IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 5 December 2023, accepted 10 January 2024, date of publication 15 January 2024, date of current version 22 January 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3354390

RESEARCH ARTICLE

Integrating Non-Fourier and AST-Structural Relative Position Representations Into Transformer-Based Model for Source Code Summarization

HSIANG-MEI LIANG AND CHIN-YU HUANG, (Member, IEEE)
Department of Computer Science, National Tsing Hua University, Hsinchu 300044, Taiwan

Corresponding author: Chin-Yu Huang (cyhuang@cs.nthu.edu.tw)

This work was supported by the National Science and Technology Council, Taiwan, under Grant MOST 110-2221-E-007-035-MY3 and Grant MOST 111-2221-E-007-079-MY3.

ABSTRACT Source code summaries play a crucial role in helping programmers comprehend the behavior of source code functions. In recent deep-learning based approaches for Source Code Summarization, there has been a growing focus on Transformer-based models. These models use self-attention mechanisms to overcome the long-range dependency issue that previous models often encounter, making them a promising solution for the Source Code Summarization task. However, these models suffer from two shortcomings: 1) they are weak in handling the semantics of keywords, and 2) they are weak to learn the source code with complex structure. To resolve these shortcomings, our study proposes integrating Non-Fourier and AST-Structural relative position representations into Transformer-based model for Source Code Summarization, which we have named NFASRPR-TRANS. NFASRPR-TRANS employs two types of positional encoding schemes in two different Transformer encoders. The first encoder handles the semantics of the keywords of the input source code sequence by using the Gaussian Embedder to encode the non-Fourier relative position representation of the sequence. The second encoder uses Tree Positional Encoding to learn the structural information of the Abstract Syntax Trees (ASTs), which provides relative position information in the ASTs for generating the source code summaries. Finally, we compared NFASRPR-TRANS with previous models and evaluated its performance on the Java and Python datasets using five metrics, including BLEU, ROUGE-L, CIDEr, METEOR, and SPICE. NFASRPR-TRANS achieves 2%-10% improvements across all five metrics on both datasets.

INDEX TERMS Abstract syntax tree, deep learning, machine translation, natural language, program comprehension, positional encoding, source code summarization, software engineering.

LIST OF ABBREVIATIONS

AST	Abstract Syntax Tree.
BFS	Breadth-First Search.
BLEU	Bilingual Evaluation Understudy.
CIDEr	Consensus-based Image Description Evaluation.
DL	Deep Learning.
DFS	Depth-First Search.
FNN	Feed-Forward Neural Network.
GNN	Graph Neural Network.
GPU	Graphics Processing Units.
GRU	Gated Recurrent Units.
IDDFS	Iterative Deepening Depth-First Search.
IR	Information Retrieval.
LSTM	Long Short-Term Memory Network.
NLP	Natural Language Processing.
NMT	Neural Machine Translation.
MT	Machine Translation.
OOV	Out of Vocabulary.
RNN	Recurrent Neural Network.

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

ROUGE	Recall-Oriented Understudy for Gisting Evaluation.
RQ	Research Question.
SBT	Structure-based Traversal.
SDLC	Software Development Life Cycle.
seq2seq	Sequence-to-sequence.
Si-SAN	Structure-Induced Self-Attention.
SPICE	Semantic Propositional Image Caption Evaluation.
SWUM	Software Word Usage Model.
TR	Text Retrieval.

I. INTRODUCTION

Over the years, there has been a spate of interest in using software in human life. To ensure the proper functioning of the software, activities such as software testing and software maintenance have received significant attention. However, these works are often time-consuming. According to research, the maintenance stage of the Software Development Life Cycle (SDLC), accounts for an estimated 59% of the total time cost, on average [1]. Program comprehension is regarded as a major factor in code maintenance, and having a source code with good code summarization is believed to help reduce future maintenance costs [2]. Source code summaries could help programmers comprehend the behavior of source code functions, which is beneficial to the programmers' work of software maintenance, code categorization, code retrieval, and other work [3], [4], [5]. The automation of Source Code Summarization is believed to free programmers from the heavy work of manually summarizing the code and to reduce the mismatch between summaries and functions of code [6].

With good summarization of source code, programmers could quickly understand the actions of the source code. Comparatively, if the summary lacks information on the internal behavior of the source code, the programmers may struggle to comprehend the purpose of the source code and have to invest more time in tracing through the entire code to understand it.

Early works [3], [6], [7], [8], [9], [10], [11], [12], [13] treat the Source Code Summarization task as an Information Retrieval (IR) task. They used templates to extract the keywords in a source code and to generate a summary, by way of synthesizing the keywords into the corresponding natural language templates. However, the IR-based models rely on pre-defined templates; they do not consider the semantics of the keywords or the contextual information between keywords and surrounding codes. As a result, the summaries generated by the IR-based models often suffer from poor readability [14].

Recently, researchers working on Source Code Summarization have been inspired by the Neural Machine Translation (NMT) task in Natural Language Processing (NLP). The latest Deep Learning (DL) models for Source Code Summarization follow an encoder-decoder framework [15], using architecture such as Recurrent Neural Network [16], Graph Neural Network [17], or Transformer [18].

Recent DL-based approaches [14], [19], [20], [21], [24], [25] are raised more attention on Transformer-based model [18] which uses self-attention mechanism to handle the long-range dependency issue that both RNN-based [19], [23] and GNN-based [17] models struggle with. Even though the Transformer-based models achieve state-of-the-art performance compared to other DL-based models and IR-based models [24], they still have several deficiencies.

Example 1: First, most of the Transformer-based models are using sequential relative positions between code tokens. This approach may ignore the structural or semantic information of the source code. For example, Figure 1 shows that the Transformer [24] cannot effectively summarize this code snippet, as its model lacks the ability to learn the structural information. Although some other Transformer-based models, such as SiT [25], have attempted to address this issue by integrating matrices containing the structural information of AST into their models' calculations, these matrices need to be fixed in size when training. In other words, models may only capture the contextual information within a single block of input source code and ignore the cross-block information. As shown in Figure 1, the *if* statement within the nested function in this example may be overlooked by SiT.

The second limitation of previous Transformer-based models is their inability to correctly learn the definition of special keywords, especially those that are out-of-vocabulary (OOV). Although they have implemented a "Copy attention mechanism [26]" to avoid the OOV problem, and can generate a complete summary without any unknow tokens, this method still does not ensure the model could learn the contextual meaning of OOV words well. For instance, as illustrated in Figure 1, the Transformer was unable to accurately interpret the meaning of certain keywords, such as 'sync' and 'wrap'. Similarly, SiT was unable to understand the meaning of 'db'.

In this study, we demonstrate that incorporating both 1) the non-Fourier relative position representation in source code tokens and 2) the AST-structural relative position representation in ASTs of source code are essential for the Transformer-based models to understand the semantics and syntax of source code. As depicted in Figure 1, our model successfully captures the definition of keywords such as 'wrap' and 'sync' and also learns the syntactic information of the nested function, including the *if* statement in it. For a more detailed analysis of this case, please refer to section V.

Our proposed model, integrating **N**on-**F**ourier and **AST**-**S**tructural **R**elative **P**osition **R**epresentations into **Trans**former (NFASRPR-TRANS), is a Transformer-based model designed for the Source Code Summarization task. We construct it by referring to the works of [14], where we apply two Transformer encoders to learn two different types of position representations respectively. The first Transformer encoder, the Source Code Sequence (SC-Seq) Encoder, directly integrates the non-Fourier relative position representation between source code tokens into the input source code sequence. This positional encoding step helps the model better capture the contextual relationships between

Code	def sync_from_db(f):
	@functools.wraps(f)
	def wrapper(self, *args, **kwargs):
	if self._time_to_sync():
	self._cell_db_sync()
	return f(self, *args, **kwargs)
	return wrapper		

Reference	BLEU	ROUGE	use as a decorator to wrap methods that use cell information to make sure they sync the latest information from the db periodically .
Transformer [24]	0.25	0.59	use as a decorator to **sync** methods that **update** cell information **from the database** .
SiT [25]	0.68	0.79	use as a decorator to wrap methods that use cell information to make sure they **do not cause the db from the database** .
NFASRPR-TRANS	1.0	1.0	use as a decorator to wrap methods that update cell information to make sure they sync the latest information from the db periodically .

FIGURE 1. A nested function example written in Python.

specific keywords. The other Transformer encoder, the AST Relative Position (AST-RP) Encoder, is based on SiT [25], blending the AST-structural relative position representation into the input data and the computation of the self-attention mechanism. This allows NFASRPR-TRANS to more effectively learn the structural information and grammar rules inherent in the programming language.

Our study presents the experimental results on two common datasets for performance competition in different programming languages, Java [27] and Python [28]. The results demonstrate that our proposed model, NFASRPR-TRANS, surpassed the comparative models [24], [25] in terms of five evaluation metrics, achieving a better performance on both datasets. Specifically, on the Java dataset, the performance of NFASRPR-TRANS outperforms the comparative models by at least 6.21%, 6.30%, 6.94%, 10.30%, and 6.63% in terms of BLEU [29], ROUGE [30], CIDEr [31], METEOR [32], and SPICE [33] scores, respectively. While in the Python dataset, NFASRPR-TRANS outperforms comparative models by at least 2.43% in BLEU, 1.84% in ROUGE-L, 5.94% in CIDEr, 4.33% in METEOR, and 4.83% in SPICE.

To conclude, our study makes the following main contributions:

- We propose NFASRPR-TRANS which introduces two distinct types of Transformer encoders. These encoders learn the non-Fourier and AST-structural relative position representations of the source code separately, resulting in improved code semantics and syntax learning.
- We demonstrate that our proposed model improves the quality of automatically summarizing the source code, as measured by five automatic evaluation metrics, and by incorporating both the non-Fourier relative position representations and the AST-structural relative position representations.
- We confirm the effectiveness of NFASRPR-TRANS in enhancing the source code summarization quality by conducting extensive experiments on benchmarks for the Source Code Summarization task. The source code and datasets for NFASRPR-TRANS are publicly available on GitHub.[1]

[1]https://github.com/hsmeiliang/NFASRPR_TRANS

For the rest of our paper, its organization is as follows. Section II introduces the previous works of the Source Code Summarization and the related DL techniques in our work. In section III, there is an explanation of the workflow and architecture of NFASRPR-TRANS in detail. Section IV shows the experimental result of our proposed model and the comparison of its performance with previous works. In section V, we provide a test case to evidence that the improvements are reflected in the generated summary. Section VI covers the research questions and the threats to validity in our study. Finally, the conclusion of our study and future research works are discussed in section VII.

II. RELATED WORKS

A. MANUALLY-DESIGNED AND IR-BASED APPROACHES FOR SOURCE CODE SUMMARIZATION

Reviewing the past, the works for Source Code Summarization have focus on exploiting the manually-designed and information retrieval (IR) approaches, to extract features and keywords in the source code and to synthesize them to generate predictive summaries.

Haiduc et al. [3] proposed employing the Text Retrieval (TR) approach, a branch of the IR-based approach, to extract the lexical information, such as keywords, from the input source code. The extracted keywords are then combined with the structural information of the source code to acquire the semantics of the programming languages, resulting in automatically summarizing the source code. The purpose of Haiduc's approach was to gain insight into the source code behavior by utilizing the identified keywords.

Sridhara et al. [6] introduced the Software Word Usage Model (SWUM), which automates the extraction of the keywords with their linguistic and structural relationships in source code. This allows for the identification of actions, themes, and secondary arguments related to a specific method. The SWUM approach combines the linguistic information of the identification with the control flow graph, along with data and control dependencies to create a detailed logic summary of the source code. After that, they proposed an extension model of the SWUM [7], which focused on capturing the high-level actions of source code methods

from a block-wise relationship perspective, in contrast to the statement-wise relationship of their earlier model. With this modification, their approach can more accurately capture the *if-else* and *loop* relationships within the source code, resulting in a more precise Source Code Summarization.

Lastly, Zhang et al. [34] introduced Rencos, a novel method that blends the IR-based and DL-based approaches for Source Code Summarization. To accomplish this, Rencos utilizes IR-based techniques to retrieve two source codes that are most syntactically and semantically similar to the input code. The DL-based approach then encodes the input and the retrieved codes, and a summary of the source code is generated by merging the input and the two retrieved codes during the decoding process. The results of the experiments indicate that Rencos leverages the complementary strengths of IR-based and DL-based approaches, improving the performance of the Source Code Summarization task.

B. DL-BASED APPROACHES FOR SOURCE CODE SUMMARIZATION

Ahmad et al. [24] put forward a Transformer-based model for Source Code Summarization, which employs relative position representations and copy attention. Unlike models that rely on structural information in ASTs, they pointed out that incorporating the sequential mutual interactions of source code tokens in the self-attention mechanism can better represent the semantics of source code. Their results show that their Transformer-based model outperforms previous models significantly in the Source Code Summarization task.

Wu et al. [25] proposed SiT, which is a Transformer-based model with a novel structure-induced self-attention (Si-SAN) mechanism to generate summaries for the source codes. SiT uses the source code sequence and the multi-view structural information of source code as input. The multi-view structural information is captured by adjacency matrices of three code graphs, including abstract syntax tree, control flow, and data dependency. Then, SiT adopts their proposed Si-SAN mechanism, which is designed to drop out irrelevant information in the self-attention matrix, based on the multi-view structural information. Their experimental results revealed that SiT, through the utilization of the Si-SAN mechanism, attains a quicker convergence rate compared to Transformer [24].

Gong et al. [14] proposed SCRIPT, a model for Source Code Summarization that enhances the learning of structural information in the source code by employing two types of transformer encoders. The first encoder is a standard Transformer encoder, which directly integrates the structural information of the code into the code sequence, while the second encoder is derived from SiT, which calculates the dependencies between the tokens in the AST. Different from SiT, SCRIPT calculate the shortest path length of the AST in order to get the structural information of the source code from the AST. They point out that the AST's shortest path length reflects the relationship of two tokens, as there exists a positive correlation between the proximity of two tokens in the AST of source code and the strength of their semantic relationship.

These studies demonstrate the potential of integrating the structural information of the ASTs into a Transformer-based model to produce high-quality summaries for the input source code. However, as the case shows in Example 1 in section I, the existing Transformer-based models suffer from two shortcomings: 1) they are weak in handling the semantics of special words, and 2) the models are weak to learn the source code with complex structure.

To resolve these shortcomings, in this study, we propose a novel approach, NFASRPR-TRANS, which employed two different methods in the positional encoding step to learn the information which relates to the two shortcomings separately, and which improves the ability of our model to comprehend the source code and which also generates more accurate summaries.

C. TRANSFORMER

Due to the ability for better long-term dependency modeling, the Transformer-based model has gained more popularity than RNNs in the NLP task. The Transformer [18], also known as vanilla Transformer, uses self-attention and parallelization for an efficient information exchange between inputs and outputs, making them more suitable for handling complex NLP tasks.

Similar to RNNs [16], Transformer is a sequence-to-sequence model with encoder-decoder architecture. The encoder of the Transformer is composed of multi-head attention with a self-attention mechanism, feed-forward neural network (FNN), and Add&Norm. The decoder of the Transformer has similar structures to the encoder, but the masked multi-head attention is added in each decoder layer.

In Transformer, both the input and target output are sequences of tokens that are embedded as a sequence of vectors. To be specific, the set of input and target output vectors is $\left\{x^i, y^i\right\}, 1 \leq i \leq N$, where N is the number of training data. An input vector $x^i = \left(x_1^i, x_2^i, \ldots, x_m^i\right)$, where $x_m^i \in R^{d_{model}}$, m is the number of code tokens of the i-th inputs and d_{model} is the dimension of the model. A target output vector $y^i = \left(y_1^i, y_2^i, \ldots, y_\omega^i\right)$, where $y_\omega^i \in R^{d_{model}}$ and ω is the number of words of the i-th target output sequence. Then, the positional encoding layer adds the position information of the tokens into the input vectors and output vectors. The input vector will be the input to the encoder, and the output vector from the encoder will be the input to the decoder.

In the self-attention mechanism, an input vector $x = (x_1 \ldots x_m)$ is transformed to an output vector $o = (o_1 \ldots o_c)$, where c is the length of the output vector. The calculation of the self-attention mechanism is as follows:

$$o_i = \sum_{j=1}^{m} \alpha_{ij} \left(x_j W^V\right), \tag{1}$$

$$e_{ij} = \frac{x_i W^Q \left(x_j W^K\right)^T}{\sqrt{d_k}}, \tag{2}$$

$$\alpha_{ij} = \frac{\exp\left(e_{ij}\right)}{\sum_{k=1}^{m} \exp\left(e_{ik}\right)}, \tag{3}$$

where W^Q and $W^K \in R^{d_{model} \times d_K}$, and $W^V \in R^{d_{model} \times d_V}$ are the parameter matrices which are attention weights that are unique.

Since the input vector x can be regarded as a large matrix, we express the self-attention calculation in the following equation, and it can be calculated in parallel.

$$Attention\left(Q, K, V\right) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \tag{4}$$

where $Q = W^Q x$, $K = W^K x$, and $V = W^V x$.

In multi-head attention, there are multiple attention heads in each layer. An attention head is a set of (W^Q, W^K, W^V) parameter matrices. The calculation of the multi-head attention is

$$\begin{aligned} &MultiheadAttention\left(Q, K, V\right) \\ &\quad = Concat\left(head_1, \ldots, head_h\right) W^O, \end{aligned} \tag{5}$$

$$\begin{aligned} &head_i \\ &\quad = Attention\left(QW_i^Q, KW_i^K, VW_i^V\right), \end{aligned} \tag{6}$$

where (W_i^Q, W_i^K, W_i^V) are i-th attention head's parameter matrices, h is the number of heads, and W^O is a parameter matrix owned by the whole multi-head attention heads.

Masked multi-head attention differs from regular multi-head attention in that the masked multi-head attention only looks to earlier positions of the output vector. That is, (1) is changed to

$$o_i = \sum_{j=1}^{i} \alpha_{ij}\left(x_j W^V\right). \tag{7}$$

The Add&Norm layer calculates the residual connection and layer normalization for a vector. That is

$$LayerNorm\left(x' + o'\right), \tag{8}$$

where x' is the input vector of the previous block of Add&Norm, and o' is the output vector of the previous block of Add&Norm.

The FNN is composed of a fully connected network and a ReLU activation function, processing the input vector. That is

$$FNN\left(x'\right) = ReLU\left(W^F x' + b^F\right), \tag{9}$$

where W^F and b^F are the fully connected network's weight matrix and bias vector, respectively.

In addition, in section II-B we have introduced that Wu et al. [25] proposed a variant self-attention mechanism, namely Si-SAN, which drops out irrelevant information in the self-attention matrix based on the multi-view structural information. The calculation of Si-SAN can be expressed with the following equation:

$$SiSAN\left(A, Q, K, V\right) = softmax\left(\frac{A \cdot QK^T}{\sqrt{d_k}}\right)V, \tag{10}$$

$$A = \alpha A_{ast} + \beta A_{fl} + \gamma A_{dp}, \tag{11}$$

where A is the multi-view structural information, which is an integrating matrix of three code graphs' adjacency matrices, including the A_{ast} abstract syntax tree, the A_{fl} control flow, and the A_{dp} data dependency graphs. And α, β, γ are the corresponding weight for each adjacency matrix.

D. POSITIONAL ENCODING SCHEMES

Recently, there has been a spate of interest in positional encoding for deep learning, particularly for Transformer-based models [35]. Viswani et al. [18] proposed the Transformer, which utilizes the self-attention mechanism to capture the long-dependencies relationship in input data. However, since the self-attention mechanism is not sensitive to the position information of the input data, it is necessary to use positional encoding to incorporate the position information into the model. For the source code which features unique structural properties and follows strict programming language grammars, there are multiple ways to encode it.

Viswani et al. [18] first propose to incorporate position information by injecting it into the Transformer only at the first layer of the encoder and decoder using a pre-defined sinusoidal function. Subsequently, Dehghani et al. [36] observe that the Transformer-based model could achieve a better performance by injecting position information into each layer of the model.

Ahmad et al. [24] proposed the use of relative positional encoding [37] within the Transformer-based model for Source Code Summarization. They demonstrated that the semantic information of the source code is not solely dependent on the absolute position of the code tokens, but rather on the relative position of any two tokens. For instance, the semantic meaning of '1+2' is the same as '2+1'.

Zheng et al. [35], [38] proposed a positional encoding scheme that leverages pre-defined non-Fourier functions, such as Gaussian function. The performance of the non-Fourier functions of positional encoding is heavily influenced by a trade-off between the stable rank of the embedding matrix and the distance preservation of the embedding coordinates. Specifically, the stable rank has a significant impact on the model's memory of the training data, while the distance preservation affects the generalization ability of the model.

Shiv et al. [39] proposed a tree positional encoding scheme for extending the Transformer model to tree-structured data, enabling the mapping of data between tree and sequence structures. Shiv et al. utilized a stack-like position representation that encodes the path information from the root node for each node, and the scheme takes two hyper-parameters: 1) the width of the tree, and 2) the depth of the tree.

Wu et al. [25], and Gong et al. [14] proposed using adjacency matrices for encoding tree-structured data in the Transformer-based model, indicating that it is important to choose a suitable positional encoding scheme for learning the specific information from the input data, and that utilizing the features in the ASTs of the source code could help the Transformer-based models to comprehend some structural information of the source code.

These studies indicate that it is important to choose a suitable positional encoding scheme for learning the specific information from the input data, and that utilizing the features in the ASTs of the source code could help the Transformer-based models to comprehend some structural information of the source code.

Inspired by these observations, to solve the shortage of the Transformer-based models which we have illustrated in Example 1 in section I, the ability of the model to comprehend the semantics of the keywords and structural information of the input source code is important. Our model uses two different methods to learn them from the input source code. Specifically, NFASRPR-TRANS applies: 1) the Gaussian Embedder to encode the non-Fourier relative position representations between source code tokens to learn the contextual relationships between specific keywords, and 2) the Tree Positional Encoding to encode the AST-structural relative position representations to make the model comprehend the complex structure of the input source code.

The details of the Gaussian Embedder and the Tree Positional Encoding are in section III-A.

III. NFASRPR-TRANS: INTEGRATING NON-FOURIER AND AST-STRUCTURAL RELATIVE POSITION REPRESENTATIONS INTO TRANSFORMER

A. WORKFLOW OF NFASRPR-TRANS

As the illustration in Figure 2 shows, NFASRPR-TRANS involves several key steps. First, we pre-process the input source codes and summaries, and also generate the ASTs of input source code for the Tree Positional Encoding step. Next, NFASRPR-TRANS employs two types of positional encoding schemes, one representing the non-Fourier relative position by the Gaussian Embedder for the source code sequences, and the other conducting the AST-structural relative position representations of the ASTs using Tree Positional Encoding. These positional encoding schemes facilitate the encoder's ability to understand the semantics and structural information of the source code. Finally, the code sequence and the two types of relative position representations are fed to the model encoder, and then the predicted summary is generated by the decoder.

1) PRE-PROCESSING OF SOURCE CODE SEQUENCE

To generate efficient input features for the source code text, we implemented the pre-processing step proposed by Ahmad et al. [24], since some user-defined code tokens are compound words in *CamelCase* and *snake_case* forms, which are rare words in the training dataset and cause the OOV problem [26]. To address this, our model splits up the *CamelCase* and *snake_case* forms source code tokens before conducting Word Embedding on the input source code. This step enables our model to capture the essential information from the source code text, and generates high-quality summaries that are both accurate and informative.

Example 2: The *CamelCase* and *snake_case* forms are two types of naming conventions in computer programming. In the *CamelCase* form, words are joined together without spaces or punctuation, and the first letter of each word is capitalized to make it easier for humans to identify each individual word. For example, the code tokens "stepValue" is *CamelCase* form code tokens. In the Pre-processing step, these tokens are split by spaces and become "step Value". On the other hand, *snake_case* joins words with an underscore and uses all lowercase letters. For example, code tokens such as "resource_patch" is written in *snake_case*. After pre-processing, these tokens become "resource patch".

The next step is Word Embedding, which is a common technique in the NLP task [40] and has been widely used in DL-based models for Source Code Summarization [24], [25]. It is a pre-train neural network to transfer the input tokens into vectors, and these vectors could represent the semantics similarity of each of the two tokens by their distance in the vector space. If any two tokens have similar meanings or characteristics, their distance in the vector space is close. Otherwise, when two tokens have different meanings or are completely irrelevant, they are far away from each other in the vector space.

2) ASTS OF THE SOURCE CODE

We employed several open-source tools to generate the ASTs of the source code. The javalang [41] is a tool that is used to parse the source code in the Java programming language, which extracts the ASTs in the dictionary form, while the *attokens* module provided by [25] is used to obtain the ASTs of the source code in the Python programming language. By leveraging these tools, the ASTs of the source codes are generated and passed to the Tree Positional Encoding.

3) GAUSSIAN EMBEDDER

To generate the non-Fourier relative position representations of the source code sequence, we use the Gaussian signal as the embedding function, as in [38]. The computation of the Gaussian Embedder is as follows [45]:

$$\phi(t, x_i) = \exp\left(-\frac{|t - x_i|^2}{2\sigma^2}\right), \tag{12}$$

$$\sigma = \frac{1}{4N\sqrt{k \times \ln 10}}, \tag{13}$$

where x_i is the i-th token of the input source code sequence x, k is the empirically chosen threshold, N is the number of samples, and σ is the standard deviation chosen log-linearly from $[10^{-4}, 10^{-1}]$.

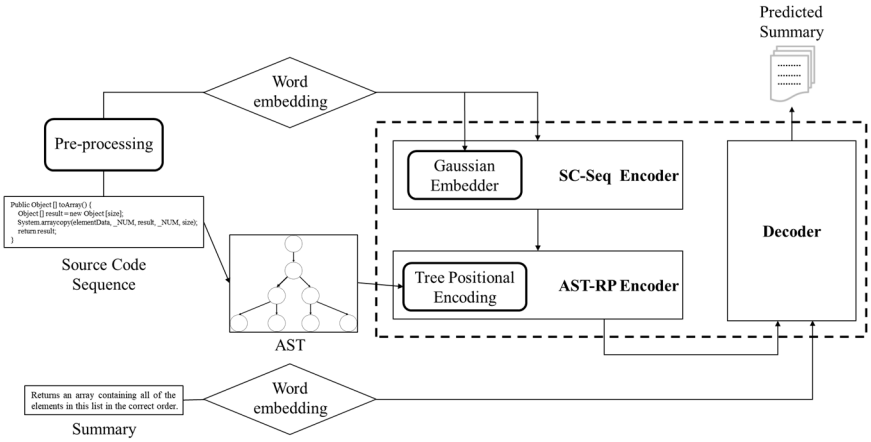


FIGURE 2. The source code summarization workflow of NFASRPR-TRANS.

The embedded distance between the two tokens x_1 and x_2 is as follows [38]:

$$D(x_1, x_2) = \exp\left(-\frac{|x_1 - x_2|^2}{4\sigma^2}\right). \tag{14}$$

As pointed out by [38], the value of σ is crucial in determining the performance of the Gaussian Embedder, which is related to the model's memorialization and generalization. A large σ leads to overly smooth generalization and poor memorization because $D(x_1, x_2)$ becomes larger. Conversely, a small σ leads to better memorization but poor generalization because the $D(x_1, x_2)$ becomes smaller.

4) TREE POSITIONAL ENCODING

In the Tree Positional Encoding step, to provide the model with the structural information of the input source code, we refer to the work of [39] to obtain relative position representation from the ASTs. Specifically, our model utilizes the Pre-order traversal algorithm to traverse an AST, and then reversing the Pre-order traversal sequence and encoding the sequence to the AST positional encoding vectors by the scheme of [39]. Finally, there is an embedding network in the NFASRPR-TRANS to generate the AST-structural relative position representations.

The encoding step of [39] is demonstrated in Figure 3. In each AST positional encoding vector, its dimension is $(g \cdot d)$, where g is the degree of the ASTs, and d is the maximum tree depth that constraints the size of the vector. The root position R is initiated as the zero vector $\vec{0}$, and the AST positional encoding vectors of other nodes are represented as follows from [39]:

$$TP(p) = D_{b_L} D_{b_{L-1}} \ldots D_{b_1} \vec{0}, \tag{15}$$

$$0 \leq b_l < g, \quad \forall l \in \{1 \ldots L\}, \tag{16}$$

where b_l is the step choice at the l-th level of the AST and L is the level of the node that p locates, and the set $\langle b_1 \ldots b_L \rangle$ can be represented as a path from root node to the node p, and D_{b_l} is an operation that pushes b_l in one-hot g bit vector to p's relative position vector.

Notably, our proposed model differs from that of [39] which used the Deep-First-Search (DFS) algorithm for traversing ASTs, in that our model used a Pre-order Reverse traversal algorithm. Specifically, after getting a Pre-order traversal sequence of an AST, we first reverse the sequence, then encode the reversed sequence to the AST positional encoding vectors by the scheme of [39].

Example 3: Figure 4 is an example of the difference Tree-traversal Strategies for the AST. In the tree traversal sequence, we can consider the *operators* as 'Verb' and the *parameters* as 'Subject'. Following this, in the Pre-order Reverse traversal algorithm, the sequence can be interpreted as: 'The *parameters* do the action of the *operator*.' Which is similar to the 'Subject + Verb' expression in the natural language.

Our model utilizes two features of the Pre-order Reverse traversal algorithm. First, by reversing the order of the Pre-order traversal sequence of the AST, our model starts reading from the parameters and learns how to describe the situation of the parameters and operators. Second, we do not modify the tree node index that the Pre-order traversal algorithm generates; that is, we keep the feature of the Pre-order traversal algorithm when encoding the tree node vector. This implicitly makes our model imitate the way compilers process syntax trees. We believe that this approach is more in line with the natural language sentence structure of the 'Subject + Verb' expression, and it also retains the programming information of the source code. These features help the model generate more human-readable and meaningful sentences.

By leveraging the Tree Positional Encoding and an embedding network, we are able to extract those AST-structural relative position representations, that could be used to support the model to summarize the source code with complex structure.

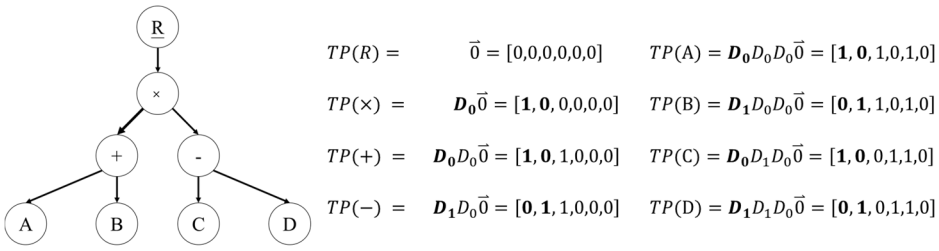


FIGURE 3. An example of AST positional encoding vectors.

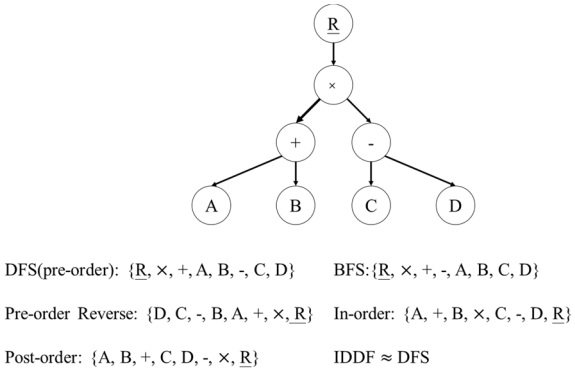


FIGURE 4. An example of an AST in six different types of tree traversal strategies.

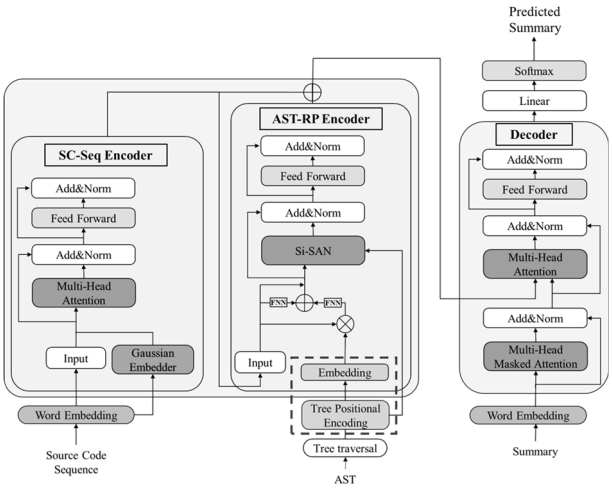


FIGURE 5. The architecture of NFASRPR-TRANS.

B. ARCHITECTURE OF NFASRPR-TRANS

Figure 5 presents the overall architecture of the model we proposed. It is made up of two different types of encoders and one decoder.

The Source Code Sequence Encoder (SC-Seq Encoder) is in charge of paying attention to keywords, and it is composed of the normal Transformer encoder with the Gaussian Embedder, which treats the source code sequence as input. The AST Relative Position Encoder (AST-RP Encoder) is composed of a normal Transformer encoder with Si-SAN and Tree Positional Encoding. It takes the SC-Seq Encoder's output and the AST information as its input to learn the structural information of the input source code. Finally, the Decoder is composed of a normal Transformer decoder, which is utilized to generate a summary.

1) SOURCE CODE SEQUENCE ENCODER

Recall that there are various positional encoding schemes that can be used in the Transformer-based model. Inspired by [36] and [38], we can leverage the memorization and generalization ability of the non-Fourier positional encoding functions and inject the position information in each layer. By doing so, the model can gain knowledge about keywords and parameters in the input source code.

Therefore, we have the Source Code Sequence Encoder (SC-Seq Encoder) which is shown in the left block of the encoder in Figure 5. The input data for the SC-Seq Encoder is the source code sequences $X = \{x^1, \ldots, x^N\}$, which have undergone the Pre-processing step mentioned in section III-A, and an input source code sequence is represented as x, $x \in X$. Then we use the Gaussian Embedder formatted in (12)-(14) to calculate the non-Fourier relative position representation vector G of X and inject it into input data. Finally, the output of n-th layer of SC-Seq Encoder is denoted as H_1^n, defined as:

$$H_1^n = EN_1^n(H^{n-1} + G) \tag{17}$$

where H^{n-1} represents the output of the previous layer. The $EN_1^n()$ represents the computation of the vanilla Transformer encoder and includes the modules we introduced in section II-C: namely the multi-head self-attention, the FFN, and the Add&Norm layers.

2) AST RELATIVE POSITION ENCODER

Although sequential relationships of source tokens are learned in the SC-Seq Encoder, it disregards the structural information or grammatical rules of the programming language. To address this, we drew inspiration from the previous works [14], [25] that parse ASTs to get the structural information of the source code.

Thus, we construct an AST Relative Position Encoder (AST-RP Encoder) which is shown in the right block of the encoder in Figure 5. By using the Tree Positional Encoding, which had been introduced in section III-A, the model is able

to gain the attributes such as the source code structure and grammatical rules of the programming language.

In the AST-RP Encoder, the input data – the traversal sequence of AST – first passes through the Tree Positional Encoding and an embedding network to compute the AST-structural relative position representation matrix $\overline{M}$. Assuming the original input of the n-th layer of the AST-RP Encoder is H_1^n, which is the output of the previous layer (SC-Seq Encoder), the AST-structural relative position representation matrix $\overline{M}$ is blended into H_1^n. After the above computation, we obtain the input matrix for the AST-RP Encoder, expressed as $\overline{H}^n$ in (19).

$$\overline{M} = Tree\_Positional\_Encoding(AST), \tag{18}$$

$$\overline{H}^n = \sigma\left(FC_n^1\left(H_1^n\right) + FC_n^2\left(\overline{M}H_1^n\right)\right), \tag{19}$$

where AST is the traversal sequence of AST, $\sigma(\cdot)$ is the sigmoid activation function, and FC_n^1 and FC_n^2 are fully-connected layers in the n-th layer of the AST-RP Encoder.

Finally, for the output of the n-th layer of the AST-RP Encoder, H_2^n, is in (20) and the overall output of the n-th layer of both encoders of NFASRPR-TRANS is H^n in (21).

$$H_2^n = EN_2^n\left(Aggr\left(H_1^n, \overline{H}^n\right)\right), \tag{20}$$

$$H^n = Aggr\left(H_1^n, H_2^n\right), \tag{21}$$

where the $Aggr(\cdot,\cdot)$ is a simple element-wise addition function used to combine the two encoders' outputs H_1^n and H_2^n. The $EN_2^n()$ represents the computation of a Transformer encoder that includes those modules we introduced in section II-C that are Si-SAN, FFN and Add&Norm layers.

3) DECODER

We do not have any modifications on the Decoder, that is we employed the vanilla Transformer decoder in NFASRPR-TRANS. So, the $DE_n()$ consists of modules such as masked multi-head self-attention, multi-head self-attention, FFN, and Add&Norm. To calculate the probability of each candidate word, the Softmax function is applied.

The decoding process is defined as

$$canditates = DE_n\left(H^n\right), \tag{22}$$

$$o_i = \max\left(Softmax\left(canditates\right)\right), \tag{23}$$

where o_i is the i-th word in the generated summary. To choose the final word, NFASRPR-TRANS employs the Greedy strategy during testing, which selects the candidate word with the highest probability.

IV. EXPERIMENTAL RESULTS AND DISCUSSION

A. EXPERIMENTAL SETUP

1) DATASETS

To ensure the reproducibility and generalizability of our experimental results, we have utilized an existing Java dataset provided by Hu et al. [27] and Python dataset provided by Wan et al. [28] to evaluate our proposed model, both datasets are widely used in evaluating the models for the Source Code Summarization task [14], [24], [25].

TABLE 1. The statistics of the Java dataset from [27].

Length of Summary (words)						
Avg.	Max	<20	<40	<50	<60	<100
17.74	503	75.6%	92.5%	95.6%	97.1%	99.2%
Length of Code (tokens)						
Avg.	Max	<100	<200	<300	<400	<500
119.26	5956	61.7%	84.9%	92.9%	96.0%	97.6%

TABLE 2. The statistics of the Python dataset from [28].

Length of Summary (words)						
Avg.	Max	<10	<20	<30	<40	<50
9.48	50	68.0%	96.8%	99.3%	99.9%	100%
Length of Code (tokens)						
Avg.	Max	<50	<100	<150	<200	<500
47.85	5398	62.7%	91.3%	99.5%	99.9%	100%

The Java dataset [27] comprises of 87,136 code/summary pairs. However, due to the computational limitation, we have only used 57,428 pairs, which have been divided into 40,000 pairs for training, 8,714 pairs for testing, and 8,714 pairs for validation.

For the Python dataset [28], which consists of 92,545 code/summary pairs, the original source codes of it are written in Python2. This makes some of the source codes to be non-executable in Python3, so we are unable to generate their ASTs. To account for the computational limitation and filter out the non-executable Python source codes, we used only 66,047 pairs from the Python dataset and divided them into 40,000 pairs for training, 18,047 pairs for testing, and 8,000 pairs for validation.

The Tables 1 and 2 display the statistics related to the length of code and summary for the Java and Python datasets, respectively.

2) PRE-PROCESSING

Apart from the data pre-processing steps by Ahmad et al. [24], we also pre-process the ASTs of the source code and their corresponding relative position representations (as explained in section III-A). Specifically, our model uses the Pre-order traversal algorithm to walking the ASTs and reverse the ASTs' sequences to generate the reversed Pre-order sequences of the ASTs. Next, our model follows the steps of Shiv et al. [39] to encode the root path of each node of the ASTs' reversed Pre-order sequences to generate the relative position matrices of ASTs, which serve as the input data of the AST-RP Encoder.

3) TRAINING DETAILS AND HYPER-PARAMETERS

We used Python 3 and Pytorch to construct our model, and train our model on a system with Ubuntu 18.04 LTS installed, equipped with a 6-core 3.1 GHz i5-8600 CPU, an NVIDIA GeForce GTX 1080 8G GPU, and 32GB RAM.

TABLE 3. The performance of different models on Java dataset.

Model	BLEU	ROUGE-L	CIDEr	METEOR	SPICE
Transformer [24]	34.06	45.19	2.59	20.78	0.335
SiT [25]	35.74	45.86	2.94	20.77	0.362
NFASRPR-TRANS (Full model)	**37.96** (↑6.21%)	**48.75** (↑6.30%)	**3.15** (↑6.94%)	**22.91** (↑10.30%)	**0.386** (↑6.63%)
NFASRPR-TRANS w/o Gaussian Embedder	37.49	48.23	3.13	22.66	0.381
NFASRPR-TRANS w/o Tree Positional Encoding	37.25	48.27	3.13	22.93	0.385

TABLE 4. The performance of different models on Python dataset.

Model	BLEU	ROUGE-L	CIDEr	METEOR	SPICE
Transformer [24]	29.28	43.76	1.89	18.16	0.264
SiT [25]	29.65	44.12	1.92	18.26	0.269
NFASRPR-TRANS (Full model)	**30.37** (↑2.43%)	**44.93** (↑1.84%)	**2.03** (↑5.94%)	**19.05** (↑4.33%)	**0.282** (↑4.83%)
NFASRPR-TRANS w/o Gaussian Embedder	30.37	44.85	2.01	18.85	0.278
NFASRPR-TRANS w/o Tree Positional Encoding	30.29	44.66	2.01	18.89	0.277

To set most of the hyper-parameters, we followed the setting of Ahmad et al. [24].

Specifically, in both datasets, the vocabulary sizes are 50000 for the encoder and 30000 for the decoder. The Adam optimizer [42] is used to train NFASRPR-TRANS on the Java dataset, while the AdamW optimizer [43] is used to train the model on the Python dataset. Both optimizers have a learning rate of 10^{-4}, a training batch size of 16, and a learning rate decay of 0.99. The weight decay is set to 0 for the Adam optimizer in the Java dataset and 0.01-0.05 for the AdamW optimizer in the Python dataset.

The number of the NFASRPR-TRANS encoder/decoder layers is 6 layers, with 3 layers each for the SC-Seq Encoder and the AST-RP Encoder and 6 layers for the Decoder. The dropout rate is 0.2 in the encoder/decoder layer. The Gaussian Embedder is constructed of 4 layers with the σ set to 0.001 in both datasets, and its dropout rate sets to 0.2 in the Java dataset and 0.7 in the Python dataset. The dropout rate of the Tree Positional Encoding layers is 0.2 in the Java dataset and 0.7 in the Python dataset.

Finally, the maximum training epochs are set to 200 epochs when training on both datasets. In addition, we adopt an early-stopping strategy that ends the training process when the performance of the validation state fails to improve for 20 consecutive epochs to avoid overfitting.

4) EVALUATION METRICS

To assess the quality of the generated summaries, we conduct experiments using some commonly employed metrics in NMT evaluation: BLEU [29], ROUGE [30], CIDEr [31], METEOR [32], and SPICE [33].

B. EXPERIMENTAL RESULTS

In this section, we provide the experimental results to support the improvement of our proposed model over the baselines, Transformer [24] and SiT [25].

To demonstrate the superiority of NFASRPR-TRANS compared to baselines, we first use the evaluation metrics to measure the similarity between the summaries generated by the models and human-written summaries. Table 3 shows the performances of our proposed model and baselines on the test data of the Java dataset, and Table 4 shows the performances on the test data of the Python dataset. In Tables 3 and 4, the best score in each evaluation metric is highlighted in bold, and the percentage improvement of NFASRPR-TRANS over the best baseline is provided in brackets.

In comparison to all baselines, NFASRPR-TRANS exhibits the best performance across all five metrics. Table 3 shows that NFASRPR-TRANS achieves improvements of at least 6.21% in BLEU, 6.30% in ROUGE-L, 6.94% in CIDEr, 10.30% in METEOR, and 6.63% in SPICE on the Java dataset. Meanwhile, in Table 4, NFASRPR-TRANS shows improvements of at least 2.43% in BLEU, 1.84% in ROUGE-L, 5.94% in CIDEr, 4.33% in METEOR, and 4.83% in SPICE on the Python dataset. The results suggest that the non-Fourier and the AST-structural relative position representations significantly improve the performance of NFASRPR-TRANS for Source Code Summarization.

In this study, we utilize two types of positional encoding schemes, namely Gaussian Embedder and Tree Positional Encoding, in NFASRPR-TRANS, that encode two types of relative position representations and that allow the model to understand the semantics and syntax of the source code.

TABLE 5. The performance of applying different tree traversal algorithms of NFASRPR-TRANS on the Java and Python datasets.

Dataset	Java					Python				
Model	BLEU	ROUGE-L	CIDEr	METEOR	SPICE	BLEU	ROUGE-L	CIDEr	METEOR	SPICE
NFASRPR-TRANS (w/ Pre-order Reverse)	**37.96**	**48.75**	**3.15**	**22.91**	**0.386**	**30.37**	**44.93**	**2.03**	**19.05**	**0.282**
NFASRPR-TRANS w/ Pre-order	37.53	48.28	3.11	22.69	0.383	30.31	44.72	2.07	18.83	0.278
NFASRPR-TRANS w/ In-order	8.37	14.12	0.07	3.95	0.027	30.31	44.81	2.02	18.96	0.279
NFASRPR-TRANS w/ Post-order	37.61	48.15	3.10	22.67	0.381	30.32	44.73	2.02	18.92	0.278
NFASRPR-TRANS w/ BFS	37.02	47.59	3.06	22.46	0.378	30.32	44.68	2.01	18.87	0.279
NFASRPR-TRANS w/ IDDFS [44] (depth=16, width=8)	37.69	48.21	3.12	22.82	0.385	28.22	43.62	1.74	17.45	0.251

To assess the contribution of these modules, we next conduct an ablation experiment in which we add or remove these two modules to examine their impact. Specifically, by removing the Gaussian Embedder, the SC-Seq Encoder becomes a vanilla Transformer and the AST-RP Encoder remains unchanged. Similarly, by removing the Tree Positional Encoding module, the AST-RP Encoder becomes a vanilla Transformer with Si-SAN and the SC-Seq Encoder does not change.

The lower half of Tables 3 and 4 show the ablation experimental results on the Java dataset and the Python dataset, respectively. According to the results, both variant models achieve higher performances compared to the baselines in both datasets, indicating that both types of relative position representations are beneficial for NFASRPR-TRANS models to learn more of the semantics or syntax of the source code and to support the model to generate higher-quality summaries. Conversely, when comparing with the full model, removing any type of relative position representation has resulted in a drop in performance. This drop demonstrates that both of these modules play a major role in NFASRPR-TRANS in learning more semantics or syntax of the source code.

In addition, Figures 6 and 7 depict the change in BLEU and ROUGE-L scores on the validation data over each of the 10 training epochs, and it is observed that the NFASRPR-TRANS exhibits a much faster convergence rate than baselines on both datasets. For example, for the convergence rate on the Java dataset shown in Figure 6, our model surpasses all baselines from the 30th to 80th epoch and achieves the best performance of baselines in about 120 epochs, while the baselines need more than 190 epochs to achieve it. Moreover, as showed in Figure 7, our model trained on the Python dataset surpasses all baselines in the first 10 epochs and achieves the best performance of baselines in about 90 epochs, while the baselines need more than 150 epochs to achieve it. The results imply that the high convergence rate of NFASRPR-TRANS highlights the necessity of both the non-Fourier and AST-structural relative position representations.

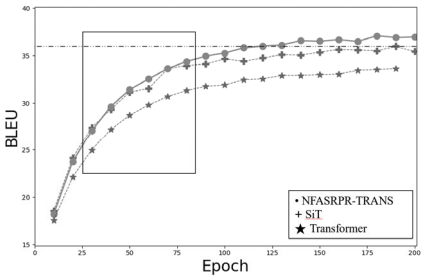
Next, we focus on the Tree Positional Encoding module of our proposed model and examine the impact of different tree traversal algorithms on the performance of the model by conducting additional experiments. Given that there are several tree traversal algorithms to choose from (e.g., Pre-order, In-order, Post-order, BFS, IDDFS [44]), we further conduct experiments employing different tree traversal algorithms in the Tree Positional Encoding module of NFASRPR-TRANS. Table 5 shows how applying different tree traversal algorithms affected the performance of the model on the Java dataset and the Python dataset, respectively. As shown in Table 5, the best scores in each evaluation metric are highlighted in bold.

In comparison to all tree traversal algorithms, Pre-order Reverse traversal algorithm, which has been adopted in NFASRPR-TRANS, achieved the best performance in both datasets, indicating that the Pre-order Reverse traversal algorithm may be more valuable for Source Code Summarization, as we supposed in section III-A. In addition, as shown from Tables 3 to 5, most other tree traversal algorithms outperformed the baselines for Source Code Summarization, which reveals that Tree Positional Encoding is a suitable choice to add position information to Transformer-based models for Source Code Summarization.

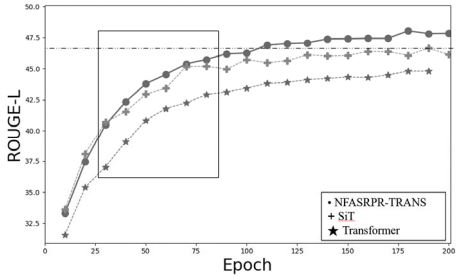
V. QUALITATIVE ANALYSIS AND VISUALIZATION

This section includes a test case that is presented to determine that NFASRPR-TRANS is capable of: 1) using the non-Fourier relative position representations encoded by the Gaussian Embedder to learn the semantics of special words, and 2) using the AST-structural relative position representations of ASTs encoded by the Tree Positional Encoding, in order to capture the complex structure of the source code.

Figure 8 displays a test case with a nested function example from the Python dataset. The lower half of the figure shows

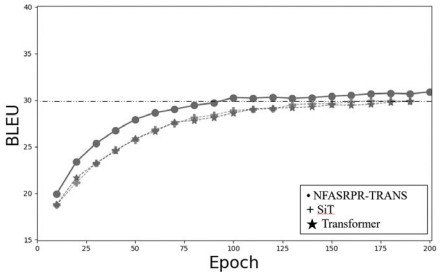


(a) BLEU scores with training epochs

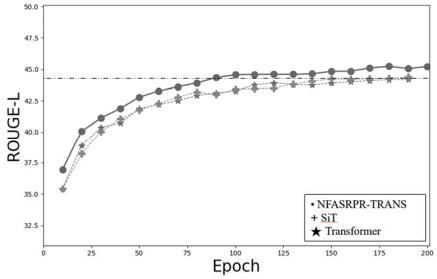


(b) ROUGE-L scores with training epochs

FIGURE 6. Convergence of different models on the Java dataset.



(a) BLEU scores with training epochs



(b) ROUGE-L scores with training epochs

FIGURE 7. Convergence of different models on the Python dataset.

<table>
<tr><td>Code</td><td colspan="3">def sync_from_db(f): ①
@functools.wraps(f) ④
def wrapper(self, *args, **kwargs):
if self._time_to_sync(): ②
self._cell_db_sync()
return f(self, *args, **kwargs) ⑤
return wrapper ③</td></tr>
<tr><td>Reference</td><td>BLEU</td><td>ROUGE</td><td>use as a decorator to wrap methods that use cell information to make sure they sync the latest information from the db periodically .</td></tr>
<tr><td>Transformer [24]</td><td>0.25</td><td>0.59</td><td>use as a decorator to sync methods that update cell information from the database .</td></tr>
<tr><td>SiT [25]</td><td>0.68</td><td>0.79</td><td>use as a decorator to wrap methods that use cell information to make sure they do not cause the db from the database .</td></tr>
<tr><td>NFASRPR-TRANS</td><td>1.0</td><td>1.0</td><td>use as a decorator to wrap methods that use cell information to make sure they sync the latest information from the db periodically .</td></tr>
</table>

FIGURE 8. A nested function example written in Python with its five attention groups.

the comparison between generated summaries, revealing that Transformer [24] does not capture the keyword 'wrap' and the behavior of the nested function, resulting in a summary that does not match the intended functionality. Similarly, SiT [25] fails to learn the nested function's behavior of the source code, and therefore, generates an erroneous summary. In contrast, NFASRPR-TRANS successfully captures both the keyword 'wrap' and learns the actions of the nested function, resulting in an accurate and complete summary.

In order to demonstrate the ability of NFASRPR-TRANS to learn the definition of special keywords and the behavior of the nested function, we present the attention heatmaps from the 3rd layer of both the SC-Seq Encoder and the AST-RP Encoder in Figure 9 and Figure 10, respectively. The source code is first dissected into 5 attention groups, which are sequentially numbered from 1 to 5, as seen in Figure 8.

The group numbers 1 through 3 in Figure 9 correspond to the circled keywords in Figure 8, which received high attention from the SC-Seq Encoder of NFASRPR-TRANS. Meanwhile, Figure 11 is the attention heatmap of the 6-th layer of the Transformer encoder, which further confirms our observation that the Transformer is inefficient at capturing these keywords. Specifically, the attention score between the keywords 'db' and 'wrapper' of group number 3 in the Transformer is almost 0, whereas the keywords 'sync' and 'wrapper' are nearly fully attended to, which may be the reason why the generated summary of Transformer is lacking these keywords.

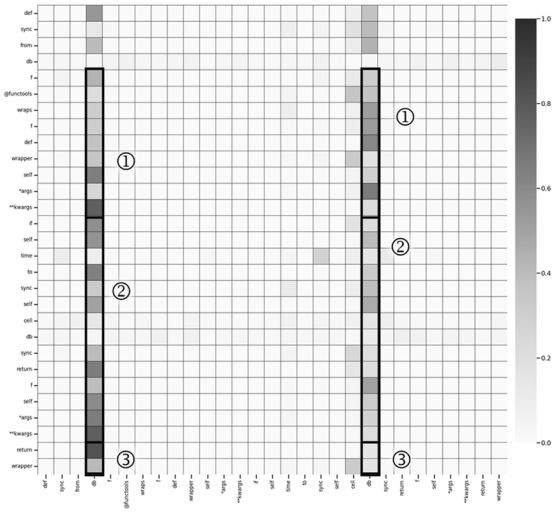


FIGURE 9. The attention heatmap of the SC-Seq Encoder.

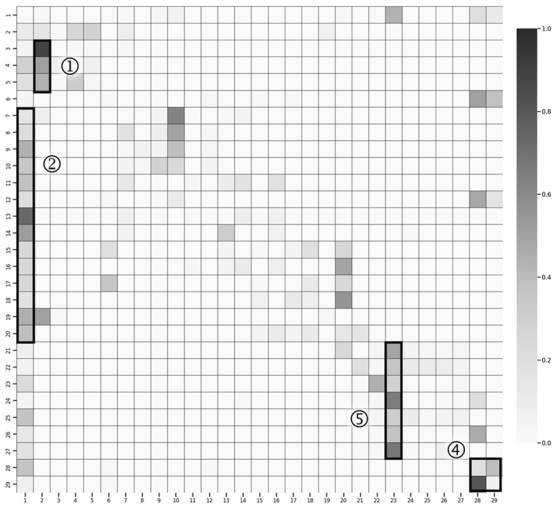


FIGURE 10. The attention heatmap of the AST-RP Encoder.

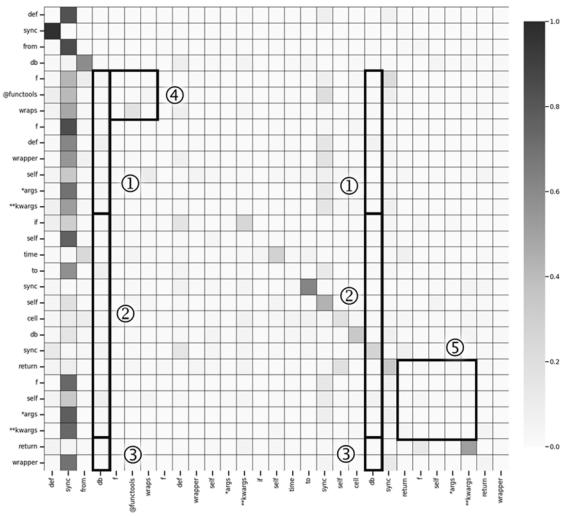


FIGURE 11. The attention heatmap of the transformer [24] Encoder.

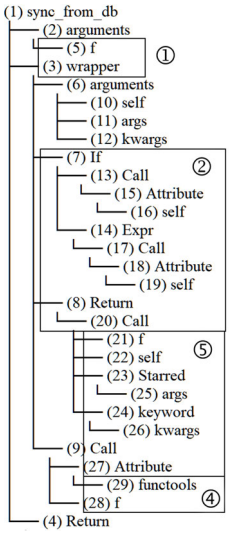


FIGURE 12. The AST graph of the source code.

Next, the group numbers in Figure 10 indicate the structure of the statements they circle in Figure 8, which are also shown in the AST of this test case in Figure 12. Group number 1 denotes the relationship between the input argument 'f' and the nested function 'wrapper', while group number 4 represents information regarding the 'functools.wraps' and the input argument 'f'. Group number 2 shows the structural information of the nested function's content, and group number 5 outlines the structure of the nested function's return statement. Figure 10 highlights that the AST-RP Encoder of NFASRPR-TRANS puts significant emphasis on the structure of the source code represented by these group numbers. Comparatively, Figure 11 illustrates that the Transformer is unable to capture the information related to these attention groups, which could be another reason for generating a mismatched summary.

VI. RESEARCH QUESTIONS AND THREATS TO VALIDITY

A. RESEARCH QUESTIONS

RQ1: Does our proposed model, NFASRPR-TRANS, generate summaries with a higher level of accuracy in explaining the behavior of source codes compared to other DL-based models?

To address the shortcomings of the DL-based models that we discussed in Example 1 in section I, we propose comprehensive solutions in our model to tackle each issue in detail. This RQ intends to verify whether our model, NFASRPR-TRANS, generates more accurate summaries in explaining source code behavior than other DL-based models.

Tables 3 and 4 show the overall performance of different models for the Source Code Summarization task on the Java [27] and Python [28] datasets, and it is observed that

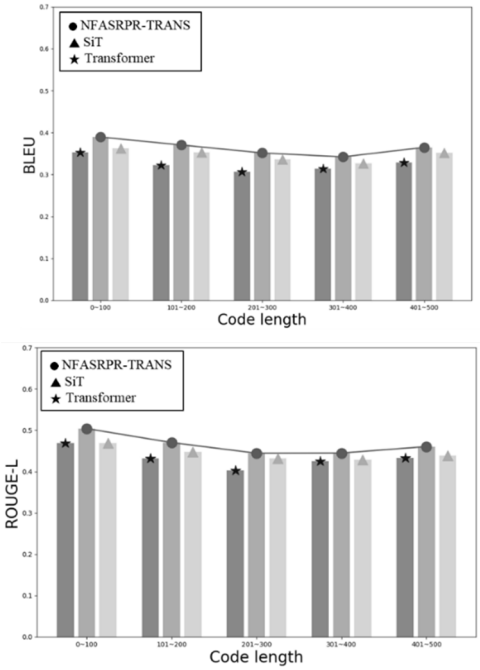


FIGURE 13. Performance of varying code length in different methods on the Java dataset.

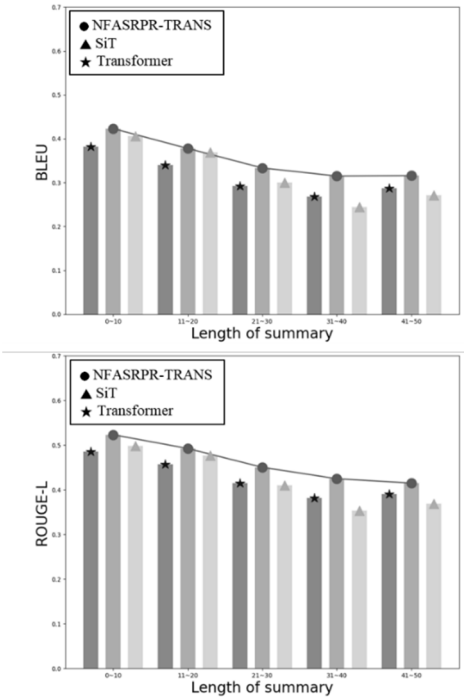


FIGURE 14. Performance of varying summary length in different methods on the Java dataset.

NFASRPR-TRANS achieves the best performance across all five metrics for both datasets compared to other DL-based baseline models. On the Java dataset, NFASRPR-TRANS achieves higher BLEU, ROUGE-L, CIDEr, METEOR, and SPICE scores than Transformer [24] (+11.45%, +7.88%, +21.60%, +10.25%, and +15.22%) and SiT [25] (+6.21%, +6.30%, +6.94%, +10.30%, and +6.63%) did. Similarly, on the Python dataset, NFASRPR-TRANS achieves higher scores in all five metrics than both Transformer [24] (+3.72%, +2.67%, +7.63%, +4.90%, and +6.82%) and SiT [25] (+2.43%, +1.84%, +5.94%, +4.33%, and +4.83%) do.

The results reveal that NFASRPR-TRANS, which utilizes both the Gaussian Embedder and the Tree Positional Encoding to integrate the non-Fourier and AST-structural relative position representations into Transformer, leads to improvement in the Source Code Summarization task.

RQ2: What are the impacts of the length of source codes and summaries on the performance of NFASRPR-TRANS in Source Code Summarization?

As previously mentioned, NFASRPR-TRANS is designed to address the challenge of learning complex structured source code, often characterized by long code and summary lengths. Hence, this RQ seeks to examine the impact of code length on the code representation learning of the model, as well as the effect of summary length on evaluating text generation performance.

As shown in Figure 13 and 14, the BLEU and ROUGE-L scores of Transformer [24] and SiT [25] decrease while code and summary lengths increase. This indicates that these models have the difficulty of learning the code representations in long code length and in generating a complete long summary.

In contrast, Figure 13 shows that NFASRPR-TRANS outperforms the baselines on both metrics in the Java [27] dataset, and also maintains a stable performance when the code lengths is increasing. This indicates that the AST-structural relative position representation of NFASRPR-TRANS is effective in helping the model handle the complex structure of the source code. Similarly, Figure 14 illustrates that NFASRPR-TRANS achieve the best performance with varying summary lengths. Although the metrics scores have slightly decreased with increasing summary lengths, the gaps of the performance compared to the baselines in the last two summary-length intervals are the largest. This indicates that the NFASRPR-TRANS has a better ability to generate complete long summaries.

RQ3: Do the features of different programming languages (e.g., average source code length) affect the model performance?

In this study, we use Java [27] and Python [28] datasets to demonstrate the performance of NFASRPR-TRANS. Since different programming languages have their own unique features when coding, such as a programming rule and an average source code length, we compare the performance of NFASRPR-TRANS on both Java and Python datasets to verify how the programming language features affect the performance of the model.

As shown in Tables 3 and 4, NFASRPR-TRANS shows a better performance on the Java dataset, yet a lower one on the Python dataset. Tables 1 and 2 demonstrate the data

characteristics of both datasets, which show that the average code length in the Python dataset is relatively short. Consequently, the corresponding AST is small, making the contribution of the AST-structural relative position representation of NFASRPR-TRANS less in the Python dataset than in the Java dataset.

However, NFASRPR-TRANS outperforms the baselines on both Java and Python datasets, indicating that applying the AST-structural relative position representation in the model can outperform models without it.

RQ4: Does NFASRPR-TRANS have a faster convergence rate than other DL-based models do?

Having a faster convergence rate means that the model takes fewer training steps to achieve the best performance compared to the other models. As Figures 6 and 7 show, NFASRPR-TRANS has a faster convergence rate than Transformer [24] and SiT [25] on the Java [27] and Python [28] datasets do. Figure 6 demonstrates that NFASRPR-TRANS surpassed all baselines from the 30th to the 80th epoch and achieved the best performance of the baselines in about 120 epochs. In comparison, baselines required more than 70 additional epochs to reach the same level of performance.

It's worth noting that the convergence rate is largely influenced by the architecture of the model, training data, and training equipment. Our two baselines, Transformer [24] and SiT [25], are both DL-based models that share a similar architecture of NFASRPR-TRANS, and they are trained using the same datasets and equipment as NFASRPR-TRANS does. Therefore, it is safe to conclude that applying non-Fourier and AST-Structural relative position representations in Transformer, NFASRPR-TRANS, converges faster than other DL-based models on both Java and Python datasets do.

RQ5: Do different hyper-parameters in NFASRPR-TRANS affect the model performance?

In NFASRPR-TRANS, we leverage the Gaussian Embedder and the Tree Positional Encoding as two major components. As mentioned in section III-A, the Gaussian Embedder is used to encode the non-Fourier relative position representations, and its hyper-parameter σ has an impact on its ability to perform positional encoding, whereas the Tree Positional Encoding is used to encode the AST-structural relative position representations, and its hyper-parameters g and d have an impact on its ability to perform positional encoding.

To analyze how different values of these hyper-parameters affected the performance of the model, we conducted ablation experiments where the value of σ varies from $[10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}]$, and the value of (g, d) varies from $[(4, 32), (8, 16), (16, 8), (32, 4)]$. The performance comparison of these variant models is shown in Tables 6 and 7, and the best score in each evaluation metric is highlighted in bold.

Table 6 demonstrates that choosing the appropriate value of $\sigma = 0.001$ is crucial for NFASRPR-TRANS, since an incorrect value can cause a slight decrease in the performance of the model. Although a smaller σ brings better memorization to the model, it may lead to poor generalization of the model, making the model fail to train. In addition, Table 7 illustrates that the performance of variant models drops dramatically when g and d are too small or too large. While the best performance (g, d) pair is also the hyper-parameters we set for NFASRPR-TRANS. It means that the (g, d) pair should be chosen to fit most of the ASTs' widths and depths to obtain the best AST-structural relative position representation.

Therefore, it is important to carefully select the optimal values of σ and (g, d) to ensure the best performance of NFASRPR-TRANS.

TABLE 6. The performance of different σ values on Java and Python datasets.

	σ	BLEU	ROUGE-L	CIDEr	METEOR	SPICE
Java	10^{-1}	33.7	44.27	2.74	20.36	0.338
	10^{-2}	35.88	46.57	2.94	21.64	0.359
	10^{-3}	**37.96**	**48.75**	**3.15**	**22.91**	**0.386**
	10^{-4}	--	--	--	--	--
Python	10^{-1}	30.37	44.76	2.01	18.75	0.278
	10^{-2}	30.22	44.82	2.01	19.00	0.280
	10^{-3}	**30.37**	**44.93**	**2.03**	**19.05**	**0.282**
	10^{-4}	30.35	44.85	2.02	18.98	0.279

TABLE 7. The performance of different (g, d) values on Java datasets.

(g, d)	BLEU	ROUGE-L	CIDEr	METEOR	SPICE
(4, 32)	14.19	21.16	0.67	6.81	0.089
(8, 16)	**37.96**	**48.75**	**3.15**	**22.91**	**0.386**
(16, 8)	37.37	48.11	3.11	22.60	0.380
(32, 4)	12.93	20.40	0.50	6.24	0.073

RQ6: How effective are the two types of relative position representations that we applied in NFASRPR-TRANS?

As we had mentioned before, NFASRPR-TRANS encodes non-Fourier relative position representations using the Gaussian Embedder to learn the semantics of source code keywords, while AST-structural relative position representations are encoded using Tree Positional Encoding to learn the syntax of the source code.

To validate the effectiveness of these two position representations, we conduct an ablation experiment in which we added or removed these two modules in NFASRPR-TRANS. The results, as shown in Tables 3 and 4, demonstrate that both variant models achieved $2\% \sim 9\%$ higher performance across all evaluation metrics compared to the baselines on both datasets. On the other hand, the performance of both variant models decreased $1\% \sim 2\%$ when compared with the full model on both datasets. In other words, both types of relative position representations are beneficial for NFASRPR-TRANS to learn more semantics or syntax of the source code, thus supporting the model in generating higher quality summaries.

RQ7: Does the use of different tree traversal algorithms in the Tree Positional Encoding module affect the performance of our model?

TABLE 8. The training time on Java and Python datasets.

	Java	Python
Transformer [24]	181741	68234
SiT [25]	**76916**	**58281**
NFASRPR-TRANS	84794	67156

In section III-A, we proposed that the Pre-order Reverse traversal algorithm is more suitable for Tree Positional Encoding to generate natural language sentences. To verify the performance of different tree traversal algorithms on the Tree Positional Encoding module, we implemented the Tree Positional Encoding with different tree traversal algorithms, including Pre-order, In-order, Post-order, BFS, and IDDFS [44]. The performance comparison of these variant models with ours is shown in Table 5.

The results demonstrate that the Pre-order Reverse traversal algorithm outperforms other tree traversal algorithms in both Java and Python datasets. Therefore, we can conclude that applying different tree traversal algorithms in the Tree Positional Encoding module affects the performance of the model. Meanwhile, by employing the Pre-order Reverse traversal algorithm in this module, we can achieve a better performance in the Source Code Summarization task.

RQ8: What is the cost of implementation of NFASRPR-TRANS?

As we had mentioned in section III, NFASRPR-TRANS employs the Gaussian Embedder and the Tree Positional Encoding. Both make NFASRPR-TRANS superior to previous works, but also leave the model with more parameters to train. Table 8 demonstrates the training time in second of different model in Java and Python dataset. The results demonstrate that the time cost of NFASRPR-TRANS is higher than SiT, but lower than Transformer.

Although NFASRPR-TRANS takes more time cost than SiT, it has the best performance, as shown in section V.

B. THREATS TO VALIDITY

There are certain undetected factors in our study that might produce unexpected experimental outcomes that result in erroneous conclusions. To assess the threats to the validity of our study, we discuss the following four aspects of validity threats in this section.

1) INTERNAL VALIDITY

The quality of the collected summaries in the datasets posed the most significant threat to the internal validity of our study. For example, both Java [27] and Python [28] datasets were collected from the GitHub, where poor quality summaries for source code, i.e., noisy data, may exist; including unrelated and unreadable summaries. Specifically, in the Java dataset, the referenced summary of each code/summary pair was taken from the first sentence of the corresponding Javadoc in the source code, while the Python dataset contains some Python2 source codes which are no longer supported by the Python programming language. These factors result in the datasets containing some noisy data, some of which are listed in Figure 15. The presence of noisy data could mislead the DL-based models in predicting the summaries. However, efficiently removing the unexpected noisy data is challenging. To address this threat, future studies could consider exploring alternative methods that eliminate redundancy data and reduce the impact of noisy data in the datasets.

```
// builds a function to retrieve tags given and endpoint .
public Builder ( ) { }
```

(a) This Java snippet lacks code content but contains a meaningful summary.

```
// description of the method.
protected void openFile ( File f ) {
    if ( f == null ) {
        return ;
    }
    if ( ! f . exists ( ) ) {
        Debug . output ( STRING + getName ( ) + STRING ) ;
        return ;
    }
    E00File = f ;
    if ( gui != null ) {
        label . setText ( E00File . getName ( ) ) ;
} }
```

(b) The summary of this Java snippet is meaningless.

```
'''inclusive_or_expression :
        inclusive_or_expression or exclusive_or_expression .'''
def p_inclusive_or_expression_2(t):
    pass
```

(c) This Python snippet lacks code content and contains a meaningless summary.

```
'''on windows'''
def rmtree_errorhandler(func, path, exc_info):
    if (os.stat(path).st_mode & stat.S_IREAD):
        os.chmod(path, stat.S_IWRITE)
        func(path)
        return
    else:
        raise
```

(d) The summary of this Python snippet is meaningless.

FIGURE 15. Some noisy data in the Java and Python datasets.

2) EXTERNAL VALIDITY

The external validity of our proposed model may be threatened by the generalizability of it, i.e., whether or not our proposed model can be applied to different programming languages. Therefore, in our study, we used two publicly available datasets, Java [27] and Python [28], which are widely used programming languages in the market [45]. As demonstrated in section IV-B, the experimental results show that NFASRPR-TRANS outperforms other DL-based models in both datasets. Additionally, as shown in section V, the generated summary by NFASRPR-TRANS is of higher-quality, more complete, and more meaningful compared to other DL-based models. Although we believe that our proposed model for Source Code Summarization can be applied

to other programming languages, further experiments are necessary to confirm it.

3) CONSTRUCT VALIDITY

In our study, the construct validity may be called into question due to the potential biases inherent in the automatic evaluation metrics we earlier had employed to assess the similarities between the generated summaries and the referenced human-written summaries. To minimize this threat in our study, we have selected five different metrics: BLEU [29], ROUGE-L [30], CIDEr [31], METEOR [32], and SPICE [33].

In BLEU's study [29], the authors conducted a series of experiments on MT tasks, involving four different natural languages translated into English, to investigate the correlation between BLEU scores and human evaluation results. Their findings revealed a strong positive correlation between BLEU and human judgement, with a correlation coefficient of 0.99. The authors pointed out that BLEU can accurately determine whether two English sentences correspond to the same translated sentence. In a similar vein, BLEU can also be applied to judge whether the two summaries correspond to the input source code.

The study of ROUGE-L [30] demonstrates that there is a high correlation between ROUGE-L scores and human judgement in the document summarization task. As both Source Code Summarization and document summarization generate brief sentences to describe the input content, we can aptly assume that ROUGE-L scores can be used to evaluate the similarity of the generated summaries and human-written summaries with a human-like view.

While many others – including CIDEr, METEOR and SPICE – have suggested, n-gram metrics such as BLEU and ROUGE-L have several weaknesses. The evaluation by BLEU and ROUGE-L is based on the n-gram approach, which involves sensitive word-to-word matching between two sentences, and that may result in incorrect matching, especially for common function words that are not critical for the meaning of the sentence.

Therefore, we chose to employ METEOR [32], which is a recall-oriented metric. Different from BLEU, METEOR has an alignment to check the mapping of each word in two ways, including exact mapping and stemmed mapping. In the study of METEOR [32], the authors evaluated both BLEU and METEOR on the MT task and found that METEOR overcomes the weaknesses of BLEU and has a higher correlation with human judgement.

In addition, the studies of CIDEr [31] and SPICE [33] have evaluated image caption tasks with several metrics, including BLEU, ROUGE, METEOR. The findings reveal that all five metrics have their best performance in different image caption tasks. For example, CIDEr is the top performer in evaluating the similarity of human judgements and generated captions, while SPICE has the best performance for evaluating the similarity between two sentences, considering other features such as the objects, attributes and relations in the input image. The image caption task shares similarities with the Source Code Summarization task in that both generate concise sentences to describe the input data. Hence, we can adopt CIDEr and SPICE for evaluating our model.

The above analysis of the validity criteria for the five metrics we used reveals that these metrics are not only valid for measuring the similarity between predicted and human-written summaries in the Source Code Summarization task, but they also compensate for each other's shortcomings. This eliminates the potential threat of construct validity in our study as much as possible.

4) CONCLUSION VALIDITY

The conclusion of our study is that NFASRPR-TRANS outperforms other DL-based models in Source Code Summarization. To analyze the validity of this conclusion, we discuss it based on two criteria: 1) Did the re-built models of baselines perform as well as their original models? 2) Does NFASRPR-TRANS obtain higher metric scores than the other DL-based models do?

To satisfy the first criterion, we trained our model and baselines on the same server to guarantee a just comparison. However, there are several undefined hyper-parameters in the baselines' studies [24], [25]. Although, we set these hyper-parameters to the same values as our model, these undefined hyper-parameters could potentially compromise the performance of the baselines. Therefore, further research is needed to examine the impact of assigning these undefined hyper-parameters in order to fully conclude that NFASRPR-TRANS outperforms other DL-based models.

Next, we discuss the second criterion for conclusion validity. Recall the experimental results shown in Tables 3 and 4: NFASRPR-TRANS outperforms all the baselines on both datasets by achieving 2% - 10% higher performance across all five metrics. As a result, we can conclude that NFASRPR-TRANS is demonstrated to be a superior performance in source code summarization compared to other DL-based models.

VII. CONCLUSION

Program comprehension is widely considered a critical factor in software development. Through the use of Source Code Summarization, programmers are able to have brief yet comprehensive summaries of source code behavior, allowing greater efficiency in program comprehension. Previous studies have shown that Transformer-based models outperform other deep learning (DL) based models in the area of Source Code Summarization. However, these models have confronted several problems, including learning the semantics of a source code's keywords, and understanding the complex structure of source code.

In this study, we propose NFASRPR-TRANS, which can address these problems and achieves a better performance compared to Transformer-based models in the Source Code Summarization task. In NFASRPR-TRANS, there are two different Transformer encoders, which employ Gaussian Embedder and Tree Positional Encoding as positional

encoding schemes, respectively. By utilizing the Gaussian Embedder, the SC-Seq Encoder can encode the non-Fourier relative position representation of the input source code sequence and handle the semantics of the source code's keywords. Meanwhile, the Tree Positional Encoding in the AST-RP Encoder can encode the AST-structural relative position representation and equips NFASRPR-TRANS with the ability to learn the complex structure by the ASTs of the input source code.

We have conducted competitive experiments with previous models on the Java and Python datasets. Our experimental results show that NFASRPR-TRANS outperforms the other DL-based models and has a faster convergence rate than other Transformer-based models. On the Java dataset, NFASRPR-TRANS achieves at least 6.21%, 6.30%, 6.94%, 10.30%, and 6.63% improvements on BLEU [29], ROUGE [30], CIDEr [31], METEOR [32], and SPICE [33], respectively. While on the Python dataset, it achieves improvements of at least 2.43%, 1.84%, 5.94%, 4.33%, and 4.83% on five metrics, respectively.

In addition, we have analyzed a qualitative test case and visualized its attention heat maps of NFASRPR-TRANS, and it is observed that the semantics between specific keywords are well learned by the SC-Seq Encoder, and the complex structure of the source code is well learned by the AST-RP Encoder.

In future planning, we will apply NFASRPR-TRANS to other programming languages in order to assess its generalizability. However, given that different programming languages have varying features, grammars, and usage patterns, we also plan on integrating specific features of each programming language into NFASRPR-TRANS for Source Code Summarization. The objective is to improve the domain-specific performance of the model.

REFERENCES

[1] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Trans. Softw. Eng.*, vol. 44, no. 10, pp. 951–976, Oct. 2018, doi: 10.1109/TSE.2017.2734091.

[2] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proc. 23rd Annu. Int. Conf. Design Communication: Documenting Designing for Pervasive Inf.*, New York, NY, USA, Sep. 2005, pp. 68–75, doi: 10.1145/1085313.1085331.

[3] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proc. ACM/IEEE 32nd Int. Conf. Softw. Eng.*, Cape Town, South Africa, May 2010, pp. 223–226, doi: 10.1145/1810295.1810335.

[4] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 1–37, Jul. 2018, doi: 10.1145/3212695.

[5] T. H. M. Le, H. Chen, and M. A. Babar, "Deep learning for source code modeling and generation: Models, applications, and challenges," *ACM Comput. Surv.*, vol. 53, no. 3, pp. 1–38, May 2021, doi: 10.1145/3383458.

[6] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, New York, NY, USA, Sep. 2010, pp. 43–52, doi: 10.1145/1858996.1859006.

[7] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Proc. 33rd Int. Conf. Softw. Eng. (ICSE)*, Honolulu, HI, USA, May 2011, pp. 101–110, doi: 10.1145/1985793.1985808.

[8] B. P. Eddy, J. A. Robinson, N. A. Kraft, and J. C. Carver, "Evaluating source code summarization techniques: Replication and expansion," in *Proc. 21st Int. Conf. Program Comprehension (ICPC)*, San Francisco, CA, USA, May 2013, pp. 13–22, doi: 10.1109/ICPC.2013.6613829.

[9] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for Java classes," in *Proc. 21st Int. Conf. Program Comprehension (ICPC)*, San Francisco, CA, USA, May 2013, pp. 23–32, doi: 10.1109/ICPC.2013.6613830.

[10] P. Rodeghero, C. Mcmillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proc. 36th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA, May 2014, pp. 390–401, doi: 10.1145/2568225.2568247.

[11] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proc. 17th Work. Conf. Reverse Eng.*, Beverly, MA, USA, Oct. 2010, pp. 35–44, doi: 10.1109/WCRE.2010.13.

[12] E. Wong, T. Liu, and L. Tan, "CloCom: Mining existing source code for automatic comment generation," in *Proc. IEEE 22nd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, Montreal, QC, Canada, Mar. 2015, pp. 380–389, doi: 10.1109/SANER.2015.7081848.

[13] P. W. McBurney and C. McMillan, "Automatic source code summarization of context for Java methods," *IEEE Trans. Softw. Eng.*, vol. 42, no. 2, pp. 103–119, Feb. 2016, doi: 10.1109/TSE.2015.2465386.

[14] Z. Gong, C. Gao, Y. Wang, W. Gu, Y. Peng, and Z. Xu, "Source code summarization with structural relative position guided transformer," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Honolulu, HI, USA, Mar. 2022, pp. 13–24, doi: 10.1109/saner53432.2022.00013.

[15] 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)*, Montreal, QC, Canada, May 2019, pp. 795–806, doi: 10.1109/ICSE.2019.00087.

[16] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, doi: 10.1162/neco.1997.9.8.1735.

[17] A. LeClair, S. Haque, L. Wu, and C. Mcmillan, "Improved code summarization via a graph neural network," in *Proc. 28th Int. Conf. Program Comprehension*, New York, NY, USA, Jul. 2020, pp. 184–195, doi: 10.1145/3387904.3389268.

[18] A. Vaswani, N. M. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, Jun. 2017, pp. 6000–6010, doi: 10.48550/arXiv.1706.03762.

[19] 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)*, Madrid, Spain, May 2021, pp. 184–195, doi: 10.1109/ICPC52881.2021.00026.

[20] X. Zhang, S. Yang, L. Duan, Z. Lang, Z. Shi, and L. Sun, "Transformer-XL with graph neural network for source code summarization," in *Proc. IEEE Int. Conf. Syst., Man, Cybern. (SMC)*, Melbourne, VIC, Australia, Oct. 2021, pp. 3436–3441, doi: 10.1109/SMC52423.2021.9658619.

[21] Z. Tang, X. Shen, C. Li, J. Ge, L. Huang, Z. Zhu, and B. Luo, "AST-trans: Code summarization with efficient tree-structured attention," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, Pittsburgh, PA, USA, May 2022, pp. 150–162, doi: 10.1145/3510003.3510224.

[22] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics*, Berlin, Germany, 2016, pp. 2073–2083, doi: 10.18653/v1/p16-1195.

[23] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proc. 26th Conf. Program Comprehension*, Gothenburg, Sweden, May 2018, pp. 200–210, doi: 10.1145/3196321.3196334.

[24] 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*, Jul. 2020, pp. 4998–5007. Accessed: May 31, 2023. [Online]. Available: https://aclanthology.org/2020.acl-main.449

[25] H. Wu, H. Zhao, and M. Zhang, "Code summarization with structure-induced transformer," in *Proc. Findings Assoc. Comput. Linguistics, ACL-IJCNLP*, Aug. 2021, pp. 1078–1090. Accessed: May 31, 2023. [Online]. Available: https://aclanthology.org/2021.findings-acl.93

[26] 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*, Vancouver, BC, Canada, Jul. 2017, pp. 1073–1083. Accessed: May 31, 2023. [Online]. Available: https://aclanthology.org/P17-1099

[27] 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. (IJCAI)*, Jul. 2018, pp. 2269–2275.

[28] 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 IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Montpellier, France, Sep. 2018, pp. 397–407, doi: 10.1145/3238147.3238206.

[29] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. Annu. Meeting Assoc. Comput. Linguistics (ACL)*, Philadelphia, PA, USA, Jul. 2002, pp. 311–318.

[30] C. Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Proc. Text Summarization Branches Out, ACL-04 Workshop*, Barcelona, Spain, Jul. 2004, pp. 74–81.

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

[32] 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*, Jun. 2005, pp. 65–72.

[33] P. Anderson, B. Fernando, M. Johnson, and S. Gould, "SPICE: Semantic propositional image caption evaluation," in *Computer Vision—ECCV 2016* (Lecture Notes in Computer Science), vol. 9909, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham, Switzerland: Springer, 2016, pp. 382–398, doi: 10.1007/978-3-319-46454-1_24.

[34] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. (ICSE)*, Seoul, South Korea, Oct. 2020, pp. 1385–1397.

[35] J. Zheng, S. Ramasinghe, and S. Lucey, "Rethinking positional encoding," Jul. 2021, *arXiv:2107.02561*.

[36] M. Dehghani, S. Gouws, O. Vinyals, J. Uszkoreit, and Ł. Kaiser, "Universal transformers," Jul. 2018, *arXiv:1807.03819*.

[37] P. Shaw, J. Uszkoreit, and A. Vaswani, "Self-attention with relative position representations," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Lang. Technol.*, New Orleans, LA, USA, vol. 2, Jun. 2018, pp. 464–468. Accessed: May 31, 2023. [Online]. Available: https://aclanthology.org/N18-2074

[38] J. Zheng, S. Ramasinghe, X. Li, and S. Lucey, "Trading positional complexity vs deepness in coordinate networks," in *Computer Vision—ECCV 2022* (Lecture Notes in Computer Science), S. Avidan, G. Brostow, M. Cissé, G. M. Farinella, and T. Hassner, Eds., vol. 13687. Cham, Switzerland: Springer, Oct. 2022, pp. 144–160, doi: 10.1007/978-3-031-19812-0_9.

[39] V. L. Shiv and C. Quirk, "Novel positional encodings to enable tree-based transformers," in *Proc. 33rd Int. Conf. Neural Inf. Process. Syst.*, no. 1082. Red Hook, NY, USA: Curran Associates, Dec. 2019, pp. 12081–12091.

[40] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Int. Conf. Neural Inf. Process. Syst. (NIPS)*, vol. 2. Red Hook, NY, USA: Curran Associates, Dec. 2013, pp. 3111–3119.

[41] *Javalang*. Accessed: Jun. 30, 2023. [Online]. Available: https://github.com/c2nes/javalang

[42] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, San Diego, CA, USA, May 2015.

[43] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," Jan. 2019, *arXiv:1711.05101*.

[44] R. E. Korf, "Depth-first iterative-deepening," *Artif. Intell.*, vol. 27, no. 1, pp. 97–109, Sep. 1985, doi: 10.1016/0004-3702(85)90084-0.

[45] Stackoverflow. *Most popular technologies*. Accessed: May 30, 2023. [Online]. Available: https://survey.stackoverflow.co/2022/#technology-most-popular-technologies

HSIANG-MEI LIANG received the B.S. and B.E. degrees (double degree) from the Department of Physics and the Department of Computer Science and Information Engineering, National Taiwan Normal University, Taipei, Taiwan, in 2021, and the M.E. degree from the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, in 2023. Her research interests include software engineering, deep learning, and NLP.

CHIN-YU HUANG (Member, IEEE) received the M.S. and Ph.D. degrees in electrical engineering from the National Taiwan University, Taipei, in 1994 and 2000, respectively. He is currently a Full Professor with the Department of Computer Science, Institute of Information Systems and Applications and the Institute of Information Security, National Tsing Hua University (NTHU), Hsinchu, Taiwan. He was with the Bank of Taiwan, from 1994 to 1999, and a Senior Software Engineer with Taiwan Semiconductor Manufacturing Company, from 1999 to 2000. Before joining NTHU, in 2003, he was a Division Chief of the Central Bank of China, Taipei. His research interests include software reliability engineering, software testing, software metrics, software testability, fault tree analysis, and system safety assessment. He received the Ta-You Wu Memorial Award of National Science Council, Taiwan, in 2008. He has been on the editorial board of Scientific Programming, since 2017, and the editorial board of *Journal of Information Science and Engineering*, since 2016. He is currently an Associate Editor of IEEE TRANSACTIONS ON RELIABILITY.