An Extensive Study of the Structure Features in Transformer-based Code Semantic Summarization

Kang Yang*, Xinjun Mao*, Shangwen Wang*, Yihao Qin*, Tanghaoran Zhang*, Yao Lu*, Kamal Al-Sabahi†
* Key Laboratory of Software Engineering for Complex Systems, National University of Defense Technology, China,
{yangkang, xjmao, wangshangwen13, qinyihao, zhangthr, luyao08}@nudt.edu.cn
† University of Technology and Applied Sciences-ibra, Oman, kamal@ict.edu.om

***Abstract*—Transformers are now widely utilized in code intelligence tasks. To better fit highly structured source code, various structure information is passed into Transformer, such as positional encoding and abstract syntax tree (AST) based structures. However, it is still not clear how these structural features affect code intelligence tasks, such as code summarization. Addressing this problem is of vital importance for designing Transformer-based code models. Existing works are keen to introduce various structural information into Transformers while lacking persuasive analysis to reveal their contributions and interaction effects. In this paper, we conduct an empirical study of frequently-used code structure features for code representation, including two types of position encoding features and AST-based structure features. We propose a couple of probing tasks to detect how these structure features perform in Transformer and conduct comprehensive ablation studies to investigate how these structural features affect code semantic summarization tasks. To further validate the effectiveness of code structure features in code summarization tasks, we assess Transformer models equipped with these code structure features on a structural dependent summarization dataset. Our experimental results reveal several findings that may inspire future study: (1) there is a conflict between the influence of the absolute positional embeddings and relative positional embeddings in Transformer; (2) AST-based code structure features and relative position encoding features show a strong correlation and much contribution overlap for code semantic summarization tasks indeed exists between them; (3) Transformer models still have space for further improvement in explicitly understanding code structure information.**

***Index Terms*—Transformer, empirical study, probing task, code summarization**

I. Introduction

Inspired by the successful application in the natural language processing (NLP) [1], [2], the state-of-the-art deep learning architecture, i.e., the Transformer [3], is widely utilized to process source code in support of code intelligence tasks [4]–[8], including method names recommendation [9], [10], code generation [11], [12], bug fixing [13], [14], code translation [15], [16], etc. Due to the global and long-range connections view for input sequences, Transformers outperform traditional sequential deep learning architectures, e.g., Convolutional Neural Network (CNN) [17] and Recurrent Neural Network (RNN) [18]–[21], as these architectures capture more on local dependencies [22]. Source code snippets are usually seen as a sequence of keywords in the programming language, identifiers, and punctuations, which are fed to the Transformer directly at the beginning of exploration [23]. In this case, the Transformer based models make predictions mainly depend on literal semantics. In contrast to natural language, the programming language is highly structured and contains rich structural information, such as abstract syntax tree, data and control flow graph. Therefore, increasing efforts are devoted to integrating the structure of source code into Transformer for effectiveness enhancement [24].

The self-attention mechanism is the key component of Transformer architecture, which treats the input sequence as an unordered bag of tokens. To remedy the loss of the order structure, the Transformer requires additional positional encodings [25], [26]. Ahmad et al. [23] adopted relative position representation [25] to model the pairwise code token relationship by injecting relative position embeddings into Transformer. Following this research line, several research works endeavour to insert non-sequential code structure features into the self-attention calculation as inductive biases to encode source code snippets [27]–[29].

Although methods that integrate code structure information into Transformer achieved promising summarization results, little is known about how these structural features affect code summarisation tasks' performance. Existing research works usually utilize structure features heuristically and validate their contributions by control variate technique, which neglects the interactive impact of these features. It will offer valuable guidance to figure out whether incorporating multi-features would achieve a sub-additive effect or result in a super-additive consequence. Therefore, there is an increasing demand to understand to what extent the embedding vectors from the Transformer encoder equipped with multiple structural features effectively capture the source code characteristics and how they affect code intelligence tasks.

To fill this gap, in this paper, we extensively study the frequently used code structure features in Transformers for two code intelligence tasks, i.e., code summarization [30], and extreme summarization [31]. The former aims to generate a sentence-level summary (usually explaining how the code snippet work), while the latter provides a summary in the form of a descriptive method name. Since generally both tasks aim at summarizing the main semantics of a given code snippet, we unified refer to them as **code semantic summarization**. We conduct a thorough empirical study of frequently used code

*Xinjun Mao and Shangwen Wang are the corresponding authors.
This research was supported by the National Science Foundation of China No.62172426.

structure features for code summarization tasks. More specifically, we would like to answer the question: *How do these code structure features utilized in Transformer affect code semantic summarization?* Addressing this question plays an important role in understanding Transformer-based code models. We propose a couple of probing tasks to detect how these structure features perform in the Transformer. Furthermore, we also conduct comprehensive ablation studies to investigate how these structural features affect code semantic summarization tasks. We aim to provide consistent observations and findings in both ablation studies and probing tasks to be more convincing. Our work is performed on the structure-induced Transformer architecture, which is the Transformer equipped with relation-aware self-attention [25] and structure-induced self-attention in SIT [32]. Through comprehensive studies, we mainly find:

- There is a conflict between the influence of the absolute positional embeddings and the relative positional embeddings in Transformer, and the combination of them interferes with understanding tokens position information and generating code summaries.
- The AST-based code structure features and relative position encoding feature show strong *Pearson* correlation in Transformer, and much contribution overlap for code semantic summarization indeed exists between them.
- In a more structural-dependent setting, in which generating summaries require a more explicit structural understanding of source code, the performance of the Transformer model drops significantly. The Transformer equipped with structure features still has space for further improvement in explicitly understanding code structure.

These findings call for more concentration on integrating code structure features into Transformer and could inspire future studies on source code representation in Transformer.

II. Preliminaries

This section reviews some background knowledge of our study and describes our research questions.

A. Vanilla Transformer and Self-Attention

Transformer [3] is stacked with multi-head attention and parameterized linear transformation layers for both the encoder and decoder. In each layer, the multi-head attention employs h attention heads and performs the self-attention mechanism. We describe Transformer architecture for a code summarization task, which maps a source code snippet to a target natural language summary. Let $c = \{w_1, w_2, \ldots, w_n\}$, where w_i is the i_{th} token in the token sequence of the source code snippet, $i \in \{1, 2, \ldots, N\}$, ($N$ is the length of source vocabulary) denote a piece of code that contains n tokens. The input tokens are mapped into a sequence of embedding vectors, $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}, x_i \in \mathbb{R}^{d_x}$ (d_x is the embedding size of x).

The self-attention mechanism is the core component of the Transformer. In a single attention head, input embeddings are transformed into a sequence of output vectors with the same length: $z = \{z_1, z_2, \ldots, z_n\}$, $z_i \in \mathbb{R}^{d_z}$. Each output element z_i is computed as a weighted sum of inputs, which can be formulated as:

$$z_i = \sum_{j=1}^{n} \alpha_{ij} \left(x_j W^V\right) \tag{1}$$

$$\alpha_{ij} = \frac{\exp e_{ij}}{\sum_{k=1}^{n} \exp e_{ik}} \tag{2}$$

$$e_{ij} = \frac{x_i W^Q \left(x_j W^K\right)^T}{\sqrt{d_z}} \tag{3}$$

where $W^Q, W^K, W^V \in \mathbb{R}^{d_x \times d_z}$ are projection matrices. These parameter matrices are uniquely applied in parallel per layer and attention head. Attention weights α_{ij} are the query-key similarity between code token i and token j, $\sum_{i=1}^{n} \alpha_{ij} = 1, \alpha_{ij} \geq 0$.

A Transformer block consists of the described multi-head self-attention, a residual connection, a layer-normalization step, and a position-wise fully-connected network. The overall Transformer model comprises L stacking layers of the Transformer blocks. In the code summarization task, future target words are masked in self-attention computation, and the consequence stacking blocks are called Transformer decoder. Otherwise, the Transformer encoder is without masking. In addition, the attention from the decoder to the encoder is also computed.

B. Positional embeddings in Transformer

Absolute positional embeddings (APE): Initially designed for NLP to account for the sequential structure order, the vanilla Transformer assigns an absolute positional representation for each token in the input sequence. After mapping input tokens to embedding vectors $\mathrm{x} = \{x_1, x_2, \ldots, x_n\}, x_i \in \mathbb{R}^{d_x}$, x_i are summed with positional embeddings p_i, which helps it determine the position of each token, or the distance between different words in the sequence, $\tilde{x}_i = x_i + p_i, \quad p_i \in \mathbb{R}^{d_x}$. Each position $i = 1, 2, \ldots, L$, indicates the index of token in the sequence, based on which p_i can be computed with sine and cosine functions [3] or be learned in an embedding matrix [23].

Relative positional encoding (RPE): In code representation learning, the mutual interactions embody the semantic functions of code snippets [23]. In expression statements $n+m$ and $m+n$, absolute positions do not make a difference in their semantic meaning. Shaw et al. [25] firstly proposed relation-aware self-attention to leverage relative positional encoding.

$$z_i = \sum_{j=1}^{n} \alpha_{ij} \left(x_j W^V + a_{ij}^V\right) \tag{4}$$

$$e_{ij} = \frac{x_i W^Q \left(x_j W^K + a_{ij}^K\right)^T}{\sqrt{d_z}} \tag{5}$$

Where a_{ij}^V and a_{ij}^K are the pairwise relationship representations between input tokens for position i and j. With the hypothesis that precise relative position information is not useful beyond a certain distance, Shaw et al. [25] suggested to clip the maximum relative positional relationship to a value of k.

We follow the suggestion in [23] to set $k = 32$ in our experiments. Moreover, to better applying relative position embeddings to code representation, directional information are ignored in [23], i.e. relative position embeddings for token i and token j, or for token j and i share the same representation.

$$a_{ij}^K = w_{\text{clip}(j-i,k)}^K, a_{ij}^V = w_{\text{clip}(j-i,k)}^V \tag{6}$$

$$\text{clip}(x,k) = \min(|x|,k) \tag{7}$$

The absolute position encoding focuses on capturing the sequence order information of code tokens, and the relative position encoding treats relationships between token pairs as edges, which focus more on the relative position differences between two code tokens.

C. AST structures in Transformer

Structure-induced self-attention: An abstract syntax tree uniquely determines a source code snippet given the language and grammar rules. Thus, AST is usually applied to represent the syntactic structure of code. In an AST, each node contains a type, which represents the syntactic unit of specific language grammar (e.g. *identifier*, *argument_list*, *for_statement*). Leaf nodes, also called terminals, contain a value (e.g., *url*, *file_path*, *opContext*), which usually are user-defined identifiers (e.g., variable names) and its values. Existing works have demonstrated the benefits of integrating AST properties into deep learning approaches [33], [34]. Inspired by relation-aware self-attention [25], much more types of pairwise relationships between code tokens are expected to enrich the code snippets representation. In Structure-induced Transformer (SIT) [32], authors expand AST into a multi-view graph based on code semantics, which are the combination of abstract syntax graph A_{ast}, control-flow graph A_{cf} and data-flow graph A_{df}. We show an example in Figure 1 to demonstrate the AST structure features of a Python code snippet. SIT adopt adjacency matrix to represent AST, based on which control-flow and data-flow edges are added to form the multi-view graph $A_{mv} = A_{ast} + A_{cf} + A_{df}$.

Self-attention mechanism equations (1-3) can be formulated as matrix form:

$$\textit{Self-Attention}(\mathbf{x}) = softmax\left(\frac{QK^T}{\sqrt{d_z}}\right)V \tag{8}$$

Where $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}, x_i \in \mathbb{R}^{d_x}$, denotes the input sequence of code tokens. In SIT, self-attention calculation is viewed as a directed cyclic graph, in which N ($= V$ in (8)) are n vector representations of code tokens (seen as vertexes in graph) and $E = \{e_{ij}\}$ ($= QK^T/\sqrt{d_z}$ in (8)) is viewed as weighted matrix of each edge, where e_{ij} represents the significance of vertex n_i attend to n_j. The matrix form Self-attention mechanism can be rewritten as follow:

$$\textit{Self-Attention}(\mathbf{x}) = E \cdot N \tag{9}$$

Note that, in vanilla Transformer, self-attention is a full connection cyclic graph. In SIT [32], multi-view graph is utilized to dropout the attention where $a_{ij} = 0$ in A_{mv}

$$\textit{Self-Attention}(\mathrm{x}) = softmax\left(\frac{A_{mv} \cdot QK^T}{\sqrt{d_z}}\right)V \tag{10}$$

D. Research Questions

RQ1: How do absolute and relative position encoding perform in code semantic summarization?

As described in the former part, various kinds of features of source code are integrated into the self-attention calculation, including the absolute/relative position encoding and the syntax information in ASTs. Experimental results in [23] and [25] suggested not to include absolute position encoding when there is pairwise relationship modeling but without explaining the rationale. It is of vital significance to probe the capability of capturing positional information, which may guide future design for source code representation. Therefore, we are motivated to validate and interpret the interactive impact of absolute/relative position encoding on code semantic summarization. We devise our first research question to seek some clues.

RQ2: To what extent do AST-based structure features and positional information contribute to code semantic summarization?

Moreover, Ahmad et al. [23] hypothesized that there is a limited advantage in exploring code structure information in code summarization because Transformer learns it implicitly with relative position representation. In contrast, later works integrated code structure information from AST into the self-attention calculation to improve code summarization. Existing approaches focus on capturing the structural information of source code, and little is known about the interaction influence of each introduced feature. Addressing this problem paves the way for future code representation works.

RQ3: How does Transformer perform in a more structural-dependent setting in code summarization?

Furthermore, to better figure out the capability of the Transformer model that utilizes code structure information, we design a code summarization dataset with a more structural dependent setting, in which generating a summary requires a deeper structural understanding of source code. The structural-dependent summarization datasets would offer a more straightforward method to evaluate the capability of code models that convert code structural information to semantic summaries.

III. Experimental setting

In this section, we describe our experimental setting for investigating the effect of code structure features in code semantic summarization tasks. First of all, we replicate structure-induced Transformer models for code semantic summarization tasks. After obtaining the trained models and the generated summaries with different structure feature settings, we design a couple of probing tasks to detect the interactive influence of these learned features in the Transformer by using the embeddings from model encoders equipped with various features. Additionally, we conduct ablation studies on the generated summaries to figure out how these features affect code semantic summarization tasks. The probing tasks could reveal how structure features preserve in the hidden representation

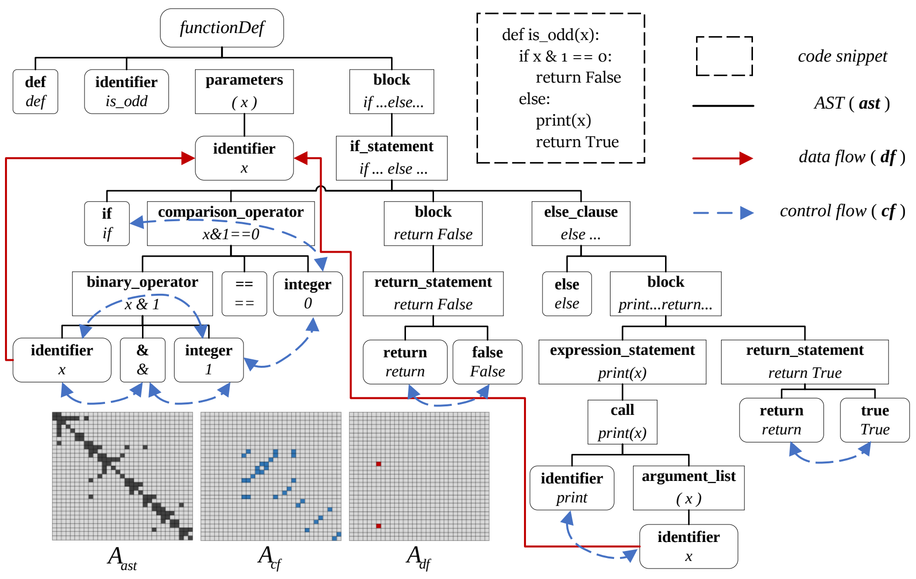
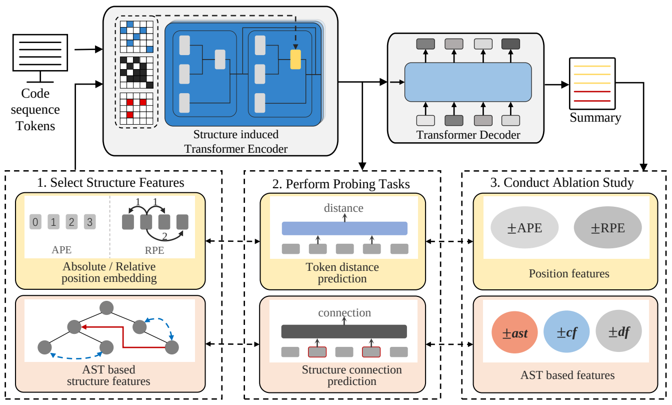


Fig. 1. An example of a Python code snippet with the corresponding ***ast***, ***cf*** and ***df*** graph dege connections and matrices.

of the Transformer, and the ablation studies illustrate how structure features perform in downstream summarization tasks. The probing tasks and the ablation studies are complementary, and the probing tasks can be the replenishment for interpreting the rationale for the effect of the ablation studies. The overall methodology of our study is illustrated in Figure 2.



Code sequence Tokens
Structure induced Transformer Encoder
Transformer Decoder
Summary
1. Select Structure Features
0 1 2 3
APE
RPE
Absolute / Relative position embedding
AST based structure features
2. Perform Probing Tasks
distance
Token distance prediction
connection
Structure connection prediction
3. Conduct Ablation Study
±APE
±RPE
Position features
±ast
±cf
±df
AST based features

Fig. 2. Overall methodology description of our work.

A. Experiment setup

Data and preprocessing. Our experiments are conducted on two code intelligent tasks. For code summarization, we use two public parallel benchmarks, Java from Wan et al. [10] and Python from Hu et al. [6], and we follow their initial training, test, and validation divisions. For code extreme summarization, the experiments are conducted on Java-small from Alon et al. [18], which contains 11 relatively large Java projects (nine projects for training, one project for validation and one project as our test set). The statistics of *Java*, *Python* and *Java-small* datasets are described in Table I. For the parser to generate ASTs, we use the open-source parser Tree-Sitter[1] to process both Java and Python code in a uniformed module, in which all the code tokens are natively mapped as terminals. For the code structure graphs used in structure-induced Transformer, we write a script to traverse AST-based features into abstract syntax tree (***ast***) adjacency matrix and control-flow (***cf***) graph matrix, and we use the data-flow (***df***) extractor provided by GraphCodeBERT [35] to obtain the data-flow graph matrix. Figure 1 shows an example of a Python code snippet with the corresponding ***ast***, ***cf*** and ***df***. All three code structure graph matrices are kept separately to allow ease of ablation. We adopt the way to split sub-tokens following [23] by applying *CamelCase* and *snake_case* tokenizers.

Training details. The implementation of the structure-induced Transformer is based on the PyTorch implementation of open-NMT and the source code in SIT [32]. We follow the hyper-parameters setting in [23] and [32]. The embedding size for source code and summary is 512. The batch size is 32, and the optimizer is Adam [36]. The learning rate was initially set to 1e-4 with a warmup rate of 0.06 and L2 weight decay of 0.01. For the code summarization task, we train the structure-induced Transformer models for a maximum of 200 epochs and perform an early stop if the validation performance does not improve for 20 consecutive iterations. For the extreme summarization task, we set the maximum epochs to 40 and perform an early stop when there is no improvement for 10 epochs. In the inference period, we use a beam search to generate a summary and set the beam size to 5.

B. Probing tasks

A probe is made up of a probing task and a probing classifier. A probing task is an auxiliary diagnostic task used to investigate whether a specific property is preserved in the

[1]https://github.com/tree-sitter/

TABLE I
STATISTICS OF THE EXPERIMENTAL DATASETS.

Datasets	Number of samples	Avg. tokens in source code	summary
Java-train	69708	106.05	17.72
Java-valid	8714	105.72	17.89
Java-test	8714	106.97	17.6
Java-if	1031	146.19	35.17
Java-loop	749	175.78	30.12
Java-struc-dpdt	929	91.06	16.07
Python-train	55538	77.67	9.46
Python-valid	18505	77.72	9.48
Python-test	18502	78.09	9.55
Python-if	1148	83.88	13.28
Python-loop	1276	101.59	12.36
Python-struc-dpdt	694	71.91	9.47
Java-small-train	600701	75.66	3.10
Java-small-valid	29144	50.79	3.54
Java-small-test	52015	84.67	2.75

embedding vectors, which are often utilized to diagnose pre-trained models. If a specific property can be predicted in a probing task given the embedding vector from a model encoder, the original model most likely has the capability to encode it in its hidden states. To minimize the interpretability problem, probing tasks should not be complicated in nature compared to the main task for the original model. Therefore, a probing classifier is usually a linear or shallow multi-layer perceptron with no or few hidden layers of a classifier on its own.

In order to determine whether or to what extent embeddings of source code from the structure-induced Transformer encoder reflect code understanding in terms of position-aware and structure-aware characteristics, we propose a set of probing tasks.

Tokens distance prediction. We conjecture that the distance between two tokens in a code snippet sequence, especially when absolute/relative position encoding methods are equipped in code transformer models, should be easily accessible and easily predicted. To investigate whether the code transformer encoders preserve such primary positional information, we train structure-induced Transformer models under different position encoding settings, and then we probe the models with a tokens distance prediction task.

Structure connection prediction. To understand whether the code structure features introduced into Transformer is well encoded in the embedding vectors, we use the code tokens representation from the encoder as input to predict the structural connection between them. To further test each single structure information, including abstract syntax tree, data-flow graph, control-flow graph and the combination of the three, we first train structure-induced Transformer models equipped with different structural adjacency matrices. After that, a probing task is performed on the code tokens embedding vectors from trained model encoders.

Probing data and labels. We select a subset from the test dataset of both Java and Python that never occurred in the training and validation dataset.

For the *Tokens distance prediction* task, we randomly generate token pairs in a code snippet sequence and label them according to their distances. In a binary classification setting, class bins are formulated as 1-16 and 17-32. If the distance between two tokens is in the range of 1 to 16, the probing task will treat this tokens-pair as a positive sample; otherwise negative sample. To further challenge the models, we split range 32 into four sub-classes, and labels are in 4 class bins (A:1-8, B:9-16, C:17-24 and D:25-32). The probing task is formulated as a classification problem with a binary setting and a 4-classes setting.

For the *Structure connection prediction* task, we generate the probing data depending on the three kinds of structure adjacency matrices and their combination. Code token pairs are randomly chosen from the sequence, and labels are the indication of whether there exists a specific structure connection between the two tokens. This probing task is formulated as a binary classification. To avoid the imbalanced data problem, we generate the same size of samples in all classes. Fig.3 illustrate the probing tasks: *tokens distance prediction* and *structure connection prediction*.

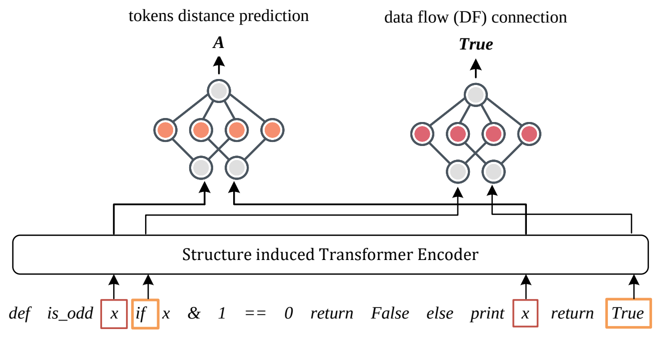
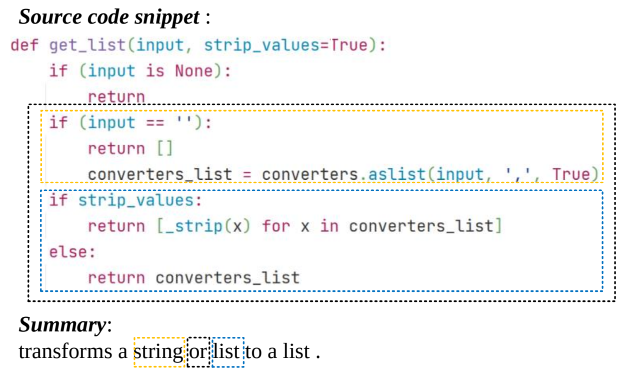

tokens distance prediction
A
data flow (DF) connection
True
Structure induced Transformer Encoder
def is_odd x if x & 1 == 0 return False else print x return True

Fig. 3. Illustration of the probing tasks (**tokens distance prediction** and **structure connection prediction**).

C. Structural-dependent summarization

To better evaluate how code Transformer models perform in utilizing structure information for code summarization task, we build a special code summarization test dataset, in which generating summaries requires an explicit structural understanding of source code. Specifically, if a code snippet contains *if-else statements* and there are natural language keywords that explicitly indicate the *if-else* structure keywords in its reference summary, such as ***if***, ***or*** and ***else***, we call this kind of $\langle code, summary \rangle$ cases as structural-dependent samples. We chose structural-dependent samples from original *Java/Python-test* splits and named them as *Java/Python-if*. We hypothesize that generating summaries for structural-dependent samples requires a more thorough understanding of source code structure information. Fig.4 shows an example of structural-dependent $\langle code, summary \rangle$ case.

Similarly, if a code snippet contains *for-loop/while-loop statements* and there are natural language keywords, such as ***from***, ***in***, in its gold summary, we also call this kind of $\langle code, summary \rangle$ cases that require explicit descriptions



Source code snippet :

```
def get_list(input, strip_values=True):
    if (input is None):
        return
    if (input == ''):
        return []
        converters_list = converters.aslist(input, ',', True)
    if strip_values:
        return [_strip(x) for x in converters_list]
    else:
        return converters_list
```

Summary:
transforms a string or list to a list .

Fig. 4. An Example of a structural-dependent $\langle code, summary \rangle$ case, in which we hypothesize that predicting *string*, *or* and *list* requiring the capability of structure-aware.

for structure understanding to summarize source code as structural-dependent samples. These samples are gathered from original *Java/Python-test* splits and named original *Java/Python-loop*. The statistics of each structural-dependent test dataset are shown in Table I.

IV. Experimental Results

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

RQ1: How do absolute and relative position encoding perform in code semantic summarization?

From Table II, we can observe from the tokens distance prediction accuracy that APE and RPE both contribute positively to predicting the distances between code tokens. The APE is learned based on the absolute indexes of each code token in the full input sequence. The RPE clipped the maximum relative position to k for the hypothesis that relative position information is not useful beyond a certain distance. Therefore, results in Table II show that the model equipped with APE alone outperforms RPE in this probing task.

TABLE II
Results of tokens distance prediction probing tasks, including binary (2-cls) and four classes (4-cls) classification in Java and Python datasets. Metrics: accuracy, the higher the better.

Model settings			Java		Python	
ASTs	APE	RPE	4-cls	2-cls	4-cls	2-cls
1	1	1	86.50	90.75	89.79	91.05
1	1	0	**88.09**	**93.02**	**92.09**	**93.07**
1	0	1	57.17	75.63	50.28	70.83
1	0	0	50.78	70.92	43.36	67.77
0	1	1	89.86	92.07	**92.85**	92.26
0	1	0	**89.92**	**92.77**	91.60	**93.24**
0	0	1	61.16	80.74	54.44	70.92
0	0	0	39.45	64.50	37.00	58.42

Probing results in Table II show that models trained with the combination of APE and RPE yield no further improvement, and there is an accuracy degradation in predicting tokens distance compared to models only equipped with APE. In most situations, models trained with APE alone achieve the best accuracy, and there occurs an accuracy drop when combing RPE with APE, especially when AST-based structure features are integrated. In the bottom of Table II, when AST-based features are not passed to Transformer, the decrease is not significant. There are two possible interpretations for this phenomenon. First, APE and RPE are two different kinds of position encoding methods, and APE, learned from the token index, could predict the distance between tokens well enough. The inaccurate position information in RPE due to the clipping mechanism decreases the classification accuracy. Second, the AST-based structure information may contain an inaccurate position relationship, which further harms the performance. In SIT [32], abstract syntax tree (***ast***), data-flow graph (***df***) and control flow graph (***cf***) are integrated into self-attention as an adjacency matrix, which stores the edge connections between tokens. The connections in ***ast*** graph and ***cf*** graph usually occur between tokens that relatively closer to each other, which may reflect rough closer distances between tokens. The location of assignment statements in the code snippet is less likely related to the position, and the tokens that correlated with ***df*** connections may present a different regularity in the distance. For this reason, connections in AST-based features may indicate inaccurate position information. Therefore, if we compare the last row of the top half and the last row of the bottom half in Table II, we can observe that AST-based structure features can also bring improvements in predicting the tokens distance probing task, although not much compared to APE or RPE.

According to Table II, we could partially answer ***RQ1*** that ***applying the combination of APE and RPE in the Transformer would harm capturing positional information.***. Naturally, it is necessary to figure out whether this finding is consistent in code semantic summarization tasks. Therefore, we perform an ablation study to investigate the interaction of APE and RPE in code summarization task and code extreme summarization task. The results are shown in Table III and Table IV.

Experiment results in Table III show that the Transformer equipped with RPE alone achieve the best performance compared to other configurations in code summarization task in both Python and Java test dataset, regardless of whether AST-based structure features are equipped. In addition, compared to only utilizing RPE, ***Transformer model will suffer a significant summarization performance degradation when equipped with the combination of APE and RPE***. Moreover, in code extreme summarization task, results in Table IV also illustrate the same observation. Utilizing RPE only in the Transformer could achieve the best extreme summarization performance, and the combination of APE and RPE will yield a notable performance degradation.

This empirical finding in the summarization task is consistent with the result of the tokens distance prediction probing task mentioned above, which also corroborates the design choice of [23]. Now we could answer ***RQ1*** that the APE and RPE do not coexist well in Transformer for code semantic summarization tasks.

TABLE III
ABLATION STUDY ON APE AND RPE ON JAVA AND PYTHON CODE SUMMARIZATION BENCHMARKS. METRICS: BLEU, ROUGE-L, AND METROR, THE HIGHER THE BETTER.

Model settings			Java			Python		
ASTs	APE	RPE	BLEU	ROUGE-L	METEOR	BLEU	ROUGE-L	METEOR
1	1	1	45.46(↓1.18)	55.58	27.57	32.88(↓1.46)	46.90	19.93
1	1	0	45.67	55.79	27.48	33.30	47.37	20.26
1	0	1	**46.64**	**56.61**	28.29	**34.34**	**48.24**	**20.91**
1	0	0	46.58	56.40	**28.31**	34.24	48.09	20.87
0	1	1	45.86(↓0.78)	55.83	27.58	33.13(↓1.02)	47.19	20.05
0	1	0	45.41	55.58	27.46	32.71	46.73	19.84
0	0	1	**46.64**	**56.74**	**28.25**	**34.15**	**48.19**	**20.85**
0	0	0	45.93	55.03	27.77	33.09	46.40	19.91

TABLE IV
RESULTS OF CODE EXTREME SUMMARIZATION ON JAVA-SMALL DATASET. METRICS: PRECISION, RECALL, AND F1 SCORE.

Model settings			Java-small		
ASTs	APE	RPE	Precision	Recall	F1
1	1	1	44.83	37.60	40.90(↓0.90)
1	1	0	47.01	37.52	41.73
1	0	1	46.48	37.98	**41.80**
1	0	0	46.42	36.96	41.15
0	1	1	44.92	35.91	39.91(↓1.09)
0	1	0	44.45	37.10	40.44
0	0	1	45.21	37.51	**41.00**
0	0	0	41.49	31.82	36.02

Finding 1. There is a conflict influence when applying the combination of APE and RPE in Transformer for encoding positional information, and the combination of them interferes with both the understanding of position information and the code summaries generation.

Before moving to ***RQ2***, we notice an interesting phenomenon. If we focus on the last row of the top half and the penultimate row of the bottom half in Table III, the model equipped with AST-based structure features and the model trained with RPE show closely similar performance for code summarization task in both Java and Python test dataset. Similarly, the results of code extreme summarization in Table IV demonstrate the same observation that performance gain from RPE is competitive to the contribution from ASTs features. This finding will be discussed in ***RQ2***.

RQ2: To what extent do AST-based structure features and positional information contribute to code semantic summarization?

In RQ1, we find that the combination of APE and RPE shows a $1 + 1 < 2$ consequence in understanding positional information as well as summarization tasks. To better investigate how Transformer performs in preserving AST-based code structure features and how the positional features and ASTs features interact, we design the structure connection prediction probing task.

In the left part of Table V, Transformer models equipped with single specific AST-based structure feature, such as ***ast***, could achieve the highest accuracy in abstract syntax tree graph (AST) connection prediction task, and the observations are the same for ***cf*** and ***df*** in control-flow graph (CF) and data-flow graph (DF) connections prediction tasks. We could claim that Transformer models are able to capture AST-based structure information introduced from specific adjacency matrices.

In the middle part of Table V, Transformer model trained with ***df*** is expected to achieve the best accuracy in DF connection prediction, and models trained with the combination of ***ast***, ***df*** and ***cf*** should have achieved the best accuracy in ADJ connection prediction. Whereas, this expectation trend in the left part of Table V is disturbed with the APE integrating to encode position information with RPE. Moreover, in the right part of Table V, we cannot observe the same trend either. We blame this inconsistency on the conflicts between APE and ASTs features in Transformer.

As we discussed in ***RQ1***, there is a conflict between APE and RPE in Transformer for capturing positional information, and APE would harm code summarization tasks. Moreover, applying APE in Transformer also has a negative effect on understanding AST-based structure information. Therefore, we only include RPE in our following experiments.

To evaluate the contribution of AST-based code structure features and positional structure information for code intelligent summarization, Table VI and Table VII show the ablation study results on models trained with different settings of RPE and ASTs features. In code summarization task, Transformer model equipped with relative position encoding (RPE) and the combination of multi-view graph of code structural features (***ast***, ***cf*** and ***df***) achieves best evaluation scores in Python dataset, and the model with the setting of RPE and ***ast*** achieves the best BLEU scores in Java dataset. Results in Table VII show that model equipped with RPE and ***ast*** perform best for code extreme summarization task in Java-small dataset. In addition, we observe that the difference between applying a multi-view graph and only ***ast*** graph in Transformer is marginal. We claim that AST-based structure information does contribute to summarization tasks, and the main contribution is attributed to ***ast***. Meanwhile, Transformer models trained solely with ***cf*** or ***df*** has a worse performance than models equipped with ***ast*** or the combination of all threes in both two code summarization tasks. A possible reason is that the ***cf*** and ***df*** adjacency matrices are much sparser than ***ast***, which may filter out many useful structure connections between tokens.

Furthermore, from Table VI, in the code summarization task, RPE alone contributes 0.71 BLEU, 1.71 ROUGE and

TABLE V
RESULTS OF STRUCTURE CONNECTION PREDICTION PROBING TASKS ON PYTHON DATASET, INCLUDING DATA-FLOW (DF), CONTROL-FLOW (CF), ABSTRACT SYNTAX TREE (AST) AND COMBINATION OF THE THREE (ADJ).

Model settings			with RPE				with APE and RPE				with APE			
-ast	***-cf***	***-df***	DF	CF	AST	ADJ	DF	CF	AST	ADJ	DF	CF	AST	ADJ
1	1	1	79.88	80.07	83.40	**72.7**	**83.21**	83.14	83.52	74.89	79.89	79.04	**82.99**	71.43
1	0	0	77.00	75.82	**85.18**	71.05	80.44	83.71	**87.81**	**76.69**	**81.03**	83.37	81.95	**76.49**
0	1	0	77.12	**86.46**	77.92	69.87	78.46	**86.56**	74.27	68.64	77.92	**84.69**	77.61	71.44
0	0	1	**81.81**	71.20	78.78	67.80	81.25	68.53	70.03	62.86	80.20	69.82	76.05	67.60
0	0	0	77.17	70.30	78.05	66.52	78.90	72.81	77.78	68.46	75.97	71.64	77.56	68.29

TABLE VI
ABLATION STUDY ON RPE AND ASTS FEATURES ON JAVA AND PYTHON CODE SUMMARIZATION BENCHMARKS. METRICES: BLEU, ROUGE-L AND METROR, THE HIGHER THE BETTER.

Model settings				Java			Python		
RPE	***ast***	***cf***	***df***	BLEU	ROUGE-L	METEOR	BLEU	ROUGE-L	METEOR
1	1	1	1	46.64	56.61	28.29	**34.34**	**48.24**	**20.91**
1	1	0	0	**46.75**	56.50	**28.29**	34.27	48.07	20.76
1	0	1	0	46.63	56.46	28.27	33.81	47.71	20.49
1	0	0	1	46.42	56.27	28.12	33.70	47.39	20.42
1	0	0	0	46.64(↑0.71)	**56.74**(↑1.71)	28.25(↑0.48)	34.15(↑1.06)	48.19(↑1.79)	20.85(↑0.94)
0	1	1	1	46.58(↑0.65)	56.40(↑1.37)	28.21(↑0.44)	34.24(↑1.15)	48.09(↑1.69)	20.87(↑0.96)
0	0	0	0	45.93	55.03	27.77	33.09	46.40	19.91

TABLE VII
ABLATION STUDY ON RPE AND ASTS FEATURES ON JAVA-SMALL CODE EXTREME SUMMARIZATION DATASET. METRICES: PRECISION RECALL AND F1 SCORE.

Model settings				Java-small		
RPE	***ast***	***cf***	***df***	Precision	Recall	F1
1	1	1	1	46.48	37.98	41.80
1	1	0	0	**47.36**	**38.29**	**42.34**
1	0	1	0	44.20	33.65	38.21
1	0	0	1	44.73	35.45	39.55
1	0	0	0	45.21(↑3.72)	37.51(↑5.69)	41.00(↑4.98)
0	1	1	1	46.42(↑4.93)	36.96(↑5.14)	41.15(↑5.13)
0	0	0	0	41.49	31.82	36.02

0.48 METEOR scores in Java and 1.06 BLEU, 1.79 ROUGE and 0.94 METEOR in Python benchmarks. The improvements contributed from the ASTs features are 0.65 BLEU, 1.37 ROUGE, 0.44 METEOR in Java and 1.15 BLEU, 1.69 ROUGE, 0.96 METEOR in Python. In code extreme task, results in Table VII show that RPE contributes 3.72 Precision, 5.69 Recall and 4.98 F1 scores, and ASTs features contribute 4.93 Precision, 5.14 Recall and 5.13 F1 scores respectively. Both RPE and ASTs features could solely achieve a noticeable improvement in code summarization and extreme summarization tasks. Nevertheless, after RPE was introduced into the Transformer model, the improvements contributed from the ASTs features are very limited, which are 0.00 BLEU, -0.13 ROUGE, 0.04 METEOR in Java, 0.19 BLEU, 0.05 ROUGE-L,0.06 METEOR in Python and 1.27 Precision, 0.47 Recall and 0.80 F1 scores in Java-small datasets, and vice versa.

We hypothesize that the contributions of RPE and ASTs feature to code semantic summarization overlap to a large extent. To validate our hypothesis, we propose to establish the correlation between RPE and ASTs by using *Pearson* correlation coefficient. The main evaluation metric, BLEU score distributions of models equipped with different feature settings, are utilized to calculate *Pearson* correlations of ASTs and RPE, and the results are displayed in Figure 5. Our interpretation of *Pearson* r correlation is based on Hinkle et al.'s scheme [37]: negligible correlation ($|r| < 0.3$), low correlation ($0.3 \leq |r| < 0.5$), moderate correlation ($0.5 \leq |r| < 0.7$), high correlation ($0.7 \leq |r| < 0.9$), and very high correlation ($0.9 \leq |r| \leq 1$). Figure 5 shows that there exists high *Pearson* correlations between ASTs and RPE in three datasets. The combination feature ASTs&RPE also shares high correlations with ASTs and RPE. Although the absolute *Pearson* correlation values are moderately high, the NONE feature has a relatively low correlation with RPE and ASTs, which shows a notable gap from the correlations between ASTs and RPE features. As we can observe from the left two sub-figures in Figure 5, the moderately high correlations between the NONE feature and ASTs and RPE features are more significant in code summarization task since the basic Transformer model without any structure feature could achieve competitive performance in Java and Python datasets.

Now, we could answer ***RQ2*** that code structure features contribute positively to code semantic summarization, and RPE, together with ASTs code structure features, could achieve the best performance in Transformer architecture. In addition, our experimental results show an insightful finding that the contribution of RPE and ASTs structure information to code summarization tasks overlap to a large extent since they have a strong *Pearson* correlation.

Finding 2. The ASTs code structural features and relative position encoding feature contribute positively to code semantic summarization tasks, but they have a strong Pearson correlation, and much feature overlap exists between them.

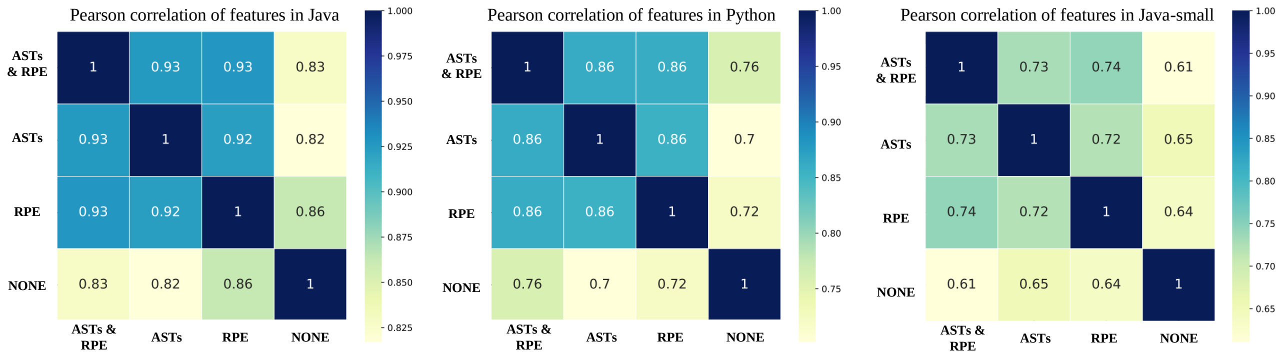


Fig. 5. Pearson correlations of RPE and ASTs features. The left two sub-figures are evaluated in code summarization task, and the last one is in extreme summarization task. **ASTs**&**RPE** is the combination of **ASTs** and **RPE**, and the **NONE** means both are not included.

TABLE VIII
RESULTS ON STRUCTURAL-DEPENDENT SUMMARIZATION DATASETS.

Model settings				*Python-if*			*Python-loop*			*Java-if*			*Java-loop*		
RPE	***ast***	***cf***	***df***	BLEU	ROUGE-L	METEOR	BLEU	ROUGE-L	METEOR	BLEU-4	ROUGE-L	METEOR	BLEU	ROUGE-L	METEOR
1	1	1	1	**30.00**	**45.26**	20.18	26.98	41.80	17.75	38.51	49.01	24.16	40.92	50.80	22.49
1	1	0	0	29.56	44.93	19.88	**27.31**	**42.05**	**17.85**	38.74	**49.19**	**24.34**	40.52	50.46	22.35
1	0	1	0	29.70	45.23	20.04	26.58	41.14	17.30	38.24	48.55	24.02	39.82	49.88	22.11
1	0	0	1	29.38	44.24	19.53	26.75	41.38	17.52	38.42	48.82	24.05	39.99	49.89	22.16
1	0	0	0	29.64	44.97	20.08	26.95	41.73	17.60	**38.83**	49.12	24.24	**41.58**	**51.83**	**23.15**
0	0	0	0	28.61	43.63	19.16	25.83	39.90	16.66	36.99	47.52	23.23	38.55	48.70	20.96

RQ3: How does Transformer perform in a more structural dependent setting in code summarization task?

To better evaluate the ability of code Transformer models that convert the understanding of code structure information into natural language code summary. We gather a special code summarization test dataset, in which generating a summary requires a more explicit structural understanding of source code. Experimental results of the Transformer equipped with various structural feature settings on the structural-dependent summarization test dataset are displayed in Table.VIII.

We can observe that in *Python-if*, the model equipped with RPE and the combination of all ASTs structure features achieves the highest evaluation metric scores. In *Python-loop*, RPE together with ***ast*** performs best. In both *Java-if* and *Java-loop*, Transformer equipped with RPE alone achieves best BLEU scores. Transformer models equipped with various code structure settings achieve best performance in different structural-dependent summarization benchmarks, which may suggest that it's not the perfect choice to treat code structure features (***ast***, ***cf*** and ***df***) equally in one adjacency matrix.

Structural-dependent datasets are selected from the original *Java* and *Python* test datasets following the rules we stated in Section III. It's obvious to see that overall evaluation metrics decrease a large margin, about 8 BLEU scores drop in Java-if, 5 BLEU scores in Java-loop, 4 BLEU scores in Python-if, and 7 BLEU scores in Python-loop, compared to the best performances evaluated on the whole benchmarks in *Java/Python-test* shown in Table VI. We blame the performance gap for two reasons. First, according to the statistics of datasets in Table I, both the average tokens of *Java/Python-if/loop* in code and in summary are much longer. The average length summary in *Java-if* is almost twice as long as in *Java-test*. Second, we assume the selected Structural-dependent test datasets are challenging for Transformer that integrated with code structure features. To eliminate the effects of the code length and summary length, we further sample *Java/Python-if/loop* to match the length distribution of *Java/Python-test*. The structural-dependent datasets that match the length distribution of *Java/Python-test* are noted as ***Java/Python-struc-dpdt***, and the statistics are listed in Table I. We re-evaluate code Transformer models with various structural feature settings on *Java/Python-struc-dpdt* summarization datasets, and the experimental results are demonstrated in Table IX. Compared to the performance decreasing margin in *Java/Python-if/loop*, although smaller, we also observe 1.46 BLEU scores decreased in *Java-struc-dpdt* and 0.98 BLEU scores decreased in *Python-struc-dpdt* compared to the best performances in Table VI. This observation indicates that it is still challenging for structure-induced Transformer models to convert their implicit understanding of code structure information into explicit natural language summary.

On the other hand, the *Structural-dependent summarization* test datasets offer a more straightforward and challenging method to evaluate the capability of converting code structural information into explicit natural language summary, which may inspire the evaluation of code summarization. We could answer ***RQ3*** that in a more structural-dependent setting, the Transformer model equipped with code structure features shows a significant performance decrease.

Finding 3. In a more structural-dependent summarization scenario, explicitly understanding code structure information in Transformer still has space for further improvement.

TABLE IX
RESULTS ON *Java/Python-struc-dept* SUMMARIZATION DATASET.

Model settings				*Java-struc-dpdt*			*Python-struc-dpdt*		
RPE	***ast***	***cf***	***df***	BLEU-4	ROUGE-L	METEOR	BLEU-4	ROUGE-L	METEOR
1	1	1	1	44.45	55.19	31.32	**33.36**	**48.77**	**22.85**
1	1	0	0	44.54	55.31	31.25	32.69	48.05	22.12
1	0	1	0	44.07	54.61	31.03	32.55	48.31	22.04
1	0	0	1	43.97	54.86	30.93	32.21	47.36	21.63
1	0	0	0	**45.29**	**55.85**	**31.88**	32.25	47.46	22.21
0	0	0	0	43.02	53.69	30.10	31.74	47.72	22.20

V. THREATS TO VALIDITY

One potential threat to the validity of our work is the design of probing tasks. Although we obtain the consistent conclusion that APE and RPE yield a conflict influence in both the tokens distance probing task and summarization task, there is still an inconsistency phenomenon. The embedding vectors from the Transformer encoder equipped with APE could achieve the best performance in the tokens distance prediction probing task. However, RPE is proved to be more beneficial in the Transformer for code semantic summarization tasks. The absolute accuracy values in the probing task are not the focus of this study; instead, the probe is utilized to assess whether APE or RPE helps in encoding position information and the effect of using the combination of them. In our future work, we will try to explore more suitable probing tasks that not only could reflect specific features but keep consistent with the target downstream task. Concerning the input embedding vectors of the probing task, our limitation is that only the last layer outputs of the Transformer encoder are considered. It would be interesting to explore the capability of preserving specific properties in various layers in the Transformer encoder. Besides, in this paper, we only consider the method utilizing code structure features in Transformer proposed in [32]. Therefore, our findings and suggestions are suitable for scenarios that integrate code structure features into Transformer as an inductive bias. It will be more revealing to extend this study to more methods that utilize code structure information in Transformer.

VI. RELATED WORKS

A. Code semantic summarization.

Recently, in the code summarization task, researchers have focused on encoding source code structural information, such as positional relationships and the structure feature from ASTs. Ahmad et al. [23] adopted relative position representation [25] to model the pairwise code token relationship by injecting relative position embeddings into Transformer. Along this technique track, Wu et al. [32] propose a structure-induced Transformer (SIT) to integrate ASTs structure features into the self-attention calculation, which combines the AST tree, data-flow graph, and control-flow graph as a multi-view graph matrix to filter attention connections between tokens. Following SIT, Gao et al. [27] proposed to introduce the AST relative position relationship between tokens into the Transformer to further enhance the capturing of code structure information.

B. Empirical study on source code models

In the natural language processing (NLP) community, the research works that attempt to demonstrate what BERT [38] learn and how it affects downstream tasks by analyzing attention and task probing have formed a "BERTology" [39] subspecialty. The counterpart research in software engineering mostly focused on the pre-trained language models for source code. For example, Karmakar and Robbes [40], and Troshin and Chirkova [26] introduced a set of heuristic diagnostic probing tasks to test whether or to what extent vectors from source code pre-trained models reflect diverse aspects of code understanding. Wan et al. [8] also conducted a structural analysis on pre-trained code models. In [40] and [26], researchers focused on pre-trained models, such as CodeBERT [41], GraphCodeBERT [35] and PLBART [42] and CodeT5 [43]. However, these studies only show in which probing tasks a code embedding technique works better, i.e., in capturing which kind of features a model performs better. It is still not clear how these features affect code intelligence tasks. Chirkova and Troshin [26] conducted an empirical study to investigate what is the best way of utilizing syntactic structure information in different scenarios while lacking analysis on code summarization. Sontakke et al. [44] performed an empirical study to analyze the state-of-the-art summarization models, to which extent these models understand the code they attempt to summarize. Insightful observations on code summarization are found, and their study granularity is on the model level.

VII. CONCLUSION

In this paper, we conduct an empirical study concerning the code structure features in Transformer-based code semantic summarization tasks. Through extensive experiments, several insightful findings are found. (1) There is a conflict between the influence of the APE and the RPE in Transformer, and we suggest that utilizing only RPE in code summarization tasks yields better performance. (2) The AST-based code structure features and RPE feature show a strong correlation, and much contribution overlap for code semantic summarization tasks do exist between them. (3) In a more structural-dependent setting, explicitly understanding code structure information in Transformer still has space for further improvement. These findings revealed in this paper could inspire future studies on source code representation in Transformer. All data in the study are publicly available at: https://drive.google.com/file/d/1cbdaxsVEC_tzZtfJ0yE-jNJeI2rwYs2J/view?usp=share_link

References

[1] M. T. R. Laskar, E. Hoque, and J. X. Huang, "Domain adaptation with pre-trained transformers for query-focused abstractive text summarization," *Computational Linguistics*, vol. 48, pp. 279–320, June 2022.

[2] Y. Lu, J. Zeng, J. Zhang, S. Wu, and M. Li, "Learning confidence for transformer-based neural machine translation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Dublin, Ireland), pp. 2353–2364, Association for Computational Linguistics, May 2022.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *In Advances in Neural Information Processing Systems 30*, p. 5998–6008, 2017.

[4] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International conference on machine learning*, pp. 2091–2100, PMLR, 2016.

[5] Y. Choi, J. Bak, C. Na, and J.-H. Lee, "Learning sequential and structural information for source code summarization," in *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pp. 2842–2851, 2021.

[6] X. HU, G. LI, X. XIA, D. LO, S. LU, and Z. JIN, "Summarizing source code with transferred api knowledge.(2018)," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelli-gence (IJCAI 2018), Stockholm, Sweden, 2018 July 13*, vol. 19, pp. 2269–2275.

[7] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th international conference on program comprehension*, pp. 184–195, 2020.

[8] Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu, and H. Jin, "What do they capture?–a structural analysis of pre-trained language models for source code," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pp. 2377–2388, 2022.

[9] L. Jiang, H. Liu, and H. Jiang, "Machine learning based recommendation of method names: how far are we," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 602–614, IEEE, 2019.

[10] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pp. 397–407, 2018.

[11] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, *et al.*, "Competition-level code generation with alphacode," *arXiv preprint arXiv:2203.07814*, 2022.

[12] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 473–485, 2020.

[13] V. J. Hellendoorn, C. Sutton, R. Singh, P. Maniatis, and D. Bieber, "Global relational models of source code," in *International Conference on Learning Representations*, 2020.

[14] U. Z. Ahmed, P. Kumar, A. Karkare, P. Kar, and S. Gulwani, "Compilation error repair: For the student programs, from the student programs," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pp. 78–87, 2018.

[15] B. Roziere, M.-A. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *Advances in Neural Information Processing Systems*, vol. 33, pp. 20601–20611, 2020.

[16] V. Shiv and C. Quirk, "Novel positional encodings to enable tree-based transformers," *Advances in neural information processing systems*, vol. 32, 2019.

[17] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang, "Tbcnn: A tree-based convolutional neural network for programming language processing," *arXiv preprint arXiv:1409.5718*, 2014.

[18] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *International Conference on Learning Representations*, 2019.

[19] B. Lin, S. Wang, Z. Liu, X. Xia, and X. Mao, "Predictive comment updating with heuristics and ast-path-based neural learning: A two-phase approach," *IEEE Transactions on Software Engineering*, pp. 1–20, 2022.

[20] C. Niu, C. Li, V. Ng, J. Ge, L. Huang, and B. Luo, "Spt-code: sequence-to-sequence pre-training for learning source code representations," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 2006–2018, 2022.

[21] B. Lin, S. Wang, M. Wen, and X. Mao, "Context-aware code change embedding for better patch correctness assessment," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, may 2022.

[22] N. Chirkova and S. Troshin, "Empirical study of transformers for source code," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 703–715, 2021.

[23] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 4998–5007, 2020.

[24] C. B. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, and N. Sundaresan, "Pymt5: multi-mode translation of natural language and python code with transformers," *arXiv preprint arXiv:2010.03150*, 2020.

[25] P. Shaw, J. Uszkoreit, and A. Vaswani, "Self-attention with relative position representations," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pp. 464–468, 2018.

[26] S. Troshin and N. Chirkova, "Probing pretrained models of source code," *arXiv preprint arXiv:2202.08975*, 2022.

[27] S. Gao, C. Gao, Y. He, J. Zeng, L. Y. Nie, and X. Xia, "Code structure guided transformer for source code summarization," *arXiv preprint arXiv:2104.09340*, 2021.

[28] H. Peng, G. Li, W. Wang, Y. Zhao, and Z. Jin, "Integrating tree path in transformer for code representation," *Advances in Neural Information Processing Systems*, vol. 34, pp. 9343–9354, 2021.

[29] S. Wang, M. Wen, B. Lin, and X. Mao, "Lightweight global and local contexts guided method name recommendation with prior knowledge," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 741–753, 2021.

[30] Q. Chen, X. Xia, H. Hu, D. Lo, and S. Li, "Why my code summarization model does not work: Code comment improvement with category prediction," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, feb 2021.

[31] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of The 33rd International Conference on Machine Learning* (M. F. Balcan and K. Q. Weinberger, eds.), vol. 48 of *Proceedings of Machine Learning Research*, (New York, New York, USA), pp. 2091–2100, PMLR, 20–22 Jun 2016.

[32] H. Wu, H. Zhao, and M. Zhang, "Code summarization with structure-induced transformer," in *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pp. 1078–1090, 2021.

[33] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[34] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pp. 200–20010, IEEE, 2018.

[35] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *International Conference on Learning Representations*, 2021.

[36] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *ICLR (Poster)*, 2015.

[37] D. E. Hinkle, *Applied Statistics For The Behavioral Sciences*. Houghton Mifflin Company, 2002.

[38] J. D. M.-W. C. Kenton and L. K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of NAACL-HLT*, pp. 4171–4186, 2019.

[39] A. Rogers, O. Kovaleva, and A. Rumshisky, "A primer in bertology: What we know about how bert works," *Transactions of the Association for Computational Linguistics*, vol. 8, pp. 842–866, 2020.

[40] A. Karmakar and R. Robbes, "What do pre-trained code models know about code?," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1332–1336, IEEE, 2021.

[41] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, 2020.

[42] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Associ-*

ation for Computational Linguistics: Human Language Technologies, pp. 2655–2668, 2021.

[43] Y.-A. Wang and Y.-N. Chen, "What do position embeddings learn? an empirical study of pre-trained language model positional encoding," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 6840–6849, 2020.

[44] A. N. Sontakke, M. Patwardhan, L. Vig, R. K. Medicherla, R. Naik, and G. Shroff, "Code summarization: Do transformers really understand code?," in *Deep Learning for Code Workshop*, 2022.