Modeling Hierarchical Syntax Structure with Triplet Position for Source Code Summarization

Anonymous ACL submission

Abstract

Automatic code summarization, which aims to describe the source code in natural language, has become an essential task in software maintenance. Our fellow researchers have attempted to achieve such a purpose through various machine learning-based approaches. One key challenge keeping these approaches from being practical lies in the lacking of retaining the semantic structure of source code, which has unfortunately been overlooked by the state-of-the-art. Existing approaches resort to representing the syntax structure of code by modeling the Abstract Syntax Trees (ASTs). However, the hierarchical structures of ASTs have not been well explored. In this paper, we propose CODESCRIBE to model the hierarchical syntax structure of code by introducing a novel triplet position for code summarization. Specifically, CODESCRIBE leverages the graph neural network and Transformer to preserve the structural and sequential information of code, respectively. In addition, we propose a pointer-generator network that pays attention to both the structure and sequential tokens of code for a better summary generation. Experiments on two real-world datasets in Java and Python demonstrate the effectiveness of our proposed approach when compared with several state-of-the-art baselines[1].

1 Introduction

Code documentation in the form of code comments has been an integral component of software development, benefiting software maintenance (Iyer et al., 2016a), code categorization (Nguyen and Nguyen, 2017) and retrieval (Gu et al., 2018). However, few real-world software projects are well-documented with high-quality comments. Many projects are either inadequately documented due to missing important code comments or inconsistently documented due to different naming conventions by developers, e.g., when programming in legacy code bases, resulting in high maintenance costs (de Souza et al., 2005; Kajko-Mattsson, 2005). Therefore, automatic code summarization, which aims to generate natural language texts (i.e., a short paragraph) to describe a code fragment by extracting its semantics, becomes critically important for program understanding and software maintenance.

Recently, various works have been proposed for code summarization based on the encoder-decoder paradigm, which first encodes the code into a distributed vector, and then decodes it into natural-language summary. Similarly, several works (Iyer et al., 2016a; Allamanis et al., 2016) proposed to tokenize the source code into sequential tokens, and design RNN and CNN to represent them. One limitation of these approaches is that they only consider the sequential lexical information of code. To represent the syntax of code, several structural neural networks are designed to represent the Abstract Syntax Trees (AST) of code, e.g., TreeLSTM (Wan et al., 2018), TBCNN (Mou et al., 2016), and Graph Neural Networks (GNNs) (LeClair et al., 2020). To further improve the efficiency on AST representation, various works (Hu et al., 2018a; Alon et al., 2018) proposed to linearize the ASTs into a sequence of nodes or paths.

Despite much progress on code summarization, there are still some limitations in code comprehension for generating high-quality comments. Particularly, when linearizing the ASTs of code into sequential nodes or paths, the relationships between connected nodes are generally discarded. Although the GNN-based approaches can well preserve the syntax structure of code, they are insensitive to the order of nodes in AST. For example, given the expressions `a=b/c` and `a=c/b`, current approaches cannot capture the orders of variables `b` and `c`. However, these orders are critical to accurately preserve the semantics of code.

To address the aforementioned limitation, this

[1]The source code of CODESCRIBE is available at https://github.com/anonymousrepoxxx/CODESCRIBE

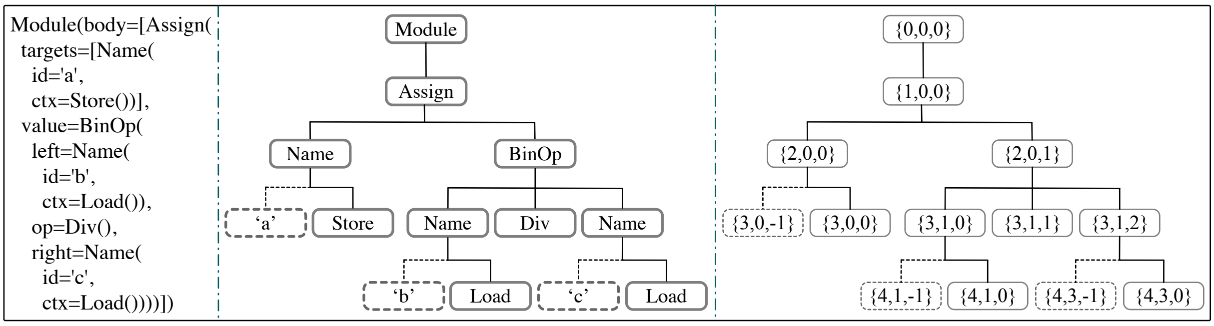

Figure 1: The AST of Python code snippet "`a = b / c`". The left is the text form of AST, the middle shows the tree structure of AST, and the right specifies triplet positions for all nodes of AST structure.

paper proposes to model the hierarchical syntax structure of code using triplet position, inspired by the positional encoding used in sequence modeling (Gehring et al., 2017; Vaswani et al., 2017), and incorporates the triplet position into current GNNs for better code summarization. The triplet position records the depth, width position of its parent, and width position among its siblings for each node.

To utilize the triplet position in AST, this paper proposes CODESCRIBE, an encoder-decoder-based neural network for source code summarization. Specially, we initialize the embedding of each AST node by incorporating the triplet positional embeddings, and then feed them into an improved GNN, i.e., GraphSAGE (Hamilton et al., 2017) to represent the syntax of code. In addition, we also account for the sequential information of code by using a Transformer encoder (Vaswani et al., 2017). In such a case, the decoding process is performed over the learned structural features of AST and sequential features of code tokens with two multi-head attention modules. To generate summaries with higher quality, we further design a pointer-generator network based on multi-head attention (Vaswani et al., 2017), which allows the summary tokens to be generated from the vocabulary or copied from the input source code tokens and ASTs. To validate the effectiveness of our proposed CODESCRIBE, we conduct experiments on two real-world datasets in Java and Python.

Overall, the contributions of this paper are two-fold: (1) It is the first time that we put forward a simple yet effective approach of triplet position to preserve the hierarchical syntax structure of source code accurately. We also incorporate the triplet position into an adapted GNN (i.e., GraphSAGE) for source code summarization. (2) We conduct comprehensive experiments on two real-world datasets in Java and Python to evaluate the effectiveness of our proposed CODESCRIBE. Experimental results demonstrate the superiority of CODESCRIBE when comparing with several state-of-the-art baselines. For example, we get 3.70/5.10/4.77% absolute gain on BLEU/METEOR/ROUGE-L metrics on the Java dataset, when comparing with the most recent mAST+GCN (Choi et al., 2021).

2 Hierarchical Syntax in Triplet Position

Recent studies have showed promising results by using AST context for tasks based on code representation learning (Yao et al., 2019; Zhang et al., 2019; Choi et al., 2021). Therefore, our work also relies on AST information besides source code tokens. As a type of intermediate representation, AST represents the hierarchical syntactic structure for source code, which is an ordered tree with labeled nodes (cf. Figure 1). In this work, we divide the nodes into two categories: (1) *function node* that controls the structure of AST and function realization, e.g., `Module` and `Assign` in Figure 1, and (2) *attribute node* that provides the value or name of its parent function node, which is always visualized as leaf node, such as '`a`' and '`b`' in dotted boxes of Figure 1.

Due to the strict construction rules of AST, positions are crucial for AST nodes. For example in Figure 1, the node `BinOp` has two children with the same label `Name`. If the positions of the two siblings are swapped, the source code will become `a=c/b`, which is totally different from the intent of the code `a=b/c`. However, GNNs are insensitive to the positions of neighbouring nodes when encoding such tree structures. Based on this observation, we specify triplet positions for AST nodes to retain accurate structural information in AST learning. The triplet position of a node includes: (1) the depth of the node in the AST, (2) the width position of its parent node in the layer, and (3) the node's width position among its siblings, which can also distinguish function node from attribute

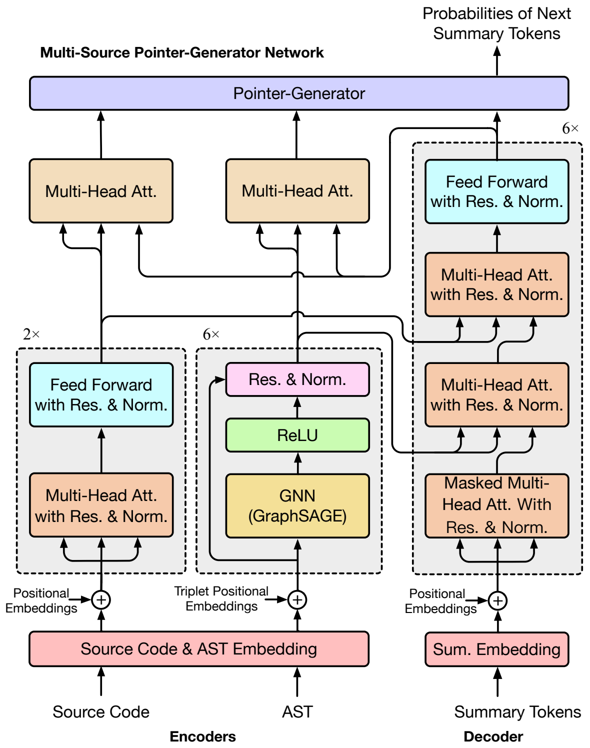

Figure 2: The architecture of CODESCRIBE model. Att., Res., and Norm. denote attention, residual connection, and layer normalization, respectively.

node. That is, the width position of a function node is a non-negative integer starting from 0, while the width position of an attribute node is a negative integer counting from -1. Note that, width positions are estimated in a breadth traversal from left to right. With such triplet indices specified, all nodes can be marked with unique positions in a given AST.

Taking a Python code snippet `a=b/c` as an example, Figure 1 illustrates its AST structure with triplet positions of nodes. Specifically, by traversing the tree, we can represent the function node `(Name,{2,0,0})` as the first child node of node `(Assign,{1,0,0})`: the depth position `2` means the third level (counting from the top to bottom starting with 0; the second width position `0` means that the parent node `Assign` is the first function node at this level (counting from the left to right); and the third position `0` indicates that the node is the first (counting from left to right) among its siblings (i.e., all children nodes of node `Assign`). Another example is the node `('a',{3,0,-1})`. The difference lies in the third position that represents it is an attribute node and it is the first among the siblings. In particular, we set the position of root node `Module` to `{0,0,0}` as it has no parent node. This triplet positioning is very precise and unique, allowing to track and discriminate among the `Name` nodes which also include `(Name,{3,1,0})` and `(Name,{3,1,2})`.

3 CODESCRIBE Approach

3.1 Notations and Framework Overview

Given a code snippet with l_c tokens $\mathbf{T}_c = (c_1, c_2, \dots, c_{l_c})$ and sequential positions $\mathbf{P}_c = (1, 2, \dots, l_c)$, and its AST with l_n nodes $\mathbf{T}_n = (n_1, n_2, \dots, n_{l_n})$ and triplet positions $\mathbf{P}_n = (\{x_1, y_1, z_1\}, \{x_2, y_2, z_2\}, \dots, \{x_{l_n}, y_{l_n}, z_{l_n}\})$, CODESCRIBE predicts the next summary token s_m based on the existing tokens $\mathbf{T}_s = (\text{</s>}, s_1, s_2, \dots, s_{m-1}, \dots)$ with the sequential positions $\mathbf{P}_s = (1, 2, \dots, l_s)$, where `</s>` is a special starting tag for summary input. Note that $\mathbf{T}_s$ is padded to a maximum length of l_s with special padding tags (e.g., `<pad>`s).

Figure 2 illustrates the architecture of CODESCRIBE model, which is mainly composed of four modules: source code encoder, AST encoder, summary decoder and multi-source pointer-generator network (MPG) for output. As shown in Figure 2, the source code, AST, and summary tokens are firstly mapped into embedding vectors $\mathbf{E}_c^0 \in \mathbb{R}^{l_c \times d}$, $\mathbf{E}_n^0 \in \mathbb{R}^{l_n \times d}$, and $\mathbf{E}_s^0 \in \mathbb{R}^{l_s \times d}$ where d is the embedding size. In the encoding process, the embedded code and AST are fed into Transformer encoder (Vaswani et al., 2017) and GNN layers respectively for learning the source code representation $\mathbf{E}'_c \in \mathbb{R}^{l_c \times d}$ and the AST representation $\mathbf{E}'_n \in \mathbb{R}^{l_n \times d}$. Then, the decoding process is performed to yield the decoded vector $\mathbf{e}'_s \in \mathbb{R}^d$ for the predicted summary token by fusing the learned source code and AST features (i.e., $\mathbf{E}'_c$ and $\mathbf{E}'_n$) as an initial state for decoding $\mathbf{E}_s^0$. At the decoding stage, we build MPG stacked on the decoder and encoders to predict the next summary token s_m by selecting from summary vocabulary or copying from the input source code and AST tokens. The detailed process will be further described in the following sub-sections.

3.2 Initial Embeddings

Before feeding code tokens, AST nodes, and summary tokens into neural networks, it is essential to embed them into dense numerical vectors. In this work, the source code tokens $\mathbf{T}_c$, AST nodes $\mathbf{T}_n$, and summary tokens $\mathbf{T}_s$ are all embedded into numeric vectors with their related positions $\mathbf{P}_c$, $\mathbf{P}_n$, and $\mathbf{P}_s$ by employing learnable positional embeddings (Gehring et al., 2017). In particular for AST, we take each triplet position as an individual tuple, and directly map each tuple into a positional embedding. The embedding processes are formulated as follows:

$$\begin{aligned}\mathbf{E}_c^0 &= \mathit{CNEmb}(\mathbf{T}_c) * \sqrt{d} + \mathit{CPEmb}(\mathbf{P}_c)\,,\\ \mathbf{E}_n^0 &= \mathit{CNEmb}(\mathbf{T}_n) * \sqrt{d} + \mathit{NPEmb}(\mathbf{P}_n)\,,\\ \mathbf{E}_s^0 &= \mathit{SEmb}(\mathbf{T}_s) * \sqrt{d} + \mathit{SPEmb}(\mathbf{P}_s)\,,\end{aligned} \tag{1}$$

where *CNEmb* denotes the shared embedding operation for source code tokens and AST nodes; *SEmb* means the token embedding operation for summary text; *CPEmb*, *NPEmb*, and *SPEmb* are the corresponding positional embedding operations.

3.3 Code Representation

Source Code Encoder. As shown in Figure 2, the code encoder is composed of two identical layers. And each layer consists of two sub-layers: multi-head attention mechanism and fully connected position-wise feed-forward network (FFN). In addition, residual connection (He et al., 2016) and layer normalization (Ba et al., 2016) are performed in the two sub-layers for the sake of vanishing gradient problem in multi-layer processing and high offset of vectors in residual connection. For the k-th layer, the process can be formulated as:

$$\begin{aligned}\mathbf{H}_c^k &= \mathit{LayerNorm}(\mathbf{E}_c^{k-1} + \mathit{Att}(\mathbf{E}_c^{k-1}, \mathbf{E}_c^{k-1}, \mathbf{E}_c^{k-1}))\,,\\ \mathbf{E}_c^k &= \mathit{LayerNorm}(\mathbf{H}_c^k + \mathit{FFN}(\mathbf{H}_c^k))\,,\end{aligned} \tag{2}$$

where $\mathbf{E}_c^{k-1} \in \mathbb{R}^{l_c \times d}$ is the output vectors from the $(k-1)$-th layer ; *LayerNorm* denotes layer normalization; and *Att* means the multi-head attention (Vaswani et al., 2017) that takes query, key, and value vectors as inputs.

AST Encoder. Considering that AST is a kind of graph, it can be learned by GNNs. Since GraphSAGE (Hamilton et al., 2017) shows high efficiency and performance dealing with graphs, we introduce the idea of GraphSAGE and improve it by adding residual connection for AST encoding, as shown in Figure 2. The encoding layer processes the AST by firstly aggregating the neighbors of the nodes with edge information and then updating the nodes with their aggregated neighborhood information. For a node i and its neighbors in the k-th layer, the process can be formulated as follows:

$$\mathbf{h}_i^k = \mathbf{W}_1 \cdot \mathbf{e}_i^{k-1} + \mathbf{W}_2 \cdot \mathit{Aggr}(\{\mathbf{e}_j^{k-1}, \forall j \in \mathcal{N}(i)\})\,, \tag{3}$$

where $\mathbf{e}_i^{k-1} \in \mathbb{R}^d$ means the vector representation of i-th node from the $(k-1)$-th layer; $\mathcal{N}(i)$ is the neighbors of the node i; $\mathbf{e}_j^{k-1} \in \mathbb{R}^d$ denotes the j-th neighbor vector for node i; $\mathbf{W}_1, \mathbf{W}_2 \in \mathbb{R}^{d \times d}$ are learnable weight matrices; *Aggr* represents aggregation function.

After updating the node information, the node vectors are put together into a *ReLU* activation for non-linear transformation:

$$\mathbf{H}_n^k = \mathit{ReLU}([\mathbf{h}_1^k, \mathbf{h}_2^k, \ldots, \mathbf{h}_i^k, \ldots])\,. \tag{4}$$

With the increase of the number of layers, a node aggregates the neighborhood information from a deeper depth. In order to achieve strong capability of aggregation, the AST encoder is composed of six layers. And to mitigate gradient vanishing and high offset caused by multi-layer processing, we adopt residual connection (He et al., 2016) and layer normalization (Ba et al., 2016) in each layer for improvement, which is formulated as follows:

$$\mathbf{E}_n^k = \mathit{LayerNorm}(\mathbf{H}_n^k + \mathbf{E}_n^{k-1})\,. \tag{5}$$

Note that, $\mathbf{E}_n^{k-1} \in \mathbb{R}^{l_n \times d}$ in this formula denotes the output vectors of nodes from the $(k-1)$-th layer.

3.4 Summary Decoder

The decoder of CodeScribe is designed with six stacks of modified Transformer decoding blocks. Given the existing summary tokens, the k-th decoding block firstly encodes them by masked multi-head attention with residual connection and layer normalization, which is formalized as:

$$\mathbf{H}_s^k = \mathit{LayerNorm}(\mathbf{E}_s^{k-1} + \mathit{MaskAtt}(\mathbf{E}_s^{k-1}, \mathbf{E}_s^{k-1}, \mathbf{E}_s^{k-1}))\,, \tag{6}$$

where $\mathbf{E}_s^{k-1} \in \mathbb{R}^{l_s \times d}$ is the output vectors from the $(k-1)$-th layer and $MaskAtt$ denotes the masked multi-head attention (Vaswani et al., 2017).

After that, we expand the Transformer block by leveraging two multi-head attention modules to interact with the two encoders for summary decoding. One multi-head attention module is performed over the AST features to get the first-stage decoded information, which will then be fed into the other over the learned source code for the second-stage decoding. Then the decoded summary vectors are put into FFN for non-linear transformation. The process can be formalized as follows:

$$\begin{aligned}\mathbf{H}_{s,n}^k &= \mathit{LayerNorm}(\mathbf{H}_s^k + \mathit{Att}(\mathbf{H}_s^k, \mathbf{E}_n', \mathbf{E}_n'))\,,\\ \mathbf{H}_{s,c}^k &= \mathit{LayerNorm}(\mathbf{H}_{s,n}^k + \mathit{Att}(\mathbf{H}_{s,n}^k, \mathbf{E}_c', \mathbf{E}_c'))\,,\\ \mathbf{E}_s^k &= \mathit{LayerNorm}(\mathbf{H}_{s,c}^k + \mathit{FFN}(\mathbf{H}_{s,c}^k))\,,\end{aligned} \tag{7}$$

where $\mathbf{E}'_n$ and $\mathbf{E}'_c$ are the learned features of AST nodes and code tokens, respectively.

3.5 Multi-Source Pointer-Generator Network

We present a multi-source pointer-generator network (MPG) on top of the decoder and encoders to yield the final probability distribution of the next summary token. Considering that tokens such as function names and variable names appear both in code and summary text (Ahmad et al., 2020), MPG is designed to allow CODESCRIBE to generate summary tokens both from the summary vocabulary and from the AST and source code.

Taking the m-th output token as an example, to get the first probability distribution $\mathbf{p}_v$, a *Linear* sub-layer with *Softmax* is applied over the decoded summary token vector $\mathbf{e}'_s \in \mathbb{R}^d$, as follows:

$$\mathbf{p}_v = \mathit{Softmax}(\mathit{Linear}(\mathbf{e}'_s)) \,. \tag{8}$$

For a token w, $\mathbf{p}_v(w) = 0$ if w is an out-of-vocabulary word to the summary vocabulary.

As for the distributions $\mathbf{p}_c$ and $\mathbf{p}_n$, we only describe $\mathbf{p}_c$ since the two have the similar calculation process. In detail, our model applies an additional multi-head attention layer stacked on the last code encoding block and summary decoding block. It takes the decoded summary token vector $\mathbf{e}'_s \in \mathbb{R}^d$ as query and the encoded code information $\mathbf{E}'_c \in \mathbb{R}^{l_c \times d}$ as key and value:

$$\begin{aligned}
\boldsymbol{\delta}_c &= \mathit{Att}(\mathbf{e}'_s, \mathbf{E}'_c, \mathbf{E}'_c) \,, \\
\boldsymbol{\alpha}_c &= \mathit{Softmax}(\mathit{Mean}(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_i, \ldots)) \,, \\
\mathbf{a}_i &= \mathit{Softmax}\left(\frac{\mathbf{e}'_s \mathbf{W}_i^Q (\mathbf{E}'_c \mathbf{W}_i^K)^T}{\sqrt{d}}\right)(\mathbf{E}'_c \mathbf{W}_i^V) \,,
\end{aligned} \tag{9}$$

where $\mathbf{W}_i^Q$, $\mathbf{W}_i^K$, and $\mathbf{W}_i^V$ are learnable parameters. The context vector $\boldsymbol{\delta}_c \in \mathbb{R}^d$ will be used for the final distribution. Through the function *Mean* and *Softmax*, the attention vectors $(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_i, \ldots)$ of all heads are averaged as $\boldsymbol{\alpha}_c \in \mathbb{R}^{l_c}$. For the token w, its probability $\mathbf{p}_c(w)$ is formulated as follows:

$$\mathbf{p}_c(w) = \textstyle\sum_{i: w_i = w} \boldsymbol{\alpha}_{ci} \,, \tag{10}$$

where w_i means the i-th token in the source code.

Similarly, we can get $\boldsymbol{\delta}_n$ and $\mathbf{p}_n$ corresponding to the AST. After that, the final probability $p_s(w)$ of the token w is defined as a mixture of the three probabilities:

$$\begin{aligned}
&\mathbf{p}_s(w) = \lambda_v \cdot \mathbf{p}_v(w) + \lambda_c \cdot \mathbf{p}_c(w) + \lambda_n \cdot \mathbf{p}_n(w) \,, \\
&[\lambda_v, \lambda_c, \lambda_n] = \mathit{Softmax}(\mathit{Linear}([\mathbf{e}'_s, \boldsymbol{\delta}_c, \boldsymbol{\delta}_n])) \,,
\end{aligned} \tag{11}$$

where λ_v, λ_c, and λ_n are the weight values for $\mathbf{p}_v(w)$, $\mathbf{p}_c(w)$, and $\mathbf{p}_n(w)$. The higher the probability $\mathbf{p}_s(w)$ is, the more likely the token w is considered as the next summary token.

4 Experiments

We conduct experiments to answer the following research questions: *(1) How effective is* CODESCRIBE *compared with the state-of-the-art baselines? (2) How effective is the structure design of* CODESCRIBE*? (3) What is the impact of model size on the performance of* CODESCRIBE*?* We also perform a qualitative analysis of two detailed examples.

4.1 Datasets

The experiments are conducted based on two benchmarks: (1) Java dataset (Hu et al., 2018a) and (2) Python dataset (Wan et al., 2018). The two datasets are split into *train/valid/test* sets with *69,708/8,714/8,714* and *55,538/18,505/18,502*, respectively. In the experiments, we follow the divisions for the fairness of the results.

In the data preprocessing, *NLTK* package (Loper and Bird, 2002) is utilized for the tokenization of source code and summary text. And we apply *javalang* [2] and *ast* [3] packages to parsing Java and Python code into ASTs. In addition, the tokens in forms of "*CammelCase*", "*snake_case*", and "*concatenatecase*" are split into sub-tokens as "*Cammel Case*", "*snake case*", and "*concatenate case*".

4.2 Implementation Details

We leverage PyTorch 1.9 for CODESCRIBE implementation. The model runs under the development environment of Python 3.9 with NVIDIA 2080 Ti GPUs and CUDA 10.2 supported.

We follow the previous works (Ahmad et al., 2020; Choi et al., 2021) and set both the embedding size of code tokens, AST nodes, and summary tokens to 512, and the number of attention headers to 8. As described in Section 3, the numbers of layers of code encoder, AST encoder, and summary decoder are 2, 6, and 6, respectively.

The model is trained with Adam optimizer (Kingma and Ba, 2014). We initialize the learning rate as $5e^{-4}$ that will be decreased by 5% after each training epoch until to $2.5e^{-5}$. The

[2] https://github.com/c2nes/javalang

[3] https://github.com/python/cpython/blob/master/Lib/ast.py

Model	Java			Python		
	BLEU(%)	METEOR(%)	ROUGE-L(%)	BLEU(%)	METEOR(%)	ROUGE-L(%)
CODE-NN (Iyer et al., 2016b)	27.60	12.61	41.10	17.36	09.29	37.81
Tree2Seq (Eriguchi et al., 2016)	37.88	22.55	51.50	20.07	08.96	35.64
RL+Hybrid2Seq (Wan et al., 2018)	38.22	22.75	51.91	19.28	09.75	39.34
DeepCom (Hu et al., 2018a)	39.75	23.06	52.67	20.78	09.98	37.35
API+CODE (Hu et al., 2018b)	41.31	23.73	52.25	15.36	08.57	33.65
Dual Model (Wei et al., 2019)	42.39	25.77	53.61	21.80	11.14	39.45
CopyTrans (Ahmad et al., 2020)	44.58	26.43	54.76	32.52	19.77	46.73
mAST+GCN (Choi et al., 2021)	45.49	27.17	54.82	32.82	20.12	46.81
CODESCRIBE	**49.19**	**32.27**	**59.59**	**35.11**	**23.48**	**50.46**

Table 1: Comparison with the baselines on the Java and Python datasets.

dropout rate is set to 0.2. We set the batch size to 96 and 160 for the Java and Python datasets, respectively. The training process will terminate after 100 epochs or stop early if the performance does not improve for 10 epochs. In addition, we leverage beam search (Koehn, 2004) during the model inference and set the beam width to 5.

4.3 Baselines

We introduce eight state-of-the-art works as baselines for comparison, including six RNN-based models and two Transformer-based models.

RNN-based Models. Among these baselines, CODE-NN (Iyer et al., 2016b), API+CODE (Hu et al., 2018b), and Dual Model (Wei et al., 2019) learn source code for summarization. Tree2Seq (Eriguchi et al., 2016) and DeepCom (Hu et al., 2018a) generate summaries from AST features. RL+Hybrid2Seq (Wan et al., 2018) combines source code and AST based on LSTM.

Transformer-based Models. The two baselines include CopyTrans (Ahmad et al., 2020) and mAST+GCN (Choi et al., 2021), both of which leverage Transformer for code summary generation. The main difference is that CopyTrans learns sequential source code, and mAST+GCN is built based on AST.

For the model evaluation, three metrics are introduced: BLEU (Papineni et al., 2002), METEOR (Banerjee and Lavie, 2005), and ROUGE (Lin, 2004).

4.4 Comparison with the Baselines (RQ1)

We first evaluate the performance of CODESCRIBE by comparing it with eight state-of-the-art baselines. The results of baselines are all from Choi et al. (2021) and are shown in Table 1.

The overall results in Table 1 illustrate that the recent Transformer-based models (Ahmad et al., 2020; Choi et al., 2021) are superior to the previous works based on RNNs (Iyer et al., 2016b; Eriguchi et al., 2016; Wan et al., 2018; Hu et al., 2018a,b; Wei et al., 2019). Although the two models CopyTrans and mAST+GCN have high performance in code summarization, our approach CODESCRIBE performs much better than them both on the two datasets. Intuitively, CODESCRIBE improves the performance (i.e., BLEU/METEOR/ROUGE-L) by 4.46/5.84/4.83% on the Java dataset and 2.59/3.71/3.73% on the Python dataset compared to CopyTrans. In comparison with mAST+GCN, the performance of CODESCRIBE improves by 3.70/5.10/4.77% on the Java dataset and 2.29/3.36/3.65% on the Python dataset.

The comparison demonstrates the outperformance of CODESCRIBE. It indicates that: (1) Transformer-like models are more effective than RNN-based models in code summarization task; (2) AST information contributes significantly to code comprehension; and (3) by incorporating both AST and source code into CODESCRIBE based on GraphSAGE and Transformer, the performance can be greatly improved due to its more comprehensive learning capacity for code and better decoding for summary generation.

4.5 Ablation Study (RQ2)

This section validates the effectiveness of CODESCRIBE's structure to by performing an ablation study. To this end, we firstly design five models for comparison that remove one of important components in CODESCRIBE including: (1) the AST encoder (R-AST), (2) the source code encoder (R-Code), (3) the triplet positions (R-ASTPos), (4) the MPG (R-Copy), and (5) the residual connection in the AST encoder (R-ASTRes). We further investigate the rationality of CODESCRIBE's structure through the comparison with five variants: (1) V-Copy that replaces MPG with the copying mechanism (See et al., 2017) used in Ahmad et al. (2020), (2) V-GCN that replaces GraphSAGE with GCN (Kipf and Welling, 2016), (3) V-GAT that replaces GraphSAGE with GAT (Kipf and Welling, 2016), (4) V-Emb that replaces the shared embedding layer for code tokens and AST nodes with two independent embedding layers, and (5) V-Dec that reverses the decoding order for the source code and AST features.

Model	BLEU(%)	METEOR(%)	ROUGE-L(%)
R-AST	46.45	29.37	56.42
R-Code	47.06	30.06	57.03
R-ASTPos	48.53	31.62	58.84
R-Copy	48.64	31.71	58.68
R-ASTRes	13.03	2.59	5.89
V-Copy	48.59	31.82	58.73
V-GCN	48.84	31.96	58.95
V-GAT	48.84	32.03	59.23
V-Emb	49.05	31.93	58.95
V-Dec	48.99	32.11	59.31
CODESCRIBE	**49.19**	**32.27**	**59.59**

Table 2: Ablation study on the Java dataset.

As shown in Table 2, the performance of CODESCRIBE is affected if the components are removed. The results of R-AST and R-Code show that the two encoders are the most significant learning components to the framework. Moreover, the AST encoder is more important than the code encoder as R-Code performs better than R-AST. It further demonstrates that AST contains richer structural features than source code that is beneficial to summary generation. The performances of R-ASTPos and R-Copy indicate that the triplet positions for nodes and copying mechanism (MPG) we proposed are effective for CODESCRIBE in code summarization. In addition, we find that R-ASTRes suffers from under-fitting on the Java dataset, which indicates that the residual connection in AST encoder has a powerful influence on CODESCRIBE.

As illustrated in Table 2, CODESCRIBE improves the performance by 0.26/0.22/0.30% on the Java dataset compared with V-Copy. It indicates that our proposed MPG is more effective than the copying mechanism in Ahmad et al. (2020). As for the GNN module in AST encoding, it can be observed that CODESCRIBE still has the higher performance than V-GCN and V-GAT. This demonstrates the superiority of GrahpSAGE for the architecture of CODESCRIBE compared to GCN and GAT. Compared with V-Emb, it shows that the shared embedding layer works better than two separated embedding layers for AST and source code. The result of V-Dec turns out that the performance will not be affected sinificantly if the order of decoding over AST and code features is reversed. The results on the Python dataset are presented in Table 7 in Appendix A.

4.6 Study on the Model Size (RQ3)

This section studies the performance of CODESCRIBE with the change of model size. To that end, we modify the number of layers of the encoders and the decoder respectively for observation.

AST Layers	Model Size($\times 10^6$)	BLEU(%)	METEOR(%)	ROUGE-L(%)
2	38.89	48.68	31.76	58.77
4	39.94	48.76	31.99	59.10
6	*40.99*	***49.19***	***32.27***	***59.59***
8	42.05	49.11	32.20	59.49
10	43.10	48.97	32.12	59.23
12	44.15	48.84	32.06	59.10

Table 3: Performance of CODESCRIBE with different numbers of AST encoding layers on the Java dataset.

Code Layers	Model Size($\times 10^6$)	BLEU(%)	METEOR(%)	ROUGE-L(%)
2	*40.99*	***49.19***	***32.27***	***59.59***
4	47.30	48.80	32.15	59.32
6	53.60	48.92	32.10	59.30
8	59.91	48.73	31.95	58.95
10	66.21	49.11	31.97	59.09
12	72.52	48.36	31.59	58.59

Table 4: Performance of CODESCRIBE with different numbers of code encoding layers on the Java dataset.

Summary Layers	Model Size($\times 10^6$)	BLEU(%)	METEOR(%)	ROUGE-L(%)
2	19.97	47.99	31.21	58.50
4	30.48	48.80	32.02	59.32
6	*40.99*	*49.19*	*32.27*	***59.59***
8	51.51	49.16	32.20	59.33
10	62.02	49.16	**32.33**	59.56
12	72.53	**49.24**	32.31	59.41

Table 5: Performance of CODESCRIBE with different numbers of decoding layers on the Java dataset.

Table 3 presents the performance of CODESCRIBE when the number of AST encoding layers varies from 2 to 12. The results show that the performance improves as the number of AST encoding layers increases from 2 to 6. With the increase of the number from 6 to 12, the performance does not improve any more and is even impacted slightly. As illustrated in Table 4, CODESCRIBE has the best performance with 2 code encoding layers. With the number of code layers growing from 4 to 12, there is a trend of gradual decrease of the performance. For the model size concerned with summary decoding layers, as shown in Table 5, the performance is getting better when the number of layers ranges

	Java	Python

Code	`public void addMessage(String message){`
	`  messages.addLast(message);`
	`  if (messages.size() > MAX_HISTORY) {`
	`    messages.removeFirst();`
	`  }`
	`  pointer=messages.size();`
	`}`
		`@_get_client`
		`def image_create(client, values,`
		`↪  v1_mode=False):`
		` return client.image_create(values=values,`
		` ↪  v1_mode=v1_mode)`

Summary	Gold: add a message to the history R-AST: add a message to the end of the list R-Copy: add a message to the history . V-GCN: add a message to the list V-Dec: add a message to the list CODESCRIBE: add a message to the history .	Gold: create an image from the value dictionary . R-AST: create an image cli example : . R-Copy: create an image mode that can exist from the give value . V-GCN: create an image from a v <number> image . V-Dec: create an image object . CODESCRIBE: create an image from the value dictionary .

Table 6: Qualitative examples on the Java and Python datasets.

from 2 to 6, and can not be improved as the number continues to increase. The overall results show that it the performance of CODESCRIBE will not be improved if the encoders and the decoder become too deep (i.e. with more layers), especially for the source code encoder. More experimental results are provided in Table 8 - 11 in Appendix B.

4.7 Case Study

Table 6 shows the qualitative examples of R-AST, R-Copy, V-GCN, V-Dec, and CODESCRIBE. From the table, it can be observed that CODESCRIBE with the whole architecture generates better code summaries compared with the four variants. In the case on the Java dataset, only R-Copy and CODESCRIBE get the right intent of the code. The other variants miss out the key word "*history*". In the case on the Python dataset, CODESCRIBE generates the most accurate summary compared to the other variants. In contrast, although the four variants output the first half of the summary (i.e., "*create an image*"), the rest information "*from the value dictionary .*" can not be generated correctly. More qualitative examples are referred to Table 12 and 13 in Appendix C.

5 Related Work

With the development of deep learning, most works have considered code summarization as a sequence generation task. In many of the recent approaches, source code snippets are modeled as plain texts based on RNNs (Iyer et al., 2016b; Hu et al., 2018b; Wei et al., 2019; Ye et al., 2020). Most recently, Ahmad et al. (2020) applied Transformer to encoding the source code sequence to improve the summarization performance.

Since considering source code as plain text ignores the structural information in code, recent works have explored the AST of code and modeled the tree-based structure for code summarization. Typically, some approaches (Hu et al., 2018a; Alon et al., 2018) converted the AST to node sequence(s) and used LSTMs for learning. Others leveraged Tree-LSTM (Shido et al., 2019) or GNNs (Liu et al., 2020) to capture the structural features of code. The latest work (Choi et al., 2021) performed graph convolutional network (GCN) (Kipf and Welling, 2016) before Transformer framework to learn AST representation for summary generation.

To represent the code comprehensively, more and more works pay attention to both the source code and the AST for code summarization. For example, Hu et al. (2020) integrated both AST node sequence and source code into a hybrid learning framework based on GRUs. Wei et al. (2020) and Zhang et al. (2020) both utilized the information retrieval techniques to improve the quality of code comments that are generated from the code snippets and ASTs. The rest methods (Wan et al., 2018; LeClair et al., 2020; Wang et al., 2020) learned the source code based on RNNs and modeled the tree structure of code using AST-based LSTM or GCNs. Different from all these works, we combine GraphSAGE and Transformer to both learn the AST and source code.

6 Conclusion

This paper has presented CODESCRIBE, an encoder-decoder-based neural network for source code summarization. CODESCRIBE designs a triplet position to model the hierarchical syntax structure of code, which is then incorporated into Transformer and GNNs for better representation of lexical and syntax information of code, respectively. The performance of CODESCRIBE is further enhanced by the introduced multi-source pointer generator in decoding. Experiments on two benchmarks reveal that the summaries generated by CODESCRIBE are of higher quality compared to recent state-of-the-art works.

References

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)*.

M. Allamanis, H. Peng, and C. Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, pages 2091–2100.

Uri Alon, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *CoRR*, abs/1808.01400.

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

Satanjeev Banerjee and Alon Lavie. 2005. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments.

Y. S. Choi, J. Y. Bak, C. W. Na, and J. H. Lee. 2021. Learning sequential and structural information for source code summarization. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*.

S. C. B. de Souza, N. Anquetil, and K. M. 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*, pages 68–75. ACM.

A. Eriguchi, K. Hashimoto, and Y. Tsuruoka. 2016. Tree-to-sequence attentional neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. 2017. Convolutional sequence to sequence learning. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1243–1252. PMLR.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE.

William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1025–1035, Red Hook, NY, USA. Curran Associates Inc.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778. IEEE.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018a. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*, 25(3):2179–2217.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018b. Summarizing source code with transferred api knowledge.

S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. 2016a. Summarizing source code using a neural attention model. In *ACL (1)*.

S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. 2016b. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Mira Kajko-Mattsson. 2005. A survey of documentation practice within corrective maintenance. *Empirical Software Engineering*, 10(1):31–55.

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

Philipp Koehn. 2004. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In *Conference of the Association for Machine Translation in the Americas*, pages 115–124. Springer.

Alex LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *2020 IEEE/ACM International Conference on Program Comprehension (ICPC)*.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. page 10.

Shangqing Liu, Yu Chen, Xiaofei Xie, Jingkai Siow, and Yang Liu. 2020. Automatic code summarization via multi-dimensional semantic fusing in gnn. *arXiv preprint arXiv:2006.05405*.

Edward Loper and Steven Bird. 2002. Nltk: The natural language toolkit. In *In Proceedings of the ACL Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics. Philadelphia: Association for Computational Linguistics*.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, pages 1287–1293. AAAI Press.

A. T. Nguyen and T. N. Nguyen. 2017. Automatic categorization with deep neural network for open-source java projects. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 164–166. IEEE Press.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation.

Abigail See, Peter J Liu, and Christopher D Manning. 2017. Get to the point: Summarization with pointer-generator networks. *arXiv preprint arXiv:1704.04368.*

Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. c. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

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*, pages 5998–6008. Curran Associates, Inc.

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*, pages 397–407. ACM.

Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, Philip Yu, and Guandong Xu. 2020. Reinforcement-learning-guided source code summarization via hierarchical attention. *IEEE Transactions on Software Engineering*.

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*, pages 6563–6573.

Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and refine: exemplar-based neural comment generation. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 349–360. IEEE.

Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. Coacor: Code annotation for code retrieval with reinforcement learning. In *The World Wide Web Conference*, pages 2203–2214. ACM.

Wei Ye, Rui Xie, Jinglei Zhang, Tianxiang Hu, Xiaoyin Wang, and Shikun Zhang. 2020. Leveraging code generation to improve code retrieval and summarization via dual learning. In *Proceedings of The Web Conference 2020*, pages 2309–2319.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *Proceedings of the 42nd International Conference on Software Engineering. IEEE.*

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 IEEE/ACM 41st International Conference on Software Engineering*, pages 783–794. IEEE.

A Results of Ablation Study

Table 7 shows the results of ablation study on the Python dataset. It can be observed that CODESCRIBE has the best performance in contrast with all the variants except V-Dec. Although there is no under-fitting for R-ASTRes on the Python dataset, we can find that the performance (i.e., BLEU/METEOR/ROUGE-L) is reduced by 1.02/0.89/1.51 if the residual connection in AST encoder is excluded. So it also demonstrates the effectiveness of this component to the AST encoder. In addition, the result of V-Dec still confirms the conclusion that the order of decoding over AST and source code features won't impact the performance of CODESCRIBE.

Model	BLEU(%)	METEOR(%)	ROUGE-L(%)
R-AST	32.97	21.24	47.70
R-Code	33.54	21.91	48.61
R-ASTPos	34.50	22.91	49.79
R-Copy	34.55	23.16	49.88
R-ASTRes	34.09	22.59	48.95
V-Copy	34.85	23.26	50.16
V-GCN	34.73	23.24	50.11
V-GAT	34.88	23.27	50.25
V-Emb	34.55	22.80	49.16
V-Dec	35.04	23.41	50.40
CODESCRIBE	**35.11**	**23.48**	**50.46**

Table 7: Ablation study on the Python dataset.

B Results of Study on the Model Size

The additional results of study on the model size on the Python dataset are described in the Table 8, 9, and 10. The performances show the similar change trends with that on the Java dataset. For example, Table 9 shows that the performance of CODESCRIBE does not improve with the number increasing from 2 to 12.

AST Layers	Model Size($\times 10^6$)	BLEU(%)	METEOR(%)	ROUGE-L(%)
2	38.89	34.81	23.27	50.12
4	39.94	34.76	23.26	50.25
6	*40.99*	***35.11***	***23.48***	***50.46***
8	42.05	35.02	23.38	50.34
10	43.10	34.88	23.35	50.22
12	44.15	34.97	23.26	50.14

Table 8: Performance of CODESCRIBE with different numbers of AST encoding layers on the Python dataset.

Code Layers	Model Size($\times 10^6$)	BLEU(%)	METEOR(%)	ROUGE-L(%)
2	*40.99*	*35.11*	*23.48*	*50.46*
4	47.30	34.99	23.43	50.37
6	53.60	34.86	23.32	50.33
8	59.91	35.08	**23.58**	**50.61**
10	66.21	**35.16**	23.41	50.18
12	72.52	34.94	23.21	49.87

Table 9: Performance of CODESCRIBE with different numbers of code encoding layers on the Python dataset.

Summary Layers	Model Size($\times 10^6$)	BLEU(%)	METEOR(%)	ROUGE-L(%)
2	19.97	34.16	22.92	49.70
4	30.48	34.75	23.32	50.29
6	*40.99*	*35.11*	*23.48*	*50.46*
8	51.51	34.90	23.43	50.37
10	62.02	35.08	23.49	50.56
12	72.53	**35.19**	**23.59**	**50.58**

Table 10: Performance of CODESCRIBE with different numbers of summary decoding layers on the Python dataset.

We further provide the results of CODESCRIBE by varying the embedding size from 128 to 1024 with the interval of 128. As depicted in Table 11, CODESCRIBE has the worst performance with the embedding size 128, and performs much better when the size becomes 256. Then the performance improves steadily as the embedding size increases until to 512. After that, although CODESCRIBE can be boosted with the growth of embedding size (from 512 to 1024), the improvement is not so obvious. These observations suggest that expanding the embedding size properly is indeed effective to CODESCRIBE. However, excessive expansion will not help much for the improvement.

C Qualitative Examples

Table 12 and 13 provide qualitative examples of R-AST, R-Copy, V-GCN, V-Dec, and our CODESCRIBE on the Java and Python datasets for case study. The overall results show that CODESCRIBE generates better summaries for the given code snippets. For instance, in the first case in Table 12, only R-Copy and CODESCRIBE get the right intent of the code. In the third case in Table 12, only CODESCRIBE grasps the key information, i.e., "status panel". In the first case in Table 13, CODESCRIBE generates the most accurate summary compared to the other variants, which is the same in the second case.

Emb. Size	Model Size($\times 10^6$)	Java S-BLEU(%)	Java METEOR(%)	Java ROUGE-L(%)	Python S-BLEU(%)	Python METEOR(%)	Python ROUGE-L(%)
128	2.58	33.55	22.31	47.89	26.83	18.53	43.68
256	10.27	44.24	28.62	55.36	32.19	21.50	47.54
384	23.08	48.16	31.56	58.67	34.34	22.99	49.73
512	*40.99*	*49.19*	*32.27*	***59.59***	*35.11*	*23.48*	*50.46*
640	64.02	49.17	32.29	59.45	35.31	23.62	**50.59**
768	92.16	49.20	**32.32**	59.28	**35.35**	23.69	50.59
896	125.41	49.19	32.26	59.34	35.55	23.75	50.56
1024	163.78	**49.32**	32.29	59.35	35.20	23.56	50.29

Table 11: Performance of CODESCRIBE with different embedding sizes on the Java and Python datasets.

Code	`public void addMessage(String message){`
	`  messages.addLast(message);`
	`  if (messages.size() > MAX_HISTORY) {`
	`    messages.removeFirst();`
	`  }`
	`  pointer=messages.size();`
	`}`

Summary	Gold: add a message to the history R-AST: add a message to the end of the list R-Copy: add a message to the history . V-GCN: add a message to the list V-Dec: add a message to the list CODESCRIBE: add a message to the history .

Code	`public void hspan(double start,double end,Paint color,String legend){`
	`  LegendText legendText=new LegendText(color,legend);`
	`  comments.add(legendText);`
	`  plotElements.add(new HSpan(start,end,color,legendText));`
	`}`

Summary	Gold: draw a horizontal span into the graph and optionally add a legend . R-AST: plot request data a a vertical and optionally add a legend . R-Copy: draw a vertical span into the graph and optionally add a legend . V-GCN: draw the current legend . V-Dec: plot request data a a line , use the color and the line width specify . CODESCRIBE: draw a vertical span into the graph and optionally add a legend .

Code	`public CStatusPanel(final BackEndDebuggerProvider debuggerProvider){`
	`  super(new BorderLayout());`
	`  Preconditions.checkNotNull(debuggerProvider,"IE1094: Debugger provider argument can not be`
	`  ↪   null");`
	`  m_label.setForeground(Color.BLACK);`
	`  add(m_label);`
	`  m_synchronizer=new CStatusLabelSynchronizer(m_label,debuggerProvider);`
	`}`

Summary	Gold: create a new status panel . R-AST: create a new panel . R-Copy: create a new panel object . V-GCN: create a new debugger panel . V-Dec: create a new panel object . CODESCRIBE: create a new status panel object .

Code	`private Spannable highlightHashtags(Spannable text){`
	`  if (text == null) {`
	`    return null;`
	`  }`
	`  final Matcher matcher=PATTERN_HASHTAGS.matcher(text);`
	`  while (matcher.find()) {`
	`    final int start=matcher.start(1);`
	`    final int end=matcher.end(1);`
	`    text.setSpan(new`
	`    ↪   ForegroundColorSpan(mHighlightColor),start,end,Spanned.SPAN_EXCLUSIVE_EXCLUSIVE);`
	`    text.setSpan(new`
	`    ↪   StyleSpan(android.graphics.Typeface.BOLD),start,end,Spanned.SPAN_EXCLUSIVE_EXCLUSIVE);`
	`  }`
	`  return text;`
	`}`

Summary	Gold: highlight all the hash tag in the pass text . R-AST: highlight all the text in the pass text . R-Copy: highlight all the hash text in the pass text . V-GCN: highlight all the span of the text . V-Dec: highlight all the occurrence of a match tag in the pass text . CODESCRIBE: highlight all the hash line in the pass text .

Table 12: Qualitative examples on the Java dataset.

Code	`@_get_client`
	`def image_create(client, values, v1_mode=False):`
	` return client.image_create(values=values, v1_mode=v1_mode)`

Summary	Gold: create an image from the value dictionary . R-AST: create an image cli example : . R-Copy: create an image mode that can exist from the give value . V-GCN: create an image from a v <number> image . V-Dec: create an image object . CODESCRIBE: create an image from the value dictionary .

Code	`def test_help_command_should_exit_status_ok_when_no_cmd_is_specified(script):`
	` result = script.pip('help')`
	` assert (result.returncode == SUCCESS)`

Summary	Gold: test help command for no command . R-AST: test help command for exist command . R-Copy: test help command for exist command . V-GCN: test help command for exist command . V-Dec: test help command for exist command . CODESCRIBE: test help command for no command .

Code	`def all_editable_exts():`
	` exts = []`
	` for (language, extensions) in sourcecode.ALL_LANGUAGES.items():`
	`  exts.extend(list(extensions))`
	` return [('.' + ext) for ext in exts]`

Summary	Gold: return a list of all editable extension . R-AST: return a list of all python extension . R-Copy: return a list of tuples extension for all editable s . V-GCN: return a list of all file extension that be editable by the extension . V-Dec: return a list of all available extension . CODESCRIBE: return a list of all editable s extension .

Code	`def update_featured_activity_references(featured_activity_references):`
	` for activity_reference in featured_activity_references:`
	`  activity_reference.validate()`
	` activity_hashes = [reference.get_hash() for reference in featured_activity_references]`
	` if (len(activity_hashes) != len(set(activity_hashes))):`
	`  raise Exception('The activity reference list should not have duplicates.')`
	` featured_model_instance =`
	` ↪  activity_models.ActivityReferencesModel.get_or_create(activity_models. \`
	` ACTIVITY_REFERENCE_LIST_FEATURED)`
	` featured_model_instance.activity_references = [reference.to_dict() for reference in`
	` ↪  featured_activity_references]`
	` featured_model_instance.put()`

Summary	Gold: update the current list of feature activity reference . R-AST: add the specify activity reference to the list of feature activity reference . R-Copy: update the specify activity reference from the list of feature activity reference . V-GCN: update the specify activity reference from the list of feature activity reference . V-Dec: update the specify activity reference from the list of feature activity reference . CODESCRIBE: update the list of feature activity reference .

Table 13: Qualitative examples on the Python dataset.