes, and definitions.

3) Training Details

Our source code is available in the GitHub repository.[2] ALSI-Transformer was implemented with Tensorflow 1.13.1 and Python 3.7. All the experiments were run on a computer having following specifications: NVIDIA Corporation GP102, GeForce GTX 1080 Ti GPU, and the operating system was Linux.

We adopted the hyperparameters commonly used in other code comment generation studies: Hybrid-Deepcom [11] Se-Transformer [4] and transformer [22]. In addition, to ensure fairness, additional comparative experiments were conducted using a test set and validation set. During training, the model was validated every 5000 steps using the BLEU and METEOR score as terms. As a result of hyperparameter adjustments, we determined the optimal combination of parameters as the final parameters of our proposed method. The hyperparameters of the model were set as follows:

- We used a CNN with 2 layers, a 3 × 1 convolution kernel, and a 2 × 1 filter for max pooling.
- Transformer had 3 layers, the hidden state was 768 dimensions, and the head of the multi-head attention was 8. Both the encoder and decoder had the same settings. We analyzed the size of the hidden layer of the neural network. The hidden state was tested with 256, 518, and 768 sizes. The results are presented in Table 2.

TABLE 2. Performance comparison under different hidden state sizes

Hidden State Size	BLEU	METEOR
256	47.12	61.50
512	51.95	64.96
768	53.80	66.11

- Because the length of the code sequence or CAT was less than 200, we set both the code sequence and CAT input length to 200 respectively. Thus, the total input length was 400(=200+200). To validate this input length, the performance was compared when the input length of the code sequence and CAT was set to 600(=300+300), and the resulting performance was better when set to 400(=200+200). The results are presented in Table 3.
- Both the code sequence and CAT were embedded in 768 dimensions. These two embeddings were connected in 1556 dimensions and fed to the Gate Network. The

[2]https://github.com/KIE-KID/ALSI-Transformer

TABLE 3. Performance comparison under different input length

Input Length	BLEU	METEOR
400	53.80	66.11
600	52.67	65.28

hidden state of the FC layer was adjusted to 768 dimensions.

- We chose the Adam optimizer with an initial learning rate of 1e-4. The learning rate was decayed at a rate of 0.99. The dropout of all models was set to 0.2. Since the models were tested with dropout rates ranging from 0.1 to 0.3, and the best performance was achieved with a dropout rate of 0.2. The results are presented in Table 4.

TABLE 4. Performance comparison under different dropout

Dropout	BLEU	METEOR
0.1	53.35	65.97
0.2	53.80	66.11
0.3	52.67	65.28

- Cross-entropy was used as the loss function. The best model was selected after training 50 epochs in the experiments.

B. RESULTS

1) Code Comment Generation

We compared the performance of our proposed method, ALSI-Transformer, with six state-of-the-art baselines in code comment generation. We used BLEU (n-gram BLEU) and METEOR to evaluate the performance by measuring the similarity between the reference comment and hypothesis comment generated by the ALSI-Transformer and the six baselines. The overall performance evaluation results are illustrated in Table 5; ALSI-Transformer outperformed the six baselines. ALSI-Transformer improved by 4.8%–33.54% in BLEU and 3.12%–42.82% in METEOR compared to baselines.

The results indicate that our proposed method achieved state-of-the-art performance in comment generation. This result proves the importance of using lexical information and syntactic information together. Therefore, simultaneously considering these two multi-modal information helps improve performance. Notably, BLEU 1 of SeTransformer is higher than that of ALSI-Transformer, but the scores of other indicators are low. BLEU 1, which indicates the accuracy of unigram token translation, is 5.52% lower than SeTransformer but 5.41% higher in BLEU 4, which indicates a high continuity of words being compared. This is because that the similarity between the reference comment and the comment created by ALSI-Transformer is high. Specifically, our proposed approach has an excellent ability to naturally generate comments. This can be confirmed through the example of the actual test set presented in Table 6.

In Table 6, we give examples of test cases to demonstrate the actual comments generated by the models. In Case A, the robustness for OOVs of the ALSI-Transformer is shown. SeTransformer [4] and Hybrid-DeepCom [11] generate <unk> tokens to represent out-of-vocabulary (OOV) words, whereas the ALSI-Transformer makes '*topic*' instead. This is due to the effect of the Byte Pair Encoding (BPE) algorithm in our proposed approach, which solves the OOV problem. DeepCom [6] performs poorly and generates irrelevant words and unnatural sentences. In Case B, it is demonstrated that the ALSI-Transformer is good at handling a long length of source code. The ALSI-Transformer employs specific words, such as '*array*' and '*offset*' and consequently generates comments matching the reference comments. In Case C, we observe that the ability to resolve OOVs of our proposed method is still effective with long source codes. While SeTransformer and Hybrid-DeepCom generate <unk> tokens for OOV words, our proposed method generates the same sentence as the reference comment by using 'sample' in the sentence, which other models miss. DeepCom generates numerous words that are irrelevant to the source code. In the last case, it is demonstrated that the ALSI-Transformer can deal with user-defined identifiers. The ALSI-Transformer is the only one that clarifies user-defined identifiers '*cachexmlgenerator*' and '*xml*' and completes the comment almost identically to the reference.

2) Training Efficiency

To demonstrate the training efficiency of ALSI-Transformer, we compared our proposed model with the state-of-the-art

TABLE 5. BLEU, n-gram BLEU, and METEOR score for ALSI-Transformer compared with six baselines.

Models	BLEU	BLEU 1	BLEU 2	BLEU 3	BLEU 4	METEOR
Seq2Seq (attention) [9]	37.87	46.53	41.53	37.81	35.04	23.29
Transformer [10]	45.55	55.62	46.30	41.57	38.69	29.06
DeepCom [6]	20.26	32.88	21.91	18.93	17.35	31.72
Hybrid-DeepCom [11]	38.20	51.63	40.56	36.70	34.41	51.26
ComFormer [7]	42.99	55.31	47.57	41.72	36.26	59.12
SeTransformer [4]	49.00	62.78	51.91	47.47	44.62	62.99
ALSI-Transformer	53.80	57.26	56.29	52.49	50.03	66.11

TABLE 6. Examples of generated comments by ALSI-Transformer and other baselines. These examples cover both long and short code snippets. Four code examples are selected from the test set to compare the reference comments and hypothesis comments. The hypothesis comments are generated by ALSI-Transformer and three baselines.

Case	Code Snippet and Comments
A	`public BaseTopicProperty( final String topic ) { this . topic = new SimpleStringProperty ( topic ) ; }` Reference comment: creates the base topic property object with the provided topic value . ALSI-Transformer: creates the topic property with the provided topic value . SeTransformer: creates the <UNK> object with the provided topic value . Hybrid-DeepCom: creates the <UNK> object with the provided topic id . DeepCom: create a multi valued rdn .
B	`public void write(byte b, int off, int len) throws IOException { ensureCapacity(len, BOOL_); if (bufferedBlockCipher != null) { int outLen = bufferedBlockCipher.processBytes(b, off, len, buf, NUM_); if (outLen != NUM_) { out.write(buf, NUM_, outLen); } } else if (aeadBlockCipher != null) { int outLen = aeadBlockCipher.processBytes(b, off, len, buf, NUM_); if (outLen != NUM_) { out.write(buf, NUM_, outLen); } } else { streamCipher.processBytes(b, off, len, buf, NUM_); out.write(buf, NUM_, len); } }` Reference comment: writes len bytes from the specified byte array starting at offset off to this output stream . ALSI-Transformer: writes len bytes from the specified byte array starting at offset off to this output stream . SeTransformer: writes a compressed message to this output stream . Hybrid-DeepCom: writes len bytes from this output stream . DeepCom: is the keycode currently being pressed ?
C	`public SampleInfo(DataInputStream is) throws IOException { numberOfChannels = is.readInt () ; sampleRate = is . readInt () ; coeffMin = is . readFloat () ; coeffRange = is . readFloat () ; postEmphasis = is . readFloat () ; residualFold = is . readInt () ; }` Reference comment: constructs a sample info from the given input stream ALSI-Transformer: constructs a sample info from the given input stream . SeTransformer: constructs a <unk> from the given input stream Hybrid-DeepCom: constructs a <UNK> that reads the pipeline from the given stream . DeepCom: construct a signalstrength object from the given parcel where the token is already been processed .
D	`private CacheXmlGenerator(ClientCache cache, boolean useSchema, String version, boolean includeKeysValues) { this .cache = (Cache) cache; this .useSchema = useSchema; this . version = CacheXmlVersion.valueForVersion(version) ; this .includeKeysValues = includeKeysValues; this . generateDefaults = BOOL_; if (cache instanceof ClientCacheCreation) { this . creation = (ClientCacheCreation) cache; this . creation . startingGenerate () ; } else { this . creation = new ClientCacheCreation () ; if (generateDefaults () || cache.getCopyOnRead()) { this . creation .setCopyOnRead(cache.getCopyOnRead()); } } }` Reference comment: creates a new cachexmlgenerator that generates xml for a given clientcache . ALSI-Transformer: creates a new cachexmlgenerator that generates xml for a given cache . SeTransformer: creates a new generator for a given cache . Hybrid-DeepCom: creates a counter so that it doesn t exist . DeepCom: creates a new <UNK> using the given parameters create a blocking .

model, SeTransformer [4], which has the most competitive performance. For accurate comparison, the experiment was conducted under the same conditions as the experimental environment. ALSI-Transformer adopted the hyperparameter that achieved the best performance, and SeTransformer was implemented according to [4]. Each experiment was conducted three times, and the average values were compared.

The results of comparison between ALSI-Transformer and SeTransformer are shown in Table 7. The experimental results indicate that ALSI-Transformer is computationally more efficient than SeTransformer. ALSI-Transformer outperforms the state-of-the-art model with a simple transformer model of one encoder. SeTransformer has a total number of parameters of 167,308,038 and ALSI-Transformer has 146,939,910, which shows a smaller number of parameters. In terms of training time, ALSI-Transformer took 174,279 seconds, whereas SeTransformer took 265,140 seconds. The training time of ALSI-Transformer was 1.52 times less than that of SeTransformer, and even considering the cost of calculating CAT in the original code, our proposed approach can be considered efficient. Additionally, the size of our proposed method was also 8.3 GB, which is smaller than the 10 GB of SeTransformer.

3) Aggregation Methods of Multi-modal Embeddings

The ALSI-Transformer uses Gate Network as an aggregation method of multi-modal information. To demonstrate the effectiveness of this embedding aggregation layer design, we designed and compared five methods of aggregating code sequences and CAT. In detail, *Alternation* and *Seperation* methods aggregate the code sequence and CAT in the step before the CNN layer. *Alternation* alternately merges the tokens of the code sequence and CAT, and *Seperation* uses a special token called <code> to connect the code sequence, <code> token, and CAT in that order. On the other hand, *Addition*, *Average* and *Concatenation* methods aggregate the code sequence and CAT embeddings in various ways after the CNN layer. *Addition* adds the values of the code sequence embedding and CAT embedding to combine the two, and *Average* uses the average of the two values. *Concatenation* uses data obtained by concatenating the embedding of code sequence and embedding value of CAT. In all the methods, the input was set to 768 dimensions, the same as the ALSI-Transformer. In particular, *Concatenation* adjusted the embedding ratio of code sequence to CAT to match the dimensions of the input embedding to 768. From the results of the experiments with the embedding ratios of 756 to 12, 750 to 18, and 700 to 68, we adopted the embedding ratio of 756 to 12, which yielded the best results. An example of the five methods is illustrated in Figure 3.

As shown in Table 8, the results of comparing various aggregation methods of code sequence and CAT confirm that the model performed best when using the *Gate Network*. The *Gate Network* improved by 0.59%–3.05% in BLEU and 0.22%–2.65% in METEOR compared to other models. This indicates that the *Gate Network* helps effectively aggregate code sequences and CAT. Therefore, in this paper, Gate Network was adopted as the optimal method and the model was called ALSI-Transformer.

TABLE 7. Comparison of the number of model parameters, training time, and model size between ALSI-Transformer and SeTransformer.

Models	# of Parameters	Training Time(s)	Model Size (GB)
ALSI-Transformer	146,939,910	174,279	8.3
SeTransformer [10]	167,308,038	265,140	10

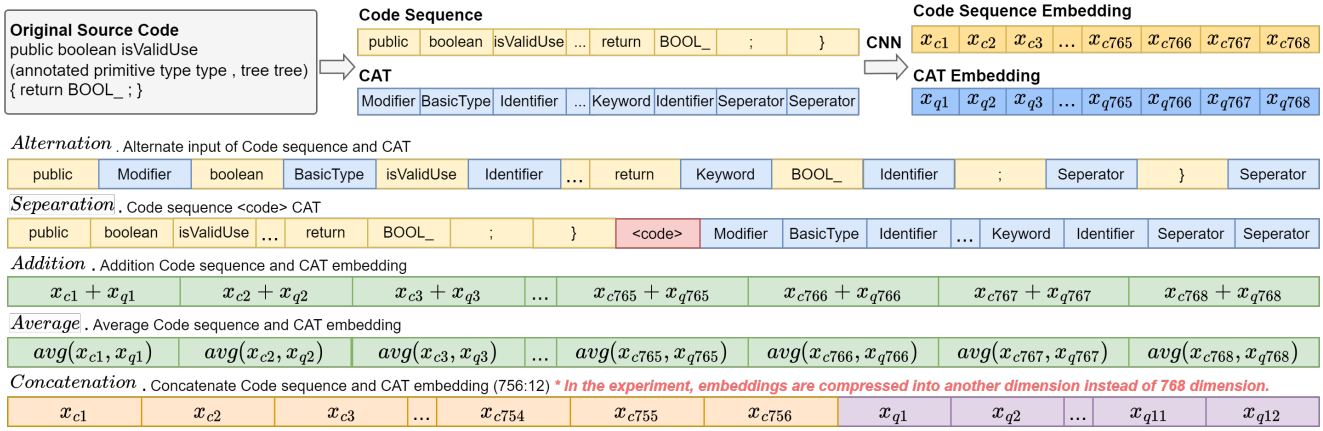


FIGURE 3. Example of five aggregation methods. Code sequence and CAT extracted from original source code used for $Alternation$ and $Separation$ methods. Code sequence embedding and CAT embedding are generated through the CNN layer. Two embeddings are used for $Addition$ and $Avgerage$. $Concatenation$ makes embedding in different ratios.

TABLE 8. Results of comparison between ALSI-Transformer (*Gate Network*) and five aggregation methods in terms of BLEU and METEOR.

Aggregation Method	BLEU	METEOR
Alternation	51.26	64.05
Separation	52.40	65.20
Addition	53.21	65.89
Average	50.75	63.46
Concatenation	51.41	64.23
Gate Network	53.80	66.11

4) Encoder Design

ALSI-Transformer uses one encoder to simultaneously encode multi-modal information. To verify the effectiveness of this encoder design, we compared it with a model that encodes lexical and syntactic information separately using two encoders. For a more accurate comparison, we designed a two-encoder ALSI-Transformer model using each code sequence and CAT as inputs instead of leveraging the existing model using two encoders. All other settings were the same. To ensure model adequacy, we also checked the model size after training. Model sizes on disk are measured in gigabytes (GB).

The comparison results are presented in Table 9. ALSI-Transformer was superior to the two-encoder ALSI-Transformer in all two indicators with BLEU 3.75% and METEOR 2.81%, and the model size was small at 8.3 GB. Furthermore, the single-encoder model achieved better performance than the two-encoder model. Because a single encoder is more suitable for this task, as shown in Table 5, our ALSI-Transformer model using one encoder had higher BLEU and METEOR scores than Hybrid-DeepCom and SeTransformer, which use two encoders.

5) Different Lengths of Source code and Comments

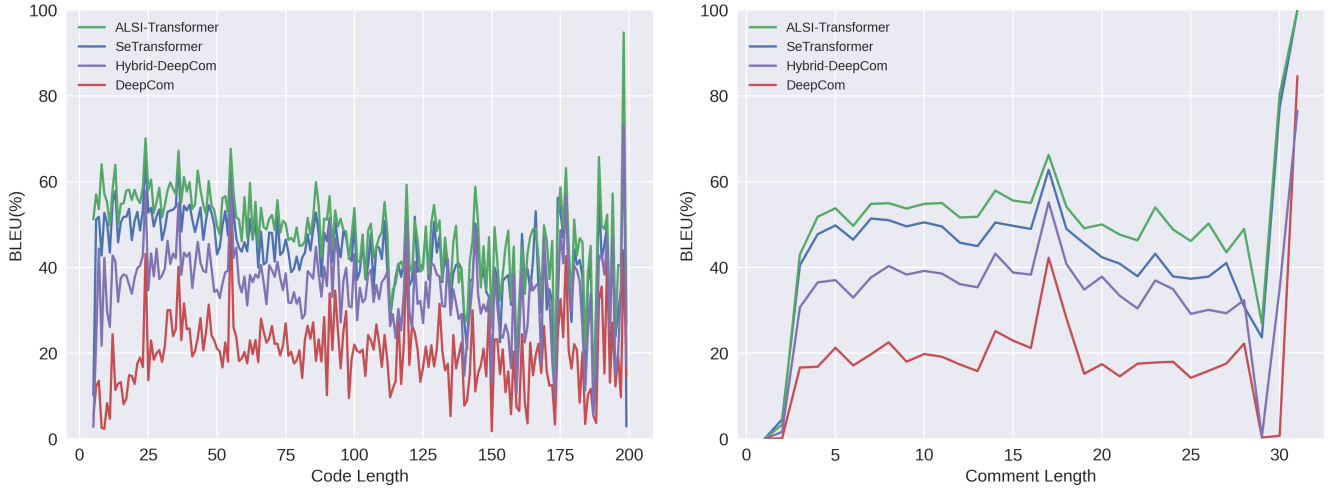
We further analyzed the impact of code and comment length on the performance of ALSI-Transformer. Code and comment length is one of the main factors affecting the performance of the code comment generation model. Therefore, we compared the performance of three baselines, DeepCom [6], Hybrid-Deepcom [11], and SeTransformer [4] with ALSI-Transformer when the code and comment lengths are different. The performance of our proposed approach and the other three baselines with different lengths of code and comments is illustrated in Figure 4.

In Figure 4(a), ALSI-Transformer outperforms the baselines in terms of BLEU score. As the length of the source code increases, the score tends to decrease, and the model has excellent performance when the length of the code is 50 or less. This shows that the model can better extract lexical information and syntactic information when the source code length is short. Performance fluctuates significantly when the length of the source code is 175 or more. This phenomenon can be attributed to the learning ability of our proposed approach being limited because there is little data with a length of 175 or more in the java method-comment code corpus we used.

As shown in Figure 4(b), our proposed approach does

TABLE 9. Comparison results between ALSI-Transformer and ALSI-Transformer (two-encoder).

Models	BLEU	METEOR	Model Size (GB)
ALSI-Transformer	53.80	66.11	8.3
ALSI-Transformer (two-encoder)	50.05	63.30	9.5



(a) BLEU score for different source code lengths

(b) BLEU score for different comment lengths

FIGURE 4. Performance of ALSI-Transformer, Deepcom, Hybrid-Deepcom, and SeTransformer with different lengths of code and comments

not show significant differences in performance in terms of comment length and has consistently higher BLEU scores than the baselines. When the length is 15-20, the score rises sharply, but this is the same phenomenon in all models, and it can be considered a dataset distribution problem rather than a problem of the model. For comments with a length of 30, ALSI-Transformer significantly outperformed DeepCom and Hybrid-Deepcom. However, long code comment generation task is still a challenge. This aspect is left for future work.

From the analysis, we surmised that the code comment generation ability was reduced when there are too many variable names, when a user-defined API was used, and when the source code processing complexity increased because the source code is written in one line. It is insufficient to generalize because the above cases were analyzed through human evaluation by three authors, but it would be interesting to find additional processing methods for source codes that yield poor performance through whole-case analysis.

V. THREATS TO VALIDITY

In this section, we discuss the following threats to our study:

A. INTERNAL THREATS

The first internal threat is the potential error of our experimental code. To alleviate this threat, the program code was checked and validated through multiple experiments. In addition, we used well-known mature open-source, such as TensorFlow[3], to ensure correct operation of the neural network.

The second internal threat is our training environment and hyperparameter configuration. We adopted the hyperparameters commonly used in other code comment generation studies [4], [11], [22]. In addition, to ensure fairness, additional comparative experiments were conducted using a test set and validation set, which is discussed in Section IV-A3.

The last internal threat is the result correctness of the baseline methods. To alleviate this threat, we re-implemented the structure and parameter settings of the baselines according to their respective studies.

B. EXTERNAL THREATS

The first external threat comes from code-comment corpus. To alleviate this threat, we selected a corpus, which was provided by Hu *et al.* [18]. This corpus consists of Java, which is the most popular programming language. Moreover, the quality of this corpus has been improved by preprocessing. It has also been used in previous studies on code comment generation [6], [7], [11], [29], [30], [33].

However, code comment generation ability may reduced when there are too many variable names, when a user-defined API was used, and when the source code processing complexity increased because the source code is written in one line. Therefore, In the future, we intend to collect more higher quality code-comment corpus and to apply our proposed method to the corpus of other programming languages (Python and Javascript).

[3][Online]. Available: https://www.tensorflow.org/

C. CONSTRUCT THREATS

The construct threat is the performance metric used in our study. We use three metrics, BLEU and METEOR, which are popular performance metrics in NMT. These measures have also been widely used in previous code comment generation studies [4], [6], [7], [11], [24], [34].

VI. CONCLUSION AND FUTURE WORK

In this study, we proposed a transformer-based code comment generation model ALSI-Transformer. Inspired by natural language translation, we first introduced CAT, which is aligned to code sequences. We improved its performance by further learning the syntactic representation. We also compressed the dimensions of the data to speed up the training process of neural networks with CNNs.

We conducted experiments to compare the performance of ALSI-Transformer with baselines on Java datasets. ALSI-Transformer achieved state-of-the-art performance. On comparing the training efficiency of ALSI-Transformer with existing state-of-the-art models, it outperformed the model because the size and training time were relatively small in ALSI-Transformer. In addition, we experimented with various methods of aggregating CAT and Code sequence and evaluated performance according to the number of encoders.

Because CAT, a key element in ALSI-Transformer, can be created by extracting the structure information of the source code using a parser, a parser specialized for each programming language can be used to generate CAT from the source code written in the corresponding programming language.

The proposed approach is a an NMT task from programming language to natural language. ALSI-Transformer can generate better functional-level code comments than existing models; however, it still does not provide relevance between functions or project-level descriptions. However, this is a common challenge in the study of code comment generation. The proposed model is applicable to project-level comment generation, which will be studied in the future.

In future work, we intend to apply the proposed model to different code corpus (e.g., Python and Javascript) and evaluate its effectiveness. Furthermore, to improve the performance of the proposed method, we plan to explore additional information available and apply other deep learning models. Recently, integrated models, such as CodeBERT [35] and CodeT5 [36], which simultaneously perform programming language understanding and generation tasks simultaneously have also emerged. Our proposed method can also be extended to tasks other than code comment generation. Additionally, because no case of using a new data type such as CAT has been reported, we intend to create a model applying these data types.

ACKNOWLEDGMENT

This research was supported by Culture, Sports and Tourism R&D Program through the Korea Creative Content Agency grant funded by the Ministry of Culture, Sports and Tourism in 2022 (Project Name: Development of software copyright application technology for fair trade and distribution, Project Number: R2022020041, Contribution Rate: 90%)

REFERENCES

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

[2] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," in *Proceedings of the 5th international conference on Software engineering*, 1981, pp. 215–223.

[3] H. He, "Understanding source code comments at large-scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1217–1219.

[4] Z. Li, Y. Wu, B. Peng, X. Chen, Z. Sun, Y. Liu, and D. Paul, "Setransformer: A transformer-based code semantic parser for code comment generation," *IEEE Trans. Reliab.*, pp. 1–16, 2022.

[5] F. Wen, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on code-comment inconsistencies," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 53–64.

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

[7] G. Yang, X. Chen, J. Cao, S. Xu, Z. Cui, C. Yu, and K. Liu, "Comformer: Code comment generation via transformer and fusion method-based hybrid code representation," in *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*, 2021, pp. 30–41.

[8] A. Graves, "Long short-term memory," *Supervised sequence labelling with recurrent neural networks*, pp. 37–45, 2012.

[9] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[10] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014.

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

[12] T. Baldwin and H. Tanaka, "The effects of word order and segmentation on translation retrieval performance," in *COLING 2000 Volume 1: The 18th International Conference on Computational Linguistics*, 2000.

[13] R. Murthy V, A. Kunchukuttan, and P. Bhattacharyya, "Addressing word-order divergence in multilingual neural machine translation for extremely low resource languages," *arXiv preprint arXiv:1811.00383*, 2018.

[14] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *2011 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 101–110.

[15] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 279–290.

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

[17] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *2010 acm/ieee 32nd international conference on software engineering*, vol. 2, 2010, pp. 223–226.

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

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

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

[21] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang*, vol. 3, no. POPL, pp. 1–29, 2019.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[23] S. Aljumah and L. Berriche, "Bi-lstm-based neural source code summarization," *Appl. Sci.*, vol. 12, no. 24, p. 12587, 2022.

[24] Z. Li, Y. Wu, B. Peng, X. Chen, Z. Sun, Y. Liu, and D. Yu, "Secnn: A semantic cnn parser for code comment generation," *J. Syst. Softw*, vol. 181, p. 111036, 2021.

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

[26] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.

[27] F. Jiang, W. Tao, S. Liu, J. Ren, X. Guo, and D. Zhao, "An end-to-end compression framework based on convolutional neural networks," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 28, no. 10, pp. 3007–3018, 2017.

[28] S. Khan, M. Naseer, M. Hayat, S. W. Zamir, F. S. Khan, and M. Shah, "Transformers in vision: A survey," *ACM Comput. Surv.*, vol. 54, no. 10s, pp. 1–41, 2021.

[29] W. Wang, Y. Zhang, Y. Sui, Y. Wan, Z. Zhao, J. Wu, P. Yu, and G. Xu, "Reinforcement-learning-guided source code summarization via hierarchical attention," *IEEE Trans. Softw. Eng.*, vol. 48, no. 1, pp. 102–119, 2022.

[30] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," *Advances in neural information processing systems*, vol. 32, 2019.

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

[32] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 2005, pp. 65–72.

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

[34] 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 *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 397–407.

[35] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv:2002.08155*, 2020.

[36] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv:2109.00859*, 2021.

YOUNGMI PARK received the B.S. degree in Department of Information Technology Engineering and Department of Biological Sciences from the Sookmyung women's University, South Korea, in 2021. She is currently pursuing the M.S. degree in Department of Information Technology Engineering with the Sookmyung Women's University, South Korea. Her research interests include deep learning, machine learning, and natural language processing.

AHJEONG PARK received the B.S. degree and the M.S. degree in Department of Information Technology Engineering from the Sookmyung women's University, South Korea, in 2021 and 2023. She is currently pursuing the Ph.D. degree in Department of Information Technology Engineering with the Sookmyung Women's University, South Korea. Her research interests include artificial intelligence, machine learning, natural language processing and ensemble.

CHULYUN KIM (M'14) received the B.S. degree in computer engineering, the M.S. degree in cognitive science, and the Ph.D. degree in electrical engineering and computer science from Seoul National University, South Korea, in 1996, 1998, and 2010, respectively. He received the Microsoft Research Asia Fellowship in 2004. He is currently an Associate Professor at Sookmyung Women's University, South Korea. Before that, he was an Assistant Professor at Gachon University, the Chief Technical Officer at NTROS, Ltd., South Korea, and a Research Assistant at the National Center for Supercomputing Applications, Urbana, IL, USA. He has been working in the area of data engineering focusing on machine learning, data mining, big data analysis, databases, artificial intelligence and interdisciplinary informatics. His writings have appeared in many professional conferences and journals, including VLDB and IEEE publications. He previously served as a Chair/PC member for ICDE, PAKDD, IEEE BigComputing, and WWW conferences.