Received: 23 February 2023 | Revised: 5 October 2023 | Accepted: 11 November 2023

DOI: 10.1002/spe.3291

RESEARCH ARTICLE

WILEY

# PYSCRIBE–Learning to describe python code

**Juncai Guo**[1] | **Jin Liu**[1,2] | **Xiao Liu**[3] | **Yao Wan**[4] | **Yanjie Zhao**[5] | **Li Li**[6] | **Kui Liu**[7] | **Jacques Klein**[8] | **Tegawendé F. Bissyandé**[8]

[1]School of Computer Science, Wuhan University, Wuhan, China

[2]Key Laboratory of Network Assessment Technology, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China

[3]School of Information Technology, Deakin University, Burwood, Melbourne, Australia

[4]School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China

[5]Faculty of Information Technology, Monash University, Clayton, Victoria, Australia

[6]School of Software, Beihang University, Beijing, China

[7]Huawei Software Engineering Application Technology LabHangzhou, China

[8]SnT Centre, University of Luxembourg, Esch-sur-Alzette, Luxembourg

**Correspondence**
Jin Liu, School of Computer Science, Wuhan University, Wuhan, China.
Email: jinliu@whu.edu.cn
Xiao Liu, School of Information Technology, Deakin University, Melbourne, Australia.
Email: xiao.liu@deakin.edu.au

**Funding information**
China Scholarship Council; National Natural Science Foundation of China, Grant/Award Number: 61972290; Open Fund of Key Laboratory of Network Assessment Technology from Chinese Academy of Sciences, Grant/Award Number: 201906270158

**Abstract**

Code comment generation, which attempts to summarize the functionality of source code in textual descriptions, plays an important role in automatic software development research. Currently, several structural neural networks have been exploited to preserve the syntax structure of source code based on abstract syntax trees (ASTs). However, they can not well capture both the long-distance and local relations between nodes while retaining the overall structural information of AST. To mitigate this problem, we present a prototype tool titled PYSCRIBE, which extends the Transformer model to a new encoder-decoder-based framework. Particularly, the triplet position is designed and integrated into the node-level and edge-level structural features of AST for producing Python code comments automatically. This paper, to the best of our knowledge, makes the first effort to model the edges of AST as an explicit component for improved code representation. By specifying triplet positions for each node and edge, the overall structural information can be well preserved in the learning process. Moreover, the captured node and edge features go through a two-stage decoding process to yield higher qualified comments. To evaluate the effectiveness of PYSCRIBE, we resort to a large dataset of code-comment pairs by mining Jupyter Notebooks from GitHub, for which we have made it publicly available to support further studies. The experimental results reveal that PYSCRIBE is indeed effective, outperforming the state-ofthe-art by achieving an average BLEU score (i.e., av-BLEU) of ≈0.28.

**KEYWORDS**

code comprehension, code documentation, code embedding, code summarization, deep learning, representation learning

**Abbreviations:** AST, Abstract syntax tree; API, Application programming interface; BLEU, Bilingual evaluation understudy; CNN, Convolutional neural network; FFN, Feed-forward network; GNN, Graph neural network; GRU, Gated recurrent unit; GCN, Graph convolutional network; LSTM, Long short-term memo; METEOR, Metric for evaluation of translation with explicit ordering; NTLK, Natural language toolkit; NLP, Natural language processing; ROUGE, Recall-oriented understudy for gisting evaluation; RNN, Recurrent neural network; SBT, Structure-based traversal.

This is an open access article under the terms of the Creative Commons Attribution-NonCommercial License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited and is not used for commercial purposes.
© 2023 The Authors. *Software: Practice and Experience* published by John Wiley & Sons Ltd.

## 1 INTRODUCTION

Program comprehension is a challenging, and yet critical, activity of software development. It is generally facilitated by the availability of documentation. Unfortunately, code comments are often either missing or outdated in practice. The literature has thus invested effort into various research directions for mitigating the problem. Some researchers have investigated self documenting code* as a way to ensure that program comprehension can still be achieved.[1] Nevertheless, the commonly accepted perception within the practice community is that code comments are important,[2] justifying the development of techniques to generate such comments. The literature refers to this endeavour in various terms: code summarization,[3] comment generation,[4] description generation.[5] In this article, the terms *description* and *comment* will be used interchangeably since they both refer to *natural-language text* that describes the functionality of source code.

Early techniques of code comment generation are built on formal specifications to infer pseudo-code,[6] which is not applicable for legacy code. Then some efforts were made to automatically detect topic-relevant words and phrases from source code as descriptions.[7,8] Sirdhara et al.[4] then proposed an approach that summarizes each statement in a method into sentences that must be composed. This approach heavily depends on the developer's use of meaningful naming. Follow-up work considered manually designed templates,[3,5] or leveraged information retrieval methods to find similar code snippets with descriptions from which to extract the needed comments.[9] Extending the latter idea, Wong et al.[10,11] utilized the code clone detection technique to identify similar code fragments with the relevant comments. All these approaches mostly focus on the lexical and syntactic details, however, ignoring deep semantics in the code.

Recent advances in code representation learning and natural language processing provide an opportunity to deeply learn hidden semantic relationships between code and comments. Many works have considered source code as natural language sequence and adopted sequence-to-sequence networks such as RNNs and transformer[12] with attention mechanism for comment generation.[13–15] Nevertheless, the structural features of source code, which are crucial for code understanding, are ignored in these studies. To mitigate the issue, there are more and more approaches exploiting the structural information in the abstract syntax trees (ASTs) of source code. Some of these approaches convert the AST to node sequences and use RNNs to model the sequences.[16–18] The other works introduce graph neural networks (GNNs)[19,20] or tree-based RNNs[21,22] to learn on the AST directly.

Despite much progress having been made on automatic code comment generation, there still exists much space for further improvement. From our investigation, current approaches that model AST node sequences based on RNNs can not capture the structural information well. First, it damages the local relations between connected nodes and loses positional information converting ASTs to node sequences. Moreover, RNNs are incapable of well handling the long-term dependency issue when representing long sequences. Although GNN models are designed to capture the relations among connected nodes, they are sensitive to local features and limited to extracting the long-distance relations among AST nodes. Additionally, GNNs are also insensitive to the positions of neighboring nodes in AST, which are important to indicate the functionality of source code. Figure 1 shows an example to better illustrate the importance of positions in ASTs. Given two code snippets "`b=g/m`" and "`b=m/g`", Figure 1 shows their corresponding ASTs. It is clear to see that these two ASTs are very similar except on the positions of variables "`g`" and "`m`". We argue that current GNNs can not well capture the subtle positional information of those two variables.

Based on the aforementioned limitations, this paper first explains the AST as its nodes and edges with their triplet positions. In AST, an edge indicates the local relations between connected nodes, which conveys semantic features at a higher level for AST compared to the node. A triplet position for each node comprises the depth, sequential position of its parent node in the layer, and sequential position among its sibling nodes, which can precisely locate a node/edge in the AST. To integrate AST nodes and edges with their positional information, this paper expands the Transformer model[12] and investigates a novel neural architecture for code comment generation. Specifically, a Transformer encoder is used to represent the AST nodes. Then we learn the local relations between connected nodes by explicitly embedding the edges of AST through another Transformer encoder. Inspired by the positional encoding used in sequence modeling,[12,23] we design a triplet position for each AST node to preserve the hierarchical structure of ASTs. Due to the fully-connected structure of Transformer and triplet positions specified for nodes/edges, the overall structural information of AST can be well learned, mitigating the issues of local sensitivity and long-term dependency. To generate the comments with higher quality, our approach applies a two-stage decoding process that contains two multi-head attention modules over the learned node and edge features sequentially.

*Self-documenting code is written following naming conventions that makes it easy for humans to read and understand code as sentences.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

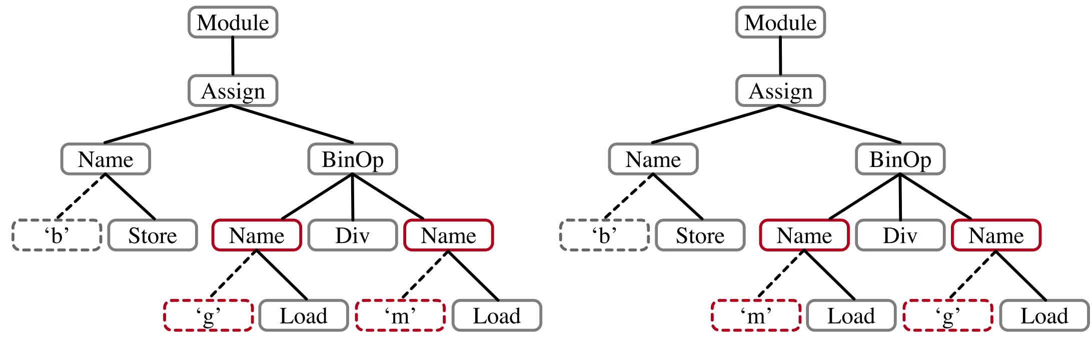


**FIGURE 1** The ASTs of the two Python code snippets "`b=g/m`" (left) and "`b=m/g`" (right). The two ASTs are very similar except on the positions of variables "`g`" and "`m`". The GNN-like models are not able to distinguish between the two code snippets as they can not well capture the subtle positional information of those two variables.

**TABLE 1** Examples of <code, comment> pairs collected from Jupyter Notebooks and from the dataset in Reference 24.

| | Sample in Jupyter Notebooks | Sample in Python dataset[24] |
|---|---|---|
| **Code** | `richpeople.plot(kind='scatter', x = 'age', y='networthusbillion', figsize=(10,10), alpha=0.3)` | `def is_classifier(estimator): return (getattr(estimator, '_estimator_type', None) == 'classifier')` |
| **Comment** | maybe plot their net worth vs age (scatterplot) | returns true if the given estimator is a classifier . |

We instantiate the approach on Python code, which is known as a fast-growing programming language. As data source for evaluation, we explore a large number of Jupyter Notebooks available in open source repositories. Jupyter Notebooks provide a unique channel, via a web-based interface, for capturing the diversity of information in the computation process, including the developed code, the documentation and the results. In such an interactive development environment, users have great freedom to edit their code and descriptions, which is very different from that in the formal software development. Hence, the data in Jupyter Notebooks are much more diverse compared with the existing datasets[13,24,25] consisting of <method,comment> samples obtained from real-world software projects. On the one hand, the code snippets are not limited to methods with clear functional purposes. Instead, the code may be just a line to plot a figure, for example in Table 1. On the other hand, the textual descriptions are written more flexibly (e.g., the word "*maybe*" in the comment in Table 1). As a result, our dataset based on Jupyter Notebooks can provide a new space for code comment generation task.

*Note that*, this research aims to develop a novel encoder-decoder neural architecture that employs the triplet positional AST nodes and edges to represent the comprehensive structure of source code, thereby boosting the generation of code comments. Thus, our work is orthogonal to large pre-trained language models, such as Codex,[26] CodeT5,[27] and the most recent ChatGPT,[28] which depend on massive corpora and model source code only as textual sequence. Despite that, the pre-trained paradigm can further improve our proposed PyScribe approach, which is left to future work.

In short, this paper makes the contributions as below:

① We introduce PyScribe, a novel neural architecture for Python code comment generation. The approach expands the power of Transformer to exploit more detailed AST features while maintaining the structural information by encoding both edges and nodes with triplet positions. To our knowledge, it is the first time in this area to (1) explicitly embed the edges for AST learning, and (2) define triplet positions for both the AST nodes and edges to maintain the overall AST structure. In addition, a two-stage decoding process is applied over the learned edge and node features sequentially for yielding the comment texts.

② We build and share a large dataset of Python code+description pairs with high quality to support the community effort in advancing the state-ofthe-art in code comment generation. The dataset includes 47*k* pairs of code-description samples.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

③ We extensively assess the performance of PyScribe, and conduct an ablation study to highlight the effectiveness of the major components and design choices of our model. We also compare PyScribe against the traditional sequence-to-sequence models, and more recent approaches that propose different code abstraction models or that leverage the Transformer. Quantitative results indicate that PyScribe can generate better code comments than the state-ofthe-art baselines.

## 2 | RELATED WORK

Automated code-to-text translation has been a topic of intense interest in the scientific community for many years.[16,29] It has been applied to resolve software engineering tasks, for example, automated code comment generation and code annotation generation. With the rapid advancement of machine learning and deep learning in recent years, most state-ofthe-art works, nowadays achieve code-to-text translations through learning-based approaches.[30] In these approaches, code snippets are either regarded as natural language sequences or structural trees (i.e., ASTs of the code).

**Code as natural language.** Numerous essential solutions of neural source code summarization approaches are proposed to transform the problem into a sequence generation task.[13,14,29] For example, Iyer et al.[29] proposed a sequence-to-sequence neural network based on LSTMs and attention mechanism, namely GODE-NN, to produce code descriptions. Wei et al.[14] built a dual learning framework to jointly model the tasks of code generation and code summarization. More recently, Ahmad et al.[15] presented a method called NeuralCodeSum for automatically annotating code snippets with natural language. While taking source code as natural language, their approach attempts to further learn information from the code by maintaining the pairwise relationships between source code tokens. Nevertheless, since the initial architecture of the Transformer is proposed for handling natural language rather than programming language, the Transformer-based approach can only achieve limited performance for code comment generation, as shown in the evaluation section.

**Code as AST.** By considering code as natural language, the structural information of source code is ignored. Consequently, the power of the corresponding code-to-text models may be affected. To this end, more and more researchers have started to train the learning models with code's structural information, which can be obtained from the code's AST[16,17,19–22,25,31–35] and the Application Programming Interface (API) sequence.[13,36,37] For example, Wan et al.[21] put forward a framework based on reinforcement learning to automatically summarize code snippets. Their framework respectively encodes the AST structure as well as the plain code sequence via AST-based LSTM and LSTM, which are integrated together by a hybrid attention layer. Hu et al.[16,17] converted the AST to linear node sequence through structure-based traversal (SBT) for code comment generation. Zhou et al.[33] performed CNN[38] with N-Ary Tree-LSTM[39] for better AST representation, and proposed a switch network to dynamically decode the syntactical and lexical representations of source code for comment generation. LeClair et al.[19] utilized a single GRU layer[40] to encode the source code sequence and graph convolutional network (GCNs)[41] to model the AST of Java method. Choi et al.[34] performed GCNs[41] before transformer framework to learn AST representation for code comment generation. In addition to AST structural information, Wang et al.[42] recently proposed another approach that utilizes a hierarchical attention model by consolidating different features, including not only AST but also code sequence and control flow.

**Our Approach.** Unfortunately, despite the fact that AST structures have been recurrently leveraged to learn programming code, their capabilities could have been under-estimated. For instance, converting AST to node sequence may lead to loss of structural information. GNN-based methods are oversensitive to local features and are limited to capture the long-distance relations in the tree structure. Indeed, state-ofthe-art works either ignore the edge information or consider the edges only as the local relations hidden between nodes, but not explicit components as nodes that can be embedded. Therefore, we propose a novel approach titled PyScribe in this work, which attempts to learn the AST representation from the node-level and edge-level perspectives separately while retaining the overall structural information by incorporating the triplet positions.

## 3 | APPROACH

We now overview the details of our approach, namely PyScribe, for generating Python code descriptions based on learning with neural networks. The overview of its phases is presented in Figure 2: data is first (1) pre-processed to prepare the required input for (2) model training. Given a code snippet, PyScribe first parses it to produce its AST nodes and edges

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

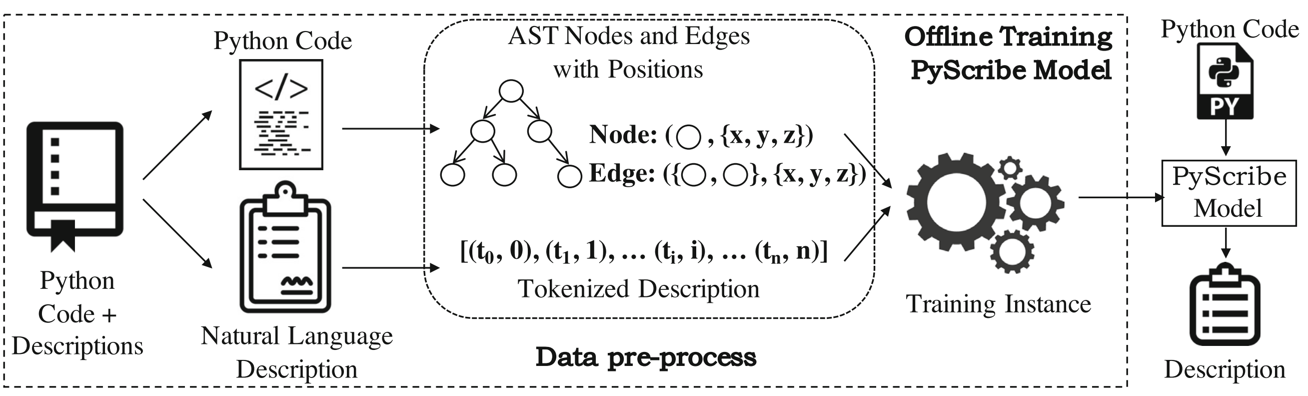


**FIGURE 2** Overview of PyScribe code-to-description generation process.

with their triplet positions in their AST context. The natural language description of Python code is also tokenized for the training purpose.† In the training process, the AST nodes, edges and description tokens with positions flow as input into the proposed PyScribe model. After multiple epochs of offline training on the whole training dataset, the model will be further used to generate the text description corresponding to the given new Python code. We now detail both phases.

## 3.1 | Data preprocessing

### 3.1.1 | Code to AST with triplet positions

Program code is written with tokens forming textual representations that can be processed as natural language text. Structural details of code however carry semantic information that is not expressed in textual representations. In practice, code analysis is performed by building its AST. Recent studies in the literature have reached promising results by using AST context as input to code-related learning models.[17,18,39,44–47] This work also relies on AST information, however, goes further. On the one hand, we consider the edges as an explicit component of AST that can be embedded (i.e., not just hidden relations between connected nodes). On the other hand, we present triplet positions for the nodes/edges to maintain the overall structural information of AST.

As a basic component of AST, the nodes in this work are divided into two categories: (1) function nodes that are crucial to AST structure and function realization, such as `Assign` and `Module` in Figure 1, and (2) attribute nodes that denote the values or names of their parent nodes, which also exist in the source code and are visualized as dotted boxes in Figure 1, e.g., “`a`” and “`g`”. Formally, we define a triplet positional AST node $n$ in this study as below:

$$n = (n_t, \{x, y, z\}), \tag{1}$$

where $n_t$ is the label associated to an AST node. $\{x, y, z\}$ represents the triplet position of node $n$, which consists of: (1) the node's depth $x$ in the AST, (2) the left-to-right sequential position $y$ of its parent node in the layer, and (3) the left-to-right sequential position $z$ among its sibling nodes. Both $x$ and $y$ start with 0. To differentiate attribute node from function node, the position value $z$ of an attribute node is a negative integer starting from −1. In contrast, a function node's position $z$ is a non-negative integer beginning with 0.

We take for example the left side of Figure 1 that depicts the AST of code fragment `b=g/m`. By traversing the AST layer by layer and left to right, the nodes with triplet positions are listed in the left side of Figure 3. For instance, the triplet position of function node `(BinOp,{2,0,1})` includes: the depth position `2` that represents the third level from top to bottom, the sequential position 0 indicating that the parent node `Assign` is the first left-to-right function node in `Assign`'s layer, and the position 1 which means `BinOp` is the second among its siblings. For node `("g",{4,1,-1})` as another example, the third position `-1` indicates that it is an attribute node and is the first node among its siblings. Since

† In this paper, we utilize the ast package in Python for converting code to AST and NTLK package[43] for tokenization.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

| **AST nodes:** | **AST edges:** |
|---|---|
| ( Module, {0, 0, 0} ), | ( {Module, Assign}, {1, 0, 0} ), |
| ( Assign, {1, 0, 0} ), | ( {Assign, Name}, {2, 0, 0} ), |
| ( Name, {2, 0, 0} ), | ( {Assign, BinOp}, {2, 0, 1} ), |
| ( BinOp, {2, 0, 1} ), | ( {Name, 'b'}, {3, 0, -1} ), |
| ( 'b', {3, 0, -1} ), | ( {Name, Store}, {3, 0, 0} ), |
| ( Store, {3, 0, 0} ), | ( {BinOp, Name}, {3, 1, 0} ), |
| ( Name, {3, 1, 0} ), | ( {BinOp, Div}, {3, 1, 1} ), |
| ( Div, {3, 1, 1} ), | ( {BinOp, Name}, {3, 1, 2} ), |
| ( Name, {3, 1, 2} ), | ( {Name, 'g'}, {4, 1, -1} ), |
| ( 'g', {4, 1, -1} ), | ( {Name, Load}, {4, 1, 0} ), |
| ( Load, {4, 1, 0} ), | ( {Name, 'm'}, {4, 3, -1} ), |
| ( 'm', {4, 3, -1} ), | ( {Name, Load}, {4, 3, 0} ) |
| ( Load, {4, 3, 0} ) | |

FIGURE 3 AST nodes and edges extracted from the left AST in Figure 1.

the root node `Module` has no parent node, we set its triplet position to `{0,0,0}` in particular. In an AST, all the triplet positions are specified uniquely and precisely, which allows for the tracking and differentiation of nodes with identical labels. (e.g., `(Name,{3,1,0})` and `(Name,{3,1,2})`).

Since an edge reflects the relationship between a node and its child node that is essential to AST structure, we consider the edges as another component for AST that can be embedded and encoded directly. Thus, we combine a node and its child node as an explicit edge and record the triplet position of its child node as the position of edge. Such an edge with a triplet position can be formulated as below:

$$e = (\{n_t, n_t^c\}, \{x, y, z\}), \tag{2}$$

where $n_t$ is the parent node of the node $n_t^c$, and $\{x, y, z\}$ represents the triplet position of $n_t^c$ that has been mentioned above.

For example, according to the left AST of Figure 1, all edges with triplet positions shown in Figure 3 can be extracted. The edge `({BinOp,Name},{3,1,0})` starts from the node `(BinOp,{2,0,1})` and ends with its first child node `(Name,{3,1,0})`. `3` denotes the depth position of the edge, of which the child node is in the fourth layer from top to bottom in the AST. Its sequential position 1 indicates that the edge is the second that starts with function node in this layer from the left to right. And 0 illustrates that the edge is for the first child function node `Name` of `BinOp`. Its triplet position can thus be used to discriminate it from its sibling edge `({BinOp,Name},{3,1,2})`.

It should be noted that such a three-tuple index can mark the position of a node/edge uniquely in a given AST. It indicates that the nodes/edges with triplet positions can represent the whole AST structure.

### 3.1.2 | Comment to tokens

Code description/comment is a document provided in natural language. We thus build on text pre-processing strategies widely used in the NLP community and employ NTLK package[43] to tokenize Python code description and lemmatize each word in description. Furthermore, some special texts such as URLs (e.g., "http://www.timeout.com/") and specific delimiters (e.g., "******") are replaced with pre-defined tokens (respectively "*<url>*" and "–"). Besides the token sequences, the index of each token in the related sequence (as shown in Figure 2) is also forwarded as input to the PyScribe neural networks.

Like the most natural language processing tasks, code comment generation approaches based on neural networks are generally challenged by the out-of-vocabulary issue as well. Considering the vocabulary size and that rare words in the training data offer little possibility to learn their embeddings, any token that occurs less than three times in the training dataset will be regarded as an out-of-vocabulary token. To deal with these out-of-vocabulary tokens, we marked them as a special symbol "*<unk>*".

## 3.2 | PyScribe model

Figure 4 depicts the framework of our proposed PyScribe model. It is made up of three modules: two encoders and one decoder that are connected with attention neural networks, inspired by the Transformer[12] language-to-language

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

FIGURE 4 The PYSCRIBE model. R&N represents residual connection and normalization.

translation architecture. The encoders and decoder implement multi-head attention and position-wise feed-forward networks with a stack of N identical layers. Each layer is fed with embedded AST nodes, edges, and comment tokens. The two encoders are dedicated to extracting the AST features of source code, layer by layer. In the decoding process, the comment features are first captured by the masked multi-head attention neural networks, and further fused with the extracted AST node and edge features continuously. Finally, through a stack of decoder blocks and the output layer, the model can predict the probabilities of descriptive text tokens for the given Python code. The details of this process are further laid out in the following sub-sections, with the notations listed in Table 2.

### 3.2.1 | Embeddings

When passing the AST nodes, edges, and comment tokens with their positions to PYSCRIBE, it is necessary to embed them into numerical vectors[40,46,48] in the first place. For example, given the AST with $l_n$ nodes $\mathbf{S}_n = [n_1, n_2, \dots, n_{l_n}]$ with the triplet positions $\mathbf{P}_n = (\{x_1, y_1, z_1\}, \{x_2, y_2, z_2\}, \dots, \{x_{l_n}, y_{l_n}, z_{l_n}\})$, the nodes will be mapped into $\mathbf{E}_n^s = [\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_{l_n}]$

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 2 Summary of notations.

| | |
|---|---|
| $l_n$ | Number of AST nodes |
| $l_e$ | Number of AST edges |
| $l_t$ | Number of summary tokens |
| $\mathbf{S}_n$ | Set of AST nodes |
| $d$ | Embedding size |
| $\mathbf{E}_n^s$ | Embeddings of AST nodes |
| $\mathbf{E}_e^s$ | Embeddings of AST edges |
| $\mathbf{E}_t^s$ | Embeddings of summary tokens |
| $\mathbf{E}_n^p$ | Triplet positional embeddings of AST nodes |
| $\mathbf{E}_e^p$ | Triplet positional embeddings of AST edges |
| $\mathbf{E}_t^p$ | Sequential positional embeddings of summary tokens |
| $\mathbf{E}_n$ | Initialized representation of AST nodes |
| $\mathbf{E}_e$ | Initialized representation of AST edges |
| $\mathbf{E}_t$ | Initialized representation of summary tokens |
| $N$ | Number of encoding/decoding layers |
| $\mathbf{E}^k$ | Vectors output by the $k$th Transformer encoding layer |
| $\mathbf{H}^{k+1}$ | Self-attention vectors in the $(k+1)$th Transformer encoding layer |
| $h$ | Number of attention heads |
| $\mathbf{Q}$ | Query input of multi-head attention operation |
| $\mathbf{K}$ | Key input of multi-head attention operation |
| $\mathbf{V}$ | Value input of multi-head attention operation |
| $\mathbf{A}_i$ | Scores of the $i$th attention head |
| $\mathbf{E}_n^N$ | Encoded AST nodes |
| $\mathbf{E}_e^N$ | Encoded AST edges |
| $\mathbf{E}_t^k$ | Vectors output by the $k$th summary decoding layer |
| $\mathbf{H}_t^{k+1}$ | Self-attention vectors in the $(k+1)$th summary decoding layer |
| $\mathbf{H}_{t,n}^{k+1}$ | Decoded vectors in the $(k+1)$th summary decoding layer performed over the encoded AST nodes |
| $\mathbf{H}_{t,e}^{k+1}$ | Decoded vectors in the $(k+1)$th summary decoding layer performed over the encoded AST edges |
| $\mathbf{E}_t^N$ | Decoded summary tokens |
| $\mathbf{P}$ | Probability distribution for predicted summary tokens |

where $\mathbf{e}_i \in \mathbb{R}^d$ and d is the embedding size. Inspired by the learned positional embeddings for sequence,[12,23] we consider each triplet position $\{x_i, y_i, z_i\}$ as an individual token that can be embedded as $\mathbf{e}'_i \in \mathbb{R}^d$. Consequently, the triplet positions $\mathbf{P}_n$ can be mapped into $\mathbf{E}_n^p = [\mathbf{e}'_1, \mathbf{e}'_2, \dots, \mathbf{e}'_{l_n}]$. Particularly, the embedding operation of edges is similar to that of triplet positions by treating each edge consisting of node pair as an individual.

After getting the embedding vectors of AST nodes, edges, comment tokens (i.e., $\mathbf{E}_n^s \in \mathbb{R}^{l_n \times d}$, $\mathbf{E}_e^s \in \mathbb{R}^{l_e \times d}$, $\mathbf{E}_t^s \in \mathbb{R}^{l_t \times d}$), and their related positional embeddings (i.e., $\mathbf{E}_n^p \in \mathbb{R}^{l_n \times d}$, $\mathbf{E}_e^p \in \mathbb{R}^{l_e \times d}$, $\mathbf{E}_t^p \in \mathbb{R}^{l_t \times d}$), we follow Vaswani et al.[12] to initialize the representations of nodes, edges, description tokens as below:

$$
\begin{aligned}
\mathbf{E}_n &= \mathbf{E}_n^s * \sqrt{d} + \mathbf{E}_n^p, \\
\mathbf{E}_e &= \mathbf{E}_e^s * \sqrt{d} + \mathbf{E}_e^p, \\
\mathbf{E}_t &= \mathbf{E}_t^s * \sqrt{d} + \mathbf{E}_t^p,
\end{aligned} \tag{3}
$$

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

where $d$ is the embedding size; $l_n$, $l_e$, and $l_t$ are the numbers of AST nodes, edges, and description tokens. In this way, the triplet positional information can be integrated into node/edge embeddings to maintain the AST structure.

After that, the vector representations of AST nodes, edges, and textual descriptions flow into the encoders and decoder of the PYSCRIBE model for further processing.

### 3.2.2 | Encoders

To encode the two-level AST information, i.e., nodes and edges, with their triplet positions, PYSCRIBE expands the Transformer encoder to two independent encoders with the same architecture. They differ, however in the targeted inputs: for one, it is the embedded representations of AST nodes, and for the other, it is the embedded AST edges. As depicted in Figure 4, the encoder comprises $N$ identical Transformer encoding layers. Each layer is made up of a multi-head attention mechanism as well as a feed-forward network. Besides, residual connection[49] and layer normalization[50] are employed with the two modules to alleviate the problems of vanishing gradients in multilayer computing and excessive vector offset in residual connecting.

Multi-head attention is a self-attention technique that accepts query, key, and value vectors as input. Since it mines the semantic relevance among all the tokens, there exist no local sensitivity or long-term dependency issues. The multi-head attention mechanism is formulated as below:

$$
\begin{aligned}
\mathbf{H}^{k+1} &= LayerNorm(\mathbf{E}^k + Att(\mathbf{E}^k, \mathbf{E}^k, \mathbf{E}^k)), \\
Att(\mathbf{Q}, \mathbf{K}, \mathbf{V}) &= [\mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_h]\mathbf{W}^O, \\
\mathbf{A}_i &= Softmax\left(\frac{\mathbf{Q}_i \mathbf{K}_i^T}{\sqrt{d}}\right)\mathbf{V}_i, \\
\mathbf{Q}_i = \mathbf{Q}\mathbf{W}_i^Q, &\quad \mathbf{K}_i = \mathbf{K}\mathbf{W}_i^K, \quad \mathbf{V}_i = \mathbf{V}\mathbf{W}_i^V,
\end{aligned} \tag{4}
$$

where $\mathbf{E}^k \in \mathbb{R}^{l \times d}$ denotes the vectors output by last layer. *Att* means the multi-head attention operation that needs query, key, and value as input. The vector $\mathbf{E}^k$ and *Att*'s output are residually connected as the input of the layer normalization *LayerNorm*. In *Att*, $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{l \times d}$ represent the query, key, and value vectors, respectively. $h$ is the number of attention heads. $\mathbf{W}_i^Q, \mathbf{W}_i^K, \mathbf{W}_i^V \in \mathbb{R}^{d \times (d/h)}$, and $\mathbf{W}^O \in \mathbb{R}^{d \times d}$ are learnable parameters.

In order to effectively extract the semantics hidden in the input nodes/edges, the feed-forward network attached with residual connection and layer normalization is then applied to each position separately to achieve non-linear transformation, which is defined as follows:

$$
\begin{aligned}
\mathbf{E}^{k+1} &= LayerNorm(\mathbf{H}^{k+1} + FFN(\mathbf{H}^{k+1})), \\
FFN(\mathbf{v}) &= Linear(ReLU(Linear(\mathbf{v}))),
\end{aligned} \tag{5}
$$

where *FFN* denotes the process implemented by the feed-forward network, which is composed of two *Linear* transformations separated by a *ReLU* activation as non-linear transformation. Note that the output of *FFN* has the same shape as its input.

### 3.2.3 | Decoder

The decoder of PYSCRIBE is also composed of $N$ (same as that of the encoders) decoding blocks. Different from the encoders, each block in the decoder comprises four sub-layers: one masked multi-head attention sub-layer for self-attention encoding, two multi-head attention sub-layers for two-stage decoding, and one *Linear* sub-layer, which are all followed by residual connection and layer normalization.

In the decoding block, the existing tokens of textual description are first encoded based on the masked multi-head attention mechanism, which can be formalized as:

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

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

where $\mathbf{E}_t^k \in \mathbb{R}^{l \times d}$ is the vectors output by last decoding layer. *MaskAtt* denotes the masked multi-head attention[12] utilized in the decoder. It also takes query, key, and value vectors as input and attempts to extract the relations among the description tokens. The only difference between *MaskAtt* and *Att* is the calculation of $\mathbf{A}_i$, which is as follows:

$$\mathbf{A}_i = Softmax\left(Mask\left(\frac{\mathbf{Q}_i \mathbf{K}_i^T}{\sqrt{d}}\right)\right)\mathbf{V}_i. \tag{7}$$

The mask operation is applied to achieve fast output generation (e.g., in parallel) while avoiding recurrent generation of output tokens in the training process (e.g., like RNNs). In detail, *Mask* is responsible for setting up the matrix (e.g., by assigning $-\infty$ to all the cells above the principal diagonal), which can prevent the information in rightward positions from attending to the current position.

After that, two multi-head attention modules are applied to continuously fuse token features of descriptions with the extracted features of AST nodes and edges for proceeding with the decoding. Then, the *FFN* sublayer takes the decoded token vectors for non-linear transformation. The whole procedure can be formulized as follows:

$$\begin{aligned}
\mathbf{H}_{t,n}^{k+1} &= LayerNorm(\mathbf{H}_t^{k+1} + Att(\mathbf{H}_t^{k+1}, \mathbf{E}_n^N, \mathbf{E}_n^N)), \\
\mathbf{H}_{t,e}^{k+1} &= LayerNorm(\mathbf{H}_{t,n}^{k+1} + Att(\mathbf{H}_{t,n}^{k+1}, \mathbf{E}_e^N, \mathbf{E}_e^N)), \\
\mathbf{E}_t^{k+1} &= LayerNorm(\mathbf{H}_{t,e}^{k+1} + FFN(\mathbf{H}_{t,e}^{k+1}))\ .
\end{aligned} \tag{8}$$

The *Att* and *FFN* sub-layers in this formula have the same architectures as that of the encoders, respectively. The differences between them lie in the input data.

After the stacked two-stage decoding processes, an *Output Layer* is attached to the decoder to generate the final probabilities of the next predicted token for a given Python code snippet. The probability **P** is calculated with the following formula:

$$\mathbf{P} = Softmax(Linear(\mathbf{E}_t^N)). \tag{9}$$

The linear function *Linear* projects the decoder output to the vectors of which the dimensionality is the same with the token vocabulary size of description corpus, and the softmax function *Softmax* coverts the vectors into the probabilities. In this paper, the token with the highest probability will be chosen as the next token in the description.

# 4 | EXPERIMENTAL SETUP

The performance of PyScribe is assessed via a variety of experiments. Before presenting the results, this section provides the research questions, data collection process, and experimental configurations. Our aim is to facilitate experimental replication.

## 4.1 | Research questions

We design the experiments to answer the following research questions:

- **RQ-1.** How effective is PyScribe in automatically generating descriptions for Python code?
- **RQ-2.** How sensitive is the performance of PyScribe w.r.t. the size of training datasets, the size of code and the length of descriptions?
- **RQ-3.** How does the model size influence the performance of PyScribe?
- **RQ-4.** How does our method perform compared with other advanced code summarization techniques?
- **RQ-5.** How scalable is PyScribe on other datasets?

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

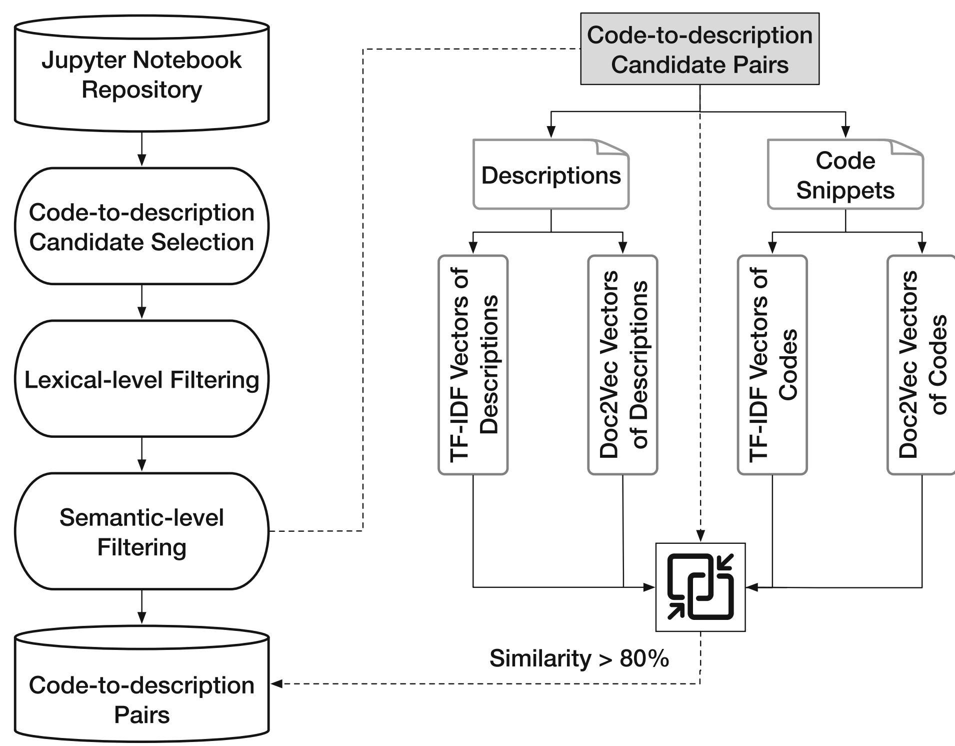


**FIGURE 5** The working steps of our data collection process.

## 4.2 | Data collection

Large-scale and high-quality datasets are necessary to evaluate neural network-based approaches. Since the literature does not propose relevant datasets associated with Python code fragments and their description texts based on Jupyter Notebooks, we propose to build such a dataset by ourselves through mining from GitHub. Jupyter Notebooks have indeed gained momentum in the Python developer community. They are leveraged as a single channel (in the form of a document) to share code, along with its explanatory text, multimedia resources, and even computation outputs. Figure 5 illustrates the steps that we followed in our data collection process, namely (1) to select candidate code/description pairs (2) filter relevant pairs based on lexical information, and (3) filter based on semantic details. As experimentally demonstrated by Sun et al.,[51] it is essential to filter out irrelevant samples from the dataset when performing neural code learning.

**Step 1: Code-to-description candidates selection.** A simplified view of an example Jupyter Notebook is provided in Figure 6. Jupyter Notebooks enable interactive programming by first describing the functionality of the code (in a `Markdown` cell) before writing the source code while reading the Notebook. The execution outputs (not visible in the illustrated example) will then be presented right after the code snippets. As the first step, given a Jupyter Notebook, we simply group two adjacent cells (`Markdown` cell followed by a `Code` cell) together to form a possible code-to-description candidate. Other combinations (such as two continuous code cells) will be ignored. In the example of Figure 6, we will retain only two code-to-description candidates, namely cells 1-2, and cells 3-4.

We crawled GitHub and collected 1,630,818 code-to-description candidates. Unfortunately, not all the code-to-description candidates collected in the first step are suitable (e.g., language is not English, code syntax is incorrect). We add filtering steps and curate the dataset.

**Step 2: Lexical-level Filtering.** In the second step, we specifically target irrelevant or malformed samples. We set rules to reduce noise in the dataset:

- Non-English text descriptions are excluded (by a Python tool named *langid*).
- The code snippet should be parsable (i.e., with a correct syntax): we consider the syntax to be correct if an AST can be successfully generated for the code snippet.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

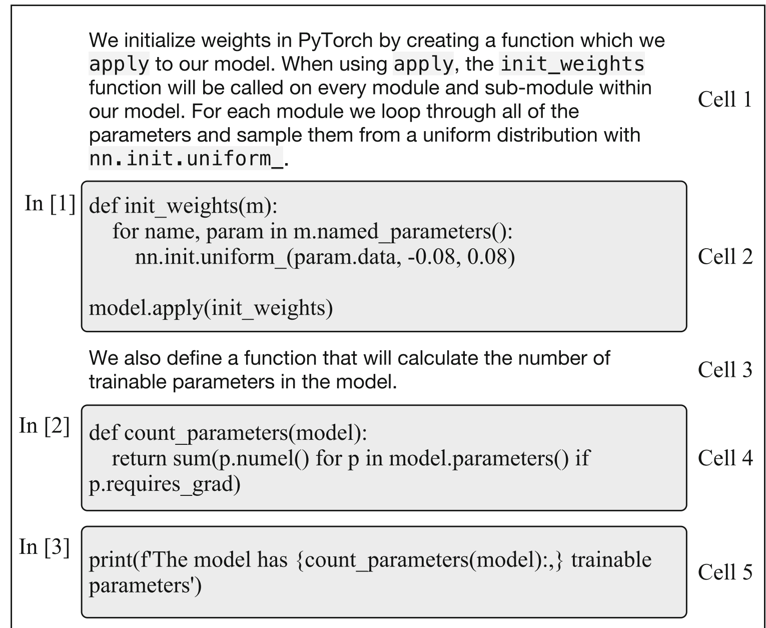


**FIGURE 6** A simplified example of Jupyter notebook.

- Small-size descriptions are discarded: the text description (after tokenization, such as removing stop words and punctuations) should have at least five words.
- Small-size code is discarded: the generated AST of the code snippet should contain at least five nodes.

After applying those rules, our collected dataset is reduced and now contains 956,728 code-to-description pairs.

**Step 3: Semantic-level filtering.** We now go one step further to perform semantic-level filtering since we have observed that some descriptions are not actually related to their associated code snippets. We, therefore, propose to calculate the semantic relevance between the description and the code snippet for each pair, so as to only select such pairs that have their descriptions or code snippets closely correlated.

The rationale behind our semantic relevance calculation method is presented as follows. Given two independent code-to-description pairs, say $(c_1, d_1)$ and $(c_2, d_2)$, if $c_1$ is similar to $c_2$ meanwhile $d_1$ is also similar to $d_2$, we will consider both of these two pairs are semantically relevant and hence will be selected to fulfill our training dataset. In this work, we introduce two techniques to calculate the similarity of descriptions and code snippets: TF-IDF[52] and Doc2Vec.[48] The code-to-description pairs (i.e., descriptions and the code) are first tokenized and converted into lower cases. The Python code snippets are further converted into AST edge sequences. Then, the tokenized descriptions and the edge sequences are respectively leveraged to learn embeddings through both TF-IDF and Doc2Vec models. This process eventually leads to four models: (1) TF-IDF model built on descriptions, (2) Doc2Vec model trained on descriptions, (3) TF-IDF model built on code, and (4) Doc2Vec model trained on code. After that, the four models are respectively leveraged to calculate cosine similarities between descriptions and code snippets (cf. Fig.5). With a pre-defined similarity threshold at 80% (for both descriptions and code), we then perform a quadruple analysis to filter out such code-to-description pairs that have no neighbor pairs fulfilling our aforementioned criteria: two descriptions (or code snippets) have a similarity less than 0.8, concerning either TF-IDF model or Doc2Vec model. In other words, if the cosine similarities of two code-to-description pairs (w.r.t. the aforementioned four models) are all higher than 0.8, the descriptions are considered to be correct for the corresponding code snippets. Consequently, both code-to-description pairs will be regarded as valid ones and hence will be considered for fulfilling the training dataset.

Eventually, after passing the three steps of data selection, 86,788 code-to-description pairs are retained. The distributions of the code AST size and the description length are illustrated in Figure 7. Based on the distribution quartile values, we focus on selecting the code-to-description pairs whose code AST size is less than 220 and whose description text length is less than 70. Finally, **47,689 code-to-description** sample pairs are retained. For the assessment experiments in this paper, we randomly select 47,000 pairs and divide the dataset into *training/validation/testing* sets with *45,000/1,000/1,000* sample pairs, respectively.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

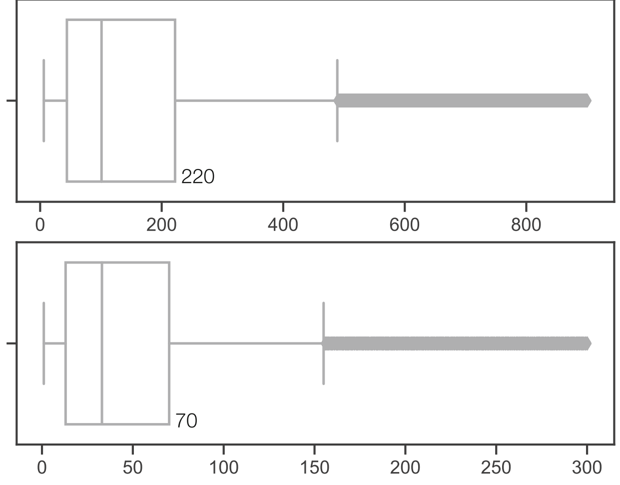


**FIGURE 7** Distributions of the AST size (top) and description length (bottom) per candidate pair. Outliers are excluded for better visualization.

**TABLE 3** Parameter settings.

| Batch size | Learning rate | Decay | Minimum learning rate | Drop | Layers | Hidden dimensions of attention | Hidden dimensions of FFN | Heads | Epochs |
|---|---|---|---|---|---|---|---|---|---|
| 160 | 5 $e^{-4}$ | 0.95 | 2.5$e^{-5}$ | 0.2 | 4 | 512 | 2048 | 8 | 20 |

*Note*: **Decay** and **Drop** indicate the decay of the learning rate and the dropout ratio; **Heads** and **Epochs** are the number of attention heads and training epochs over the whole dataset.

## 4.3 | Experiment configurations

Our implementation of PYSCRIBE leverages PyTorch 1.2.0. We choose Python 3.7 as the development environment, CUDA 10.0 as the parallel computing interface, and Ubuntu 16.04 as the backend operating system, running on a machine with 4 NVIDIA 2080 Ti GPUs. Our experiments adopt Adam[53] as the optimizer. The main hyper-parameters we set for evaluation are also provided in Table 3.

## 4.4 | Evaluation metrics

In this work, we resort to seven metrics (which are actually variants of three main metrics) that have been frequently relied upon in similar studies to evaluate the state-ofthe-art:

- ***BLEU***.[54] Bilingual evaluation understudy (BLEU) is among the earliest metrics proposed for calculating the similarity of two textual sequences. It is the geometric mean of n-gram precision values multiplied by a brevity penalty for short sentences. Currently, BLEU is widely employed for assessing machine translation tasks. In our experiments, we use BLEU-1, BLEU-2, BLEU-3, and BLEU-4 as BLEU scores (measured on 1-gram, 2-gram, 3-gram, and 4-gram) that are commonly used in other related works.[21,42] Additionally, av-BLEU that is smoothly averaged over the four BLEU scores is introduced as the overall BLEU for performance evaluation.
- ***METEOR***.[55] Metric for evaluation of translation with explicit ordering (METEOR) is an important metric utilized in translation-like tasks. It is defined as the harmonic mean of recall and precision of uni-gram matching between texts. METEOR mitigates some issues in BLEU, e.g., the lack of explicit word matching and recall calculation. As a recall-oriented metric, it measures how effectively a neural translation technique grabs the reference content in the generated sequence.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

- ***ROUGE***.[56] Recall-oriented understudy for gisting evaluation (ROUGE) was initially presented for evaluating summarization systems. It does so by comparing the overlaps between word pairs, word sequences, and n-gram data. This study introduces the ROUGE-L variant, which measures the longest co-occurring subsequences between sentences and takes sentence-level structural similarity into consideration naturally.

It is noted that we report all the result scores in percentage.

# 5 EXPERIMENTAL RESULTS

## 5.1 Performance assessment

This section evaluates PyScribe by comparing its performance against the performance that can be achieved with three baselines and six variants. These comparisons aim to assess the impact of the major components and design choices in the PyScribe architecture: The three baselines are implemented by replacing PyScribe's Transformer-based encoder and decoder algorithms with the Gated Recurrent Unit (GRU) algorithm.[40] We select GRU since it is one of the most advanced recurrent neural networks (RNN) and has been demonstrated effective in sequence-to-sequence modeling tasks.

- **Baseline-1.** The two encoders and the decoder of PyScribe are replaced by two two-layer bidirectional GRUs (one for each encoder) and a one-layer unidirectional GRU, respectively.
- **Baseline-2.** Only the decoder of PyScribe is replaced (i.e., by one-layer unidirectional GRUs).
- **Baseline-3.** Only the two encoders of PyScribe are replaced (by two two-layer bidirectional GRUs).

It should be noted that the numbers of the GRU layers are determined according to their performance.

Recall that, when embedding code snippets, our approach takes both AST nodes and edges into consideration. We proposed two variants to assess the contribution of the design choices in an ablation study.

- **Variant-1.** Edge information is not considered in the model training (i.e., -Edge). Everything else remains the same.
- **Variant-2.** Node information is not considered in the model training (i.e., -Node). Everything else remains the same.

In addition, our proposed three-dimension positions are used in the embeddings to provide position information for nodes and edges. Then, two variants are designed to evaluate their effect on our model.

- **Variant-3.** Triplet position information (for both nodes and edges) is not considered in the model training (i.e., -Position). Everything else remains the same.
- **Variant-4.** The nodes and edges are regarded as sequences. Triplet positions for nodes and edges are both replaced by traditional sequential positions (i.e., Sequential Position), like that in code descriptions. Everything else remains the same.

Finally, as shown in Figure 4, PyScribe, by default, considers edge information before node information when performing multi-head attentions in the decoding blocks. We propose to assess whether this ordering may have an impact on the performance of PyScribe. To this end, we implement two more variants.

- **Variant-5.** The two attentions are calculated in parallel and then added together. There will be no order between them.
- **Variant-6.** The order of the two-stage decoding is reversed. PyScribe sequentially takes edge information into consideration before node information.

Table 4 summarizes the experimental results with all metrics for all implementations. All of the experiments are launched at the same dataset (cf. Section 4.2: 4,5000 code-to-description pairs for training set, 1,000 pairs for validation set, and 1,000 pairs for testing set). Overall, PyScribe outperforms the eight baselines and variants on all performance metrics and shows comparable performance with one variant.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 4 Performance of PYSCRIBE and its baselines.

| Model | BLEU-1 | BLEU-2 | BLEU-3 | BLEU-4 | av-BLEU | METEOR | ROUGE-L |
|---|---|---|---|---|---|---|---|
| Baseline-1 (GRU-GRU) | 16.00 | 6.39 | 4.32 | 3.57 | 6.30 | 15.62 | 18.80 |
| Baseline-2 (Trans.-GRU) | 15.17 | 5.31 | 3.47 | 2.83 | 5.30 | 14.42 | 17.34 |
| Baseline-3 (GRU-Trans.) | 29.98 | 17.13 | 13.84 | 12.26 | 17.18 | 14.51 | 27.68 |
| Variant-1 (Edge) | 36.36 | 25.01 | 22.05 | 20.53 | 25.33 | 18.70 | 33.35 |
| Variant-2 (Node) | 37.38 | 25.73 | 22.52 | 20.88 | 25.93 | 18.94 | 33.49 |
| Variant-3 (Position) | 37.85 | 26.67 | 23.65 | 22.10 | 26.95 | 20.13 | 35.48 |
| Variant-4 (Sequential position) | 37.89 | 25.56 | 22.10 | 20.30 | 25.68 | 18.77 | 34.38 |
| Variant-5 (Parallel) | 36.99 | 25.50 | 22.22 | 20.43 | 25.58 | 19.04 | 34.13 |
| Variant-6 (Edge first) | **40.73** | 27.97 | 24.49 | **22.65** | 28.19 | 20.26 | 35.52 |
| PYSCRIBE | 40.17 | **28.29** | **24.51** | 22.47 | 28.12 | **20.35** | **35.65** |

Comparing PYSCRIBE with Baseline-3 demonstrates that replacing the Transformer-based encoder with GRUs has a substantial impact on PYSCRIBE's performance. What's more, the performance suffers much more when the Transformer-based decoder or both the encoders and the decoder are replaced with GRUs (i.e., comparing PYSCRIBE with Baseline-2 and Baseline-1).

With respect to the AST encoder design that takes both nodes and edges into account, it can be seen that the lack of either will hinder the performance of PYSCRIBE by comparing PYSCRIBE with Variant-1 and Variant-2. And the out-performance of Variant-2 to Variant-1 indicates that the edges play a greater role than nodes in the architecture. Even so, the result of PYSCRIBE shows that the combination of nodes and edges contributes greatly to the whole Python code description generation process. In terms of the triplet positions for nodes and edges that are used in their embeddings, the comparison of PYSCRIBE to Variant-3 indicates that excluding triplet position will impact PYSCRIBE's performance. Furthermore, the performance of Variant-4 shows that it won't help if the AST nodes and edges are considered as sequences like code descriptions with sequence position information added. Overall, these experimental results suggest that the positions of AST nodes and edges are indeed useful information for embedding code snippets towards learning semantic features for code comment generation.

Compared to PYSCRIBE with Variant-5 and Variant-6, PYSCRIBE achieves higher and comparable performance scores, respectively. It indicates the effectiveness of the designed two-stage decoding process. And the order of the two stages does not affect the performance.

**RQ-1**: *PYSCRIBE yields performance metrics that are consistently high across all metrics. The ablation study further revealed that all components and design choices have contributed to some extent to its performance. In particular, relying on Transformer for implementing a decoder and including AST node position information have been shown effective.*

## 5.2 | Sensitivity to training dataset and testing sample sizes

With the second research question, we investigate the sensitivity of PYSCRIBE to the experimental training and testing sets. First, we consider the dataset size property. We thus prepare nine training datasets (all samples being randomly selected from the original dataset) having sizes ranging between 5000 and 45,000 code-description pair samples, with a step of 5000. The testing dataset, however, remains the same, i.e., 1000 initial samples. Figure 8 illustrates the experimental results. Clearly, the larger the training dataset, the better the performance of PYSCRIBE on all evaluation metrics. We further observe that the performance increases rapidly when the sizes of training datasets are relatively small and tends to stabilize towards larger datasets.

We then look at the impact of testing sample properties in terms of size (different AST sizes and description lengths) on the performance of PYSCRIBE. To this end, we randomly split the testing dataset into ten size-related groups before computing the performance metric scores for each group. Figure 9 illustrates the experimental results. From the results, it can be shown that PYSCRIBE is getting better and better performance when the AST sizes and description lengths increase at first. In the fifth group (i.e., (88,110] for AST size and (28, 35] for description length), PYSCRIBE achieves the best

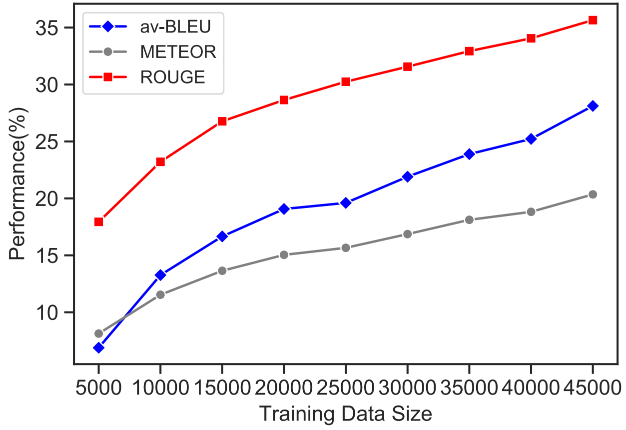


**FIGURE 8** Performance w.r.t. training dataset size.

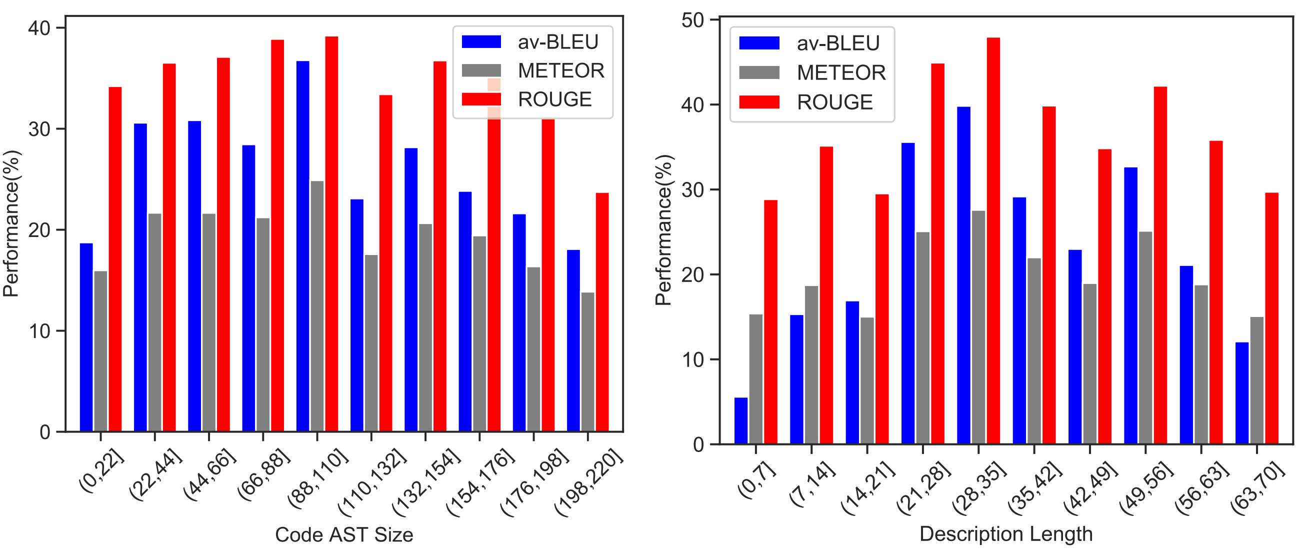


**FIGURE 9** Performance w.r.t. AST sizes (left) and description lengths (right) of testing samples.

result. Then with the growth of the AST sizes and description lengths, the performance of PyScribe becomes worse from the overall trend. Nevertheless, PyScribe shows certain stability on the whole: PyScribe is thus reliable in generating descriptions for Python code, even for large code snippets.

**RQ-2**: *Understandably, PyScribe will perform poorly with small training datasets. It also yields relatively low scores for predicting very-short and very-long descriptions. Nevertheless, PyScribe is globally effective for large code snippets and is robust overall.*

## 5.3 | Investigation on the model size

This section analyzes PyScribe's performances with modified model sizes to answer the third question. To achieve this, we first change the number of encoding and decoding layers from 2 to 6 for observation. Then we make the embedding size increase from 256 to 768 for performance comparison.

Table 5 displays the outcomes of PyScribe on our dataset when the number of encoding and decoding layers ranges from 2 to 6. The results indicate that the performance improves as the number of layers increases. For example, the

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 5 Performances of PyScribe with varying numbers of encoding and decoding layers.

| Layers | Model size | BLEU-1 | BLEU-2 | BLEU-3 | BLEU-4 | av-BLEU | METEOR | ROUGE-L |
|---|---|---|---|---|---|---|---|---|
| 2 | $57.04 \times 10^6$ | 37.01 | 24.83 | 20.84 | 18.61 | 24.48 | 18.27 | 33.27 |
| 3 | $68.60 \times 10^6$ | 39.64 | 27.68 | 23.71 | 21.55 | 27.35 | 19.91 | 34.92 |
| *4* | *$80.16 \times 10^6$* | *40.17* | *28.29* | *24.51* | *22.47* | *28.12* | *20.35* | *35.65* |
| 5 | $91.72 \times 10^6$ | 40.64 | 28.56 | 24.87 | 22.92 | 28.48 | 20.30 | 35.69 |
| 6 | $103.28 \times 10^6$ | 41.24 | 29.07 | 25.27 | 23.17 | 28.94 | 20.75 | 35.94 |

*Note*: The model size means the number of training parameters.

TABLE 6 Performances of PyScribe with different embedding sizes.

| Emb. size | Model size | BLEU-1 | BLEU-2 | BLEU-3 | BLEU-4 | av-BLEU | METEOR | ROUGE-L |
|---|---|---|---|---|---|---|---|---|
| 256 | $28.55 \times 10^6$ | 36.91 | 23.50 | 19.08 | 16.71 | 23.10 | 17.33 | 31.02 |
| 384 | $51.47 \times 10^6$ | 37.21 | 26.20 | 22.40 | 20.38 | 25.85 | 19.25 | 33.43 |
| *512* | *$80.16 \times 10^6$* | *40.17* | *28.29* | *24.51* | *22.47* | *28.12* | *20.35* | *35.65* |
| 640 | $114.62 \times 10^6$ | 39.61 | 28.45 | 25.00 | 23.28 | 28.43 | 20.76 | 35.99 |
| 768 | $154.84 \times 10^6$ | 38.55 | 27.97 | 24.40 | 22.46 | 27.67 | 20.25 | 35.45 |

*Note*: The model size means the number of training parameters.

av-BLEU/METEOR/ROUGE-L scores improve by 2.87/1.64/1.65% and 0.77/0.44/0.73% when the number of layers turns into 3 from 2 and 4 from 3, respectively. As the number of layers grows from 4 to 6, although there is still potential for improvement, the performance does not improve too much. In fact, the av-BLEU improves only by 0.36% when the number of layers turns into 5 from 4, with little change in the METEOR/ROUGE-L values. While the number of layers grows to 6, the av-BLEU/METEOR/ROUGE-L improves only by 0.46/0.35/0.25%, compared with the model with 5 layers.

Table 6 illustrates PyScribe's performances with the change of its embedding size. We observe that PyScribe's performance is improved significantly as the embedding size increases from 256 to 512. Intuitively, the av-BLEU/METEOR/ROUGE-L improve by 5.02/3.02/4.63%. When the embedding size becomes 640, the performance still improves a little. For instance, the av-BLEU/METEOR/ROUGE-L scores improve by 0.31/0.41/0.34%. However, when the embedding size grows to 768, PyScribe performs worse than that with the embedding size of 640 and 512. In fact, the av-BLEU/METEOR/ROUGE-L results decrease by 0.76/0.51/0.54% and 0.45/0.10/0.20%, respectively.

**RQ-3**: *With the increase of the model size in a certain range, PyScribe's performance will be influenced and improved greatly. Nevertheless, when the model becomes too large, it may not be much stronger or even be impacted.*

## 5.4 Comparison to the state-of-the-art

The literature includes various approaches targeting code-to-text generation tasks for different programming languages. We propose to compare PyScribe against the most recent advanced approaches and two baselines in these works. In total, we consider six approaches presented in major venues:

- **DeepCom**.[16] DeepCom introduces the LSTM and attention-based neural machine translation model to solve code comment generation. To provide the AST of source code as input of the sequence-to-sequence model, it converts the AST into a specifically formatted node sequence, which largely increases the sequence lengths.
- **CSCGDual**.[14] CSCGDual builds a dual learning-based model that trains code summarization (CS) and code generation (CG) jointly. It tries to use CG to improve the CS task. To enhance the relationship between CG and CS, it applies a constraint on probability and a constraint exploiting the nature of attention. This approach does not use the ASTs of code snippets.
- **NeuralCodeSum**.[15] NeuralCodeSum introduces Transformer architecture and incorporates the copying attention mechanism[57] to model the source code summarization. It uses source code as input for code learning.

TABLE 7 Comparison results between PyScribe and its recent closely related state-ofthe-art tools.

| Model | BLEU-1 | BLEU-2 | BLEU-3 | BLEU-4 | av-BLEU | METEOR | ROUGE-L |
|---|---|---|---|---|---|---|---|
| DeepCom[16] | 28.15 | 14.47 | 12.34 | 11.50 | 15.50 | 12.15 | 23.42 |
| CSCGDual[14] | 19.85 | 14.05 | 12.71 | 12.27 | 14.44 | 13.95 | 28.59 |
| RNN-based[15] | 30.78 | 16.66 | 13.81 | 12.69 | 17.31 | 14.35 | 28.40 |
| NeuralCodeSum[15] | 33.53 | 24.05 | 21.63 | 20.51 | 24.46 | 19.04 | 33.56 |
| NMT-only[58] | 30.98 | 16.29 | 12.87 | 11.26 | 16.44 | 13.90 | 27.19 |
| Rencos[58] | 38.34 | 24.34 | 20.66 | 18.61 | 24.47 | 18.38 | 33.91 |
| PyScribe | **40.17** | **28.29** | **24.51** | **22.47** | **28.12** | **20.35** | **35.65** |

- **RNN-based.**[15] In the implementation of NeuralCodeSum, there is a variant that replaces the Transformer with RNNs. We consider it as a comparative baseline.
- **Rencos.**[58] Rencos is a retrieval-based neural framework proposed for code comment generation. It combines the retrieval-based strategies and the strengths of the NMT model for producing better comments.
- **NMT-only.**[58] In the work,[58] NMT-only excludes the retrieval-based module and considers only the NMT-based module.

We run all the approaches on the same code-to-description Jupyter Notebook-based dataset that we have constructed. The testing and validating are performed under the same settings as in our performance assessment. Among these approaches, DeepCom, CSCGDdual, RNN-based, and MNT-only adopt encoder-decoder frameworks that are based on RNNs. Through the comparison of NeuralCodeSum and PyScribe that utilize Transformer to these RNN-based methods, it indicates that Transformer does better in learning-based code description generation.

As shown in Table 7, although CSCGDual tries to use code generation to enhance code summarization, it shows poor performance compared to all other approaches and PyScribe. For example, the av-BLEU/METEOR/ROUGE-L scores of CSCGDual are 10.03/4.43/5.32% and 13.68/6.4/7.06% lower than Rencos and PyScribe, respectively. In contrast, DeepCom adopts LSTMs to encode the AST nodes, which is still inferior to our PyScribe. The results show that PyScribe outperforms DeepCom by 12.62/8.2/12.23% in terms of av-BLEU/METEOR/ROUGE-L metrics. It can be inferred that converting an AST to its node sequence in DeepCom may lead to structural information loss during AST encoding, thus impacting the generated code comments. It is also worth mentioning that, like ours, NeuralCodeSum is also a Transformer-based approach. By replacing the Transformer module with RNNs, the performance reduction of the baseline RNN-based (e.g., 7.15/4.69/5.16% for the av-BLEU/METEOR/ROUGE-L scores) illustrates the power of Transformer for code comment generation. However, NeuralCodeSum only regards code snippets as natural language and therefore has ignored their structural information. Although it introduces the copying attention mechanism and outperforms other RNN-based approaches, PyScribe performs still better from the result shown in Table 7. Intuitively, the improvement of PyScribe over NeuralCodeSum is 3.66/1.31/2.09% in terms of av-BLEU/METEOR/ROUGE-L metrics. This result further confirms our initial hypothesis that code structural information is useful for embedding code so as to achieve code-to-text translation. Since Rencos combines the retrieval method and RNN-based neural model, it has the best result among all the baselines using RNNs. Even so, our approach PyScribe shows much better than Rencos. Particularly, it can be observed that PyScribe improves the BLEU-2/BLEU-3/BLEU-4 scores by 3.95/3.85/3.86% in contrast to Rencos, demonstrating PyScribe's superiority for longer sequence generation.

**Example generated descriptions.** Table 8 illustrates two examples to qualitatively compare the output of PyScribe, and the baseline methods. Since all the words are lemmatized with no stemming applied, it can be seen that all the descriptions generated are still well understandable.[‡] In the first case, the code length is small, while the description length is large. The result shows that only PyScribe yields a very similar description to the gold truth. Among the baseline methods, CSCGDual understands half part of the core information in the code snippet, and the text generated is not

[‡]It should be noticed that the word "a" in the phrase "a we ve" in the second example is actually "as". Since "as" is recognized as the plural word of "a" when being lemmatized by NLTK, it will become "a" after that. In fact, such minor grammatical issues are inevitable in the common pre-processing operations by NLTK, which, however, does not affect the human understanding of natural language comments too much.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

**TABLE 8** Two examples of code descriptions generated by our approach and its state-ofthe-art counterparts.

**Code**

```
a = slice(5, 50, 2)
s = 'HelloWorld'
a.indices(len(s))
for i in range(*a.indices(len(s))):
    print(s[i])
```

**Comment**

Gold: in addition , you can map a slice onto a sequence of a specific size by use it index ( size ) method . this return a tuple ( start , stop , step ) where all value have be suitably limit to fit within bind ( a to avoid index error exception when index ) . for example ,

PyScribe: in addition , you can map a slice onto a specific size by use it index ( size ) method . this return a tuple ( start , stop , step ) where all value have be <unk> limit to fit within bind ( a to fit within bind to fit bind when index ) . for example ,

DeepCom: for each medal off the total number of medal ?

CSCGDual: set a slice to a slice ( start , stop , step ) where all value have be <unk> limit to fit within bind ( a to avoid index error exception when index ) . for example ,

RNN-based: a you can see in the image above the path be not an array . when index have a value in a list , high - mean square error , and at what you re go to be generate from a start point . in this case , a list of all the index , or * bind * a a list ,

NeuralCodeSum: index slice index - slice also be use in useful for example , if you have a numpy array a sequence of index start at this index . you can use the index method to grab a sequence of index to get the default ,

NMT-only: in parallel process , it be a sequence of a list use a when you define a sequence of a sequence of size and a size of step - a you be useful when you learn about it ,

Rencos: for each medal off the total number of medal ?

**Code**

```
from sklearn.cross_validation import train_test_split
final_data.columns
x = final_data.drop('not.fully.paid',axis=1)
y = final_data['not.fully.paid']
x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.3)
```

**Comment**

Gold: train test split split our data into a train set and a test set - use sklearn to split your data into a train set and a test set a we ve do in the past -

PyScribe: train test split now it time to split our data into a train set and a test set - use sklearn to split your data into a train set a test set a we ve do in the past -

DeepCom: define the different root dataset and print it . you can find it . here , it be a time file and be it .

CSCGDual: train test split now it time to split our data into a train set and a test set a we ve do in the past -

RNN-based: split data into train and test select it use the train set

NeuralCodeSum: + prediction to split it into a train set and a test set and a test set in the past data frame to split your data into a train set and a test set -

NMT-only: train a test set

Rencos: define the different root dataset and print it . you can find it . here , it be a time file and be it .

so smooth. The rest others output the wrong descriptions, although some of them (i.e., RNN-based, NeuralCodeSum, and NMT-only) can predict several key words in the descriptions. In the second case, the code is relatively more complex than that in the first case and the description is shorter. The code in this case intends to divide the dataset into a training set and a testing set by the sklearn toolkit. All the methods predict that the code is concerned with "*dataset*". Among them, PyScribe, CSCGDual, RNN-based, and NeuralCodeSum generate the core information, while the others do not. More detailedly, the description generated by PyScribe is just slightly different from ground truth in form. CSCGDual, RNN-based, and NeuralCodeSum predict the data splitting. However, they can not infer the "*sklearn*" package. And RNN-based and NeuralCodeSum generate some redundant information, which makes the output descriptions read unsmoothly.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

More examples can be found in Table A1 in Appendix A.

**RQ-4**: PYSCRIBE *outperforms recent state-ofthe-art code to text generation approaches. Qualitative analysis of sample cases illustrates how our approach can produce relevant pieces of description for long and potentially complex code snippets.*

## 5.5 | Performance on other datasets

To further study the scalability of PYSCRIBE, we introduce two benchmarks for evaluating PYSCRIBE's performance, including: (1) a Python dataset[21,24] and (2) a Java dataset.[13] Both of the two datasets are obtained from GitHub and widely used in code comment generation.[13–15,34] Each item of data consists of a Python/Java method and its description extracted from the docstring that follows the method declaration. The two datasets are divided into *training/validation/testing* sets with *55,538/18,505/18,502* and *69,708/8,714/8,714* samples, respectively. For the result fairness, we adhere to these divisions in the experiment.

For the comparison, we introduce eleven sate-ofthe-art approaches as baselines. Apart from the three methods (DeepCom,[16] CSCGDual,[14] and NeuralCodeSum[15]) used in Section 5.4, the other eight are presented as follows:

- **CODE-NN**.[29] CODE-NN uses LSTM to model the plain source code and generate the code comment combined with attention mechanism.
- **Tree2Seq**.[59] Tree2Seq extends the sequence-to-sequence model to the tree-to-sequence model by introducing tree-based LSTM as its encoder.
- **RL+Hybrid2Seq**.[21] To enhance the correctness of generated code comments, RL+Hybrid2Seq incorporates AST and source code with LSTM and AST-based LSTM into a deep reinforcement framework.
- **API+CODE**.[13] API+CODE generates code comments with the aid of API information transferred from an API sequence summarization task.
- **mAST+GCN**.[34] mAST+GCN performs graph convolutional networks (GCNs) on AST to obtain node representations with structural information. Then the learned node sequence is fed into the Transformer framework for further encoding and decoding.
- **SiT**.[60] SiT presents a multi-view neighbouring matrix defining the relations between source code tokens based on AST. The relations are then employed to compute self-attention in the introduced structure-induced Transformer for generating code comments. The model is further fine-tuned by the pre-trained RoBERTa.[62]
- **CodeT5**.[27] As a pre-trained encoder-decoder Transformer model, CodeT5 is designed on the T5 architecture[63] for various code-related tasks, such as code generation and summarization, etc.
- **CodeBERT**.[61] CodeBERT is a pre-trained Transformer framework based on RoBERTa.[62] Although the objective of CodeBERT does not include generation tasks, it can be modified by introducing a Transformer decoder for code comment generation.[64]

Table 9 presents the results of PYSCRIBE and the baselines. We refer to the baseline performances reported by Choi et al.[34] The overall results show that the recent Transformer-based approaches NeuralCodeSum[15] and mAST+GCN[34] outperform the previous works based on RNNs.[13,14,16,21,29,59] For instance, the performance (i.e., the av-BLEU/METEOR/ROUGE-L scores) of NeuralCodeSum is 10.72/8.63/7.28% and 2.19/0.66/1.15% higher than the best RNN-based method CSCGDual on the Python and Java datasets, respectively. Compared to CSCGDual, the performance improvements of mAST+GCN correspond to 11.02/8.98/7.36% and 3.1/1.4/1.21% on the two datasets. Besides, mAST+GCN outperforms NeuralCodeSum by 0.3/0.35% and 0.91/0.74% in terms of av-BLEU/METEOR metrics on the two datasets, respectively, indicating that the AST structure is superior to the plain code for comment generation. Despite that RL+Hybrid2Seq and DeepCom introduce AST, their inferior performances in contrast to mAST+GCN and PYSCRIBE reveal the LSTM's limitation in AST encoding for code summarization. Intuitively, PYSCRIBE outperforms RL+Hybrid2Seq by 14.34/12.58/9.11% and 9.96/8.58/6.11% in terms of av-BLEU/METEOR/ROUGE-L metrics on the Python and Java datasets. In sipte of the findings above, PYSCRIBE performs much better than NeuralCodeSum and mAST+GCN on both datasets. For example, PYSCRIBE improves the av-BLEU/METEOR/ROUGE-L scores by

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 9 Comparison with the baselines on the Python and Java datasets.

| Model | Python av-BLEU | Python METEOR | Python ROUGE-L | Java av-BLEU | Java METEOR | Java ROUGE-L |
|---|---|---|---|---|---|---|
| CODE-NN[29] | 17.36 | 09.29 | 37.81 | 27.60 | 12.61 | 41.10 |
| Tree2Seq[59] | 20.07 | 08.96 | 35.64 | 37.88 | 22.55 | 51.50 |
| RL+Hybrid2Seq[21] | 19.28 | 09.75 | 39.34 | 38.22 | 22.75 | 51.91 |
| DeepCom[16] | 20.78 | 09.98 | 37.35 | 39.75 | 23.06 | 52.67 |
| API+CODE[13] | 15.36 | 08.57 | 33.65 | 41.31 | 23.73 | 52.25 |
| CSCGDual[14] | 21.80 | 11.14 | 39.45 | 42.39 | 25.77 | 53.61 |
| NeuralCodeSum[15] | 32.52 | 19.77 | 46.73 | 44.58 | 26.43 | 54.76 |
| mAST+GCN[34] | 32.82 | 20.12 | 46.81 | 45.49 | 27.17 | 54.82 |
| SiT[60] | 33.46 | 20.28 | 47.50 | 45.19 | 27.52 | 55.87 |
| CodeT5*[27] | 33.92 | 21.16 | 49.03 | 46.32 | 29.11 | 56.76 |
| CodeBERT*[61] | **34.01** | 21.53 | **49.78** | 46.90 | 29.18 | 56.88 |
| PyScribe | 33.62 | **22.33** | 48.45 | **48.18** | **31.33** | **58.02** |

*Means we rerun the model on the two datasets.

1.10/2.56/1.72% on the Python dataset and 3.60/4.90/3.26% on the Java dataset when compared to NeuralCodeSum. Compared with mAST+GCN, PyScribe's performance is enhanced by 0.80/2.21/1.64% on the Python dataset and 2.69/4.16/3.20% on the Java dataset. Evidently, the usage of triplet position for AST nodes and edges contributes significantly to PyScribe's superiority over the baselines.

As shown in Table 9, the three pre-training-based baselines (i.e., SiT, CodeT5, and CodeBERT), which benefit from the knowledge supported by large corpora, achieve excellent code summarization performance on the two datasets. For example, the ROUGE-L scores of SiT are 0.69% and 1.05% higher than mAST+GCN on the Python and Java datasets, respectively. On the Python dataset, the performance gains of CodeT5 and CodeBERT over mAST+GCN are 1.1/1.04/2.22% and 1.19/1.41/2.97%, respectively, as measured by av-BLEU/METEOR/ROUGE-L metrics. Nevertheless, our PyScribe performs even better than SiT, CodeT5, and CodeBERT based on the overall results. The table shows that PyScribe outperforms SiT by 0.16/2.05/0.95% and 2.99/3.81/2.15% on the Python and Java datasets, respectively. Besides, PyScribe's av-BLEU/METEOR/ROUGE-L scores on the Java dataset are 1.86/2.22/1.26% higher than CodeT5, despite that they have comparable performances on the Python dataset. In contrast to CodeBERT on the Python dataset, the av-BLEU/ROUGE-L scores of PyScribe decrease by 0.39/1.33% while the METEOR score improves by 0.8%. Moreover, PyScribe enhances the overall performance (i.e., av-BLEU/METEOR/ROUGE-L scores) by 1.28/2.15/1.14% when compared with CodeBERT on the Java dataset. These outcomes may be due to the fact that the pre-trained knowledge is only derived from the sequential source code tokens, without any structural information incorporated. In general, the findings above further demonstrate the importance of structural features to code comment generation.

The result analysis above illustrates the scalability and outperformance of PyScribe. It also turns out that: (1) The models using Transformer are quite more powerful than the methods based on RNNs in automatic code comment generation; (2) the AST contains more structural features than the source code, enabling models to learn better code representations for comment generation; (3) by integrating triplet positions and edge information into the Transformer-based structure, PyScribe has more comprehensive learning capacity for ASTs to generate code comments with higher quality.

**RQ-5**: *PyScribe improves the code comment generation performance on two other popular benchmarks. As a result, it indicates that PyScribe is scalable enough in such a task.*

# 6 | DISCUSSION

This section discusses the implications of our work and threats to validity.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

## 6.1 | Implication

**Considering more programming languages for code summarization.** Through the preliminary literature review we have conducted in this paper, we find that most state-ofthe-art studies, at the moment, focus on summarizing Java code.[16–19] Python has not been a popular target, although the language has become more and more popular in recent years. Other programming languages are even more rarely targetted. Therefore, we argue that our community should make more efforts to consider more programming languages when targeting code summarization approaches. One possible approach would be to make the code summarization approach more generic, i.e., suitable for explaining all types of languages (with minimal modifications, if any). In future work, we intend to enhance our method by integrating additional pre-trained models (e.g., BERT[65]) to implement the summarization of other programming languages.

**Building more diverse and realistic datasets for learning code summarization.** Traditionally, code summarization studies mainly leverage code comments to learn for summarization, which makes the learning dataset quite monotonous. Indeed, comments are mainly applied to explain well-written APIs and will often include special considerations for accessing the methods. However, this style of explanation may not be perfectly suitable for describing a more loosely written code (such as code snippets written in Jupyter Notebooks). To this end, we build and present to the research community a novel dataset leveraging code explanations presented in popular Python-based Notebooks. Due to its interactive design, programmers have much more freedom to take notes on what they code. And it leads to varieties of styles of code and descriptions. As a result, the data with this diversity will be much closer to real-world programming activities. As shown in Figure 9, the larger the size of the dataset, the better the model performs on it. This evidence suggests that it is necessary to build an even larger, more realistic, and more diverse dataset for learning code summaries.

**Considering more structural information from code to learn code representation.** In this study, we have put forward a Transformer-based network PYSCRIBE and demonstrated its effectiveness on our dataset. PYSCRIBE achieves its purpose by modeling a code AST with structural information by jointly incorporating three types of features (i.e., node, edge, and triplet position). The experimental results (as displayed in Table 4) confirm our hypothesis that more structural information should be learned from code to deliver better code representation. This promising result further suggests that our community should explore more rich code representations for performing code-based neural network learning.

## 6.2 | Threats to validity

One of the validity threats relates to the experimental process, including the bugs in our code and bias in the baseline replication. As for the implementation of our approach, the code is double-checked to reduce the bugs that harm the experiment. And to mitigate the impact concerned with bugs, the source code with the dataset will be published online for further study. While implementing the methods of which the code is unavailable would lead to severe bias to validity, we select the existing state-of-art methods with all the code open-sourced by their authors. However, the parameters should be modified in replication because our dataset has differences from the datasets used in these baseline works, such as the size of datasets, maximum length of code and descriptions, etc. In order to eliminate this threat, we have tried our best to tune the parameters to make these baseline methods generate better results.

The dataset proposed in this work is also a potential threat to validity, including the quality and generalization. We crawled a huge number of Jupyter Notebook projects from GitHub for building the dataset needed. To improve the dataset's quality as much as possible, several steps concerning layouts of cells, lexical rules, and semantic calculations are concatenated for data extraction. Despite our best efforts to remove noisy samples, such as repeated samples,[66] the dataset size may limit its generalizability in applications. In future work, we hope to enlarge our dataset from GitHub and other online platforms.

The third potential threat is the scalability of our model in the code comment generation task. We have verified the superiority of PYSCRIBE compared to the state-ofthe-art on our proposed dataset in Section 5.4. However, one may argue that it is uncertain on other datasets. To validate PYSCRIBE's scalability, we have extended the experiment on two other popular datasets and refer to the baseline results from Reference 34. The comparison

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

still demonstrates the outperformance of PyScribe. Furthermore, recent studies[67–69] have started to investigate the impact on the code-to-text translation models by leveraging cross-project datasets. So, in the future, we plan to assess PyScribe on more datasets concerning other programming languages (e.g., Javascript and C) and cross-project tasks.

One other threat to the validity of this study is the evaluation metrics. For the natural language generated, contradictions could happen between these metrics and human judgment. To mitigate the potential pitfalls, we have applied several metrics to comprehensively validate how PyScribe and other methods perform. These metrics are all widely used in such scenarios. The results in terms of all metrics successfully demonstrate PyScribe's outstanding performance. Besides, we do an in-depth study of two examples based on human judgment and provide four examples in Appendix. It further helps interpret what PyScribe can learn compared to the baseline methods. Overall, it is convinced that the threat to the evaluation metrics' validity is minimal. In spite of this, a human evaluation can be performed in future work.

## 7 | CONCLUSION

Code comment generation via structural AST feature learning has garnered significant interest in recent years. Nonetheless, there are still unresolved issues, such as the positional information loss in AST representation. Therefore, this paper improved the Transformer model into a novel encoder-decoder neural architecture PyScribe for producing Python code descriptions. Given a snippet of Python code, PyScribe first attaches a triplet position to each AST edge and node to comprehensively preserve structural information in source code. The positional edges and nodes then flow into two separate Transformer encoders for AST encoding. Afterward, PyScribe implements two-stage decoding processes over the extracted node-level and edge-level features to generate textual code comments. To assess PyScribe's effectiveness, we have built a large Python dataset with 47,000 code-description sample pairs that are carefully filtered and collected from the real-world Jupyter Notebook repository. Finally, based on standard performance metrics, we ran comprehensive experiments to demonstrate that PyScribe's design choices contribute to making it superior over the baselines. Even further investigation on other widely used datasets turns out that PyScribe outperforms the state-ofthe-art approaches in comment generation.

To promote reproducibility studies, we have made available online our tool implementation and our carefully constructed code-to-description dataset at the following link: https://github.com/SMAT-Lab/PyScribe.

### AUTHOR CONTRIBUTIONS

**Juncai Guo:** Conceptualization, Methodology, Software, Investigation, Visualization, Writing-Original Draft, Data Curation, Funding acquisition. **Jin Liu:** Supervision, Resources, Project administration, Funding acquisition. **Xiao Liu:** Supervision, Conceptualization, Formal analysis, Validation, Writing-Review & Editing. **Wan Yao:** Conceptualization, Formal analysis, Validation, Writing-Review & Editing. **Yanjie Zhao:** Investigation, Writing-Original Draft, Writing-Review & Editing. **Li Li:** Supervision, Conceptualization, Data Curation, Formal analysis, Writing-Review & Editing. **Kui Liu:** Conceptualization, Writing-Original Draft, Writing-Review & Editing. **Jacques Klein:** Visualization, Formal analysis, Writing-Review & Editing. **Tegawendé F. Bissyandé:** Visualization, Formal analysis, Validation, Writing-Review & Editing.

### ACKNOWLEDGMENT

This research is supported by the National Natural Science Foundation of China (Grant No.61972290), the Open Fund of Key Laboratory of Network Assessment Technology from Chinese Academy of Sciences, and the China Scholarship Council (Grant No.201906270158). Open access publishing facilitated by Deakin University, as part of the Wiley - Deakin University agreement via the Council of Australian University Librarians.

### DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in https://github.com/SMAT-Lab/PyScribe.

### ORCID

*Juncai Guo* https://orcid.org/0009-0002-0048-6517

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

*Jacques Klein* https://orcid.org/0000-0003-4052-475X

## REFERENCES


1. Stueben M. *Self-DOCUMENTING CODE: 67–90*. Apress; 2018.
2. Raskin J. Comments Are More Important Than Code. *ACM Queue*. 2005;3:64. doi:10.1145/1053331.1053354
3. McBurney PW, McMillan C. Automatic documentation generation via source code summarization of method context. In: Roy CK, Begel A, Moonen L, eds. *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*. ACM; 2014:279-290.
4. Sridhara G, Hill E, Muppaneni D, Pollock L, Vijay-Shanker K. Towards automatically generating summary comments for Java methods. *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM; 2010:43-52.
5. Sridhara G, Pollock L, Vijay-Shanker K. Automatically detecting and describing high level actions within methods. *2011 33rd International Conference on Software Engineering (ICSE)*. ACM; 2011:101-110.
6. Robillard P. Schematic pseudocode for program constructs and its computer automation by SCHEMACODE. *Commun ACM*. 1986;29:1072-1089. doi:10.1145/7538.7541
7. Ohba M, Gondow K. Toward mining "concept keywords" from identifiers in large software projects. *ACM SIGSOFT Softw Eng Notes*. 2005;30(4):1-5. doi:10.1145/1082983.1083151
8. Maskeri G, Sarkar S, Heafield K. Mining business topics in source code using latent dirichlet allocation. *Proceedings of the 2008 1st India Software Engineering Conference, ISEC'08*. ACM; 2008:113-120.
9. Haiduc S, Aponte J, Marcus A. Supporting program comprehension with source code summarization. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. ACM; 2010:223-226.
10. Wong E, Yang J, Tan L. AutoComment: Mining question and answer sites for automatic comment generation. *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE; 2013:562-567.
11. Wong E, Liu T, Tan L. CloCom: Mining existing source code for automatic comment generation. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE Computer Society; 2015:380-389.
12. Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need. *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems*. Curran Associates, Inc; 2017:5998-6008 http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf
13. Hu X, Li G, Xia X, Lo D, Lu S, Jin Z. Summarizing source code with transferred API knowledge. *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018*. ijcai.org; 2018:2269-2275.
14. Wei B, Li G, Xia X, Fu Z, Jin Z. Code generation as a dual task of code summarization. *Advances in Neural Information Processing Systems*. Curran Associates, Inc; 2019:6563-6573 http://arxiv.org/abs/1910.05923
15. Ahmad W, Chakraborty S, Ray B, Chang KW. A Transformer-based approach for source code summarization. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics; 2020:4998-5007.
16. Hu X, Li G, Xia X, Lo D, Jin Z. Deep code comment generation. *ICPC '18: Proceedings of the 26th Conference on Program Comprehension*. ACM; 2018:200-210.
17. Hu X, Li G, Xia X, Lo D, Jin Z. Deep code comment generation with hybrid lexical and syntactical information. *Empir Softw Eng*. 2020;25:2179-2217. doi:10.1007/s10664-019-09730-9
18. Alon U, Brody S, Levy O, Yahav E. code2seq: Generating sequences from structured representations of code. *7th International Conference on Learning Representations (ICLR)*. OpenReview.net; 2019.
19. LeClair A, Haque S, Wu L, McMillan C. Improved code summarization via a graph neural network. *2020 IEEE/ACM International Conference on Program Comprehension (ICPC)*. ACM; 2020:184-195.
20. Liu S, Chen Y, Xie X, Siow J, Liu Y. Automatic code summarization via multi-dimensional semantic fusing in GNN. *CoRR*. 2020; abs/2006.05405; https://arxiv.org/abs/2006.05405
21. Wan Y, Zhao Z, Yang M, et al. Improving automatic source code summarization via deep reinforcement learning. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM; 2018:397-407.
22. Liang Y, Zhu KQ. Automatic generation of text descriptive comments for code blocks. *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18)*. AAAI Press; 2018:5229-5236.
23. Gehring J, Auli M, Grangier D, Yarats D, Dauphin YN. Convolutional sequence to sequence learning. *Proceedings of the 34th International Conference on Machine Learning*. Vol 70. PMLR; 2017:1243-1252.
24. Barone AVM, Sennrich R. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *Proceedings of the Eighth International Joint Conference on Natural Language Processing, IJCNLP 2017*. Asian Federation of Natural Language Processing; 2017:314-319.
25. LeClair A, Jiang S, McMillan C. A neural model for generating natural language summaries of program subroutines. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE / ACM; 2019:795-806.
26. Chen M, Tworek J, Jun H, et al. Evaluating large language models trained on code. *CoRR*. 2021; abs/2107.03374.


1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

27. Wang Y, Wang W, Joty SR, Hoi SCH. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*. Association for Computational Linguistics; 2021:8696-8708.
28. OpenAI. *ChatGPT*. Computer Software; 2023.
29. Iyer S, Konstas I, Cheung A, Zettlemoyer L. Summarizing source code using a neural attention model. *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. The Association for Computer Linguistics; 2016:2073-2083.
30. Tian H, Liu K, Kaboré AK, et al. Evaluating representation learning of code changes for predicting patch correctness in program repair. *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*. IEEE; 2020:981-992.
31. Wan Y, Shu J, Sui Y, et al. Multi-modal attention network learning for semantic source code retrieval. *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*. IEEE; 2019:13-25.
32. Shido Y, Kobayashi Y, Yamamoto A, Miyamoto A, Matsumura T. Automatic source code summarization with extended tree-LSTM. *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE; 2019:1-8.
33. Zhou Z, Yu H, Fan G. Effective approaches to combining lexical and syntactical information for code summarization. *Softw Pract Exp*. 2020;50(12):2313-2336. doi:10.1002/spe.2893
34. Choi Y, Bak J, Na C, Lee J. Learning sequential and structural information for source code summarization. *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021*. Association for Computational Linguistics; 2021:2842-2851.
35. Wang X, Wang Y, Mi F, et al. SynCoBERT: Syntax-guided multi-modal contrastive pre-training for code representation. arXiv preprint, arXiv:2108.04556 2021.
36. Zhao Y, Li L, Wang H, He Q, Grundy J. APIMatchmaker: Matching the right APIs for supporting the development of android apps. *IEEE Trans Softw Eng*. 2022;49:113-130.
37. Zhao Y, Li L, Sun X, Liu P, Grundy JC. *Icon2Code*: Recommending code implementations for Android GUI components. *Inf Softw Technol*. 2021;138:106619. doi:10.1016/j.infsof.2021.106619
38. Allamanis M, Peng H, Sutton C. A convolutional attention network for extreme summarization of source code. In: Balcan M, Weinberger KQ, eds. *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. Vol 48. JMLR.org; 2016:2091-2100.
39. Wei H, Li M. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. ijcai.org; 2017:3034-3040.
40. Cho K, Merrienboer vB, Gülçehre Ç, et al. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*. ACL; 2014:1724-1734.
41. Kipf TN, Welling M. Semi-supervised classification with graph convolutional networks. *Proceedings of the 5th International Conference on Learning Representations*. OpenReview.net; 2017.
42. Wang W, Zhang Y, Sui Y, et al. Reinforcement-learning-guided source code summarization via hierarchical attention. *IEEE Trans Softw Eng*. 2020;48:102-119. doi:10.1109/TSE.2020.2979701
43. Bird S. NLTK: The natural language toolkit. *ACL 2006, 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*. The Association for Computer Linguistics; 2006:69-72.
44. Liu K, Kim D, Bissyandé TF, Yoo S, Le Traon Y. Mining fix patterns for FindBugs violations. *IEEE Trans Softw Eng*. 2018;47(1):165-188. doi:10.1109/TSE.2018.2884955
45. Liu K, Kim D, Bissyandé TF, et al. Learning to spot and refactor inconsistent method names. *Proceedings of the 41st International Conference on Software Engineering*. IEEE; 2019:1-12.
46. Mou L, Li G, Zhang L, Wang T, Jin Z. Convolutional neural networks over tree structures for programming language processing. *Proceedings of the 30th AAAI Conference on Artificial Intelligence*. AAAI Press; 2016:1287-1293.
47. Zhang J, Wang X, Zhang H, Sun H, Wang K, Liu X. A novel neural source code representation based on abstract syntax tree. *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering*. IEEE / ACM; 2019:783-794.
48. Le QV, Mikolov T. Distributed representations of sentences and documents. *Proceedings of the 31st International Conference on Machine Learning*. JMLR.org; 2014:1188-1196.
49. He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society; 2016:770-778.
50. Ba LJ, Kiros JR, Hinton GE. Layer normalization. *CoRR*. 2016; abs/1607.06450; http://arxiv.org/abs/1607.06450
51. Sun Z, Li L, Liu Y, Du X, Li L. On the importance of building high-quality training datasets for neural code search. *The 44th International Conference on Software Engineering (ICSE 2022)*. ACM: 2022.
52. Repplinger J, Chowdhury GG. *Introduction to Modern Information Retrieval*. 3rd ed.; 2019 *Coll Res Libr*, Facet publishing: 2011;72(2):194-195. http://crl.acrl.org/content/72/2/194.full.pdf
53. Kingma DP, Ba J. Adam: A method for stochastic optimization. *3rd International Conference on Learning Representations*. OpenReview.net; arXiv:2015.
54. Papineni K, Roukos S, Ward T, Zhu W. Bleu: A method for automatic evaluation of machine translation. *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. ACL; 2002:311-318.


1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

55. Banerjee S, Lavie A. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. *Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*. Association for Computational Linguistics; 2005:65-72.
56. Lin CY. ROUGE: A Package for Automatic Evaluation of Summaries. *Proceedings of the ACL Workshop: Text Summarization Braches Out 2004*. Association for Computational Linguistics; 2004:74-81.
57. See A, Liu PJ, Manning CD. Get to the point: summarization with pointer-generator networks. *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics; 2017:1073-1083.
58. Zhang J, Wang X, Zhang H, Sun H, Liu X. Retrieval-based neural source code summarization. *ICSE '20: 42nd International Conference on Software Engineering*. ACM; 2020:1385-1397.
59. Eriguchi A, Hashimoto K, Tsuruoka Y. Tree-to-sequence attentional neural machine translation. *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*. The Association for Computer Linguistics; 2016:823-833.
60. Wu H, Zhao H, Zhang M. Code Summarization with Structure-induced Transformer. *ACL/IJCNLP 2021 of Findings of ACL*. Association for Computational Linguistics; 2021:1078-1090.
61. Feng Z, Guo D, Tang D, et al. CodeBERT: A pre-trained model for programming and natural languages. *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics; 2020:1536-1547.
62. Liu Y, Ott M, Goyal N, et al. RoBERTa: A robustly optimized BERT pretraining approach. *CoRR*. 2019; abs/1907.11692.
63. Raffel C, Shazeer N, Roberts A, et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *J Mach Learn Res*. 2020;21(140):1-67.
64. Wang Y, Dong Y, Lu X, Zhou A. GypSum: Learning hybrid representations for code summarization. In Rastogi A, Tufano R, Bavota G, Arnaoudova V, Haiduc S. eds. *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. ICPC 2022, Virtual Event, May 16-17. ACM; 2022:abs/2204.12916. doi:10.48550/arXiv.2204.12916
65. Devlin J, Chang M, Lee K, Toutanova K. BERT: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019*. Association for Computational Linguistics; 2019:4171-4186.
66. Zhao Y, Li L, Wang H, et al. On the impact of sample duplication in machine-learning-based android malware detection. *ACM Trans Softw Eng Methodol*. 2021;30(3):40:1-40:38. doi:10.1145/3446905
67. Tao W, Wang Y, Shi E, et al. On the evaluation of commit message generation models: An experimental study. *IEEE International Conference on Software Maintenance and Evolution, ICSME 2021*. IEEE; 2021:126-136.
68. LeClair A, McMillan C. Recommendations for datasets for source code summarization. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019*. Association for Computational Linguistics; 2019:3931-3937.
69. Allamanis M. The adverse effects of code duplication in machine learning models of code. In: Masuhara H, Petricek T, eds. *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019, Athens, Greece, October 23-24, 2019*. ACM; 2019:143-153.


**How to cite this article:** Guo J, Liu J, Liu X, et al. PyScribe–Learning to describe python code. *Softw Pract Exper*. 2023;1-27. doi: 10.1002/spe.3291

## APPENDIX . QUALITATIVE EXAMPLES

Table A1 provide four qualitative examples of PyScribe and the baselines. The overall results show that PyScribe generates better comments for the given Python code snippets. For example in the first case in Table A1, only PyScribe produces the comment that is closest to the ground truth. The comments generated by RNN-based and NeuralCodeSum are not complete. CSCGDual and NMT-only do not predict readable texts. DeepCom and Rencos output irrelevant information for the code.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE A1 Qualitative examples on the Python dataset.

| | |
|---|---|
| **Code** | `min_price = min(zip(prices.values(), prices.keys()))`<br>`max_price = max(zip(prices.values(), prices.keys()))`<br>`# max_price is (612.78, 'AAPL')` |
| **Comment** | Gold: in order to perform useful calculation on the dictionary content , it be often useful to invert the key and value of the dictionary use zip () . for example , here be how to find the minimum and maximum price and stock name ,<br>PyScribe: in order to perform useful calculation on the dictionary content , it be often useful to invert the key and value of the dictionary use - zip () - for example , here be how to find the minimum and stock name ,<br>DeepCom: check for the data<br>CSCGDual: here , some dictionary - useful function for the dictionary use * zip () - for the dictionary<br>RNN-based: in order to perform useful calculation on the dictionary content , it be often useful to invert the key and value of the dictionary use - path () - method ,<br>NeuralCodeSum: in order to perform useful calculation on the dictionary content , it be often useful to invert the key and value of the dictionary use = = = = = = example ,<br>NMT-only: in order to find nearby minimum and minimum stock in the dictionary<br>Rencos: check for the data |
| **Code** | `params = {'project' : 'en.wikipedia.org',`<br>`'access' : 'mobile-app',`<br>`'agent' : 'user',`<br>`'granularity' : 'monthly',`<br>`'start' : '2015070100',`<br>`'end' : '2017100100'}`<br>`api_call = requests.get(endpoint.format(**params))`<br>`response_pageviews_mobile_app_201507_201709 = api_call.json()`<br>`with open('data/pageviews_mobile_app_201507_201709.json', 'w') as outfile:`<br>`json.dump(response_pageviews_mobile_app_201507_201709, outfile)` |
| **Comment** | Gold: call the page view api ( documentation , endpoint ) that provide access to mobile app traffic data from july 2015 through september 2017 and save the raw result into a json source data file .<br>PyScribe: call the page view api ( documentation , endpoint ) that provide access to mobile web traffic data from july 2015 through september 2017 and save the raw result into a json source data file .<br>DeepCom: create a new list to student , filter it and the student - dictionary be filter by role $ {<br>CSCGDual: make a web page use july 2015 to mobile app .<br>RNN-based: json , july 2016 the raw data the user be provide from the mobile package , and get the raw source traffic . the raw data documentation that be from mobile . through the mobile , july 2016 and save the raw result into a json data . we ll be use the mobile and page on account . it be save in mobile , july 2016 .<br>NeuralCodeSum: call the step generate a json api , , for save the result from january return data generate json .<br>NMT-only: get all page view api the api key api be collect api out of the json web september 2017 . the api , json view the request from the json file a well a the json file ,<br>Rencos: create a new list to student , filter it and the student - dictionary be filter by role $ { |
| **Code** | `words = []`<br>`for dataset in ["positive_tweets.json", "negative_tweets.json"]:`<br>`for tweet in twitter_samples.tokenized(dataset):`<br>`words.extend(tweet)`<br>`len(words)` |
| **Comment** | Gold: create a list of all word before perform sentiment analysis , let 's first inspect the dataset a little bite more by create a list of all word .<br>PyScribe: exercise 3 , create a list of all word before perform sentiment analysis , let 's first inspect the dataset a little bite more by create a list of all word .<br>DeepCom: convert the new date into datetime object and the date be<br>CSCGDual: create it , let 's first inspect the dataset a little bite more by create a list of all word .<br>RNN-based: create a list of all word before perform sentiment analysis<br>NeuralCodeSum: exercise 3 , create a list of all word before perform sentiment analysis ( word ) that be perform by create a list of word in the dataset ,<br>NMT-only: exercise 3 . 5 , let 's create a list of all word that have a word in the dataset . let 's inspect the first 10 word ,<br>Rencos: convert the new date into datetime object and the date be |
| **Code** | `z3 = [-0.9463*v1/0.9884 + 2.1279/0.9884 for v1 in x1]`<br>`plt.scatter(x1,x2, c=ex_df['y'])`<br>`plt.plot(x1,z3, linestyle='-', color='r')`<br>`plt.title('statsmodels logistic regression line')` |
| **Comment** | Gold: plot the line divide the x1 , x2 plane into the two region , from the stats model regression<br>PyScribe: plot the line divide the x1 , x2 plane into the two region , from the sklearn regression<br>DeepCom: create a pipeline with the follow step ,<br>CSCGDual: plot the regression model<br>RNN-based: create a field with the value of ) find the square root of the variance .<br>NeuralCodeSum: plot the line even if we plot the y return the online ,<br>NMT-only: and display the line .<br>Rencos: create a pipeline with the follow step , |

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3291 by Cochrane Luxembourg, Wiley Online Library on [11/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License