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 lack of 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 baselines1.

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, very 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

---

1The source code of CODESCRIBE is available at https://github.com/anonymousrepoxxx/ CODESCRIBE
The AST of Python code snippet \( \frac{a}{b} = c \). 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.

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 node’s width position among its siblings.
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 \( \text{a=b/c} \) 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 \( T_c = (c_1, c_2, \ldots, c_{l_c}) \) and sequential positions \( P_c = (1, 2, \ldots, l_c) \), and its AST with \( l_n \) nodes \( T_n = (n_1, n_2, \ldots, n_{l_n}) \) and triplet positions \( P_n = \{(x_1, y_1, z_1), (x_2, y_2, z_2)\ldots, (x_{l_n}, y_{l_n}, z_{l_n})\} \), CODESCRIBE predicts the next summary token \( s_m \) based on the existing tokens \( T_s = (r/s_1, s_1, s_2, \ldots, s_{m−1}, \ldots) \) with the sequential positions \( P_s = (1, 2, \ldots, l_s) \), where \( r/s \) is a special starting tag for summary input. Note that \( T_s \) is padded to a maximum length of \( l_s \) with special padding tags (e.g., \( \langle \text{pad} \rangle \)).

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 \( E^{l'_c}_{s} \in \mathbb{R}^{l_c \times d} \), \( E^{0}_{n} \in \mathbb{R}^{l_n \times d} \), and \( E^{0}_{s} \in \mathbb{R}^{l_s \times d} \) 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 \( E'^{l'}_{c} \in \mathbb{R}^{l_c \times d} \) and the AST representation \( E'^{l'}_{n} \in \mathbb{R}^{l_n \times d} \). Then, the decoding process is performed to yield the decoded vector \( e'_s \in \mathbb{R}^d \) for the predicted summary token by fusing the learned source code and AST features (i.e., \( E'^{l'}_{c} \) and \( E'^{l'}_{n} \)) as an initial state for decoding \( E^{0}_{s} \). 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 \( T_c \), AST nodes \( T_n \), and summary tokens \( T_s \) are all embedded into numeric vectors with their related positions \( P_c, P_n, P_s \).
and $P_s$ 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:

$$
E_b^s = CNEmb(T_s) \ast \sqrt{d} + CPEmb(P_s),
$$

$$
E_b^c = CNEmb(T_c) \ast \sqrt{d} + NPEmb(P_n),
$$

$$
E_b^p = SEmb(T_s) \ast \sqrt{d} + SPEmb(P_s),
$$

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:

$$
H^k_s = \text{LayerNorm}(E^k_s) + \text{Att}(E^k_s, E^k_c, E^k_e),
$$

$$
E^k_c = \text{LayerNorm}(H^k_s + \text{FFN}(H^k_s)),
$$

where $E^k_s \in \mathbb{R}^{l_s \times d}$ is the output vectors from the $(k-1)$-th layer; $\text{LayerNorm}$ denotes layer normalization; and $\text{Att}$ 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:

$$
h_i^k = W_1 \cdot e_i^{k-1} + W_2 \cdot \text{Aggr}(\{e_j^{k-1} | \forall j \in \mathcal{N}(i)\}),
$$

where $e_i^{k-1} \in \mathbb{R}^d$ means the vector representation of $i$-th node from the $(k-1)$-th layer; $\mathcal{N}(i)$ is the neighbors of the node $i$; $e_j^{k-1} \in \mathbb{R}^d$ denotes the $j$-th neighbor vector for node $i$; $W_1, W_2 \in \mathbb{R}^{d \times d}$ are learnable weight matrices; $\text{Aggr}$ represents aggregation function.

After updating the node information, the node vectors are put together into a $\text{ReLU}$ activation for non-linear transformation:

$$
H^k_n = \text{ReLU}([h_1^k, h_2^k, \ldots, h_{l_n}^k, \ldots]).
$$

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:

$$
E^k_n = \text{LayerNorm}(H^k_n + E^{k-1}_n).
$$

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

### 3.4 Summary Decoder

The decoder of CODECRIE 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:

$$
H^k_s = \text{LayerNorm}(H^k_s + \text{MaskAtt}(E^k_s, E^k_c, E^k_e)),
$$

where $E^k_s \in \mathbb{R}^{l_s \times d}$ is the output vectors from the $(k-1)$-th layer and $\text{MaskAtt}$ 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:

$$
H^k_{s,n} = \text{LayerNorm}(H^k_s + \text{Att}(H^k_s, E_n^k, E_n^k)),
$$

$$
H^k_{s,c} = \text{LayerNorm}(H^k_{s,n} + \text{Att}(H^k_{s,n}, E_c^k, E_c^k)),
$$

$$
E^k_s = \text{LayerNorm}(H^k_{s,c} + \text{FFN}(H^k_{s,c})).
$$
where $E'_n$ and $E'_c$ 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 $p_v$, a Linear sub-layer with Softmax is applied over the decoded summary token vector $e'_s \in \mathbb{R}^d$, as follows:

$$p_v = \text{Softmax} \left( \text{Linear} \left( e'_s \right) \right).$$

For a token $w$, $p_v(w) = 0$ if $w$ is an out-of-vocabulary word to the summary vocabulary.

As for the distributions $p_c$ and $p_n$, we only describe $p_c$ 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 $e'_s \in \mathbb{R}^d$ as query and the encoded code information $E'_c \in \mathbb{R}^{l_c \times d}$ as key and value:

$$\begin{align*}
\delta_c &= \text{Att} \left( e'_s, E'_c, E'_c \right), \\
\alpha_c &= \text{Softmax} \left( \text{Mean} \left( a_1, a_2, \ldots, a_{l_c} \right) \right), \\
\alpha_i &= \text{Softmax} \left( \epsilon'_s W'^Q \left( E'_c W'^K \right)^T \right) \left( E'_c W'^V \right),
\end{align*}$$

where $W'^Q$, $W'^K$, and $W'^V$ are learnable parameters. The context vector $\delta_c \in \mathbb{R}^d$ will be used for the final distribution. Through the function $\text{Mean}$ and $\text{Softmax}$, the attention vectors $(a_1, a_2, \ldots, a_{l_c})$ of all heads are averaged as $\alpha_c \in \mathbb{R}^{l_c}$. For the token $w$, its probability $p_c(w)$ is formulated as follows:

$$p_c(w) = \sum_{i=w_i}^{w} \alpha_{ci},$$

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

Similarly, we can get $\delta_n$ and $p_n$ 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:

$$p_s(w) = \lambda_v \cdot p_v(w) + \lambda_c \cdot p_c(w) + \lambda_n \cdot p_n(w),$$

where $\lambda_v, \lambda_c,$ and $\lambda_n$ are the weight values for $p_v(w)$, $p_c(w)$, and $p_n(w)$. The higher the probability $p_s(w)$ 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 "CammelCase", "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
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

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

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

The result of V-Dec turns out that the performance will not be affected significantly 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.

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

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

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 GraphSAGE 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.
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.

### 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


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


Yuysuke 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.


## 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.

<table>
<thead>
<tr>
<th>Model</th>
<th>BLEU(%)</th>
<th>METEOR(%)</th>
<th>ROUGE-L(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-AST</td>
<td>32.97</td>
<td>21.24</td>
<td>47.70</td>
</tr>
<tr>
<td>R-Code</td>
<td>33.54</td>
<td>21.91</td>
<td>48.61</td>
</tr>
<tr>
<td>R-ASTPos</td>
<td>34.50</td>
<td>22.91</td>
<td>49.79</td>
</tr>
<tr>
<td>R-Copy</td>
<td>34.55</td>
<td>23.16</td>
<td>49.88</td>
</tr>
<tr>
<td>R-ASTRes</td>
<td>34.09</td>
<td>22.59</td>
<td>48.95</td>
</tr>
<tr>
<td>V-Copy</td>
<td>34.85</td>
<td>23.26</td>
<td>50.16</td>
</tr>
<tr>
<td>V-GCN</td>
<td>34.73</td>
<td>23.24</td>
<td>50.11</td>
</tr>
<tr>
<td>V-GAT</td>
<td>34.88</td>
<td>23.27</td>
<td>50.25</td>
</tr>
<tr>
<td>V-Emb</td>
<td>34.55</td>
<td>22.80</td>
<td>49.16</td>
</tr>
<tr>
<td>V-Dec</td>
<td>35.04</td>
<td>23.41</td>
<td>50.40</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Layers Size</th>
<th>BLEU (%)</th>
<th>METEOR (%)</th>
<th>ROUGE-L (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>38.89</td>
<td>34.81</td>
<td>23.27</td>
</tr>
<tr>
<td>4</td>
<td>39.94</td>
<td>34.76</td>
<td>23.26</td>
</tr>
<tr>
<td>6</td>
<td>40.99</td>
<td>35.11</td>
<td>23.48</td>
</tr>
<tr>
<td>8</td>
<td>42.05</td>
<td>34.88</td>
<td>23.35</td>
</tr>
<tr>
<td>10</td>
<td>43.10</td>
<td>34.97</td>
<td>23.26</td>
</tr>
<tr>
<td>12</td>
<td>44.15</td>
<td>34.97</td>
<td>23.26</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Layers Size</th>
<th>BLEU (%)</th>
<th>METEOR (%)</th>
<th>ROUGE-L (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>40.99</td>
<td>35.11</td>
<td>23.48</td>
</tr>
<tr>
<td>4</td>
<td>47.30</td>
<td>34.99</td>
<td>23.43</td>
</tr>
<tr>
<td>6</td>
<td>53.60</td>
<td>34.86</td>
<td>23.32</td>
</tr>
<tr>
<td>8</td>
<td>59.91</td>
<td>35.08</td>
<td>23.58</td>
</tr>
<tr>
<td>10</td>
<td>66.21</td>
<td>35.16</td>
<td>23.41</td>
</tr>
<tr>
<td>12</td>
<td>72.52</td>
<td>34.94</td>
<td>23.21</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Layers Size</th>
<th>BLEU (%)</th>
<th>METEOR (%)</th>
<th>ROUGE-L (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>19.97</td>
<td>34.16</td>
<td>22.92</td>
</tr>
<tr>
<td>4</td>
<td>30.48</td>
<td>34.75</td>
<td>23.32</td>
</tr>
<tr>
<td>6</td>
<td>40.99</td>
<td>35.11</td>
<td>23.48</td>
</tr>
<tr>
<td>8</td>
<td>51.51</td>
<td>34.90</td>
<td>23.43</td>
</tr>
<tr>
<td>10</td>
<td>62.02</td>
<td>35.08</td>
<td>23.49</td>
</tr>
<tr>
<td>12</td>
<td>72.53</td>
<td>35.19</td>
<td>23.59</td>
</tr>
</tbody>
</table>

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 CODE-
<table>
<thead>
<tr>
<th>Emb. Model</th>
<th>Java</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S-BLEU (%)</strong></td>
<td><strong>METEOR (%)</strong></td>
<td><strong>ROUGE-L (%)</strong></td>
</tr>
<tr>
<td>128</td>
<td>3.55</td>
<td>22.31</td>
</tr>
<tr>
<td>256</td>
<td>4.24</td>
<td>28.62</td>
</tr>
<tr>
<td>384</td>
<td>48.16</td>
<td>31.56</td>
</tr>
<tr>
<td>512</td>
<td>49.19</td>
<td>32.27</td>
</tr>
<tr>
<td>640</td>
<td>49.17</td>
<td>32.29</td>
</tr>
<tr>
<td>768</td>
<td>49.20</td>
<td><strong>32.32</strong></td>
</tr>
<tr>
<td>896</td>
<td>49.19</td>
<td>32.26</td>
</tr>
<tr>
<td>1024</td>
<td><strong>49.32</strong></td>
<td>32.29</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Code</th>
<th>Summary</th>
</tr>
</thead>
</table>
| public void addMessage(String message) {
  messages.addLast(message);
  if (messages.size() > MAX_HISTORY) {
    messages.removeFirst();
  }
  pointer=messages.size();
} | 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 |
| | CODESCRIBE: add a message to the history |

<table>
<thead>
<tr>
<th>Code</th>
<th>Summary</th>
</tr>
</thead>
</table>
| public void hspan(double start, double end, Paint color, String legend) {
  if (start >= end) {
    return null;
  }
  final Matcher matcher=PATTERN_HASHTAGS.matcher(text);
  while (matcher.find()) {
    final int start=matcher.start(1);
    text.setSpan(new ForegroundColorSpan(mHighlightColor), start, end, Spanned.SPAN_EXCLUSIVE_EXCLUSIVE);
  }
  return text;
} | 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 |
| | CODESCRIBE: draw a vertical span into the graph and optionally add a legend |

<table>
<thead>
<tr>
<th>Code</th>
<th>Summary</th>
</tr>
</thead>
</table>
| 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);
} | 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 |
| | CODESCRIBE: create a new status panel object |

<table>
<thead>
<tr>
<th>Code</th>
<th>Summary</th>
</tr>
</thead>
</table>
| 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();
    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;
} | 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 |
| | CODESCRIBE: highlight all the hash line in the pass text |

<table>
<thead>
<tr>
<th>Code</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void CODESCRIBE(JavaModel model, PythonModel model) {</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Table 12: Qualitative examples on the Java dataset.</td>
</tr>
</tbody>
</table>
def image_create(client, values, v1_mode=False):
    return client.image_create(values=values, v1_mode=v1_mode)

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

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

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()