Code Structure Guided Transformer for Source Code Summarization

SHUZHENG GAO, Harbin Institute of Technology, Weihai, China
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
LUN YIU NIE, Tsinghua University, China
XIN XIA, Software Engineering Application Technology Lab, Huawei, China
MICHAEL R. LYU, The Chinese University of Hong Kong, Hong Kong, China

Code summaries, also called code comments, 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 paper, 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., data flow 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 1.8% and 2.9% in terms of BLEU-4 score, a metric widely used for measuring generation quality, respectively on two benchmark datasets.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

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

ACM Reference Format:
Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lun Yiu Nie, Xin Xia, and Michael R. Lyu. 2018. Code Structure Guided Transformer for Source Code Summarization. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Program comprehension is crucial for developers during software development and maintenance, and developers' cognitive efforts in comprehending programs can be significantly minimized by a text summary accompanying the source code [19]. Source code summarization, also known as code comment generation, thus aims at automatically generating a concise text description of the functionality of a program.

*Corresponding author.

Manuscript submitted to ACM

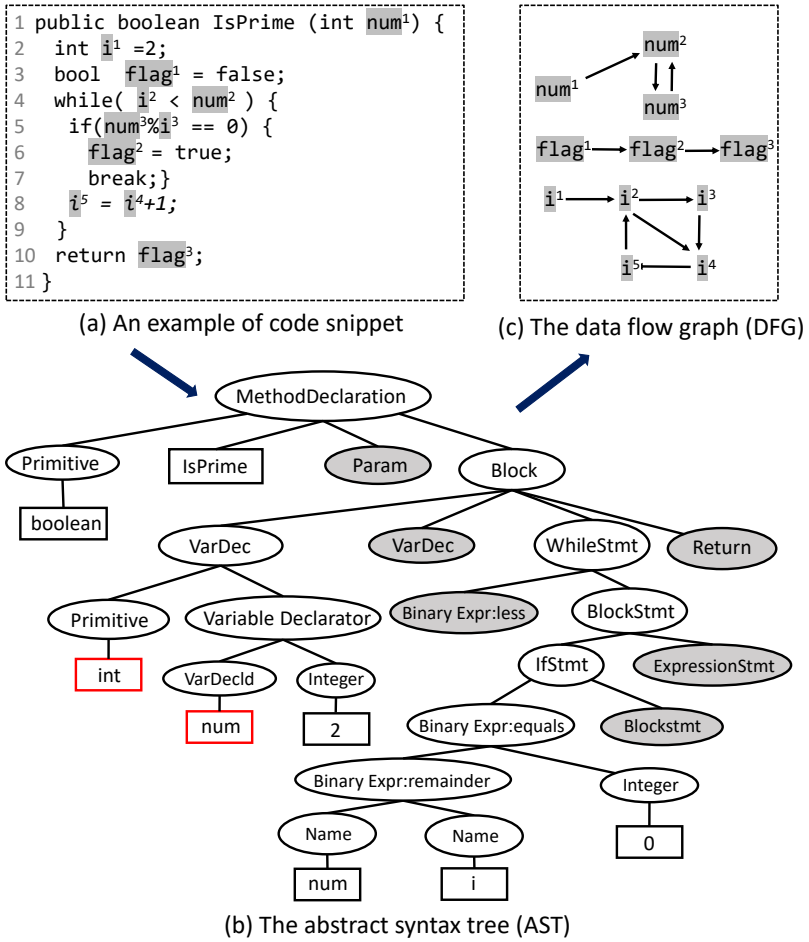


Fig. 1. An example of Java code snippet (a), with the corresponding AST (b) and DFG (c) illustrated. Entities in grey ellipse in (b) mean unexpanded branches. The arrows in the DFG represent the relations of sending/receiving messages between the variables (highlighted in grey in the code).

Existing leading approaches have demonstrated the benefits of integrating code structural properties such as Abstract Syntax Trees (ASTs) [4, 19] into deep learning techniques for the task. An example of 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, 19] or sequences of small statement trees split from large ASTs [38, 50]. The sequences are usually fed into a Recurrent Neural Network (RNN)-based sequence-to-sequence network for generating a natural language summary [19, 27]. However, due to the deep nature of ASTs, the 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 [13, 26]. Although these GNN-based approaches can capture the long-range relations between code tokens, they are proven sensitive to local information and ineffective in capturing the global structure [21]. Take the AST in Figure 1 (b) as an example, token nodes "*int*" and "*num*" (highlighted with red boxes) are in the same statement but more than one hop exists between them.

Recent study [1] shows that Transformer model [42], which can capture long-range dependencies with its self-attention mechanism, outperforms other deep learning approaches for the task. However, how to effectively integrate the code structure information into Transformer is still unexplored. One challenge is that since the position encoding in Transformer already learns the dependency relations between code tokens, trivial integration of the structure information may not bring an improvement for the task [1]. Besides, an issue of Transformer is that its attention is purely data-driven [16]. 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, 43].

To overcome the above challenges in this paper, we propose a novel model named SG-Trans, i.e., 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, i.e., data flow graph (DFG), then represent them as adjacent matrices before injecting 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 [16, 41], 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 1.8% and 2.9% in terms of BLEU-4 on two Java and Python benchmark datasets, respectively.

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 SG-Trans outperforms the state-of-the-art models.

Paper structure. Section 2 illustrates the background knowledge of the work. Section 3 presents our proposed methodology for source code summarization. Section 4 introduces the experimental setup. Section 5 describes the evaluation results, followed by the discussions in Section 6. Section 7 presents related studies. Finally, Section 8 concludes the paper and outlines future research work.

2 BACKGROUND

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

2.1 Vanilla Transformer

Transformer [42] is a kind of deep self-attention network which has demonstrated its powerful text representation capability in many NLP applications, e.g., machine translation and dialogue generation [39, 51]. It removes the recurrent and convolutional parts in conventional neural networks such as Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN), and is solely based on attention mechanism and multi-layer perceptron (MLP). Transformer follows the sequence-to-sequence [9] architecture with stacked encoders and decoders, with the general architecture

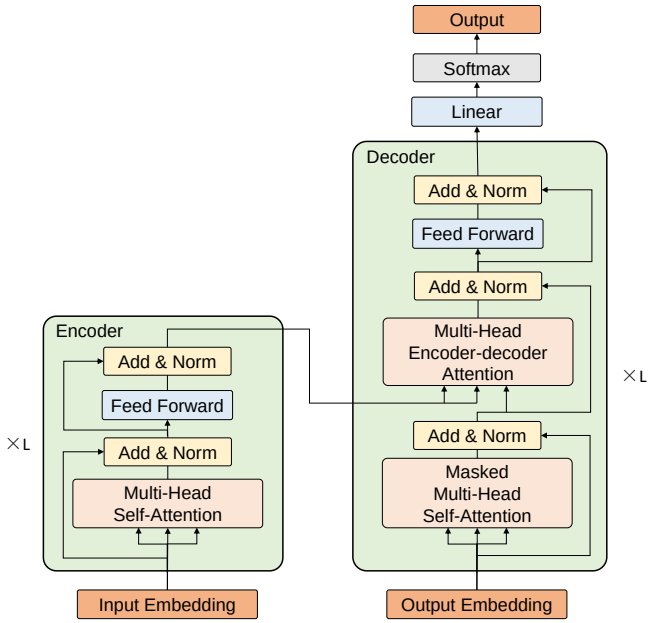


Fig. 2. Architecture of vanilla transformer with L layers in the encoder and decoder, respectively.

illustrated in Figure 2. Each encoder block and decoder block consist of multi-head self-attention sub-layer and feed forward sub-layer. Residual connection [18] 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 details as following.

2.1.1 Multi-Head Self-Attention. Multi-head attention is the key component of Transformer. Given an input sequence $X = (x_1, x_2, ..., x_i, ..., x_n)$ 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, Key vector, and Value vector for each input token by multiplying the input vector with three matrices W^q, W^k, W^v. Then it calculates the attention weight of each token x_i by scoring the query vector q_i against the key vector K of the input sentence. The scoring process is conducted by the scaled dot product, as shown in Equ. (3), where dimension d on the denominator is used to scale the dot product. Softmax is then used to normalize the attention score and outputs attention weight α_i. Finally, the self-attention vector c_i is computed as a weighted sum of the input vectors.

$$Q = XW^q, K = XW^k, V = XW^v, \tag{1}$$

$$\alpha_i = \text{softmax}(\frac{q_i K^T}{\sqrt{d}}) \tag{2}$$

$$c_i = \alpha_i V, \tag{3}$$

where c_i and q_i are the i-th self-attention vector and query vector, respectively, and W^q, W^k, W^v are trainable parameters. The attention weight here can be viewed as a relation measurement between the output vector and input vector, and a higher weight indicates that the output vector is more related to the corresponding input vector.

Instead of performing a single self-attention function, Transformer adopts multi-head self-attention (MHSA) which perform the self-attention function with different parameters in parallel and ensembles the output of each head by

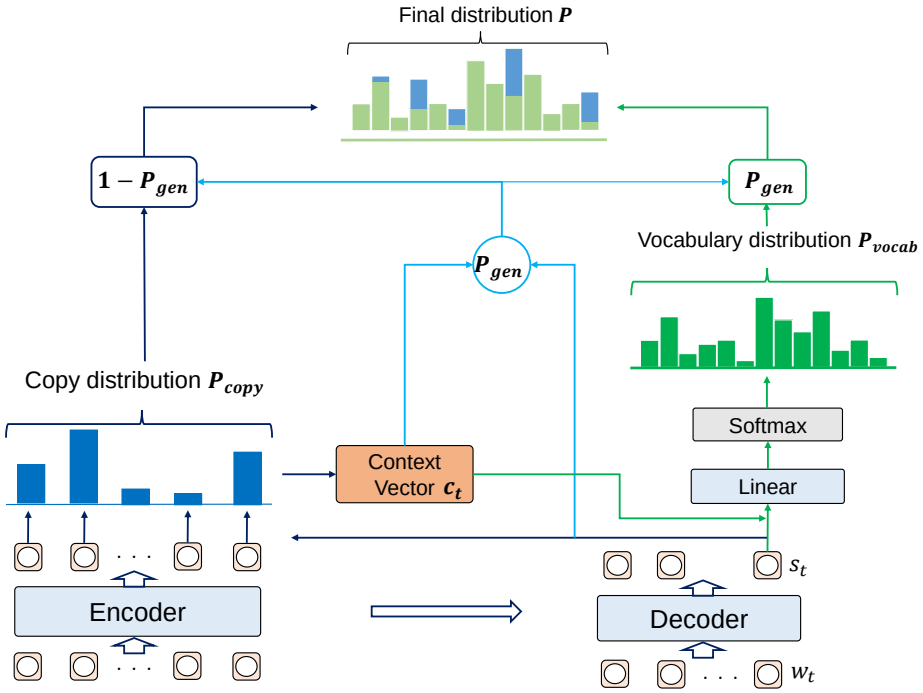


Fig. 3. Architecture of copy mechanism.

concatenating their output together. The MHSA allows the model to jointly attend to information from different representation subspaces at different positions. Formally, the MHSA is computed as following:

$$Q_i = XW_i^q, K_i = XW_i^k, V_i = XW_i^v, \tag{4}$$

$$head_i = \text{softmax}(\frac{Q_i K_i^T}{\sqrt{d}})V_i, \tag{5}$$

$$MHSA(X) = [head_1^l \circ head_2^l \circ \cdots head_i^l \circ \cdots head_h^l]W^O, \tag{6}$$

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. Feed forward network is the only nonlinear part in Transformer. It consists of two linear transformation layer and an ReLU activation function between the two linear layers.

$$FFN(X) = ReLU(XW_1 + b_1)W_2 + b_2, \tag{7}$$

where W_1, W_2, b_1, and b_2 are trainable parameters which are shared on each position.

2.2 Copy Mechanism

Copy mechanism [14] has been widely equipped in text generation models for extracting words from the source sequence into the target sequence during text generation. It has been demonstrated that copy mechanism can alleviate the out-of-vocabulary issue in the code summarization task domain [1, 49]. In this work, we adopt pointer generator [37],

a more popular form of copy mechanism, for the task. Figure 3 illustrates the architecture of the pointer generator model. Specifically, given the input sequence $X = (x_1, x_2, ..., x_n)$, decoder input w_t, decoder hidden state s_t, and context vector c_t computed by attention mechanism in time step t, pointer generator first calculates a constant P_{gen} which 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:

$$P_{gen} = \text{sigmoid}(\omega_s^\top s_t + \omega_w^\top w_t + \omega_c^\top c_t + b_{gen}), \tag{8}$$

$$P_{vocab}(w_t) = \text{softmax}(W_a s_t + V_a c_t) \tag{9}$$

$$P(w_t) = P_{gen} P_{vocab}(w_t) + (1 - P_{gen}) P_{copy}(w_t), \tag{10}$$

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 overall vocabulary. Copy distribution $P_{copy}(w_t)$ determines where to attend in time step t, computed as:

$$P_{copy}(w_t) = \sum_{i:x_i=w} \alpha_{t,i}\ , \tag{11}$$

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

The framework of SG-Trans is mostly consistent with the vanilla Transformer, but consists of two major improvements, namely *structure-guided self-attention* and *hierarchical structure-variant attention*. Figure 4 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 adjacent 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 difference levels in a hierarchical manner (introduced in Section 3.2).

3.1 Structure-Guided Self-Attention

In the standard multi-head self-attention model [42], every node in the adjacent layer is allowed to attend to all the input nodes, as shown in Figure 4 (a). In this work, we propose to use the structural relations in source code for introducing explicit constraints to the multi-head self-attention. In order to capture the hierarchical characteristic 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) **data flow** between two tokens. For each structure type, we design the corresponding head attention, named as *token-guided self-attention*, *statement-guided self-attention*, and *data flow-guided self-attention*, respectively.

Token-guided self-attention. It is common for developers to name a variable with camel case or snake case, and such tokens are generally split for alleviating the OOV (Out-Of-Vocabulary) issue in code summarization, e.g., the method name "*IsPrime*" in the code example shown in Figure 1 (a) is split as a sequence of sub-tokens containing "*Is*" and "*Prime*". We regard the semantic relations between the sub-tokens are relatively stronger than the relations

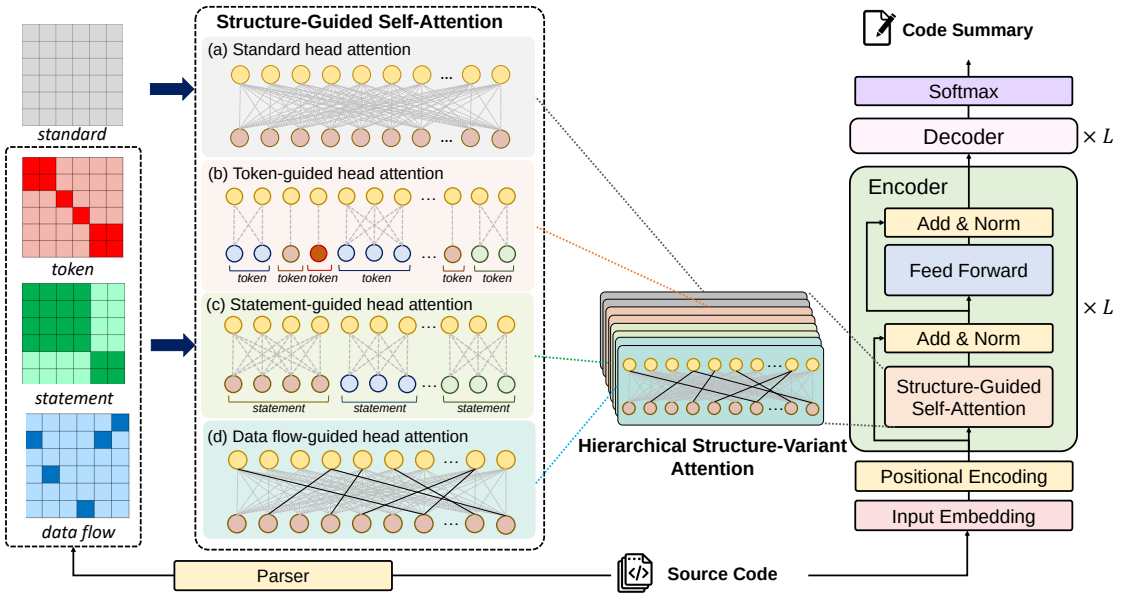


Fig. 4. Overall framework of the proposed SG-Trans. The "Structure-Guide Self-Attention" part illustrates different self-attention mechanisms between adjacent layers.

between the other tokens. Therefore, the attention can be built upon the extracted token-level structure, i.e, whether two sub-tokens are originally from the same source code token. We use an adjacent 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 message passing to only among the sub-tokens belonging to the same code token in self-attention, as shown in Figure 4 (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. We assume the three matrices as $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ for denoting the query, key, and value matrix, respectively. The token-guided single-head self-attention $head_t$ can be calculated as:

$$head_t = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}} + \mathbf{T}\right)\mathbf{V}, \tag{12}$$

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 the tokens 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. So we design another adjacent matrix $\mathbf{S}$ for representing the pairwise token relations regarding 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 head attention to only allow the message passing among the tokens from the same statement, as illustrated in Figure 4 (c). The statement-guided single-head self-attention $head_s$ is defined as below similar to the token-guided head attention.

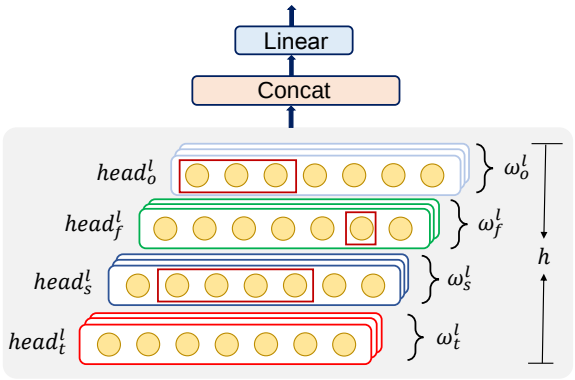


Fig. 5. A diagram of hierarchical-variant attention. Different red boxes illustrate different scales.

$$head_s = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\intercal}{\sqrt{d}} + \mathbf{S}\right)\mathbf{V}. \tag{13}$$

Data flow-guided self-attention. Prior studies have proven the effectiveness of utilizing the code structural properties such as abstract syntax trees (ASTs) for code summarization. Since ASTs are deep in nature, we employ the data flow graphs (DFGs), which are relatively neat and much shallower [15], for capturing the global structure feature.

DFGs, denoted as $V = \{v_1, v_2, ...\}$, can model the data dependencies between variables in the code, including message sending/receiving. Figure 1 (c) shows an example of the extracted data flow graph. Variables with 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 the value of the j-th variable comes from the i-th variable. For the example in Figure 1, the value i^5 comes from the variable i^4. Based on the DFGs, we build the adjacent 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, all the split sub-tokens of the two tokens are regarded as possessing the dependency relation. Figure 4 (d) illustrates the data flow-guided single-head self-attention. Due to the sparseness of the matrix $\mathbf{D}$ and to highlight the relations of data dependencies, we propose the data flow-guided self-attention $head_f$ as below:

$$head_f = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\intercal + \mu * \mathbf{Q}\mathbf{K}^\intercal\mathbf{D}}{\sqrt{d}}\right)\mathbf{V}, \tag{14}$$

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

3.2 Hierarchical Structure-Variant Attention

Considering the principle of compositionality in logic semantics: the high-level semantics is the composition of low-level terms [16, 41], we propose hierarchical structure-variant attention to make our model focus on local structure at the lower layers and global structure at the higher layers. The diagram of the hierarchical structure-variant attention is illustrated in Figure 5. Specifically, the token-guided head attention $head_t$ and statement-guided head attention $head_s$ are more distributed in the heads of lower layers; while the data flow-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. h indicates the number of heads in each layer and k is a hyper-parameter to control the distribution of four types of head attentions, including $head_t$, $head_s$, $head_f$, and $head_o$,

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

where $head_o$ indicates the standard head attention without constraints. 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 below:

$$\omega_t^l = \omega_s^l = \lfloor h * \frac{k-l}{2*k-l} \rfloor, \tag{15}$$

$$\omega_f^l = \lfloor h * \frac{l}{2*k-l} \rfloor, \tag{16}$$

$$\omega_o^l = h - (\omega_t^l + \omega_s^l + \omega_f^l), \tag{17}$$

where k is a positive integer hyperparameter, and $\lfloor \cdot \rfloor$ denotes rounding the value down to the next lowest integer. The design is to enable more heads attend to the global structure with the growth of l, i.e., ω_f^l will get larger at a higher layer l; meanwhile few heads can catch the local structure, i.e., 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. Especially, 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:

$$\text{HSVA}^l = [head_1^l \circ \cdots \circ head_h^l]\mathbf{W}^O, \tag{18}$$

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 [25]. 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 [1], an additional attention layer is added to learn the copy distribution on top of the decoder stack [35]. The mechanism enables the proposed SG-Trans to copy low-frequency words, e.g., API names, from source code, thus mitigating the OOV issue.

4 EXPERIEMENTAL 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 respectively contain Java and Python source code following the previous work [1, 49]. Specifically, the Java dataset publicly released by [19] comprises 87, 136 ⟨Java method, comment⟩

pairs collected from 9,714 GitHub repositories, and the Python dataset consists of 92,545 functions and corresponding documentation as originally collected by [8] and later processed by [45].

For the sake of fairness, we directly use the benchmarks open sourced by the previous state-of-the-art [1] with the dataset split into training set, validation set and test set in a consistent proportion of 8 : 1 : 1 for the Java dataset and 6 : 2 : 2 for the Python dataset. The statistics are shown in Table 1. For the data flow extraction of the Java dataset, we use the tool in [10] to generate augmented ASTs first and then extract DFGs from the ASTs. Regarding the Python dataset, we follow the design in [3] and extract four kinds of edge (*LastRead*, *LastWrite*, *LastLexicalUse*, *ComputeFrom*) from code.

4.2 Evaluation Metrics

To verify the efficacy of SG-Trans over the baselines, we use the most commonly-used automatic evaluation metrics, BLEU-4 [36], METEOR [7] and ROUGE-L [28].

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) [24, 30, 34, 48]. 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:

$$BLEU - N = BP \times \exp(\sum_{n=1}^{N} \tau_n \log P_n), \tag{19}$$

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 sequence and τ_n is the uniform weight $1/N$. We use corpus-level BLEU-4, i.e., $N = 4$, as our evaluation metric since it is demonstrated to be more correlated with human judgements than other evaluation metrics [29].

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

$$METEOR = (1 - \gamma \cdot frag^{\beta}) \cdot \frac{P \cdot R}{\alpha \cdot P + (1 - \alpha) \cdot R}, \tag{20}$$

where P and R are the unigram precision and recall, $frag$ 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 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 text 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:

$$P_{lcs} = \frac{LCS(X,Y)}{n}, R_{lcs} = \frac{LCS(X,Y)}{m}, F_{lcs} = \frac{(1+\beta^2)P_{lcs}R_{lcs}}{R_{lcs} + \beta^2 P_{lcs}}, \tag{21}$$

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

4.3 Baselines

We compare SG-Trans with following baseline approaches.

Table 2. Comparison results with baseline models. The **bold** figures indicate the best results. * denotes statistical significance in comparison to the baseline models (i.e., two-sided t-test with p-value< 0.01).

Approach	Java			Python		
	BLEU-4	METEOR	ROUGE-L	BLEU-4	METEOR	ROUGE-L
CODE-NN [22]	27.60	12.61	41.10	17.36	9.29	37.81
Tree2Seq [11]	37.88	22.55	51.50	20.07	8.96	35.64
RL+Hybrid2Seq [44]	38.22	22.75	51.91	19.28	9.75	39.34
DeepCom [19]	39.75	23.06	52.67	20.78	9.98	37.35
API+Code [20]	41.31	23.73	52.25	15.36	8.57	33.65
Dual Model [45]	42.39	25.77	53.61	21.80	11.14	39.45
NeuralCodeSum [1]	45.15	27.46	54.84	32.19	19.96	46.32
Vanilla Transformer [42]	44.20	26.83	53.45	31.34	18.92	44.39
SG-Trans	**45.97***	**27.88***	**55.86***	**33.11***	**20.58***	**47.07***

CODE-NN [22], as the first deep-learning-based work in code summarization, generates source code summaries with an LSTM network. To utilize code structure information, **Tree2seq** [11] encodes source code with a tree-LSTM architecture. **RL+Hybrid2Seq** [44] incorporates ASTs and code sequences into a deep reinforcement learning framework, while **DeepCom** [19] encodes the node sequences traversed from ASTs to capture the structural information. **API+Code** [20] involves API knowledge in the code summarization procedure. **Dual model** [45] adopts a dual learning framework to exploit the duality of code summarization and code generation tasks. The most recent approach, denoted as **NeuralCodeSum** [1], which integrates the **vanilla Transformer** [42] with relative position encoding (RPE) and copy attention, shows the state-of-the-art performance on the benchmark datasets.

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. To avoid over-fitting, we early stop the training if the performance on the validations set does not increase for 20 epochs. For the control factors of heads distribution and data flow, we set them to 1 and 5, respectively. We will discuss optimal parameters selection in Section 5.3. Our experiments are conducted on a single Tesla P100 GPU for about 45 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 design of hierarchical attention on the model performance?

RQ3: How accurate is SG-Trans under different parameter settings?

Table 3. Ablation study on different part of our model. The **bold** figures indicate the best results.

Approach	Java			Python		
	BLEU-4	METEOR	ROUGE-L	BLEU-4	METEOR	ROUGE-L
SG-Trans w/o token info.	44.91	27.39	54.72	32.28	19.87	45.98
SG-Trans w/o statement info.	44.52	27.06	54.16	32.32	19.72	45.77
SG-Trans w/o data flow info.	45.58	27.57	55.36	32.64	20.18	46.66
SG-Trans w/o hierarchical attention	45.59	27.79	55.50	32.97	20.44	46.82
SG-Trans w/o copy attention	45.12	27.41	54.89	31.92	19.35	45.21
SG-Trans	**45.97**	**27.88**	**55.86**	**33.11**	**20.58**	**47.07**

5.1 Answer to RQ1: Comparison with the Baselines

The experimental results on the benchmark datasets are shown in Table 2. For the vanilla Transformer and the NeuralCodeSum [1], we reproduce their experiments under the same hyper-parameter settings as the Transformer in SG-Trans to ensure fair comparison. Based on Table 2, we summarize the following findings:

Code structural properties are beneficial for source code summarization. Comparing Tree2Seq/DeepCom with CODE-NN, we can find that the structure information brings a great improvement in the performance. For example, both Tree2Seq and DeepCom increase the performance of CODE-NN by at least 37.2%, 78.8%, and 25.3% 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.

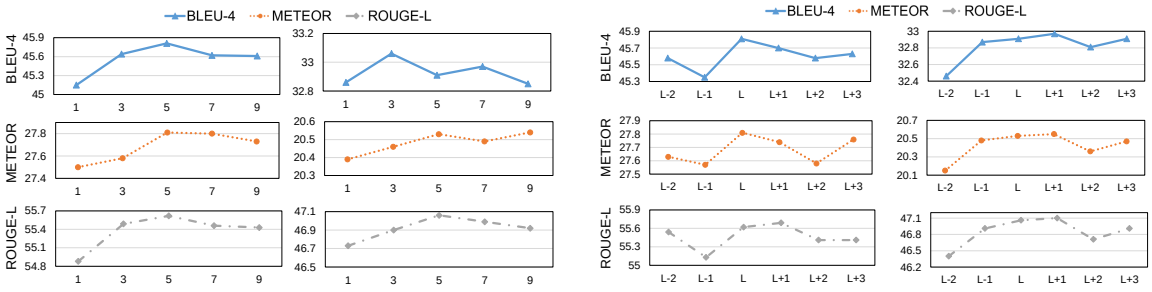
Transformer-based approaches perform better than RNN-based approaches. The two Transformer-based approaches [1, 42] outperform all the other baselines, with NeuralCodeSum [1] giving better performance compared to the Vanilla Transformer. The vanilla Transformer already achieves better performance than the top six RNN-based approaches with various types of structural information incorporated, showing the efficacy of Transformer for the task. On the Python dataset, NeuralCodeSum outperforms the best RNN-based baseline, Dual Model [45], by 47.7% and 79.2% in terms of the BLEU-4 and METEOR metrics.

The proposed SG-Trans is effective in code summarization. Comparing SG-Trans with the vanilla Transformer and NeuralCodeSum, SG-Transachieves the best results on both benchmark datasets, yet without introducing any extra model parameters. Specifically, SG-Trans improves the best baseline by 1.8% and 2.9% in terms of BLEU-4 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, and show the results in the bottom half of Table 2.

Analysis of the involved code structure. We find that all the three structure types, including code token, statement and data flow, contribute to the model performance improvement but with varied degrees. Specifically, local syntactic structures play a more important role than the global data flow structure. For example, removing the statement information leads to a significant performance drop at around 3.2% 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 data flow information eliminated, SG-Trans also suffers from a performance drop, which may indicate that the data dependency relations are hard to be learnt by Transformer implicitly.



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

Fig. 6. Influence of the hyper-parameters μ and k on the model performance.

Table 4. Human evaluation results. * denotes statistical significance in comparison to Transformer or NeuralCodeSum in Human Evaluation (i.e., two-sided t-test with p-value< 0.01).

Dataset	Metrics	Vanilla Transformer	NeuralCodeSum	SG-Trans
Java	Relevance	2.73	3.41	**4.08***
	Similarity	2.62	3.31	**4.02***
Python	Relevance	3.16	3.01	**3.36***
	Similarity	2.99	2.81	**3.69***

Analysis of the hierarchical structure-variant attention mechanism. We replace the hierarchical structure-variant attention with uniformly-distributed attention, i.e., $\Omega^l = [\omega_t^l, \omega_s^l, \omega_f^l, \omega_o^l] = [2, 2, 2, 2]$, for the ablation analysis. As can be found in Table 2, without the hierarchical structure design, the model's performance decreases on all metrics of 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 brings a significant drop to SG-Trans's performance, similar to the results in [1]. This may reflect that the copy attention is useful for alleviating the OOV issue and facilitating better code summarization.

5.3 Answer to RQ3: Parameter Sensitivity Analysis

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

The parameter μ. Figure 6 (a) shows the performance variation with the changes of μ and other hyper-parameters fixed. For the Java dataset, the model achieves the best scores when $\mu = 5$. Lower or higher parameter values do not provide better results. While for the Python dataset, the same trends appear to the BLEU-4 and ROUGE-L metrics where the models present the highest scores when μ equals to 3 and 5, respectively. In this work, we set μ to 5 since the model can produce relatively promising 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 6 (b) illustrates the results. We can find that SG-Trans can well balance the distribution of local and global structure-guided head attention 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 inaccurate code summary. In the work, we take $k = L$.

Code:

```
def is_printable(s):
    for c in s:
        if (c not in PRINTABLE_CHARACTERS):
            return False
    return True
```

Reference Summary: test if a string is printable .

Summary 1: return true if s consists of any duplicates in repr

Summary 2: true if s consists entirely of ascii characters

Summary 3: check if a string is printable .

	Very Dissatisfied				Very Satisfied
Summary 1's Relevance	○ 1	○ 2	○ 3	○ 4	○ 5
Summary 1's Similarity	○ 1	○ 2	○ 3	○ 4	○ 5
. .	. .	. .	. .	. .	. .
Summary 3's Relevance	○ 1	○ 2	○ 3	○ 4	○ 5
Summary 3's Similarity	○ 1	○ 2	○ 3	○ 4	○ 5

Fig. 7. An example of questions in our questionnaire. Reference summary is the summary for the given code in the ground truth. Summary 1, 2 and 3 correspond to the summary generated by vanilla Transformer, NeuralCodeSum, and SG-Trans, respectively. The two-dot symbols indicate the simplified rating schemes for Summary 2.

5.4 Human Evaluation

In this section, we perform human evaluation to qualitatively evaluate the summaries generated by the two best-performing baselines, i.e., vanilla Transformer and NeuralCodeSum, and SG-Trans. The human evaluation is conducted through online questionnaire. In total, 14 participants including 11 postgraduate students, 2 undergraduate students, and 1 senior researcher are invited for the questionnaire. All of the participants are not co-authors and major in computer science, 12 (85.7%) of whom have programming experience in software development for at least two years. Each participant is invited to read 50 functions and judge the quality the summaries generated by vanilla Transformer, NeuralCodeSum, and SG-Trans. Each of them will be paid 10 USD upon completing the questionnaire.

5.4.1 Survey Design. We randomly selected 50 functions, with 25 in Java and 25 in Python, for evaluation. As shown in Figure 7, in the questionnaire, each question comprises a code snippet, the corresponding reference summary and summaries generated by the three models. The order of the summaries generated by the models is randomly swapped for each question, so that the participants are not aware of which summary is generated by which model.

The quality of the provided summaries is evaluated from two aspects, including *relevance* and *similarity*, with the 1-5 Likert scale (5 for excellent, 4 for good, 3 for acceptable, 2 for marginal, and 1 for poor). We explained the meaning of the two evaluation metrics at the beginning of the questionnaire: The metric "similarity" measures the degree of the semantic similarity between the generated summary and the summary in ground truth, i.e., reference summary; And the metric "relevance" estimates the extent of semantic relation between the generated summary and the given code snippet. The volunteers are asked to complete the online questionnaires separately.

5.4.2 Results. We finally received 700 sets of scores totally and 14 sets of scores for each code-summary pair from the human evaluation. On average, the participants spent 1.1 hours on completing the questionnaire, with the median

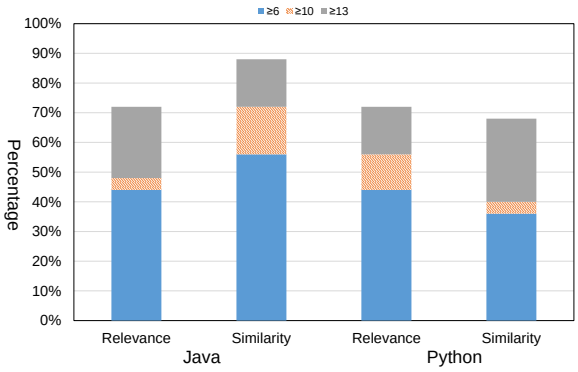
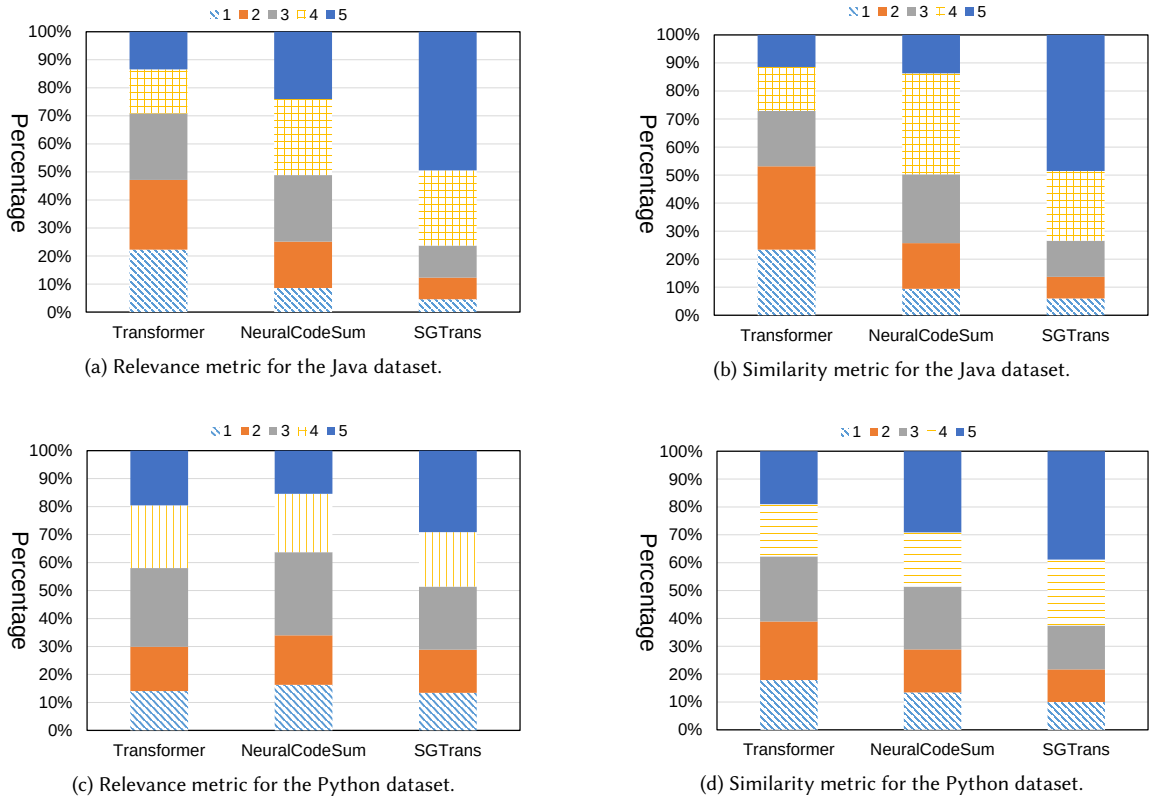


Fig. 8. Agreement rate among the participants in the human evaluation. The vertical axis indicates the percentage of the survey questions that are scored consistently from ≥ 6, ≥ 10 and ≥ 13 participants, respectively.

completion time of 0.77 hours. We first compute the agreement rate on the two aspects given by the participants, depicted in Figure 8. As can be seen, 72% and 88% of the total code-summary pairs are rated consistently by at least six annotators in terms of the "Relevance" and "Similarity" for the Java dataset, respectively. Also, 72% and 68% of the total questions received more than six consistent annotations in terms of the respective metrics for the Python dataset. Moreover, we can observe that around 20% of the questions were labeled with same scores from more than 13 participants regarding different aspects for the two programming languages, respectively. This demonstrates that the participants achieved acceptable agreement on the quality of the generated summaries.

The overall evaluation results are illustrated in Table 4 and Figure 9. We can find that the summaries generated by SG-Trans are more relevant to the given functions and exhibit the highest similarities to the summaries in the ground truth for both programming languages. For Java dataset, NeuralCodeSum is very effective which significantly outperforms the vanilla Transformer. However SG-Trans is more powerful, further boosting the performance by 19.6% and 21.5% in terms of the relevance and similarity metrics, respectively. As can be observed from Figure 9 (a) and (b), summaries generated by SG-Trans receive the most 5-star ratings and fewest 1/2-star ratings from the participants, comparing with the summaries produced by NeuralCodeSum and vanilla Transformer, with respect to both metrics. Specifically, regarding the relevance metric, 48.6% of the participants give 5-star ratings to the summaries generated by SG-Trans, with only 11.4% for the vanilla Transformer approach and 13.7% for the NeuralCodeSum approach. The score distributions indicate that the summaries generated by SG-Trans are more semantically relevant to the code snippets and also more similar to the golden summaries.

For the Python dataset, as shown in Table 4, SG-Trans also achieves best performance, increasing the performance of NeuralCodeSum by 11.6% and 31.3% with respect to the relevance and similarity metrics, respectively. According to Figure 9 (c), the vanilla Transformer even receives more 4/5-star ratings than the NeuralCodeSum approach, which implies that NeuralCodeSum may not well capture the functionality of the code snippets and thus produce less relevant summaries. However, summaries generated by SG-Trans are scored with the most 4/5-star ratings among the three approaches, which further demonstrating the effectiveness of SG-Trans in capturing the functionality of given code snippets. In terms of the similarity metric, SG-Trans receives more than 60% 4/5-star ratings while Transformer and NeuralCodeSum only obtains 37.4% and 48.6% 4/5-star ratings, respectively, indicating its superior performance in producing practical code summaries.



(a) Relevance metric for the Java dataset.

(b) Similarity metric for the Java dataset.

(c) Relevance metric for the Python dataset.

(d) Similarity metric for the Python dataset.

Fig. 9. Distribution of the received scores from the participants during human evaluation on the two datsets. The "Transformer" on the horizontal axis denotes the "vanilla Transformer" approach.

6 DISCUSSION

In this section, we mainly discuss the advantage of the proposed SG-Trans, the impact of duplicate data in the benchmark dataset on the model performance, and threats to validity.

6.1 Why does Our Model Work?

We further conduct a deep analysis on the advantages of the proposed SG-Trans in generating high-quality code summaries. Through qualitative analysis, we have identified two advantages of SG-Trans that may explain its effectiveness in the task.

Observation 1: SG-Trans can better capture the semantic relations among tokens. For the Example (1) shown in Table 5, we can observe that SG-Trans produces the summaries most similar to the ground truth among all the approaches, and the vanilla Transformer gives the worst result. We then visualize the heatmap of the self-attention scores of the three types of heads in Figure 10 for further analysis. As can be seen in Figure 10 (a) and (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 10 (a). As depicted in Figure 10 (b), we can find that SG-Trans captures that the token "*path*" is strongly related to the

Table 5. 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
NeuralCodeSum: checks if is file exist
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
NeuralCodeSum: prints out a user access control list
SG-Trans: prints out a buckets access control list for a 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: generate a token
NeuralCodeSum: construct a random text string .
SG-Trans: return a random url-safe string .
Ground truth: return a random url-safe text string .

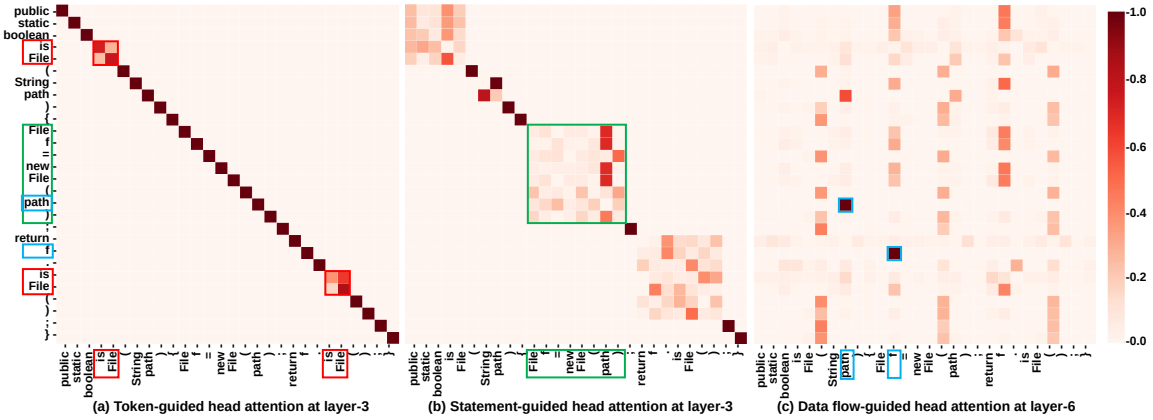


Fig. 10. Heatmap visualization of self-attention scores of the three types of heads in the encoder for the first case in Table 5. The rectangles with red edge, green edge, and blue edge indicates the tokens belonging to the same original token, the same statement, or containing data flow relation, respectively.

corresponding statement, which may be the reason the token "*path*" appears in the summary. Figure 10 (c) shows that the data flow-guided head attention focuses more on the global information, and can capture the strong relation

Table 6. Duplicate data in the Java dataset.

	Validation Set	Test set
Total data	8,714	8,714
Duplicate data	2,028 (23.3%)	2,059 (23.7%)

Table 7. Comparison results on the de-duplicated Java dataset. Data listed within brackets are computed drop rates compared with the results on original Java dataset.

Approach	BLEU-4	ROUGE-L	METEOR
NeuralCodeSum	29.37 (↓34.95%)	41.62 (↓24.11%)	19.98 (↓27.24%)
SG-Trans	**30.46** (↓33.74%)	**42.97** (↓23.10%)	**20.82** (↓25.32%)

between the tokens "*path*" and "*f*". Based on the analysis of the example (1), we speculate that the model can well capture the token relations locally and globally for code summary generation. The example (2) in Table 5 provides the same hint for the advantage of SG-Trans. All 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. For the example (3) in Table 5, SG-Trans copies the important token "*urlsafe*" from the given code to the generated summary, while both vanilla Transformer and NeuralCodeSum ignore the token and output relatively imprecise summaries. The reason that the important token is successfully copied by SG-Trans may be attributed to the structural information-guided self-attention which helps focus on the source tokens more accurately.

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 shown in Table 6, there are 23.3% and 23.7% 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 cross the training, validation, and test sets. We choose the best baseline, NeuralCodeSum, for comparison. The results after removing the duplication are shown in Table 7. As can be seen, both models present a dramatic decrease on the de-duplicated dataset. However, the proposed SG-Trans still outperforms NeuralCodeSum with an improvement on the BLEU-4, ROUGE-L and METEOR metrics, i.e., by 3.7%, 3.2% and 4.2%, respectively.

6.3 Threats to Validity

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

(1) The generalizability of our results. We use two public large datasets, which include 87,136 Java and 92,545 Python code-summary pairs, following the prior research [1, 46, 49]. The limited programming language types may influence the scalability of the proposed SG-Trans. In our future work, we will experiment with more large-scale datasets with different programming language types.

(2) More code structure information may be considered. SG-Trans now only takes token-level and statement-level syntactic structure and data flow structure into consideration. Other code structural properties such as AST

and CFG, which may further boost the model performance, are currently not involved. In this paper, we only consider the data flow information since it is demonstrated to be more effective than AST and CFG during code representation learning in [15]. In future, we will explore the impact of involving more structural properties in SG-Trans.

(3) Biases in human evaluation. We invite 14 participants to evaluate the quality of 50 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 each code-summary pair was evaluated by all the 14 participants. The order of the summaries generated by different approaches is also randomly disrupted across different code snippets in the questionnaire, in order to eliminate the influence of participants' prior knowledge on their rating.

7 RELATED WORK

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

7.1 Source Code Summarization

There have been extensive research in source code summarization, including template-based approaches [31, 32, 40], information-retrieval-based approaches [17, 33, 47] and deep-learning-based-approaches. Among these categories, deep-learning-based methods have achieved the greatest success and become the most popular in recent years, which specifically formulate the code summarization task as a neural machine translation (NMT) problem and adopt state-of-the-art NMT frameworks to improve the performance. For instance, [22] propose CODE-NN, a hierarchical Long Short Term Memory (LSTM) network with attention to generate code summaries from code snippets. [45] and [49] further take advantages of the retrieved information to enable more informed code summarization.

In order to achieve more accurate code modeling, later researchers then introduce more structural and syntactic information to the deep learning models. [19] extract structural information from source code by traversing the ASTs and processing the AST nodes into sequences that can be fed into the encoder. On the contrary, [38] adopt multi-way Tree-LSTM to directly model the code structures. For more fine-grained intra-code relationship exploitation, many works [13, 26] also incorporate code-related graphs and GNN to boost performance. With the rise of Transformer in NMT task domain, [1] also utilize transformer for better mapping the source code to their corresponding natural language summaries.

7.2 Code Representation Learning

Learning high-quality code representations is of vital importance for deep-learning-based code summarization. Apart from the above practices for code summarization, there also exist other code representation learning methods that lie in similar task domains such as source code classification, code clone detection, commit message generation, etc. For example, the ASTNN model proposed by Zhang et al.[50] splits large ASTs into sequences of small statement trees, which are further encoded into vectors as source code representations. Alon et al.[4] present CODE2SEQ that also represents the code snippets by sampling certain paths from the ASTs. Recently, inspired by the successes of pre-training methods in learning word representations, Feng et al.[12] and Guo et al.[15] also apply pre-trained models on learning source code and achieve empirical improvements on a variety of tasks. To extend the code representations

to characterize programs' functionalities, Jain et al. [23] further enriches the pre-training tasks to learn task-agnostic semantic code representations from textually divergent variants of source programs via contrastive learning.

Comparatively, our proposed model focuses more on optimizing the self-attention mechanism inside Transformer to make it incorporate both local and global features, through which our approach can exploit more accurate source code representations and enhance the code summarization.

8 CONCLUSION

In this paper, 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 the code structural information, including local structure in token-level and statement-level, and global structure, i.e., data flow. 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 domains, 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 2020, 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 2018, 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 2019, New Orleans, LA, USA, 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 2020, 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). arXiv:1607.06450 http://arxiv.org/abs/1607.06450

[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, Michigan, USA, 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] 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 2014, 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. https://doi.org/10.3115/v1/d14-1179

[10] 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 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 1475–1485.

[11] 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 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics.

[12] 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 2020, Online Event, 16-20 November 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547.

[13] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured Neural Summarization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

[14] 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 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics. https://doi.org/10.18653/v1/p16-1154

[15] 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).

[16] Qipeng Guo, Xipeng Qiu, Pengfei Liu, Xiangyang Xue, and Zheng Zhang. 2020. Multi-Scale Self-Attention for Text Classification. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 7847–7854.

[17] 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 2010, Cape Town, South Africa, 1-8 May 2010*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 223–226.

[18] 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 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 770–778. https://doi.org/10.1109/CVPR.2016.90

[19] 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 2018, Gothenburg, Sweden, May 27-28, 2018*, Foutse Khomh, Chanchal K. Roy, and Janet Siegmund (Eds.). ACM, 200–210.

[20] 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 Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, Jérôme Lang (Ed.). ijcai.org, 2269–2275.

[21] 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.

[22] 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 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics.

[23] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E. Gonzalez, and Ion Stoica. 2020. Contrastive Code Representation Learning. *CoRR* abs/2007.04973 (2020).

[24] 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 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 135–146.

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

[26] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved Code Summarization via a Graph Neural Network. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 184–195.

[27] Yuding Liang and Kenny Qili Zhu. 2018. Automatic Generation of Text Descriptive Comments for Code Blocks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 5229–5236.

[28] 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.

[29] 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 2016, Austin, Texas, USA, November 1-4, 2016*, Jian Su, Xavier Carreras, and Kevin Duh (Eds.). The Association for Computational Linguistics, 2122–2132.

[30] 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 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 176–188.

[31] 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.

[32] 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 2013, San Francisco, CA, USA, 20-21 May, 2013*. IEEE Computer Society, 23–32.

[33] 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 2013, 4-9 August 2013, Sofia, Bulgaria, Volume 2: Short Papers*. The Association for Computer Linguistics, 35–40.

[34] 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).

[35] 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 2019, 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.

[36] 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, USA*. ACL, 311–318.

[37] 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 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, Regina Barzilay and Min-Yen Kan (Eds.). Association for Computational Linguistics, 1073–1083.

[38] 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 2019 Budapest, Hungary, July 14-19, 2019*. IEEE, 1–8.

[39] 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 *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 8886–8893.

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

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

[42] 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, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008.

[43] 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 2019, 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.

[44] 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 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 397–407.

[45] Bolin Wei. 2019. Retrieve and Refine: Exemplar-Based Neural Comment Generation. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 1250–1252.

[46] 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 2019, 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. https://proceedings.neurips.cc/paper/2019/hash/e52ad5c9f751f599492b4f087ed7ecfc-Abstract.html

[47] 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 2015, Montreal, QC, Canada, March 2-6, 2015*, Yann-Gaël Guéhéneuc, Bram Adams, and Alexander Serebrenik (Eds.). IEEE Computer Society, 380–389.

[48] 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 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*. 1325–1335.

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

[50] 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 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 783–794.

[51] Xiangyu Zhao, Longbiao Wang, Ruifang He, Ting Yang, Jinxin Chang, and Ruifang Wang. 2020. Multiple Knowledge Syncretic Transformer for Natural Dialogue Generation. In *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen (Eds.). ACM / IW3C2, 752–762.