Code Structure–Guided Transformer for Source Code Summarization

SHUZHENG GAO and CUIYUN GAO, Harbin Institute of Technology, Shenzhen, China
YULAN HE, University of Warwick, UK
JICHUAN ZENG, The Chinese University of Hong Kong, Hong Kong, China
LUNYIU NIE, Tsinghua University, China
XIN XIA, Software Engineering Application Technology Lab, Huawei, China
MICHAEL LYU, The Chinese University of Hong Kong, Hong Kong, China

Code summaries help developers comprehend programs and reduce their time to infer the program functionalities during software maintenance. Recent efforts resort to deep learning techniques such as sequence-to-sequence models for generating accurate code summaries, among which Transformer-based approaches have achieved promising performance. However, effectively integrating the code structure information into the Transformer is under-explored in this task domain. In this article, we propose a novel approach named SG-Trans to incorporate code structural properties into Transformer. Specifically, we inject the local symbolic information (e.g., code tokens and statements) and global syntactic structure (e.g., dataflow graph) into the self-attention module of Transformer as inductive bias. To further capture the hierarchical characteristics of code, the local information and global structure are designed to distribute in the attention heads of lower layers and high layers of Transformer. Extensive evaluation shows the superior performance of SG-Trans over the state-of-the-art approaches. Compared with the best-performing baseline, SG-Trans still improves 1.4% and 2.0% on two benchmark datasets, respectively, in terms of METEOR score, a metric widely used for measuring generation quality.

CCS Concepts: • **Software and its engineering → Software creation and management**; **Software development techniques;**

Additional Key Words and Phrases: Code summary, Transformer, multi-head attention, code structure

This research was supported by the National Natural Science Foundation of China under grant no. 62002084, the Stable Support Plan for Colleges and Universities in Shenzhen under grant no. GXWD20201230155427003-20200730101839009, and the Research Grants Council of the Hong Kong Special Administrative Region, China (grant no. CUHK 14210920 of the General Research Fund). This research was also partly funded by the UK Engineering and Physical Sciences Research Council (grant no. EP/V048597/1, EP/T017112/1). Yulan He is supported by a Turing AI Fellowship funded by the UK Research and Innovation (grant no. EP/V020579/1).

Authors' address: S. Gao and C. Gao (corresponding author), Harbin Institute of Technology, Guangdong Province, 518055, China; email: szgao98@gmail.com; Y. He, University of Warwick, Coventry, Warwickshire, CV4 7AL, China; email: yulan.he@warwick.ac.uk; J. Zeng and M. Lyu, The Chinese University of Hong Kong, Hong Kong, SAR, 999077, China; emails: jczeng@cse.cuhk.edu.hk, lyu@cse.cuhk.edu.hk; L. Nie, Tsinghua University, 30 ShuangQing Street, Beijing, 100084; email: nlx20@mails.tsinghua.edu.cn; X. Xia, Software Engineering Application Technology Lab, Huawei Base Bantian, ShenZhen City, Guangdong Province, 518129, China; email: xin.xia@acm.org.

1049-331X/2023/02-ART23 $15.00
https://doi.org/10.1145/3522674

ACM Reference format:
Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lunyiu Nie, Xin Xia, and Michael Lyu. 2023. Code Structure–Guided Transformer for Source Code Summarization. *ACM Trans. Softw. Eng. Methodol.* 32, 1, Article 23 (February 2023), 32 pages.
https://doi.org/10.1145/3522674

1 INTRODUCTION

Program comprehension is crucial for developers during software development and maintenance. However, existing studies [42, 64] have shown that program comprehension is a very time-consuming activity that occupies over 50% of the total time in software maintenance. To alleviate the developers' cognitive efforts in comprehending programs, a text summary accompanying the source code has been proved to be useful [9, 17, 27]. However, human-provided comments are often incomplete or outdated because of the huge effort needed and the rapid update of software [13, 49]. The source code summarization task aims at automatically generating a concise comment on a program. Many studies [11, 39, 53, 61] have demonstrated that machine-generated summaries are helpful for code comprehension. A recent empirical study [29] also shows that 80% of practitioners believe code summarization tools can help them improve development efficiency and productivity.

Existing leading approaches have demonstrated the benefits of integrating code structural properties such as Abstract Syntax Trees (ASTs) [4, 27] into deep learning techniques for the task. An example of an AST is shown in Figure 1(b). The modality of the code structure can be either sequences of tokens traversed from the syntactic structure of ASTs [4, 27] or sequences of small statement trees split from large ASTs [50, 68]. The sequences are usually fed into a Recurrent Neural Network (RNN)–based sequence-to-sequence network for generating a natural language summary [27, 36]. However, due to the deep nature of ASTs, the associated RNN-based models may fail to capture the long-range dependencies between code tokens [1]. To mitigate this issue, some works represent the code structure as graphs and adopt Graph Neural Networks (GNNs) for summary generation [16, 35]. Although these GNN-based approaches can capture the long-range relations between code tokens, they are shown sensitive to local information and ineffective in capturing the global structure [30]. Taking the AST in Figure 1(b) as an example, token nodes "*int*" and "*num*" (highlighted with red boxes) are in the same statement but separated by five hops; thus, GNN-based approaches tend to ignore the relations between the two token nodes. In addition, the message passing on GNNs is limited by the predefined graph, reducing its scalability to learn other dependency relations.

A recent study [1] shows that the Transformer model [55] outperforms other deep learning approaches for the task. The self-attention mechanism in Transformer can be viewed as a fully connected graph [22], which can ensure the long-range message passing between tokens and the flexibility to learn any dependency relation from data. However, it is hard for the Transformer to learn all important dependency relations from limited training data. In addition, an issue of Transformer is that its attention is purely data driven [23]. Without the incorporation of explicit constraints, the multi-head attentions in Transformer may suffer from attention collapse or attention redundancy, with different attention heads extracting similar attention features, which hinders the model's representation learning ability [5, 56]. To solve these problems, we incorporate code structure into the Transformer as prior information to eliminate its dependency on data. However, how to effectively integrate the code structure information into Transformer is still under-explored. One major challenge is that since the position encoding in the Transformer already learns the dependency relations between code tokens, trivial integration of the structure information may not improve performance of the task [1].

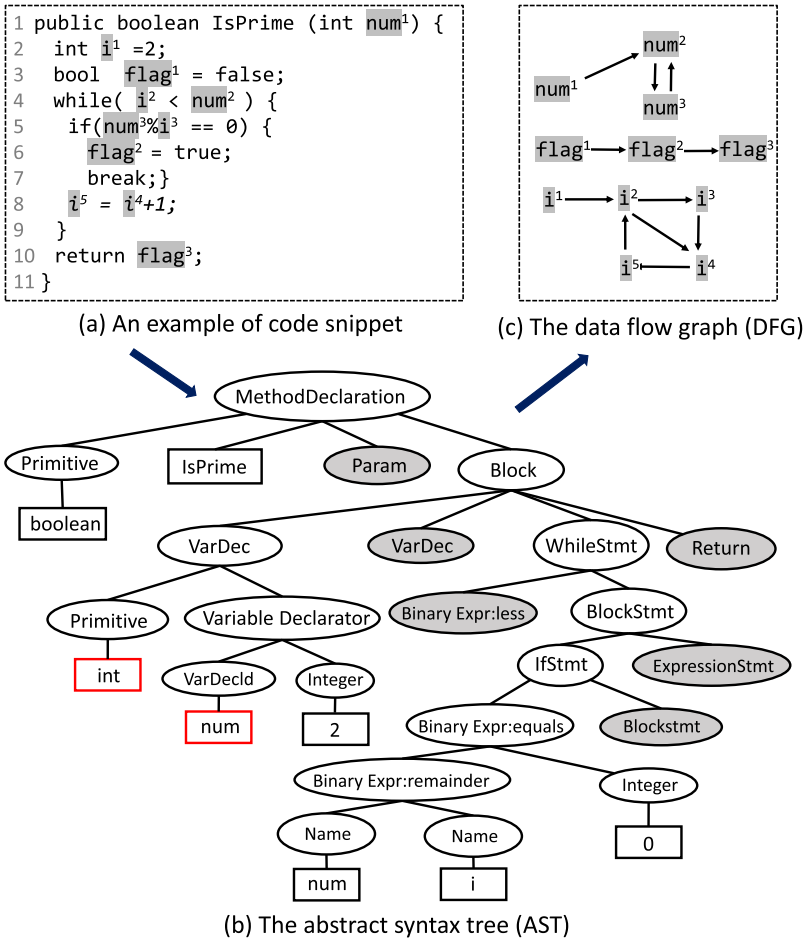


Fig. 1. An example of Java code snippet (a), with the corresponding AST (b) and DFG (c) illustrated. Entities in gray ellipse in (b) mean unexpanded branches. The arrows in the DFG represent the relations of sending/receiving messages between the variables (highlighted in gray in the code).

To overcome the challenges in this article, we propose a novel model named SG-Trans, that is, code **S**tructure **G**uided **Trans**former. SG-Trans exploits the code structural properties to introduce explicit constraints to the multi-head self-attention module. Specifically, we extract the pairwise relations between code tokens based on the local symbolic structure, such as code tokens and statements, and the global syntactic structure, that is, dataflow graph (DFG), then represent them as adjacency matrices before injecting them into the multi-head attention mechanism as inductive bias. Furthermore, following the principle of compositionality in language: the high-level semantics is the composition of low-level terms [23, 54]. We propose a hierarchical structure-variant attention approach to guide the attention heads at the lower layers attending more to the local structure and those at the higher layers attending more to the global structure. In this way, our model can take advantage of both local and global (long-range dependencies) information of source code. Experiments on benchmark datasets demonstrate that SG-Trans can outperform the state-of-the-art models by at least 1.4% and 2.0% on two Java and Python benchmark datasets, respectively, in terms of METEOR.

In summary, our work makes the following contributions:

- We are the first to explore the integration of both local and global code structural properties into Transformer for source code summarization.
- A novel model is proposed to hierarchically incorporate both the local and global structure of code into the multi-head attentions in Transformer as inductive bias.
- Extensive experiments show that SG-Trans outperforms the state-of-the-art models. We have publicly released the replication repository — including source code, datasets, prediction logs, online questionnaire, and results of human evaluation — on GitHub.[1]

Article Structure. Section 2 provides the background knowledge of the work. Section 3 contains our proposed methodology for source code summarization. Section 4 introduces the experimental setup. Section 5 describes the evaluation results, followed by discussions in Section 6. Section 7 discusses related work. We present our conclusions and plans for future work in Section 8.

2 BACKGROUND

In this section, we introduce the background knowledge of the proposed approach, including the vanilla Transformer model architecture and the copy mechanism.

2.1 Vanilla Transformer

Transformer [55] is a kind of deep self-attention network that has demonstrated its powerful text representation capability in many NLP applications, for example, machine translation and dialogue generation [51, 69]. Recently, a lot of research in code summarization also leverages Transformer as the backbone for better source code representations [1, 15]. Some work [26, 58, 70] also improves the Transformer to make it better adapt to source code or structured data. Unlike conventional neural networks, such as Convolutional Neural Network (CNNs) and Recurrent Neural Network (RNNs), it is solely based on attention mechanism and multi-layer perceptrons (MLPs). Transformer follows the sequence-to-sequence [10] architecture with stacked encoders and decoders. Each encoder block and decoder block consists of a multi-head self-attention sub-layer and a feed-forward sub-layer. Residual connection [25] and layer normalization [6] are also employed between the sub-layers. Since the two sub-layers play an essential role in Transformer, we introduce them in more detail next.

2.1.1 Multi-Head Self-Attention. Multi-head attention is the key component of Transformer. Given an input sequence $X = (x_1, x_2, \ldots, x_i, \ldots, x_n)$ where n is the sequence length and each input token x_i is represented by a d-dimension vector, self-attention first calculates the Query vector, the Key vector, and the Value vector for each input token by multiplying the input vector with three matrices W^q, W^k, and W^v. Then, it calculates the attention weight of sequence X by scoring the query vector Q against the key vector K of the input sentence. The scoring process is conducted by the scaled dot product, as shown in Equation (2), where the dimension d in the denominator is used to scale the dot product. Softmax is then used to normalize the attention score and, finally, the output vector is computed as a weighted sum of the input vectors. Instead of performing a single self-attention function, Transformer adopts multi-head self-attention (MHSA), which performs the self-attention function with different parameters in parallel and ensembles the output of each head by concatenating their outputs. The MHSA allows the model to jointly attend to information from different representation subspaces at different positions. Formally, the MHSA

[1] https://github.com/gszsectan/SG-Trans.

is computed as follows:

$$Q_i = XW_i^q, \quad K_i = XW_i^k, \quad V_i = XW_i^v, \tag{1}$$

$$head_i = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d}}\right) V_i, \tag{2}$$

$$MHSA(X) = \left[head_1^l \circ head_2^l \circ \cdots head_i^l \circ \cdots head_h^l\right] W^O, \tag{3}$$

where h denotes the number of attention heads at l-th each layer, the symbol $\circ$ indicates the concatenation of h different heads, and W_i^q, W_i^k, W_i^v, and W^O are trainable parameters.

2.1.2 Feed-Forward Network. The feed-forward network is the only nonlinear part in Transformer. It consists of two linear transformation layers and a ReLU activation function between the two linear layers:

$$FFN(X) = ReLU(XW_1 + b_1)W_2 + b_2, \tag{4}$$

where W_1, W_2, b_1, and b_2 are trainable parameters shared across input positions.

2.2 Copy Mechanism

The copy mechanism [19] has been widely equipped in text generation models for extracting words from a source sequence as part of outputs in a target sequence during text generation. It has been demonstrated that the copy mechanism can alleviate the out-of-vocabulary issue in the code summarization task [1, 67, 70]. Copying some variable name can also help generate a more precise summary. In this work, we adopt the pointer generator [48], a more popular form of copy mechanism, for the task. Figure 2 illustrates the architecture of the pointer generator model. Given an input sequence $X = (x_1, x_2, \ldots, x_n)$, a decoder input w_t, a decoder hidden state s_t, and a context vector c_t computed by the attention mechanism in timestep t, the pointer generator first calculates a constant P_{gen} that is later used as a soft switch for determining whether to generate a token from the vocabulary or to copy a token from the input sequence X:

$$P_{gen} = \text{sigmoid}\left(\omega_s^{\mathsf{T}} s_t + \omega_w^{\mathsf{T}} w_t + \omega_c^{\mathsf{T}} c_t + b_{gen}\right), \tag{5}$$

$$P_{vocab}(w_t) = \text{softmax}(W_a s_t + V_a c_t) \tag{6}$$

$$P(w_t) = P_{gen} P_{vocab}(w_t) + (1 - P_{gen}) P_{copy}(w_t), \tag{7}$$

where vectors ω_s, ω_w, ω_c, W_a, V_a, and scalar b_{gen} are learnable parameters. $P(w_t)$ is the probability distribution over the entire vocabulary. Copy distribution $P_{copy}(w_t)$ determines where to attend to in timestep t, computed as

$$P_{copy}(w_t) = \sum_{i:x_i=w} \alpha_{t,i}, \tag{8}$$

where α_t indicates the attention weights and $i : x_i = w$ indicates the indices of input words in the vocabulary.

3 PROPOSED APPROACH

In this section, we explicate the detailed architecture of SG-Trans. Let D denote a dataset containing a set of programs C and their associated summaries Z, given source code $c = (x_1, x_2, \ldots, x_n)$ from C, where n denotes the code sequence length. SG-Trans is designed to generate the summary consisting of a sequence of tokens $\hat{z} = (y_1, y_2, \ldots, y_m)$ by maximizing the conditional likelihood: $\hat{z} = \arg\max_z P(z|c)$ (z is the corresponding summary in Z).

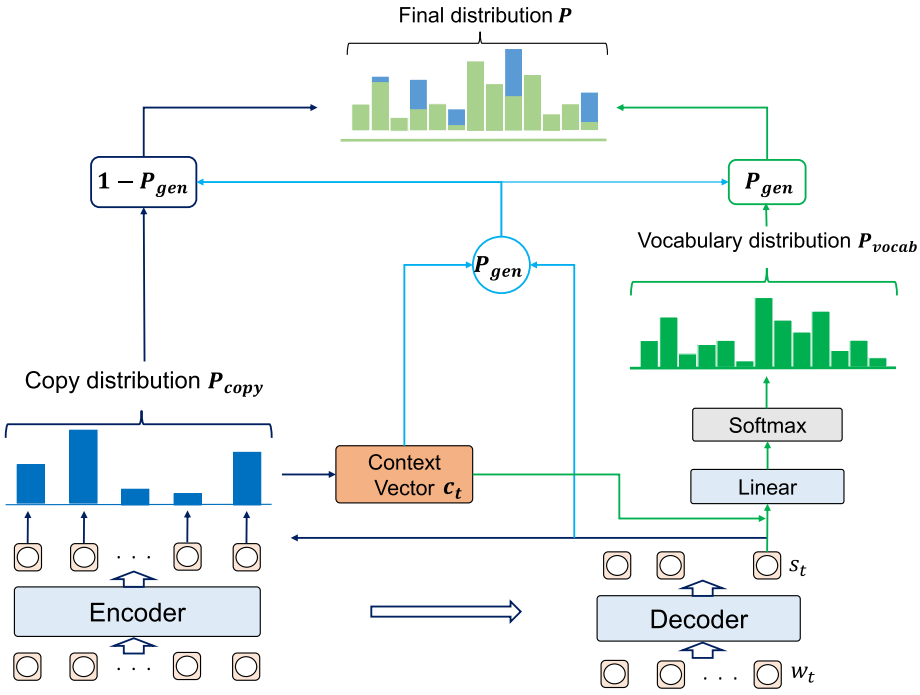


Fig. 2. Architecture of copy mechanism.

The framework of SG-Trans is mostly consistent with the vanilla Transformer, but consists of two major improvements: *structure-guided self-attention* and *hierarchical structure-variant attention*. Figure 3 depicts the overall architecture. SG-Trans first parses the input source code for capturing both local and global structure. The structure information is then represented as adjacency matrices and incorporated into the self-attention mechanism as inductive biases (introduced in Section 3.1). Following the principle of compositionality in language, different inductive biases are integrated into the Transformer at different levels in a hierarchical manner (introduced in Section 3.2).

3.1 Structure-Guided Self-Attention

In the standard multi-head self-attention model [55], every node in the adjacent layer is allowed to attend to all of the input nodes, as shown in Figure 3(a). In this work, we propose to use the structural relations in source code to introduce explicit constraints to the multi-head self-attention. In order to capture the hierarchical structure of source code, we utilize three main types of structural relations between code tokens, including local structures — whether the two split sub-tokens originally belong to the same (1) **token** or (2) **statement**, and global structure — whether there exists a (3) **dataflow** between two tokens. For each structure type, we design the corresponding head attention, named as *token-guided self-attention*, *statement-guided self-attention*, and *dataflow-guided self-attention*, respectively.

Token-Guided Self-Attention. Semantic relations between the sub-tokens are relatively stronger than the relations between the other tokens. For example, the method name "*IsPrime*" in the code example shown in Figure 1(a) is split as a sequence of sub-tokens containing "*Is*" and

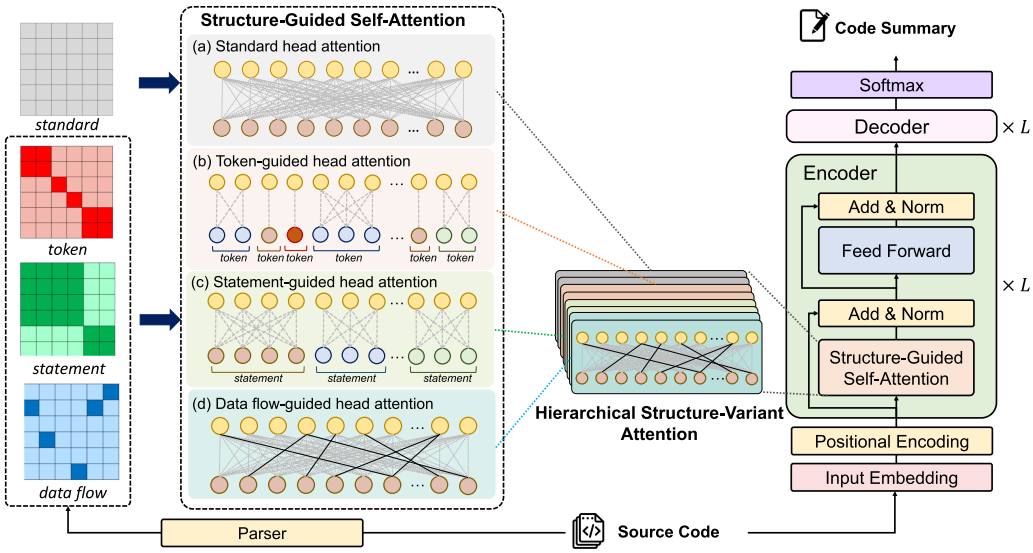


Fig. 3. Overall framework of the proposed SG-Trans. The "Structure-Guide Self-Attention" part of the figure illustrates different self-attention mechanisms between adjacent layers.

"*Prime*." Moreover, the semantic relation between them is stronger than the relation between "*Is*" and "*num*" in the same statement. Therefore, the attention can be built upon the extracted token-level structure, that is, whether two sub-tokens are originally from the same source code token. We use an adjacency matrix $\mathbf{T}^{n\times n}$ to model the relationship, where $t_{ij} = 0$ if the i-th and j-th elements are sub-tokens of the same token in the code. Otherwise, $t_{ij} = -\infty$. The matrix is designed to restrict the attention head to attend only to the sub-tokens belonging to the same code token in self-attention, as shown in Figure 3(b). Given the input token representation $\mathbf{X} \in \mathbb{R}^{n\times dh}$, where n is the sequence length, d is the input dimension of each head, and h is the number of attention heads, let $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ denote the query, key, and value matrix, respectively. The token-guided single-head self-attention $head_t$ can be calculated as:

$$head_t = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}} + \mathbf{T}\right)\mathbf{V}, \tag{9}$$

where $\sqrt{d}$ is a scaling factor to prevent the effect of large values.

Statement-Guided Self-Attention. Tokens in the same statement tend to possess stronger semantic relations than those from different statements. For the code example given in Figure 1(a), the token "*flag*" in the third statement is more relevant to the tokens "*bool*" and "*False*" in the same statement than to the token "*break*" in the 7-th statement. Thus, we design another adjacency matrix S to represent the pairwise token relations capturing whether the two tokens are from the same statement. In the matrix $\mathbf{S}$, $s_{ij} = 0$ if the i-th and j-th input tokens are in the same statement; otherwise, $s_{ij} = -\infty$. The design is to restrict the attention head to attend to only the tokens from the same statement, as illustrated in Figure 3(c). The statement-guided single-head self-attention $head_s$ is defined here, similar to the token-guided head attention:

$$head_s = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}} + \mathbf{S}\right)\mathbf{V}. \tag{10}$$

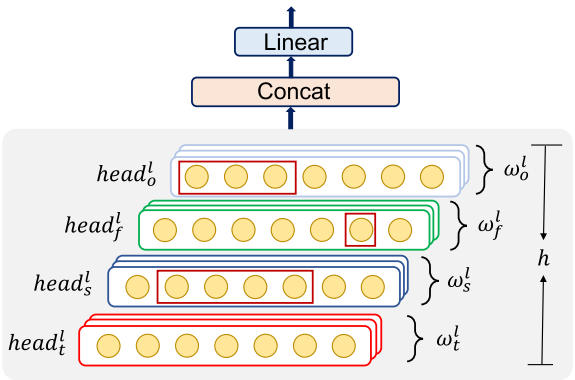


Fig. 4. A diagram of hierarchical-variant attention. Different red boxes illustrate different scales.

Dataflow-Guided Self-Attention. To facilitate the model to learn the global semantic information from code, we employ the DFGs for capturing the global semantic structure feature [21]. We do not involve ASTs as input since they are deeper in nature and contain more redundant information [3] than DFGs. DFGs, denoted as $V = \{v_1, v_2, \ldots\}$, can model the data dependencies between variables in the code, including message sending/receiving. Figure 1(c) shows an example of the extracted DFG. Variables with the same name (e.g., i^2 and i^5) are associated with different semantics in the DFG. Each variable is a node in the DFG and the direct edge $\langle v_i, v_j \rangle$ from v_i to v_j indicates that the value of the j-th variable comes from the i-th variable. We find the semantic relations among "i^2", "i^3", "i^4" and "i^5" that represent the data sending/receiving in a loop. Based on the DFGs, we build the adjacency matrix $\mathbf{D}$, where $d_{ij} = 1$ if there exists a message passing from the j-th token to the i-th token; otherwise, $d_{ij} = 0$. Note that if two variables have a data dependency, then their constituent sub-tokens also possess the dependency relation. Figure 3(d) illustrates the dataflow-guided single-head self-attention. To address the sparseness of the matrix $\mathbf{D}$ and to highlight the relations of data dependencies, we propose the following dataflow-guided self-attention $head_f$:

$$head_f = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\intercal + \mu * \mathbf{Q}\mathbf{K}^\intercal\mathbf{D}}{\sqrt{d}}\right)\mathbf{V}, \tag{11}$$

where μ is the control factor for adjusting the integration degree of the dataflow structure.

3.2 Hierarchical Structure-Variant Attention

Inspired by the principle of compositionality in logic semantics: high-level semantics is the composition of low-level terms [23, 54]. We propose a hierarchical structure-variant attention such that our model would focus on local structures at the lower layers and global structure at the higher layers. A diagram of hierarchical structure-variant attention is provided in Figure 4. Specifically, the token-guided head attention $head_t$ and the statement-guided head attention $head_s$ are used more in the heads of lower layers, whereas the dataflow-guided head attention $head_f$ is more spread in the heads of higher layers.

Let L denote the number of layers in the proposed SG-Trans, let h indicate the number of heads in each layer, and let k be a hyper-parameter to control the distribution of four types of head attentions, including $head_t$, $head_s$, $head_f$, and $head_o$, where $head_o$ indicates the standard head attention without constraints, and the distribution for each type of head attention at the l-th layer is denoted as $\Omega^l = [\omega_t^l, \omega_s^l, \omega_f^l, \omega_o^l]$, where ω_t^l, ω_s^l, ω_f^l, and ω_o^l represent the numbers

of $head_t$, $head_s$, $head_f$, and $head_o$, respectively at the l-th layer. We define the distribution as follows:

$$\omega_t^l = \omega_s^l = \left\lfloor h * \frac{k-l}{2*k-l} \right\rfloor, \tag{12}$$

$$\omega_f^l = \left\lfloor h * \frac{l}{2*k-l} \right\rfloor, \tag{13}$$

$$\omega_o^l = h - \left(\omega_t^l + \omega_s^l + \omega_f^l\right), \tag{14}$$

where k is a positive integer hyper-parameter and $\lfloor \cdot \rfloor$ denotes rounding the value down to the next lowest integer. The design is intended to enable more heads attending to the global structure with the growth of l, that is, ω_f^l will get larger at a higher layer l, whereas a few heads can catch the local structure, that is, ω_t^l and ω_s^l will become smaller. $head_o$ is involved to enable the model to be adapted to arbitrary numbers of layers and heads. With the increase of layer l, ω_t^l and ω_s^l might drop to zero. In the case of $\omega_t^l \leq 0$, no constraints will be introduced to the corresponding attention layer since the standard self-attention already captures long-range dependency information, which fits our purpose of attending to global structure at higher layers. Otherwise, the head attentions will follow the defined distribution Ω^l.

The hierarchical structure-variant attention (HSVA) at the l-th layer is computed as

$$\text{HSVA}^l = \left[head_1^l \circ \cdots \circ head_h^l\right] \mathbf{W}^O, \tag{15}$$

where $\circ$ denotes the concatenation of h different heads, and $\mathbf{W}^O \in \mathbb{R}^{dh \times dh}$ is a parameter matrix.

3.3 Copy Attention

The OOV issue is important for effective code summarization [34]. We adopt the copy mechanism introduced in Section 2.2 in SG-Trans to calculate whether to generate words from the vocabulary or to copy from the input source code. Following Ahmad et al. [1], an additional attention layer is added to learn the copy distribution on top of the decoder [46]. The mechanism enables the proposed SG-Trans to copy low-frequency words, for example, API names, from source code, mitigating the OOV issue.

4 EXPERIMENTAL SETUP

In this section, we introduce the evaluation datasets and metrics, comparison baselines, and parameter settings.

4.1 Benchmark Datasets

We conduct experiments on two benchmark datasets that contain Java and Python source code, respectively following the previous work [1, 67]. The Java dataset publicly released by Hu et al. [27] comprises 87,136 ⟨Java method, comment⟩ pairs collected from 9,714 GitHub repositories. The Python dataset consists of 92,545 functions and corresponding documentation as originally collected by Barone et al. [8] and later processed by Wei et al. [61]. We list the statistics of the datasets in Table 1.

For fair comparison, we directly use the benchmarks open sourced by the previous related studies [1, 28, 60], in which the datasets are split into training set, validation set, and test set in a proportion of 8 : 1 : 1 and 6 : 2 : 2 for Java and Python, respectively. We follow the commonly used dataset split strategy with no modification to avoid any bias introduced by dataset split.

Table 1. Statistics of the Benchmark Datasets

	Java	Python
Training Set	69,708	55,538
Validation Set	8,714	18,505
Test Set	8,714	18,502
Total	87,136	92,545

We apply *CamelCase* and *snake_case* tokenizers [1] to get sub-tokens for both datasets. As for the code statements, we apply a simple rule to extract the statements of code snippets. For the extraction of statements from the Java dataset, we split each code snippet into statements with separators including '{', '}' and ';'. The token sequence between two adjacent separators is considered as a statement. For the example shown in Figure 1(a), each line except for the separators is one statement. For the Python dataset, we define a statement by the row, which means that the tokens in the same row are considered as belonging to the same statement. For the extraction of dataflow from the Java dataset, we use the tool in GSC [12] to first generate augmented ASTs and then extract DFGs. Regarding the Python dataset, we follow the setup in the work of Allamanis et al. [3] and extract four kinds of edge (*LastRead*, *LastWrite*, *LastLexicalUse*, *ComputeFrom*) from code.

4.2 Evaluation Metrics

To verify the superiority of SG-Trans over the baselines, we employ the most commonly used automatic evaluation metrics, BLEU-4 [47], METEOR [7], and ROUGE-L [37].

BLEU is a metric widely used in natural language processing and software engineering fields to evaluate generative tasks (e.g., dialogue generation, code commit message generation, and pull request description generation) [33, 40, 45, 66]. BLEU uses n-gram for matching and calculates the ratio of N groups of word similarity between generated comments and reference comments. The score is computed as

$$BLEU - N = BP \times \exp\left(\sum_{n=1}^{N} \tau_n \log P_n\right), \tag{16}$$

where P_n is the ratio of the subsequences with length n in the candidate that are also in the reference. BP is the brevity penalty for short generated sequences and τ_n is the uniform weight $1/N$. We use corpus-level BLEU-4, that is, $N = 4$, as our evaluation metric since it is demonstrated to be more correlated with human judgments than other evaluation metrics [38].

METEOR is a recall-oriented metric that measures how well our model captures content from the reference text in our generated text. It evaluates generated text by aligning it to reference text and calculating sentence-level similarity scores:

$$METEOR = (1 - \gamma \cdot frag^{\beta}) \cdot \frac{P \cdot R}{\alpha \cdot P + (1 - \alpha) \cdot R}, \tag{17}$$

where P and R are the unigram precision and recall, and *frag* is the fragmentation fraction. α, β, and γ are three penalty parameters whose default values are 0.9, 3.0, and 0.5, respectively.

ROUGE-L is widely used in text summarization tasks in the natural language processing field to evaluate to what extent the reference text is recovered or captured by the generated text. ROUGE-L is based on the Longest Common Subsequence (LCS) between two texts and the F-measure is used as its value. Given a generated text X and the reference text Y whose lengths are m and n,

respectively, ROUGE-L is computed as

$$P_{lcs} = \frac{LCS(X,Y)}{n}, R_{lcs} = \frac{LCS(X,Y)}{m}, F_{lcs} = \frac{(1+\beta^2)P_{lcs}R_{lcs}}{R_{lcs}+\beta^2 P_{lcs}}, \tag{18}$$

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

4.3 Baselines

We compare SG-Trans with the following baseline approaches.

CODE-NN [31], the first deep-learning-based work in code summarization, generates source code summaries with a long short-term memory (LSTM) network. To utilize code structure information, **Tree2seq** [14] encodes source code with a tree-LSTM architecture. **Code2seq [4]** represents the code snippets by sampling paths from the AST. **RL+Hybrid2Seq** [57] incorporates ASTs and code sequences into a deep reinforcement learning framework, while **DeepCom** [27] encodes the node sequences traversed from ASTs to capture structural information. **API+Code** [28] involves API knowledge in the code summarization procedure. **Dual model** [61] adopts a dual learning framework to exploit the duality of code summarization and code generation tasks. One of the most recent approaches, **NeuralCodeSum** [1], integrates the **vanilla Transformer** [55] with relative position encoding (RPE) and copy attention. Another recent approach, **Transformer+GNN** [11], applies graph convolution to obtain structurally encoded node representations and passes sequences of the graph-convolutioned AST nodes into Transformer.

We also compare our approach with relational Transformers [26, 70], which involve structural information for code representation learning. **GREAT [26]** biases vanilla Transformers with relational information from graph edge types. **CodeTransformer [70]** focuses on multilingual code summarization and proposes to build upon language-agnostic features such as source code– and AST-based features.

During implementation, we either directly copy the results claimed in the original papers or reproduce the results strictly following the released repositories for most baselines except for GREAT and CodeTransformer. For GREAT [26], 12 types of information, including control flow graph and syntactic features, are adopted for model training, as no replication package is available. Due to the difficulty of complete replication, we follow the strategy in the work of Zügner et al. [70] by employing the same structural information as SG-Trans during replication. For CodeTransformer, although a replication package is provided by the authors, not all of the benchmark data can be successfully preprocessed. For Java, only 61,250 of 69,708 code snippets in the training set, 7,621 of 8,714 in the validation set, and 7,643 of 8,714 in the test set pass the preprocessing step, whereas for Python, all of the code snippets can be well preprocessed. To ensure the consistency of evaluation data, we compare SG-Trans with CodeTransformer on the Java dataset separately. We use the same model settings for implementing CodeTransformer, including the layer number, head number, and so on.

4.4 Parameter Settings

SG-Trans is composed of 8 layers and 8 heads in its Transformer architecture and the hidden size of the model is 512. We use Adam optimizer with the initial learning rate set to 10^{-4}, batch size set to 32, and dropout rate set to 0.2 during the training. We train our model for at most 200 epochs and select the checkpoint with the best performance on the validation set for further evaluation on the test set. We report the performance of SG-Trans and each ablation experiment by running three times and taking the average. To avoid over-fitting, we stop the training early if the performance on the validations set does not increase for 20 epochs. For the control factors of head distribution and dataflow, we set them to 1 and 5, respectively. We will discuss optimal parameters selection

Table 2. Comparison Results with Baseline Models

Approach	Java			Python		
	BLEU-4	METEOR	ROUGE-L	BLEU-4	METEOR	ROUGE-L
CODE-NN [31]	27.60	12.61	41.10	17.36	09.29	37.81
Tree2Seq [14]	37.88	22.55	51.50	20.07	08.96	35.64
RL+Hybrid2Seq [57]	38.22	22.75	51.91	19.28	09.75	39.34
DeepCom [27]	39.75	23.06	52.67	20.78	09.98	37.35
API+Code [28]	41.31	23.73	52.25	15.36	08.57	33.65
Dual Model [60]	42.39	25.77	53.61	21.80	11.14	39.45
Code2seq [4]	12.19	08.83	25.61	18.69	13.81	34.51
Vanilla Transformer [55]	44.20	26.83	53.45	31.34	18.92	44.39
NeuralCodeSum [1]	45.15	27.46	54.84	32.19	19.96	46.32
GREAT [26]	44.97	27.15	54.42	32.11	19.75	46.01
CodeTransformer [70]	–	–	–	27.63	14.29	39.27
Transformer+GNN [11]	45.49	27.17	54.82	32.82	20.12	46.81
SG-Trans	**45.89***	**27.85***	**55.79***	**33.04***	**20.52***	**47.01***

The **bold** figures indicate the best results. *denotes statistical significance in comparison with the baseline models we reproduced (i.e., two-sided t-test with p-value < 0.01).

in Section 5.3. Our experiments are conducted on a single Tesla V100 GPU for about 30 hours, and we train our model from scratch.

5 EXPERIMENTAL RESULTS

In this section, we elaborate on the comparison results with the baselines to evaluate SG-Trans's capability in accurately generating code summaries. Our experiments are aimed at answering the following research questions:

RQ1: What is the performance of SG-Trans in code summary generation?
RQ2: What is the impact of the involved code structural properties and the design of hierarchical attentions on the model performance?
RQ3: How accurate is SG-Trans under different parameter settings?

5.1 Answer to RQ1: Comparison with the Baselines

The experimental results on the benchmark datasets are provided in Table 2. For vanilla Transformer and NeuralCodeSum [1], we reproduce their experiments under the same hyper-parameter settings as the Transformer in SG-Trans to ensure fair comparison. We compare SG-Trans with CodeTransformer [70] on the Java dataset separately, in which both approaches are trained and evaluated on the same dataset, with results provided in Table 3. Based on Tables 2 and 3, we present the following findings:

Code Structural Properties are Beneficial for Source Code Summarization. Comparing Tree2Seq/DeepCom with CODE-NN, we find that the structure information facilitates a great improvement in the performance. For example, both Tree2Seq and DeepCom outperform CODE-NN by at least 37.2%, 78.8%, and 25.3%, respectively, regarding the three metrics on the Java dataset. Although no consistent improvement across all metrics is observed on the Python dataset, Tree2Seq/DeepCom still shows an obvious increase on the BLEU-4 metric.

Transformer-Based Approaches Perform Better than RNN-Based Approaches. The four Transformer-based approaches [1, 4, 11, 26, 55] outperform all of the other baselines, with NeuralCodeSum [1] giving better performance compared with the vanilla Transformer. The vanilla

Table 3. Comparison Results with CodeTransformer on the Dataset Preprocessed by CodeTransformer

Approach	Java		
	BLEU-4	METEOR	ROUGE-L
CodeTransformer [70]	39.81	24.22	51.96
SG-Trans	**44.59***	**27.32***	**54.41***

The **bold** figures indicate the best results. *denotes statistical significance in comparison with the baseline models (i.e., two-sided t-test with p-value < 0.01).

Table 4. Ablation Study on Different Parts of Our Model

Approach	Java			Python		
	BLEU-4	METEOR	ROUGE-L	BLEU-4	METEOR	ROUGE-L
SG-Trans w/o token info	44.92	27.35	54.69	32.18	19.87	46.14
SG-Trans w/o statement info	44.61	27.08	54.04	32.26	19.66	46.08
SG-Trans w/o dataflow info	45.52	27.65	55.40	32.58	20.16	46.57
SG-Trans w/o hierarchical attention	45.53	27.72	55.48	32.93	20.38	46.73
SG-Trans w/o copy attention	45.24	27.49	55.01	31.89	19.26	45.31
$\text{SG-Trans}_{\text{soft}}$	45.37	27.65	55.09	32.77	19.96	46.74
SG-Trans	**45.89***	**27.85***	**55.79***	**33.04***	**20.52***	**47.01***

The **bold** figures indicate the best results. *denotes statistical significance in comparison with the baseline models we reproduced (i.e., two-sided t-test with p-value < 0.01).

Transformer already achieves better performance than the top seven RNN-based approaches with various types of structural information being incorporated, showing the efficacy of Transformer for the task. On the Python dataset, NeuralCodeSum outperforms the best RNN-based baseline, Dual Model [61], by 47.7% and 17.4% in terms of the BLEU-4 and ROUGE-L metrics, respectively.

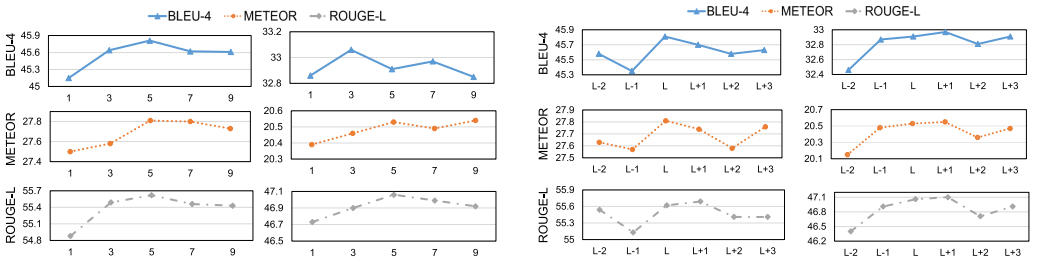
The Proposed SG-Trans is Effective in Code Summarization. Comparing SG-Trans with NeuralCodeSum and Transformer+GNN, SG-Trans achieves the best results on both benchmark datasets, yet without introducing any extra model parameters. SG-Trans improves the best baseline by 1.7% and 0.4% in terms of ROUGE-L score on the Java and Python dataset, respectively.

The Combination of the Structural Information in SG-Trans is Effective. By comparing SG-Trans with other Transformer models with structural information involved such as GREAT [26], CodeTransformer [70], and Transformer+GNN [11], SG-Trans achieves the best results on both benchmark datasets. SG-Trans improves the best baseline by 2.5% and 2.0% in terms of METEOR score on the Java and Python dataset, respectively.

5.2 Answer to RQ2: Ablation Study

We further perform ablation studies to validate the impact of the involved code structural properties and the hierarchical structure-variant attention approach. In addition, to evaluate the efficacy of the hard mask attention mechanism for combining token-level and statement-level information in SG-Trans, we create a comparative approach, $\text{SG-Trans}_{\text{soft}}$, by changing the hard mask into a soft mask. For $\text{SG-Trans}_{\text{soft}}$, we follow NeuralCodeSum [1] and add the relative position embedding for subtoken pairs x_i and x_j only if they are in the same token or statement. The results are shown in Table 4.

Analysis of the Involved Code Structure. We find that all three structure types — code token, statement, and dataflow — contribute to model performance improvement but with varied degrees. Local syntactic structures play a more important role than the global dataflow structure. For



(a) Analysis of the parameter μ.

(b) Analysis of the parameter k. $L = 8$.

Fig. 5. Influence of the hyper-parameters μ and k on model performance.

example, removing the statement information leads to a significant performance drop at around 2.8% and 2.4% regarding the BLEU-4 score. This suggests the importance of modeling the semantic relations among tokens of the same statement for code summarization. With the dataflow information eliminated, SG-Trans also suffers from a performance drop, which may indicate that it is difficult for Transformer to learn data dependency relations implicitly.

Analysis of the Hierarchical Structure-Variant Attention Mechanism. We replace the hierarchical structure-variant attention with uniformly distributed attention, that is, $\Omega^l = [\omega_t^l, \omega_s^l, \omega_f^l, \omega_o^l] = [2, 2, 2, 2]$, for the ablation analysis. As can be found in Table 2, without the hierarchical structure design, the model's performance decreases on all metrics for both datasets. The results demonstrate the positive impact of the hierarchical structure-variant attention mechanism.

Analysis of the Copy Attention. As shown in Table 2, excluding the copy attention results in a significant drop in SG-Trans's performance, similar to what has been observed in the work of Ahmad et al. [1]. This shows that copy attention is useful for alleviating the OOV issue and facilitating better code summarization.

Analysis of the Hard Mask Attention Mechanism. As shown in Table 4, we find that SG-Trans performs constantly better than SG-Trans$_{\text{soft}}$ on both Java and Python datasets with respect to all metrics. For example, on Java dataset, replacing hard masks with soft masks leads to a performance drop of 1.1% and 1.3% in terms of the BLEU-4 and ROUGE-L metrics, respectively, which indicates that the hard mask attention is effective at capturing the local information.

5.3 Answer to RQ3: Parameter Sensitivity Analysis

In this section, we analyze the impact of two key hyper-parameters on model performance, the control factor μ for adjusting the integration degree of the dataflow structure and the parameter k to control head distribution.

The Parameter μ. Figure 5(a) shows the performance variation with the changes of μ while keeping other hyper-parameters fixed. For the Java dataset, the model achieves the best scores when $\mu = 5$. Lower or higher parameter values do not give better results. For the Python dataset, a similar trend is observed for the BLEU-4 and ROUGE-L metrics—the model performs best when μ equals 3 and 5, respectively. In this work, we set μ to 5 since the model can produce relatively better results on both datasets.

The Parameter k. We observe the performance changes when the control factor k of the head distribution takes values centered on layers of SG-Trans L. Figure 5(b) illustrates the results. We can find that SG-Trans can balance the distribution of local and global structure-guided head attention well when $k = L$ or $k = L + 1$. As k gets larger, SG-Trans would be more biased by the local structure and tend to generate an inaccurate code summary. In our work here, we set $k = L$.

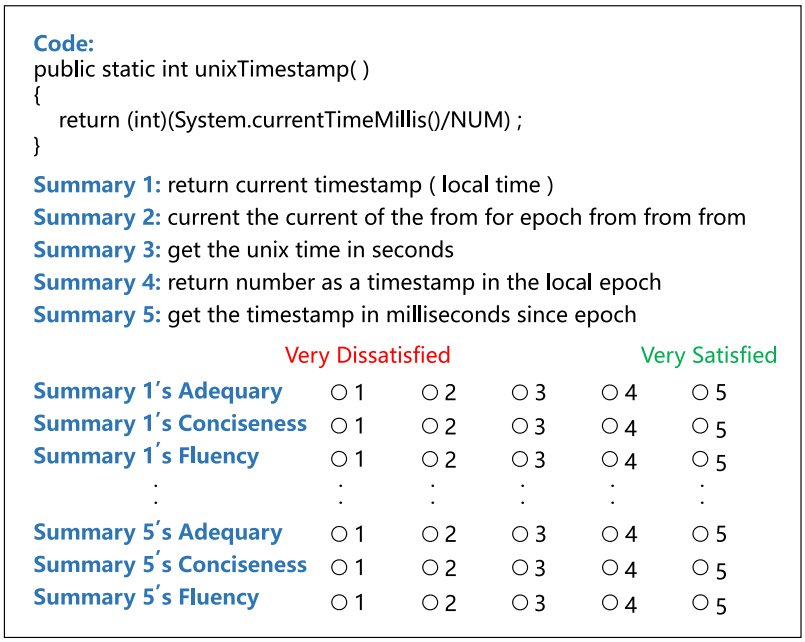


Fig. 6. An example of questions in our questionnaire. The two-dot symbols indicate the simplified rating schemes for Summaries 2, 3, and 4.

5.4 Human Evaluation

In this section, we perform human evaluation to qualitatively evaluate the summaries generated by four Transformer-based baselines, including vanilla Transformer, NeuralCodeSum, GREAT, and CodeTransformer, along with our model SG-Trans. We do not involve the baseline Transformer+GN due to the lack of replication package. The human evaluation is conducted through online questionnaire. In total, 10 software developers are invited to participate in evaluation. All participants have at least 4 years of programming experience in software development and none is a coauthor of this article. Each participant is invited to read 60 code snippets and judge the quality of summaries generated by vanilla Transformer, NeuralCodeSum, CodeTransformer, GREAT, and SG-Trans. Each will be paid 30 USD upon completing the questionnaire.

5.4.1 Survey Design. We randomly selected 200 code snippets, with 100 in Java and 100 in Python, for evaluation. As shown in Figure 6, in the questionnaire, each question comprises a code snippet and summaries generated by the five models. Each participant will be given 60 questions and each question will be evaluated by three different participants. For each question, the summaries generated by the models are randomly shuffled to eliminate order bias.

The quality of the generated summaries is evaluated in terms of *Adequacy*, *Conciseness*, and *Fluency* with the 1 to 5 Likert scale (5 for excellent, 4 for good, 3 for acceptable, 2 for marginal, and 1 for poor). We explained the meaning of the three evaluation metrics at the beginning of the questionnaire: The metric "adequacy" measures how much the functional meaning of the code is preserved after summarization; the metric "conciseness" measures the ability to express the function of code snippet without unnecessary words; and the metric "fluency" measures the quality of the generated language, such as the correctness of grammar.

Table 5. Human Evaluation Results

Dataset	Metrics	Transformer	CodeTransformer	NeuralCodeSum	GREAT	SG-Trans
Java	Adequacy	3.35	2.67	3.28	3.44	**3.65**
	Conciseness	4.20	3.49	4.32	4.36	**4.50**
	Fluency	4.32	3.25	4.36	4.50	**4.59**
Python	Adequacy	2.61	1.92	3.04	2.83	**3.21**
	Conciseness	3.84	2.62	4.01	4.05	**4.21**
	Fluency	4.06	2.39	4.21	4.26	**4.33**

The **bold** figures indicate the best results.

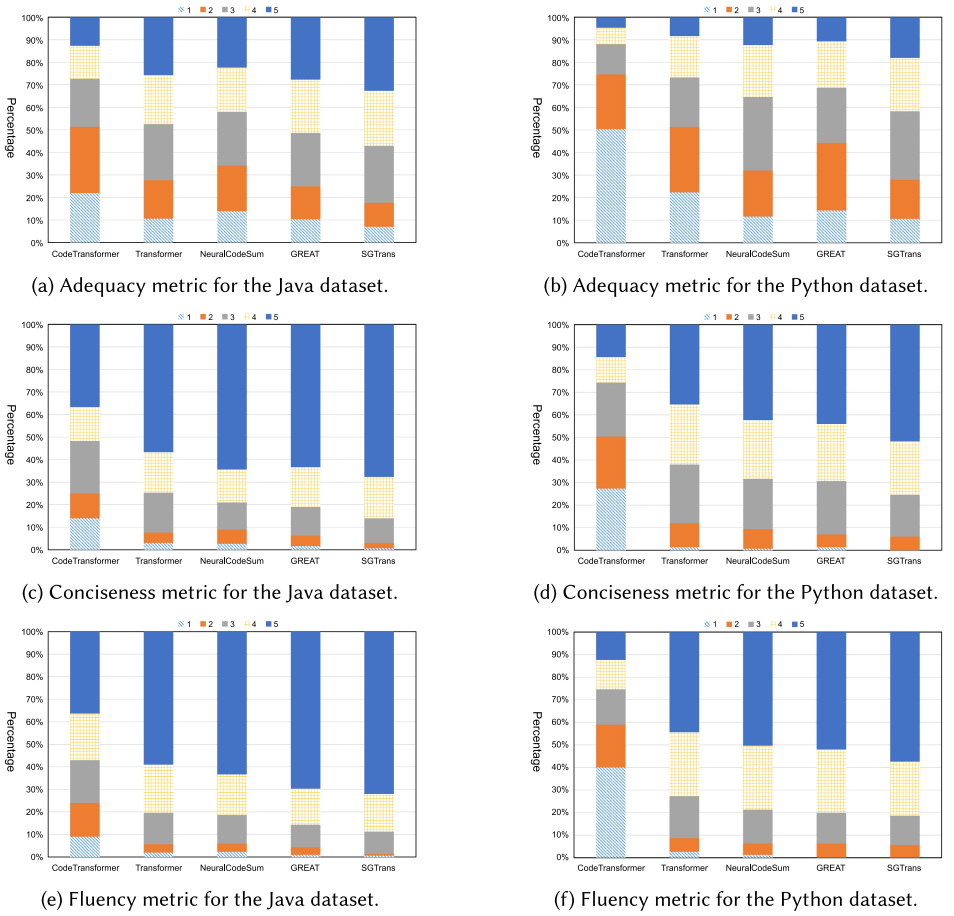
5.4.2 Survey Design. We received 600 sets of scores with 3 sets of scores for each code-summary pair from the human evaluation. On average, the participants spent 2 hours on completing the questionnaire, with the median completion time at 1.67 hours. The inter-annotator agreement of the two sets is evaluated with the widely used metric Cohen's kappa. The average Cohen's kappa scores for the Java and Python datasets are 0.66 and 0.58, respectively, indicating that the participants achieved at least moderate agreement on both datasets.

The evaluation results are provided in Table 5 and Figure 7. We find that the summaries generated by SG-Trans receive the highest scores on both datasets and with respect to all of the metrics. For the Java dataset, as shown in Table 5, SG-Trans improves the baseline models by at least 6.1%, 3.2%, and 2.0% with respect to the adequacy, conciseness, and fluency metrics, respectively. As can be observed from Figures 7(a), 7(c) and 7(e), summaries generated by SG-Trans receive the most 5-star ratings and fewest 1/2-star ratings from the participants compared with the summaries produced by other models for each metric. Regarding the fluency metric, only 1.3% of the participants gave 1/2-star ratings to the summaries generated by SG-Trans, while other approaches receive at least 6.0% 1/2-star ratings and CodeTransformer receives 24.0%. The score distributions indicate that SG-Trans better captures the functionality of given code snippets and generates a higher quality of natural language comments.

For the Python dataset, as shown in Table 5, NeuralCodeSum and GREAT significantly outperform the vanilla Transformer and CodeTransformer, whereas SG-Trans is more powerful, further boosting the best baseline approach by 5.6%, 4.0%, and 1.6% in terms of the adequacy, conciseness, and fluency, respectively. As can be observed from Figures 7(b), 7(d) and 7(f), summaries generated by SG-Trans receive the most 5-star ratings and fewest 1/2-star ratings from the annotators. Specifically, regarding the adequacy metric, 18.0% of the participants gave 5-star ratings to the summaries generated by SG-Trans, with only 4.6% for the CodeTransformer approach and 12.3% for the strongest baseline NeuralCodeSum. For the conciseness metric, 51.7% of the participants gave 5-star ratings to the summaries generated by SG-Trans and the two best baseline approaches NeuralCodeSum and GREAT receive only 42.3% and 44.0% 5-star ratings, respectively. The score distributions indicate that the summaries generated by SG-Trans can better describe the function of code snippets and have a more concise and accurate expression.

5.5 Further Evaluation on Generated Summaries

To further investigate the quality of auto-generated summaries, we invite participants to summarize the code without access to the reference summary and then ask the annotators to score the summaries. During manual code summarization, we invite four postgraduate students with more than 5 years of development experience as well as internship experience in technology companies to participate. To ease the pressure of annotation, we randomly select 80 code snippets with code lengths fewer than 250 characters. Each participant is asked to write summaries for 20 code snippets, 10 in Java and 10 in Python. Each will be paid 15 USD upon completing the questionnaire.



(a) Adequacy metric for the Java dataset.

(b) Adequacy metric for the Python dataset.

(c) Conciseness metric for the Java dataset.

(d) Conciseness metric for the Python dataset.

(e) Fluency metric for the Java dataset.

(f) Fluency metric for the Python dataset.

Fig. 7. Distribution of the rating scores in human evaluation on the two datasets. The "Transformer" on the horizontal axis denotes the "vanilla Transformer" approach.

We receive annotations of 78/80 code snippets in total. For the remaining two code snippets, as shown in Table 7, the annotators found the functionalities difficult to understand without corresponding prior knowledge. For the example in Table 7 (Example 1), the unclear meanings of "*K.zero*" and "*f[(−1)]*" hinder program comprehension. We measure the quality of summaries generated by SG-Trans and humans using the same metrics introduced in Section 4.2. The results are shown in Table 6. From the table, we find that compared with human-generated summaries, auto-generated summaries are much more similar to the reference summaries. For example, the BLEU-4 scores of the auto-generated summaries are 77.06 and 75.09 on Java and Python, respectively, while the human-generated summaries are only 19.04 and 23.63 on Java and Python, respectively. To determine the reason for the large difference between human-generated summaries and reference summaries, we manually check all of the annotated data. The two main reasons are summarized here.

- **Lack of contextual knowledge.** Some code snippets use external APIs or inner elements of a class, and the details of the APIs and elements cannot be accessed. Thus, the annotators can only infer the functions of code snippets based on the function/variable names, resulting in poorly written summaries. For example, as shown in Table 8 (Example 1), since the detail of the external API "*string_literal*" is unknown, humans can only guess its meaning from the

Table 6. The Quality of Summaries Generated by SG-Trans and Humans

Predicted Summary	Test	Java			Python		
		BLEU-4	METEOR	ROUGE-L	BLEU-4	METEOR	ROUGE-L
Auto-generated	Reference	77.06	51.67	89.95	75.09	49.04	87.56
Human-generated	Reference	19.04	13.33	32.83	23.63	19.82	38.95

The term "auto-generated" indicates the summaries output by SG-Trans.

Table 7. The Code Snippets That Cannot be Understood by the Annotators

Example (1) in Python:

```
def poly_TC(f, K):
    if (not f):
        return K.zero
    else:
        return f[(-1)]
```

Human-generated: **cannot understand**
SG-Trans: return trailing coefficient of f
Ground truth: return trailing coefficient of f

Example (2) in Java:

```
@SuppressWarnings(STRING)
public PropagationImp(Stack<CompositeTransaction>lineage, boolean serial, long timeout){
    serial_ = serial;
    lineage_ = (Stack<CompositeTransaction>)lineage.clone();
    timeout_ = timeout;}
```

Human-generated: **cannot understand**
SG-Trans: create a new instance
Ground truth: create a new instance

name. In Table 8 (Example 2), "*acl*" is an inner element of the class "*bucket*" but the definition is missing, which makes it hard for humans to comprehend the function. Our model has been provided with the knowledge that "*acl*" stands for "*access control list*" during training; thus, it can output a more accurate summary.

- **Limitation of the evaluation metrics.** As shown in Table 8 (Examples 3 and 4), although the summaries generated by humans can accurately reflect the functions of the code snippets, they are significantly different from the reference summaries, leading to low metric scores. For Example 4 in Table 8, both the human-generated summary and reference summary explain the meaning of the code snippet well. However, under the existing metrics based on n-gram matching, the metric scores between them are very low since they have only one overlapping word: "*parse.*"

We then qualitatively inspect the quality of human-generated summaries and auto-generated summaries. We invite another three annotators, who have not joined the manual code summarization part, for the inspection. The results are provided in Table 9. The Cohen's kappa scores of the annotation results are 0.69 and 0.71 on Java and Python, respectively, indicating a substantial inter-rater reliability on both datasets. As shown in Table 9, the quality of human-generated summaries is better than that of the auto-generated summaries with respect to all metrics. Specifically, the conciseness and fluency scores of human-written summaries are nearly 5 on both datasets. Moreover, the adequacy scores of human-written summaries outperform the auto-generated summaries by 21.0% and 35.5% on Java and Python datasets, respectively. The results further explain the huge difference between human-generated summaries and reference summaries under automatic evaluation, as shown in Table 6, reflecting the limitation of automatic metrics.

Table 8. Examples Illustrating the Difference Between Summaries Generated by SG-Trans and Written by Humans

Example (1) in Python:

```
def Thing2Literal(o, d):
    return string_literal(o, d)
```

Human-generated: return the literalness of a string
SG-Trans: convert something into a string representation
Ground truth: convert something into an sql string literal

Example (2) in Python:

```
def print_bucket_acl_for_user(bucket_name, user_email):
    storage_client = storage.Client()
    bucket = storage_client.bucket(bucket_name)
    bucket.acl.reload()
    roles = bucket.acl.user(user_email).get_roles()
    print roles
```

Human-generated: print the bucket acl for user
SG-Trans: prints out a buckets access control list for a given user
Ground truth: prints out a buckets access control list for a given user

Example (3) in Java:

```
public ActivityResolveInfo(ResolveInfo resolveInfo){
     this.resolveInfo = resolveInfo;}
```

Human-generated: assign a value to resolveinfo attribute
SG-Trans: creates a new activity
Ground truth: creates a new instance

Example (4) in Java:

```
public static Date parseText(String dateStr){
    try {return mSimpleTextFormat.parse(dateStr);}
    catch(ParseException e){
        e.printStackTrace();
        throw new RuntimeException(STRING);}}
```

Human-generated: parse the dateStr as a date instance
SG-Trans: parse string to datetime
Ground truth: parse string to datetime

Table 9. Human Evaluation on Summaries Generated by SG-Trans and Human-Written Summaries

Dataset	Metrics	Human-written	SG-Trans
Java	Adequacy	**4.38**	3.62
	Conciseness	**4.82**	4.48
	Fluency	**4.94**	4.58
Python	Adequacy	**4.39**	3.24
	Conciseness	**4.86**	4.23
	Fluency	**4.95**	4.34

6 DISCUSSION

In this section, we mainly discuss the key properties of the proposed SG-Trans, the impact of duplicate data in the benchmark dataset on model performance, and the limitations of our study.

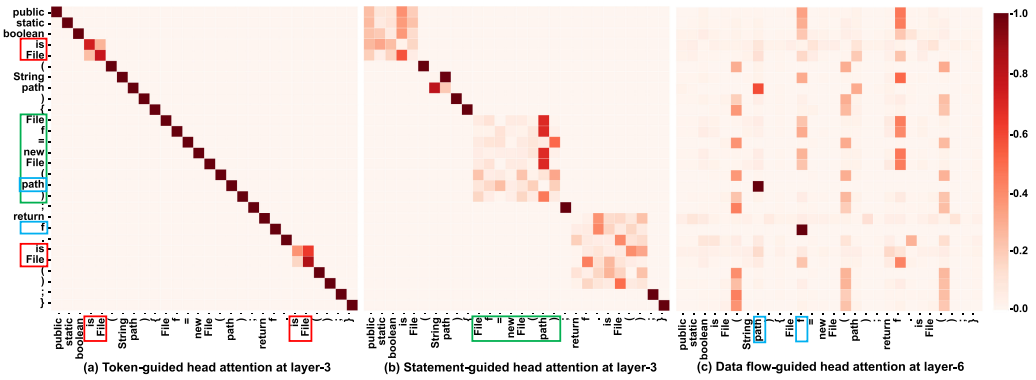


Fig. 8. Heat map visualization of self-attention scores of the three types of heads in the encoder for the first case in Table 10. The rectangles with red edges, green edges, and blue edges indicate the tokens belonging to the same original token, the same statement, or containing dataflow relation, respectively.

6.1 Why Does Our Model Work?

We further conduct an analysis to gain insights into the proposed SG-Trans in generating high-quality code summaries. Through qualitative analysis, we have identified two properties of SG-Trans that may explain its effectiveness in the task.

Observation 1: SG-Trans can better capture the semantic relations among tokens. From Example (1) in Table 10, we observe that SG-Trans produces the summaries most similar to the ground truth among all of the approaches, whereas the CodeTransformer gives the worst result. We then visualize the heat map of the self-attention scores of the three types of heads in Figure 8 for a further analysis. As can be seen in Figures 8(a) and 8(b), SG-Trans can focus on local relations among code tokens through its token-guided self-attention and statement-guided self-attention. For example, SG-Trans can learn that the two tokens "*is*" and "*File*" possess a strong relation, according to Figure 8(a). As depicted in Figure 8(b), we find that SG-Trans captures that the token "*path*" is strongly related to the corresponding statement, which may be the reason the token "*path*" appears in the summary. Figure 8(c) shows that the dataflow-guided head attention focuses more on the global information and can capture the strong relation between the tokens "*path*" and "*f*." Based on the analysis of Example 1, we speculate that the model can capture the token relations locally and globally well for code summary generation. As for the heat map of other baseline models such as NeuralCodeSum, we can see that they are very different with the heat map of SG-Trans. As shown in Figure 9, NeuralCodeSum does not capture the token, statement, and data flow information in any layer while SG-Trans can pay more attention to the token pairs with syntactic or semantic relations. A similar conclusion can be drawn from Example 2 in Table 10. All of the approaches successfully comprehend that the token "*acl*" indicates "*access control list*." However, the vanilla Transformer fails to capture the semantic relations between "*print*" and "*acl*" and NeuralCodeSum misunderstands the relations between "*user*" and "*acl*." Instead, SG-Trans accurately predicates both relations through the local self-attention and global self-attention.

Observation 2: Structural information-guided self-attention can facilitate the copy mechanism to copy important tokens. In Example 3 in Table 10, SG-Trans successfully identified the important token "*urlsafe*" in the given code while generating the summary. However, both vanilla Transformer and NeuralCodeSum ignored the token and output less accurate summaries. The important token being successfully copied by SG-Trans may be attributed to the structural information-guided self-attention, which helps focus on the source tokens more accurately.

Table 10. Examples Illustrating Summaries Generated by Different Approaches Given the Code Snippets

Example (1) in Java:

```
public static boolean isFile(String path){
    File f = new File(path);
    return f.isFile();
}
```

Vanilla Transformer: checks if the given path is a file object, is a directory it can be read. no distinction is considered exceptions
CodeTransformer: checks if the given file is a file
NeuralCodeSum: checks if file exists
GREAT: checks if the given path is a file
SG-Trans: checks if the given path is a file
Ground truth: checks if the given path is a file

Example (2) in Python:

```
def print_bucket_acl_for_user(bucket_name, user_email):
    storage_client = storage.Client()
    bucket = storage_client.bucket(bucket_name)
    bucket.acl.reload()
    roles = bucket.acl.user(user_email).get_roles()
    print roles
```

Vanilla Transformer: removes a user from the access control list
CodeTransformer: sets a the user from to to.
NeuralCodeSum: prints out a user access control list
GREAT: prints out a user access control list
SG-Trans: prints out a buckets access control list for a given user
Ground truth: prints out a buckets access control list for a given user

Example (3) in Python:

```
def token_urlsafe(nbytes=None):
    tok = token_bytes(nbytes)
    return base64.urlsafe_b64encode(tok).rstrip('=').decode('ascii')
```

Vanilla Transformer: generates a token
CodeTransformer: decodes a unicode string string string if
NeuralCodeSum: construct a random text string.
GREAT: generates a token identifier
SG-Trans: return a random url-safe string.
Ground truth: return a random url-safe text string.

The following examples are only from the test set and do not exist in the training set.

6.2 Duplicate Data in the Java Dataset

During our experimentation, we find that there are duplicate data in the Java dataset, which may adversely affect the model performance [2]. As for the Python dataset, there is no duplication across different training, validation, and test set. As shown in Table 11, there are 23.3% and 23.6% duplicate data in the validation set and the test set, respectively. To evaluate the impact of the data duplication on the proposed model, we remove the duplicate data across the training, validation, and test sets. We choose the two strongest baselines, NeuralCodeSum and GREAT, for comparison. The results after deduplication are shown in Table 12. As can be seen, all models present a dramatic decrease on the deduplicated dataset. Nevertheless, the proposed SG-Trans still outperforms GREAT on the BLEU-4, ROUGE-L, and METEOR metrics, by 3.3%, 2.7%, and 3.3%, respectively.

6.3 Analysis of the Hierarchical Structure-Variant Attention Mechanism

The hierarchical structure-variant attention mechanism in SG-Trans aims at rendering the model focus on local structures at shallow layers and global structure at deep layers. In this section, we

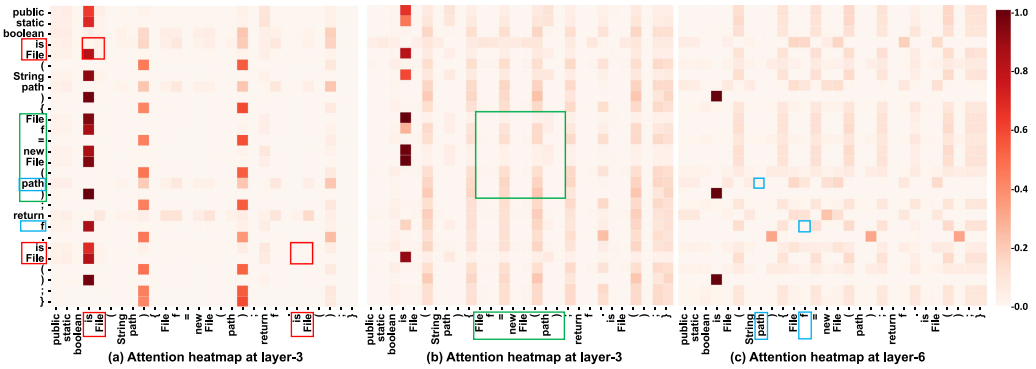


Fig. 9. Heat map visualization of self-attention scores of NeuralCodeSum. The rectangles with red edges, green edges, and blue edges indicate the tokens belonging to the same original token, the same statement, or containing dataflow relation, respectively.

Table 11. Duplicate Data in the Java Dataset

	Validation Set	Test set
Total data	8,714	8,714
Duplicate data	2,028 (23.3%)	2,059 (23.6%)

analyze whether the mechanism can assist SG-Trans in learning the hierarchical information. We visualize the distributions of attention scores corresponding to the relative token distances for the shallow layer – Layer one, middle layer – Layer four, and one deep layer – Layer seven, respectively. For each relative token distance ι, its attention distribution Y_ι is computed as Equation (19).

$$Y_\iota = \frac{\sum_{i=1}^{N} \text{attention}(i, i+\iota) + \text{attention}(i, i-\iota)}{\sum_{j=1}^{S}\sum_{i=1}^{N} \text{attention}(i, i+j) + \text{attention}(i, i-j)}, \tag{19}$$

where N denotes the number of tokens and S denotes the longest relative distance for analysis ($S = 10$ in our analysis). The attention(i, j) denotes the attention score of token x_i to x_j ($1 \le j \le N$). The attention scores reflect whether the model focuses on local information or global information. We choose the relational Transformer GREAT for comparison since it also involves structural information but is not designed hierarchically. The visualization is depicted in Figure 10. For GREAT, as shown in Figure 10(a), we find that the attention distributions across different layers present similar trends, that is, they all tend to focus on different token distances uniformly. For SG-Trans, as shown in Figure 10(b), we observe that the three layers pay various attentions to tokens of different relative distances. The shallow layer (Layer one) focuses more on tokens with short relative distances. In the middle layer (Layer four), the attention distribution among different distances is more balanced, which indicates that the model pays increasingly more attention to global tokens with the layer depth being increased. For the deep layer (Layer seven), the attention scores for tokens of long distances are larger than those of short distances, meaning that the model tends to focus on long-range relations in the deep layer. The results demonstrate that the hierarchical attention mechanism in SG-Trans is beneficial for the model to capture the hierarchical structural information that cannot be easily learned by the relational Transformer.

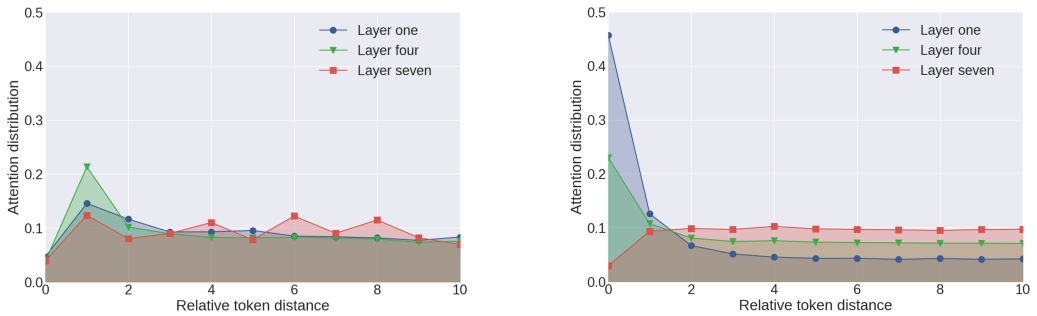
6.4 Difference with Relational Transformers

There are two main differences between SG-Trans and the relational Transformers [26, 70]:

Table 12. Comparison Results on the De-Duplicated Java Dataset

Approach	BLEU-4	ROUGE-L	METEOR
NeuralCodeSum	29.37 (↓34.95%)	41.62 (↓24.11%)	19.98 (↓27.24%)
GREAT	29.49 (↓34.52%)	41.84 (↓23.12%)	20.15 (↓25.79%)
SG-Trans	**30.46** (↓33.74%)	**42.97** (↓23.07%)	**20.82** (↓25.32%)

Data listed within brackets are computed drop rates compared with the results on original Java dataset.



(a) GREAT. (b) SG-Trans.

Fig. 10. Attention distributions regarding relative token distance for (a) GREAT and (b) SG-Trans. The horizontal axis represents the relative distance to the current token, and the vertical axis denotes the normalized attention scores along with relative distances in one layer.

(1) **Strategy in incorporating structural information.** Compared with GREAT [26] and CodeTransformer [70], which use learnable bias and sinusoidal encoding function to encode the structure information, respectively, SG-Trans incorporates the local and global structure information with different strategies, for example, introducing the local information with a hard mask. The results in Tables 2 and 4 show the effectiveness of the structural incorporation strategy in SG-Trans.
(2) **Design of hierarchical structure-variant attention mechanism.** In SG-Trans, a hierarchical attention mechanism is designed to assist the model in learning the hierarchical information, whereas the relational Transformers do not have such a design. Both the ablation study in Section 5.2 and the discussion in Section 6.3 demonstrate the benefits of this design.

6.5 Difference with GraphCodeBERT

SG-Trans takes dataflow information, which is similar to GraphCodeBERT [21]. However, the two methods are different as follows.

(1) **Role of dataflow.** GraphCodeBERT mainly uses the dataflow in two ways: (1) filtering the irrelevant signals in the variable sequence and (2) guiding the two pretraining tasks, including edge prediction and node alignment. Nevertheless, SG-Trans directly uses dataflow information to help the attention mechanism better capture the data dependency relations in source code.
(2) **Incorporation way of dataflow.** We integrate the dataflow information in a different way from GraphCodeBERT. GraphCodeBERT utilizes a sparse masking mechanism to mask the attention relations between the tokens without data dependency. SG-Trans retains the

Table 13. Comparison of the Cost of Different Models

	GPU memory usage	Training time	Preprocessing time
NeuralCodeSum	8,729 M	30.4 h	-
CodeTransformer	8,573 M	211.5 h	66.1 ms
SG-Trans	8,509 M	29.9 h	3.8 ms
CodeBERT	17,959 M	-	-

The "_" under the preprocessing time of NeuralCodeSum and CodeBERT denotes that the approaches do not need preprocessing. The "_" under the training time of CodeBERT denotes that we do not reproduce the pretraining stage due to the limitation of computing resources.

attention relations for the tokens without data dependency and also highlights the dataflow dependency through our designed dataflow-guided self-attention.

(3) **Targets of the proposed model.** The targets of the two models are different. GraphCodeBERT is proposed to utilize the inherent structure of code to facilitate the pretraining stage. SG-Trans mainly focuses on the task without large amounts of source code available and uses the incorporated code structure to alleviate dependency on the training data.

6.6 Analysis of the Complexity of SG-Trans

SG-Trans incorporates structural information based on three types of relations between code tokens. Compared with the baseline approaches, such as NeuralCodeSum, SG-Trans involves more types of relations, which could lead to an increase in the model complexity and subsequently impacting its applicability to other programming languages. To investigate to what extent SG-Trans introduces extra complexity, we conduct analysis of the cost of SG-Trans.

We first compare the cost of SG-Trans with Transformer-based baselines, including NeuralCodeSum and CodeTransformer, and a pretraining model CodeBERT, in terms of the GPU memory usage, training time cost, and preprocessing time cost. The comparison is implemented on the same server with a single Tesla V100 GPU by training on the Java dataset with 32 batch size. The results are shown in Table 13. As can be seen, the GPU memory usage and training time cost of SG-Trans are the lowest among all the approaches. Since SG-Trans does not involve the relative positive embedding, both its GPU memory usage and training time cost are even lower than NeuralCodeSum. Table 13 also shows that CodeBERT requires the highest memory usage, restricting its application to low-resource devices. With respect to the preprocessing time cost for one sample, since SG-Trans does not need to calculate the complex features used in CodeTransformer, such as shortest path length and personalized PageRank, it takes only about 3.8 ms, which is significantly faster than CodeTransformer. The results indicate that the code structure properties used in SG-Trans do not incura a larger cost than the baselines.

For the application of SG-Trans to other programming languages, the main barrier lies in the dataflow extraction procedure. In this work, we follow the main idea of Wang et al. [59] and consider only the common dataflow information, which is generally similar for different programming languages. The common dataflow information includes sequential dataflow relations and three types of non-sequential dataflow relations, such as "*if*" statements and "*for*" and "*while*" loops. The sequential dataflow relations can be easily extracted by identifying the variables for any programming language. For the non-sequential dataflow relations, the extraction procedure of different programming languages is also similar because the AST parser tree-sitter[2] can parse the dataflow relations of different languages into almost the same tree structure. Thus, it is convenient to extend SG-Trans to other popular languages.

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

Table 14. Comparison Results with CodeBERT

Approach	Java			Python		
	BLEU-4	METEOR	ROUGE-L	BLEU-4	METEOR	ROUGE-L
CodeBERT [15]	14.93	9.23	30.43	16.70	9.68	30.31
CodeBERT+fine-tune [15]	44.40	28.33	55.56	32.04	20.77	**47.45**
SG-Trans	45.89	27.85	55.79	33.04	20.52	47.01
SG-Trans$_{large}$	**46.27**	**28.37**	**56.30**	**33.53**	**20.87**	47.42

"CodeBERT" represents the CodeBERT approach without fine-tuning. "SG-Trans$_{large}$" represents SG-Trans with the same model settings as CodeBERT, 10 encoder layers and hidden size of 768.

6.7 Comparison with CodeBERT

Many pretraining models [15, 21] have been proposed recently, which can be adopted for source code summarization. Thus, we also compare the performance of SG-Trans with the most typical pretraining model, CodeBERT. We also train SG-Trans under the same model size as CodeBERT and denote it as SG-Trans$_{large}$ for further comparison.

As shown in Table 14, without fine-tuning, CodeBERT shows the worst performance among all of the approaches. After enough fine-tuning, CodeBERT improves a lot and even outperforms SG-Trans on some metrics, for example., the METEOR score on the Java dataset. However, it should be noted that the encoder layer number and hidden size of CodeBERT are much larger than SG-Trans. The numbers of encoder layers and hidden size of CodeBERT are 10 and 768, respectively, whereas SG-Trans has only 8 encoder layers and a hidden size of 512. For fair comparison, we also train SG-Trans with the same model settings as CodeBERT, denoted as SG-Trans$_{large}$. As shown in Table 14, SG-Trans$_{large}$ obtains the best performance for almost all of the metrics. On the Java dataset, SG-Trans$_{large}$ outperforms CodeBERT+fine-tune by 4.2% and 1.3% in terms of BLEU-4 and ROUGE-L, respectively. On the Python dataset, SG-Trans$_{large}$ is only slightly lower than CodeBERT+fine-tune on ROUGE-L but obviously outperforms CodeBERT+fine-tune by 4.7% in terms of the BLEU-4 score. The results demonstrate that SG-Trans is more effective than CodeBERT, even with accessing limited data.

6.8 Threats to Validity

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

(1) **The generalizability of our results.** We use two large public datasets, which include 87,136 Java and 92,545 Python code-summary pairs, following the prior research [1, 60, 67]. The limited types of programming languages may hinder the scalability of the proposed SG-Trans. In our future work, we will experiment with more large-scale datasets with different programming languages.

(2) **More code structure information could be considered.** SG-Trans takes only the token-level and statement-level syntactic structure and dataflow structure into consideration since it has been previously demonstrated that the dataflow information is more effective than AST and CFG during code representation learning [21]. Nevertheless, other code structural properties, such as AST and CFG, could be potentially useful for boosting the model performance. In the future, we will explore the use of more structural properties in SG-Trans.

(3) **Biases in human evaluation.** We invited 10 participants to evaluate the quality of 200 randomly selected code-summary pairs. The results of human annotations can be impacted by the participants' programming experience and their understanding of the evaluation metrics. To mitigate the bias of human evaluation, we ensure that the 10 participants are all software developers with at least 4 years of programming experience, and each code-summary pair

was evaluated by 3 participants. Summaries generated by different approaches were also randomly shuffled in order to eliminate order bias. In the future, we will expand the pool of human participants and will increase the size of the evaluation set.

7 RELATED WORK

In this section, we elaborate on two threads of related work: source code summarization and code representation learning.

7.1 Source Code Summarization

There has been extensive research in source code summarization, including template-based approaches [41, 43, 52], information retrieval–based approaches [24, 44, 62] and deep learning–based approaches [4, 27, 31]. Among these categories, deep learning–based methods have achieved the greatest success and have become the most popular in recent years. These methods specifically formulate the code summarization task as a neural machine translation (NMT) problem and adopt state-of-the-art NMT frameworks to improve performance. In this section, we focus on deep learning–based methods and introduce them by category. We also list an overview of the category of related works in Table 15.

RNN-Based Models: Iyer et al. [31] first propose CODE-NN, a Long Short Term Memory (LSTM) network with attention to generate code summaries from code snippets. In order to achieve more accurate code modeling, later researchers then introduced more structural and syntactic information to the deep learning models. Hu et al. [27] propose a structure-based traversal (SBT) method to traverse AST and process the AST nodes into sequences that can be fed into an RNN encoder. The authors of [28] hold the view that code API carries vital information about the functionality of the source code and incorporates the API knowledge by adding an API Sequences Encoder.

Tree/GNN-Based Models: To leverage the tree structures of AST, a multi-way Tree-LSTM [50] is proposed to directly model the code structures. For more fine-grained intra-code relationship exploitation, many works also incorporate code-related graphs and GNN to boost performance. Fernandes et al. [16] build a graph from source code and extract nodes features with a gated GNN while LeClair et al. [35] directly obtain code representation from AST with convolutional GNNs. To help the model capture more global interactions among nodes, a recent work [39] proposes a hybrid GNN that fuses the information from static and dynamic graphs via hybrid message passing.

Transformer-Based Models: With the rise of Transformer in the NMT task domain, Ahmad et al. [1] equip Transformer with copy attention and relative position embedding for better mapping of the source code to their corresponding natural language summaries. To leverage the code structure information into Transformer, Hellendoorn et al. [26] propose GREAT, which encodes structural information into self-attention with adding a learnable edge-type related bias. Another work proposed by Zügner et al. [70] focuses on multilingual code summarization and proposes to build upon language-agnostic features such as source code and AST-based features. Wu et al. [63] propose Structure-induced Self-Attention to incorporate multi-view structure information into self-attention mechanism. To capture both the sequential and structural information, a recent work [11] applies graph convolution to obtain structurally encoded node representations and passes sequences of the graph-convolutioned AST nodes into Transformer. Another recent work [18] proposes to utilize AST relative positions to augment the structural correlations between code tokens.

Information Retrieval Auxiliary Methods: Information retrieval auxiliary methods utilize information retrieval and large-scale code repositories to help the model generate high-quality summaries. Zhang et al. [67] propose to improve code summarization with the help of two retrieved

Table 15. An Overview of Existing Works Related to Code Summarization

Approach	Input code property	RNN	Tree/GNN	Transformer	IR	MT
CODE-NN [31]	–	✓				
DeepCom [27]	AST	✓				
Code2Seq [4]	AST	✓				
API+Code [28]	API Information	✓				
Dual Model [60]	–	✓				✓
Rencos [67]	–	✓			✓	
Re^{2}Com [61]	–	✓			✓	
Tree-LSTM [50]	AST		✓			
Code+GNN [35]	AST		✓			
HGNN [39]	CPG		✓		✓	
NeuralCodeSum [1]	–			✓		
DMACOS [65]	–			✓		✓
GREAT [26]	Multi relations in code			✓		
CodeTransformer [70]	AST			✓		
Transformer+GNN [11]	AST		✓	✓		

The "IR" and "MT" denote information retrieval auxiliary methods and multi-task learning strategy, respectively.

codes using syntactic and semantic similarity. To help the model generate more important but low-frequency tokens, Wei et al. [61] further use the existing comments of retrieved code snippets as exemplars to guide the summarization.

Multi-Task Learning Strategy: Some researchers try to exploit commonalities and differences across code-related tasks to further improve code summarization. Wei et al. [60] use a dual learning framework to apply the relations between code summarization and code generation and improve the performance of both tasks. A recent work [65] used method name prediction as an auxiliary task and designed a multi-task learning approach to improve code summarization performance.

Compared with existing work, our proposed model focuses on improving the Transformer architecture for source code to make it better incorporate both local and global code structures. Other improvement methods —such as information retrieval and multi-task learning, which are orthogonal to our work — are not the research target of this article.

7.2 Code Representation Learning

Learning high-quality code representations is of vital importance for deep learning–based code summarization. Apart from the practices for code summarization discussed earlier, there also exist other code representation learning methods that lie in similar task domains, such as source code classification, code clone detection, and commit message generation. For example, the ASTNN model proposed by Zhang et al. [68] splits large ASTs into sequences of small statement trees, which are further encoded into vectors as source code representations. This model is further applied on code classification and code clone detection. Alon et al. [4] present CODE2SEQ, which represents the code snippets by sampling certain paths from the ASTs. Gu et al. [20] propose to encode statement-level dependency relations through the Program Dependency Graph (PDG). Comparatively, this research on the model architecture improvement for code representation learning is relevant to us but mainly focuses on other code-related tasks, such as code classification and code clone detection.

Recently, inspired by the successes of pretraining techniques in the natural language processing field, Feng et al. [15] and Guo et al. [21] also apply pretraining models on learning source code and achieve empirical improvements on a variety of tasks. To extend the code representations to

characterize programs' functionalities, the authors of [32] further enrich the pretraining tasks to learn task-agnostic semantic code representations from textually divergent variants of source programs via contrastive learning. Their target of the research about pretraining differs significantly from ours; they mainly focus on learning from a large-scale dataset in a self-supervised way. Our research concentrates on improving the quality of generated summaries with limited training data in a supervised way.

8 CONCLUSION

In this article, we present SG-Trans, a Transformer-based architecture with structure-guided self-attention and hierarchical structure-variant attention. SG-Trans can attain better modeling of code structural information, including local structure at token level and statement level, and global structure, that is, dataflow. The evaluation on two popular benchmarks suggests that SG-Trans outperforms competitive baselines and achieves state-of-the-art performance on code summarization. For future work, we plan to extend the use of our model to other task domain and possibly build up more accurate code representations for general usage.

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL'20), Online, July 5–10, 2020*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (Eds.). Association for Computational Linguistics, 4998–5007.

[2] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019, Athens, Greece, October 23–24, 2019*, Hidehiko Masuhara and Tomas Petricek (Eds.). ACM, 143–153.

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *6th International Conference on Learning Representations (ICLR'18), Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *7th International Conference on Learning Representations (ICLR'19), New Orleans, LA, May 6–9, 2019*. OpenReview.net.

[5] Bang An, Jie Lyu, Zhenyi Wang, Chunyuan Li, Changwei Hu, Fei Tan, Ruiyi Zhang, Yifan Hu, and Changyou Chen. 2020. Repulsive attention: Rethinking multi-head attention as Bayesian inference. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP'20), Online, November 16–20, 2020*, Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 236–255.

[6] Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer normalization. *CoRR* abs/1607.06450 (2016).

[7] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization@ACL 2005, Ann Arbor, Mi, June 29, 2005*, Jade Goldstein, Alon Lavie, Chin-Yew Lin, and Clare R. Voss (Eds.). Association for Computational Linguistics, 65–72.

[8] Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275* (2017).

[9] Jie-Cherng Chen and Sun-Jen Huang. 2009. An empirical analysis of the impact of software development problem factors on software maintainability. *J. Syst. Softw.* 82, 6 (2009), 981–992.

[10] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP'14), October 25–29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, Alessandro Moschitti, Bo Pang, and Walter Daelemans (Eds.). ACL, 1724–1734.

[11] YunSeok Choi, JinYeong Bak, CheolWon Na, and Jee-Hyong Lee. 2021. Learning sequential and structural information for source code summarization. In *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1–6, 2021 (Findings of ACL'21)*, Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (Eds.), Vol. ACL/IJCNLP 2021. Association for Computational Linguistics, 2842–2851.

[12] Milan Cvitkovic, Badal Singh, and Animashree Anandkumar. 2019. Open vocabulary learning on source code with a graph-structured cache. In *Proceedings of the 36th International Conference on Machine Learning (ICML'19), 9–15 June 2019, Long Beach, CA (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, 1475–1485.

[13] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia Marçal de Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information (SIGDOC'05), Coventry, UK, September 21–23, 2005*, Scott R. Tilley and Robert M. Newman (Eds.). ACM, 68–75.

[14] Akiko Eriguchi, Kazuma Hashimoto, and Yoshimasa Tsuruoka. 2016. Tree-to-sequence attentional neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL'16), August 7–12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 823–833.

[15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings (EMNLP'20), Online Event, 16–20 November 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547.

[16] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured neural summarization. In *7th International Conference on Learning Representations (ICLR'19), New Orleans, LA, May 6–9, 2019*. OpenReview.net.

[17] Golara Garousi, Vahid Garousi-Yusifoglu, Günther Ruhe, Junji Zhi, Mahmood Moussavi, and Brian Smith. 2015. Usage and usefulness of technical software documentation: An industrial case study. *Inf. Softw. Technol.* 57 (2015), 664–682.

[18] Zi Gong, Cuiyun Gao, Yasheng Wang, Wenchao Gu, Yun Peng, and Zenglin Xu. 2022. Source code summarization with structural relative position guided transformer. *CoRR* abs/2202.06521 (2022).

[19] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O. K. Li. 2016. Incorporating copying mechanism in sequence-to-sequence learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL'16), August 7–12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 1631–1640.

[20] Wenchao Gu, Zongjie Li, Cuiyun Gao, Chaozheng Wang, Hongyu Zhang, Zenglin Xu, and Michael R. Lyu. 2021. CRaDLe: Deep code retrieval based on semantic dependency learning. *Neural Networks* 141 (2021), 385–394.

[21] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2020. GraphCodeBERT: Pre-training code representations with data flow. *CoRR* abs/2009.08366 (2020).

[22] Qipeng Guo, Xipeng Qiu, Pengfei Liu, Yunfan Shao, Xiangyang Xue, and Zheng Zhang. 2019. Star-transformer. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, June 2–7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 1315–1325.

[23] Qipeng Guo, Xipeng Qiu, Pengfei Liu, Xiangyang Xue, and Zheng Zhang. 2020. Multi-scale self-attention for text classification. In *34th AAAI Conference on Artificial Intelligence (AAAI'20), 32nd Innovative Applications of Artificial Intelligence Conference (IAAI'20), 10th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI'20), New York, NY, February 7–12, 2020*. AAAI Press, 7847–7854.

[24] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE'10), Cape Town, South Africa, 1–8 May 2010*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 223–226.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16), Las Vegas, NV, June 27–30, 2016*. IEEE Computer Society, 770–778.

[26] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global relational models of source code. In *8th International Conference on Learning Representations (ICLR'20), Addis Ababa, Ethiopia, April 26–30, 2020*. OpenReview.net.

[27] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension (ICPC'18), Gothenburg, Sweden, May 27–28, 2018*, Foutse Khomh, Chanchal K. Roy, and Janet Siegmund (Eds.). ACM, 200–210.

[28] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred API knowledge. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI'18), July 13–19, 2018, Stockholm, Sweden*, Jérôme Lang (Ed.). ijcai.org, 2269–2275.

[29] Xing Hu, Xin Xia, David Lo, Zhiyuan Wan, Qiuyuan Chen, and Tom Zimmermann. 2022. Practitioners' expectations on automated code comment generation. In *Proceedings of the 44th ACM/IEEE International Conference on Software Engineering (ICSE'22)*.

[30] Chidubem Iddianozie and Gavin McArdle. 2020. Improved graph neural networks for spatial networks using structure-aware sampling. *ISPRS Int. J. Geo Inf.* 9, 11 (2020), 674.

[31] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL'16), August 7–12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics.

[32] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E. Gonzalez, and Ion Stoica. 2020. Contrastive code representation learning. *CoRR* abs/2007.04973 (2020).

[33] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17), Urbana, IL, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 135–146.

[34] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code != big vocabulary: Open-vocabulary models for source code. In *42nd International Conference on Software Engineering (ICSE'20), Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1073–1085.

[35] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *28th International Conference on Program Comprehension (ICPC '20)*, Seoul, Republic of Korea, *July 13–15, 2020*. ACM, 184–195.

[36] Yuding Liang and Kenny Qili Zhu. 2018. Automatic generation of text descriptive comments for code blocks. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence, New Orleans, LA, February 2–7, 2018*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 5229–5236.

[37] Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*. Association for Computational Linguistics, Barcelona, Spain, 74–81.

[38] Chia-Wei Liu, Ryan Lowe, Iulian Serban, Michael Noseworthy, Laurent Charlin, and Joelle Pineau. 2016. How NOT to evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP'16), Austin, TX, November 1–4, 2016*, Jian Su, Xavier Carreras, and Kevin Duh (Eds.). The Association for Computational Linguistics, 2122–2132.

[39] Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2021. Retrieval-augmented generation for code summarization via hybrid GNN. In *9th International Conference on Learning Representations (ICLR'21), Virtual Event, Austria, May 3–7, 2021*. OpenReview.net.

[40] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. 2019. Automatic generation of pull request descriptions. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19), San Diego, CA, November 11–15, 2019*. IEEE, 176–188.

[41] Paul W. McBurney and Collin McMillan. 2016. Automatic source code summarization of context for Java methods. *IEEE Trans. Software Eng.* 42, 2 (2016), 103–119.

[42] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I know what you did last summer: An investigation of how developers spend their time. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension (ICPC'15), Florence, Italy, May 16–24, 2015*, Andrea De Lucia, Christian Bird, and Rocco Oliveto (Eds.). IEEE Computer Society, 25–35.

[43] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori L. Pollock, and K. Vijay-Shanker. 2013. Automatic generation of natural language summaries for Java classes. In *IEEE 21st International Conference on Program Comprehension (ICPC'13), San Francisco, CA, 20–21 May, 2013*. IEEE Computer Society, 23–32.

[44] Dana Movshovitz-Attias and William W. Cohen. 2013. Natural language models for predicting programming comments. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (ACL'13), 4–9 August 2013, Sofia, Bulgaria, Volume 2: Short Papers*. The Association for Computer Linguistics, 35–40.

[45] Lun Yiu Nie, Cuiyun Gao, Zhicong Zhong, Wai Lam, Yang Liu, and Zenglin Xu. 2020. Contextualized code representation learning for commit message generation. *CoRR* abs/2007.06934 (2020).

[46] Kyosuke Nishida, Itsumi Saito, Kosuke Nishida, Kazutoshi Shinoda, Atsushi Otsuka, Hisako Asano, and Junji Tomita. 2019. Multi-style generative reading comprehension. In *Proceedings of the 57th Conference of the Association for Computational Linguistics (ACL'19), Florence, Italy, July 28-August 2, 2019, Volume 1: Long Papers*, Anna Korhonen, David R. Traum, and Lluís Màrquez (Eds.). Association for Computational Linguistics, 2273–2284.

[47] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6–12, 2002, Philadelphia, PA*. ACL, 311–318.

[48] Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL'17), Vancouver, Canada, July 30-August 4, Volume 1: Long Papers*, Regina Barzilay and Min-Yen Kan (Eds.). Association for Computational Linguistics, 1073–1083.

[49] Lin Shi, Hao Zhong, Tao Xie, and Mingshu Li. 2011. An empirical study on evolution of API documentation. In *Fundamental Approaches to Software Engineering - 14th International Conference (FASE'11), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'11), Saarbrücken, Germany, March 26-April 3, 2011. Proceedings (Lecture Notes in Computer Science)*, Dimitra Giannakopoulou and Fernando Orejas (Eds.), Vol. 6603. Springer, 416–431.

[50] Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. Automatic source code summarization with extended tree-LSTM. In *International Joint Conference on Neural Networks (IJCNN'19), Budapest, Hungary, July 14–19, 2019*. IEEE, 1–8.

[51] Kai Song, Kun Wang, Heng Yu, Yue Zhang, Zhongqiang Huang, Weihua Luo, Xiangyu Duan, and Min Zhang. 2020. Alignment-enhanced transformer for constraining NMT with pre-specified translations. In *34th AAAI Conference on Artificial Intelligence (AAAI'20), 32nd Innovative Applications of Artificial Intelligence Conference (IAAI'20), 10th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI'20), New York, NY, February 7–12, 2020*. AAAI Press, 8886–8893.

[52] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori L. Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for Java methods. In *25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10), Antwerp, Belgium, September 20–24, 2010*, Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto (Eds.). ACM, 43–52.

[53] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. 2020. A human study of comprehension and code summarization. In *28th International Conference on Program Comprehension (ICPC '20), Seoul, Republic of Korea, July 13–15, 2020*. ACM, 2–13.

[54] Zoltán Gendler Szabó. 2020. Compositionality. In *The Stanford Encyclopedia of Philosophy*, Edward N. Zalta (Ed.), Metaphysics Research Lab, Stanford University.

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4–9, 2017, Long Beach, CA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008.

[56] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. 2019. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. In *Proceedings of the 57th Conference of the Association for Computational Linguistics (ACL'19), Florence, Italy, July 28 - August 2, 2019, Volume 1: Long Papers*, Anna Korhonen, David R. Traum, and Lluís Màrquez (Eds.). Association for Computational Linguistics, 5797–5808.

[57] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18), Montpellier, France, September 3–7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 397–407.

[58] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL'20), Online, July 5–10, 2020*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (Eds.). Association for Computational Linguistics, 7567–7578.

[59] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*. IEEE, 261–271.

[60] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019 (NeurIPS'19), December 8–14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 6559–6569.

[61] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and refine: Exemplar-based neural comment generation. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20), Melbourne, Australia, September 21–25, 2020*. IEEE, 349–360. https://doi.org/10.1145/3324884.3416578

[62] Edmund Wong, Taiyue Liu, and Lin Tan. 2015. CloCom: Mining existing source code for automatic comment generation. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'15), Montreal, QC, Canada, March 2–6, 2015*, Yann-Gaël Guéhéneuc, Bram Adams, and Alexander Serebrenik (Eds.). IEEE Computer Society, 380–389.

[63] Hongqiu Wu, Hai Zhao, and Min Zhang. 2021. Code summarization with structure-induced transformer. In *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1–6, 2021 (Findings of ACL'21)*, Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (Eds.), Vol. ACL/IJCNLP 2021. Association for Computational Linguistics, 1078–1090.

[64] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring program comprehension: A large-scale field study with professionals. *IEEE Trans. Software Eng.* 44, 10 (2018), 951–976.

[65] Rui Xie, Wei Ye, Jinan Sun, and Shikun Zhang. 2021. Exploiting method names to improve code summarization: A deliberation multi-task learning approach. In *29th IEEE/ACM International Conference on Program Comprehension (ICPC'21), Madrid, Spain, May 20–21, 2021*. IEEE, 138–148.

[66] Jingyi Zhang, Masao Utiyama, Eiichiro Sumita, Graham Neubig, and Satoshi Nakamura. 2018. Guiding neural machine translation with retrieved translation pieces. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT'18), New Orleans, LA, June 1–6, 2018, Volume 1 (Long Papers)*. 1325–1335.

[67] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *42nd International Conference on Software Engineering (ICSE'20)*, Seoul, South Korea, 27 June–19 July, 2020, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1385–1397.

[68] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19), Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 783–794.

[69] Xiangyu Zhao, Longbiao Wang, Ruifang He, Ting Yang, Jinxin Chang, and Ruifang Wang. 2020. Multiple knowledge syncretic transformer for natural dialogue generation. In *The Web Conference 2020 (WWW'20), Taipei, Taiwan, April 20–24, 2020*, Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen (Eds.). ACM / IW3C2, 752–762.

[70] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-agnostic representation learning of source code from structure and context. In *9th International Conference on Learning Representations (ICLR'21) Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Received 6 June 2021; revised 19 February 2022; accepted 24 February 2022