Learning Sequential and Structural Information for Source Code Summarization

YunSeok Choi, JinYeong Bak, CheolWon Na, Jee-Hyong Lee
College of Computing and Informatics
Sungkyunkwan University
Suwon, South Korea
{ys.choi, jy.bak, ncw0034, john}@skku.edu

Abstract

We propose a model that learns both the sequential and the structural features of code for source code summarization. We adopt the abstract syntax tree (AST) and graph convolution to model the structural information and the Transformer to model the sequential information. We convert code snippets into ASTs and apply graph convolution to obtain structurally-encoded node representations. Then, the sequences of the graph-convolutioned AST nodes are processed by the Transformer layers. Since structurally-neighboring nodes will have similar representations in graph-convolutioned trees, the Transformer layers can effectively capture not only the sequential information but also the structural information such as sentences or blocks of source code. We show that our model outperforms the state-of-the-art for source code summarization by experiments and human evaluations.

1 Introduction

Descriptions of source code are very important documents for programmers. Good descriptions help programmers understand the meaning of code quickly and easily.

Source code has sequential information (code tokens) and structural information (dependency and structure). To understand the content of the code and generate a good summary, both pieces of information are essential to understand the summary. However, most previous works used only one kind of information. Iyer et al. (2016); Liang and Zhu (2018); Hu et al. (2018b); Allamanis et al. (2016) simply converted the source code into sequences of tokens and tried to extract the features of the source code from the sequential information of the source code. They rarely considered the structure information about the relationship between tokens. On the other hand, Shido et al. (2019); Harer et al. (2019); LeClair et al. (2020); Scarselli et al. (2008) proposed tree-based models to capture the features of the source code. They used the structural information from parse trees but hardly considered the sequence information of code tokens.

In order to accurately understand and represent the source code, it is necessary to encode the structural information as well as the sequential information. Recently, Ahmad et al. (2020) tried to represent both the sequential and the structural information using the Transformer model with relative encoding. Since relative encoding clipped the maximum distance for attention without considering the parse tree, it is limited to represent the structure of the code.

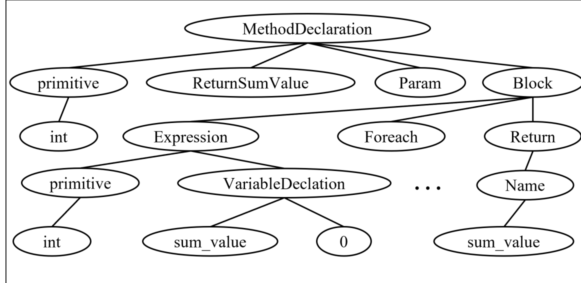
In this work, we propose a model that learns both the structural and sequential information of source code. We represent code snippets as abstract syntax trees (ASTs), and apply graph convolution (Kipf and Welling, 2017) to the ASTs to obtain the node representation reflecting the tree structure such as parents, children, and siblings. Nodes that are close to each other in a tree, such as parent and child nodes and sibling nodes, will have similar representations. Next, we convert the graph-convolutioned ASTs into sequences by the pre-order traversal and process them with Transformer layers (Vaswani et al., 2017). Since structurally-neighboring nodes have similar representations, such nodes will pay more attention to one another in the Transformer layers. Thus, the Transformer layers can easily capture not only the sequential information but also the structural information such as sentences or blocks of source code.

We also modify ASTs to better represent the structural information of the source code. We add sibling edges to represent neighboring blocks in source code and add a node representing the name of the function for Python. With the modification,

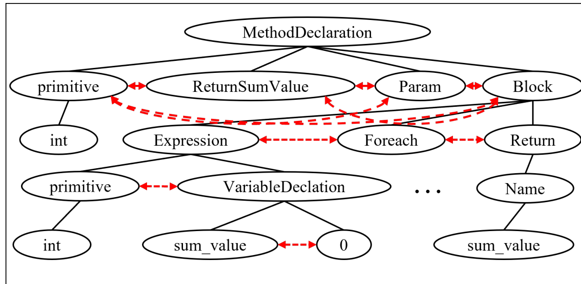
Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021, pages 2842–2851
August 1–6, 2021. ©2021 Association for Computational Linguistics

```
public static int returnSumValue(int[] values) {
    int sum_value = 0;
    for (int value : values) {
        sum_value += value;
    }
    return sum_value;
}
```

(a) Source Code



(b) AST

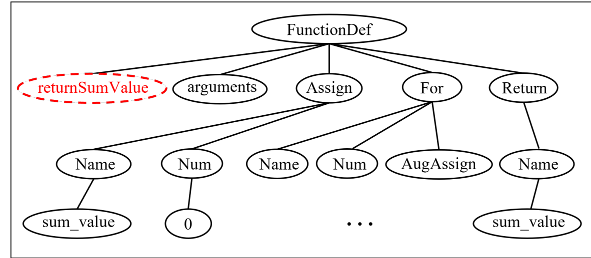


(c) mAST

Figure 1: An example of Java code, Abstract Syntax Tree and modified-Abstract Syntax Tree.

```
def returnSumValue(values):
    sum_value = 0
    for value in values:
        sum_value += value
    return sum_value
```

(a) Source Code



(b) AST

Figure 2: An example of Python code and Abstract Syntax Tree. The Python AST parser we used does not create a node for the function name unlike the Java AST parser. Since the function name is a very important keyword in generating summaries, we add the function name in the red box (a) to *FunctionDef* node in the Python AST.

our model can better catch the blocks in the source code.

In the experiment, we show that our model outperforms the state-of-the-art for source code summarization. We use two well-known Java (Hu et al., 2018b) and Python (Wan et al., 2018) datasets collected from Github. We additionally perform human evaluations and analyze the attention maps between code tokens to see how well the model has captured the structural information of code. The result proves that it is very effective to model both structural and sequential information for source code summarization.

We describe the modified AST (mAST) in Section 2 and present the proposed approach in Section 3. In Section 4, we show the superiority of our approach with experimental results and human evaluations. We describe the related work in source code summarization and compare their approaches with our proposed approach in Section 5. Finally, we conclude the paper in Section 6.

2 Representing Code as mAST

Figure 1 shows a code snippet and its AST in Java. The abstract syntax tree (AST) is a structure to represent the abstract syntactic structure of code in a programming language. Source code is separated into blocks and can be transformed into a tree structure. The leaf nodes of an AST represent code identifiers and names. The non-leaf nodes represent the grammar or the structure of the language. All non-leaf nodes in an AST have the structural information about which blocks they belong to (parent node) and which block they have (child nodes). So, we can easily catch the structure information of code from ASTs.

In order to more effectively represent structural information, we modify ASTs by adding edges between siblings. Statements or blocks at the same level in a code snippet are represented as sibling nodes. For example, *Expression* node (line 2 in the code), *Foreach* node (line 3) and *Return* node (line 7) are the statements or blocks at the same level, and they are represented as siblings in the AST. However, in ASTs, blocks at the same level (sibling nodes) are not directly connected as shown in Figure 1b. We can indirectly catch such information via parent nodes.

Neighboring blocks are very important for the sequential and structural understanding of source code. To directly represent neighboring blocks, we

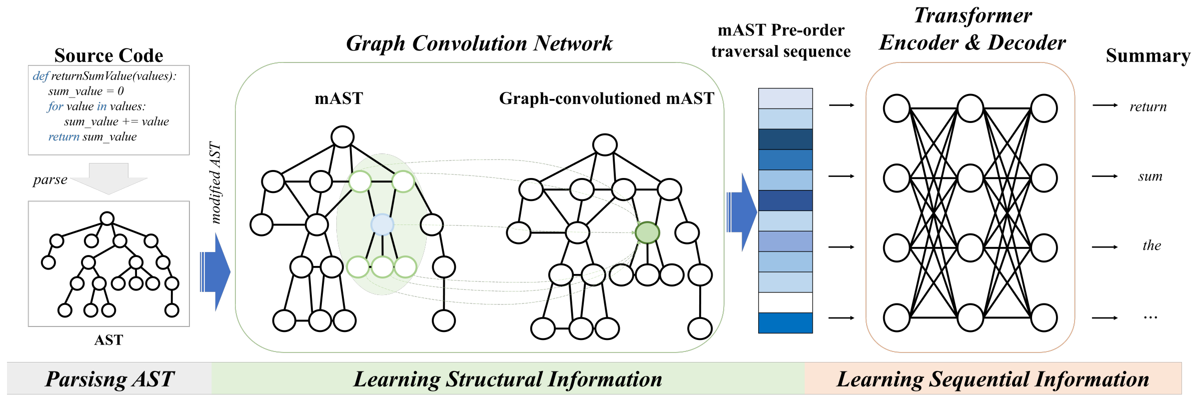


Figure 3: Overview of our proposed model.

add sibling edges to ASTs as shown in Figure 1c (red dotted lines).

In the case of Python, the Python AST parser we used does not create a node for the function name, unlike the Java AST parser. As the function name is a very important keyword in generating summaries, we add the function name to *FunctionDef* node as a child in the Python AST as shown in Figure 2. Then, we modify Python ASTs by adding sibling edges.

3 Proposed Model

We propose a model using graph convolution layers and transformer layers to summarize the source code. To encode both the structural and the sequential information of code, we combine both of the above layers.

Figure 3 shows the overview of our model. A given snippet is represented as a modified AST (mAST). The initial representation of nodes in the mAST is generated by the embedding layer. Since the embedding layer generates the representation considering only nodes themselves, we use graph convolution to capture the structural information. We apply graph convolution to each node in mAST. Then, we can have node representations considering the structural features as well as the node features.

The graph-convolutioned mASTs are converted into sequences by pre-order traversal, and the sequences are given to the Transformer encoder. Since structurally-neighboring nodes have similar representations, the Transformer encoder can effectively capture not only the sequential features but also the structural features such as sentences or blocks of source code. After the Transformer encoder generates the representations by reflecting the sequential and structural information, the Transformer decoder generates summaries.

3.1 Graph Convolution Network

Graph convolutional network (Kipf and Welling, 2017) is one of graph neural networks for representing nodes based on neighborhood features of each node in graph data. In this paper, graph convolution layers are used to capture the structure information of mASTs. Since the mAST extracted from a given code C is a graph, we denote an AST as $G(C) = \{V, E\}$, where V is a set of nodes and E is a set of edges. Initially, nodes in V are one-hot encoded tokens and then mapped into representation vectors, X, by the embedding layer.

Given representation X of nodes and E of edges, new representations of nodes are calculated by graph convolution layers as follows.

$$H^0 = X, X \in \mathbb{R}^{n \times d}$$

$$H^{(l+1)} = \sigma(AH^{(l)}W^{(l)}), W^l \in \mathbb{R}^{d \times d}$$

where A is the adjacency matrix, W^l is the graph convolution weight matrix in the l-th layer, σ is the activation function, n is the total number of nodes in an mAST, and d is the embedding dimension. The feature of each node represented by the graph convolution layer is denoted as H. In the experiment, the dimension of the weight matrix in a graph convolution is d=512.

3.2 Transformer Encoder-Decoder

After graph convolution layers, the mAST is converted into a sequence by pre-order traversal. The pre-order traversal is applied to the original AST, not the mAST, because mASTs are transformed

Dataset	Java	Python
Train	69,708	55,538
Valid	8,714	18,505
Test	8,714	18,502
Unique leaf nodes in ASTs	106	54
Unique non-leaf nodes in ASTs	57,372	101,229
Unique tokens in summaries	46,895	56,189
Avg. nodes in AST	131.72	104.11
Avg. tokens in summary	17.73	9.48

Table 1: Statistics of Java and Python dataset

into graphs by adding sibling edges. The mAST is used to obtain the structural representation of nodes by considering nodes and their neighbors. Since the original AST contains the original structure of the source code, we use it to obtain a sequence.

The Transformer encoder and decoder follow the graph convolution layers. The sequence of the mAST nodes is processed into the Transformer encoder. The Transformer architecture is good at capturing long-term dependencies in a sequence. Since we used graph convolutions, which generate similar representations for structurally-neighboring nodes, the Transformer encoder can easily capture dependencies between nodes in the same code block, between similar code blocks, and between code blocks at the same level. As a result, the Transformer encoder can generate new representation vectors which well reflect sequential and the structural information.

Next, the Transformer decoder generates the token of summary from the vectors generated by the Transformer encoder. In the experiment, the dimension of nodes and summary tokens is d_{model}=512. The Transformer encoder and decoder are respectively composed of a stack of N = 6 layers.

4 Experiment

We perform various experiments to show the superiority of our model for source code summarization.

4.1 Setup

Datasets We evaluate our model using Java dataset (Hu et al., 2018b) and Python dataset (Wan et al., 2018). The statistics of the experiment datasets are shown in Table 1. We used the Java parser used by Alon et al. (2019) and the Python parser used by Wan et al. (2018) for extracting the abstract syntax tree of the code.

Metrics We adopt 3 performance metrics: BLEU (Papineni et al., 2002), METEOR (Banerjee and Lavie, 2005), and ROUGE-L (Lin, 2004).

Baselines We compare our model with baseline models based on sequential information by Iyer et al. (2016); Hu et al. (2018a,b); Wei et al. (2019); Ahmad et al. (2020) and based on structural information by Eriguchi et al. (2016); Wan et al. (2018). We refer to the baseline results reported by Ahmad et al. (2020).

Hyper-parameters We set the maximum length to 200, and the vocabulary sizes for code and summary to 50,000 and 30,000, respectively. We train our proposed model using Adam optimizer (Kingma and Ba, 2015). The mini-batch size and dropout rate are 32 and 0.2. We set the maximum training epoch to 200, and use early stopping. We adopt beam search during inference time and set the beam size to 4.

4.2 Quantitative Result

Overall Result Table 2 shows the overall performance of the models. We present three proposed models: *AST-Only*, *AST+GCN* and *mAST+GCN*.

AST-Only is the proposed model without graph convolution layers. The model converts code snippets into ASTs and does not include graph convolutions. The sequenced AST nodes are given to the Transformer encoder and decoder. We present this model to verify how much ASTs are effective for source code summarization. It performs better than the baselines except for TransRel model proposed by Ahmad et al. (2020). This result shows that AST, which has more structural information on source code, is better than simple code for source code summarization.

AST+GCN is the proposed model with graph convolution layers but without AST modification (adding sibling edges). Code snippets are converted into ASTs and node representations are generated by graph convolutions. The sequenced graph-convolutioned AST nodes are input to the Transformer encoder and decoders. This model can verify how much the graph convolutions are useful. It shows better performance than the baselines.

mAST+GCN is the proposed model with modified ASTs by adding sibling edges and with graph convolution layers. It outperforms all baseline models. The performance improves by 0.91 and 0.3 BLEU, 0.74 and 0.35 METEOR, and 0.06 and 0.08

Methods	Java			Python		
	BLEU	METEOR	ROUGE-L	BLEU	METEOR	ROUGE-L
CODE-NN (Iyer et al., 2016)	27.60	12.61	41.10	17.36	09.29	37.81
Tree2Seq (Eriguchi et al., 2016)	37.88	22.55	51.50	20.07	08.96	35.64
RL+Hybrid2Seq (Wan et al., 2018)	38.22	22.75	51.91	19.28	09.75	39.34
DeepCom (Hu et al., 2018a)	39.75	23.06	52.67	20.78	09.98	37.35
API+CODE (Hu et al., 2018b)	41.31	23.73	52.25	15.36	08.57	33.65
Dual Model (Wei et al., 2019)	42.39	25.77	53.61	21.80	11.14	39.45
TransRel (Ahmad et al., 2020)	44.58	26.43	54.76	32.52	19.77	46.73
Proposed Model: AST-Only	44.76	26.75	53.93	31.59	19.16	45.48
Proposed Model: AST+GCN	45.30	**27.26**	54.45	32.41	19.77	46.35
Proposed Model: mAST+GCN	**45.49**	27.17	**54.82**	**32.82**	**20.12**	**46.81**

Table 2: Comparison of our proposed model with the baseline models.

ROUGE-L points in comparison to TranRel for Java and Python datasets, respectively. The proposed model with the AST modification has better performances on BLEU, METEOR, and ROUGE-L (excepts for METEOR in Java) than without the modification. This proves that modified ASTs help models learn more structural information of code than general ASTs.

Position of graph convolution layers We perform additional experiments with different positions of graph convolution layers. The positions are the front of the Transformer encoder, the back of the Transformer encoder, and both the front and back of the Transformer encoder.

Table 3 shows the performance scores according to the position of the graph convolution layers. The *front* model is the same as *mAST+GCN* which has one graph convolution layer in front of the encoder.

The *back* model does not have graph convolution layers in front of the Transformer encoder but has one next to the encoder. Nodes in an mAST are input to the encoder without graph convolutions, but graph convolutions are applied to the output of the encoder. Since the Transformer encoder can catch structural patterns in simple sequences, the graph convolution in the back of the encoder may work better than the one in front because it can feed more sharp structural information to the decoder.

The *front+back* model has two graph convolution layers: one in front and the other in the back of the Transformer encoder. It may catch much stronger structural patterns. The structurally encoded representations by a graph convolution layer are fed to the encoder and the output of the encoder is structurally enhanced once more by a graph convolution layer.

Table 3 shows the best result when located in front of the Transformer encoder. The *front+back* model is next to the *front*, and the *back* is the worst, which means that the graph convolution before the encoder is effective.

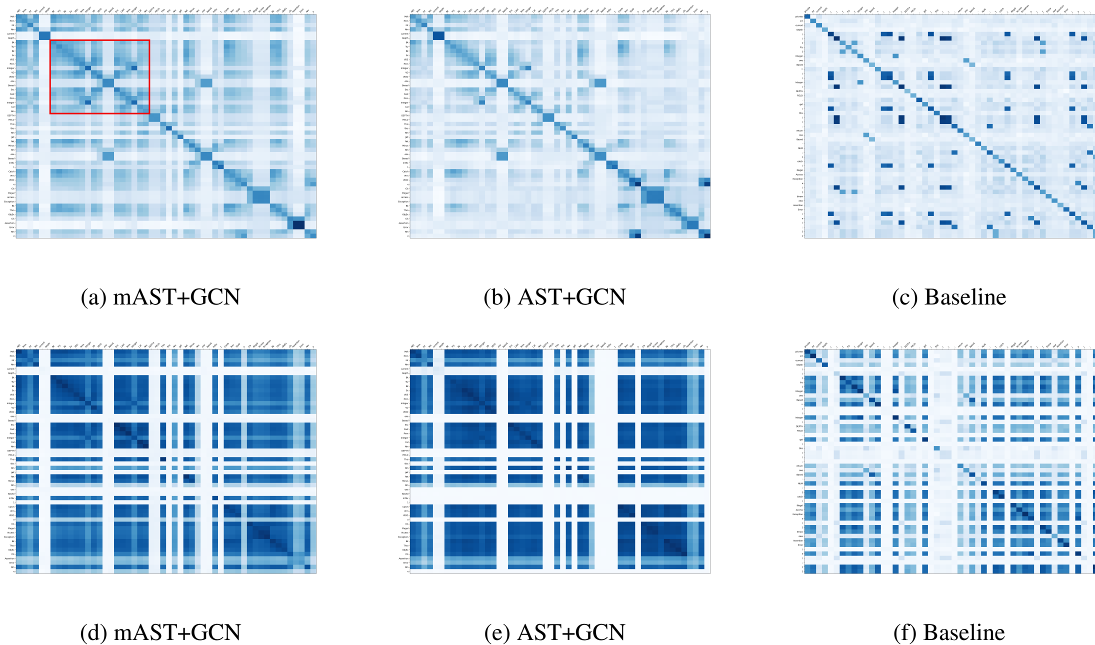
Position	BLEU	METOR	ROUGE-L
Java Dataset			
front	45.49	27.17	54.82
back	44.56	25.97	54.07
front+back	45.06	26.51	54.47
Python Dataset			
front	32.82	20.12	46.81
back	32.31	19.70	46.42
front+back	32.58	19.78	46.58

Table 3: Performance by position of graph convolution layers

Number	BLEU	METOR	ROUGE-L
Java Dataset			
1	45.49	27.17	54.82
2	44.72	26.70	53.87
3	44.14	25.62	53.46
Python Dataset			
1	32.82	20.12	46.81
2	31.80	19.31	45.56
3	30.91	18.41	44.24

Table 4: Performance by number of graph convolution layers

Since the Transformer has the ability to extract comprehensive features considering not only sequential but also structural information in sequences, the convolution layer in the back of the encoder may destruct such features and degrade the performance. However, the graph convolution layer in front of the encoder can help the encoder analyze structural patterns and to extract better features because it enhances structural information.



(a) mAST+GCN (b) AST+GCN (c) Baseline

(d) mAST+GCN (e) AST+GCN (f) Baseline

Figure 4: Attention maps of mAST+GCN, AST+GCN, and the baseline models for a code in Figure 5. (a), (b) and (c) are the attention maps of the first Transformer encoder layer and (d), (e), and (f) are the attention maps of the last Transformer encoder layer. A red box in (a) represents blocks in the snippets. We can see that the structural features are clearly captured in the red box. Our model effectively captures large and hierarchical structural features.

Number of graph convolution layers We analyze the performance according to the number of graph convolution layers. Graph convolutions are effective at capturing structural features, so more layers can help improve the performance. We tried one to three layers in front of the encoder.

Table 4 is the result of each model with 1, 2 and 3 layers. The results show that our proposed model with one graph convolution layer in front of the Transformer encoder has better performance than others. We think that this is because the graph structure of AST is not as complex as the general graph structure. So, the node representation has the over-smoothing problem when the graph convolution layer is stacked deep.

4.3 Qualitative Result

We present the qualitative analysis of our model. The attention map of an example code is compared to show how much our model catches the structural information. In order to further validate the performance metrics of our model, we perform a human evaluation on randomly sampled code snippets.

Attention Map Comparison We analyze attention maps of *mAST+GCN*, *AST+GCN* and the baseline by Ahmad et al. (2020) to verify how our model generates node representations compared to the others. Since we try to emphasize the structural information, we need to verify how much our model reflects the structural information to generate representations.

```
private int currentDepth(){
  try {
    Integer oneBased=((Integer)DEPTH_FIELD.get(this));
    return oneBased - 1;
  }
  catch ( IllegalAccessException e) {
    throw new AssertionError(e);
  }
}
```

Figure 5: An example of Java code. We draw attention maps for this in Figure 4.

We observe the attention maps for the sample code in Figure 5. We draw an attention map by evaluating the pairwise dot product of the output of a Transformer layer in the encoder. For *mAST+GCN* and *AST+GCN*, the output of a layer is the sequence of the mAST nodes, and for the baseline, it is the sequence of the program tokens.

We compare the attention maps of the first and the last Transformer layer in the encoder of each model. Figure 4a, 4b and 4c are the attention maps of the first layer of *mAST+GCN*, *AST+GCN* and the baseline. Figure 4d, 4e and 4f are the attention

maps of the last layer of *mAST+GCN*, *AST+GCN* and the baseline.

Rectangles on the diagonal as shown in a red box in Figure 4a represent blocks in the snippets. Since nodes or tokens in a block may have high similarities, we can see rectangles along with the diagonal. In the attention maps of the first layers, the rectangles are faint because the structural information has not been processed much yet, but we can see many distinct rectangles which means that the structural features are clearly captured.

If we compare the attention maps by our models and the baseline, there are many small rectangles in the baseline. On the contrary, in our models, we can see a few large rectangles in which there are small rectangles. We see a hierarchical structure in the attention maps of our model.

The fact that the baseline produces many small rectangles implies that the baseline can capture only small structural features. We can also note that these small structural features are smaller than statements, considering that the example snippets have only 4 lines. The baseline hardly captures large structural features.

On the other hand, our model effectively captures large and hierarchical structural features. We can easily identify rectangles that match with statements or blocks in the attention maps by our proposed models.

The attention maps from *mAST+GCN* and *AST+GCN* are very similar. However, we can see differences in each attention map of the first and the last layer. If we compare the first layer attention maps, the blocks of *mAST+GCN* are more distinct, which implies that the modification of ASTs by adding sibling edges is helpful to model structural information. If we compare the last layer attention maps, we can see that hierarchical structures of rectangles are clear in the map by *mAST+GCN*, which also says that the modification is effective to capture structural features.

Human Evaluation We performed human evaluation (Kryscinski et al., 2019) on the Java dataset to prove the effectiveness of how good summaries our model generates. We randomly choose 100 snippets and ask 4 people with knowledge of the Java language to evaluate the summaries. They are CS graduate students and have many years of experience in Java languages. We ask them to evaluate the 3 following aspects:

- Fluency (quality of the summary)
- Relevance (selection of the important content from source code)
- Coverage (selection of the whole content of source code)

	Fluency	Relevance	Coverage
Wins	**146**	**145**	**144**
Losses	130	135	140
Ties	124	120	116

Table 5: Human evaluation of the appropriateness of the generated summaries on the Java dataset. We ask annotators to select a more appropriate summary from two candidates generated by different models. Our proposed model outperforms the baseline.

We show pairs of summaries from our model and the baseline (Ahmad et al., 2020) to the annotators, and ask them to select one of win, tie, and loss in the three aspects, respectively. Our model shows superiority in all aspects as shown in Table 5. The scores of fluency and relevance are higher than the baseline, which means that our model generates more appropriate summaries using more natural expressions.

Figure 6 shows some examples of summaries for the qualitative comparison. We choose 6 Java snippet examples. We choose them from the snippets on which all the annotators make the same decision in each aspect. The three snippets on the left are the ones that the annotators choose win (our model is better) and the right ones for loss.

5 Related Work

As techniques and methods of deep learning have developed, researches for source code summarization have been studied based on sequence-to-sequence models. Iyer et al. (2016) proposed a model that performed source code summarization task for the first time. Allamanis et al. (2016) summarized the source code using a convolutional attention network model. Hu et al. (2018a) proposed an RNN-based sequence-to-sequence model using the pre-order traversal sequence of the abstract syntax tree. Also, Hu et al. (2018b) summarized source code with the knowledge on imported APIs using two encoders (source code encoder and API encoder). Ahmad et al. (2020) proposed a Transformer model with a relative position for summarizing source code. Wei et al. (2019) proposed a dual model that learned the code and summary sequence simultaneously. Wan et al. (2018) adopted

```
1  protected void removeClassifiers(int[] indices){
2  int i;
3  if (indices == null) {
4    m_ModelClassifiers.removeAllElements();
5  }
6  else {
7    for (i=indices.length - 1; i >= 0; i--)    m_ModelClassifiers.remove(indices[i]);
8  }
9  setModified(true);
10}
```

Fluency

Baseline Model : remove a certain indices .
Proposed Model : removes the specified datasets .
References : removes the specified classifiers .

```
1  private void buildPieces(){
2   pieces=new Piece[pathArray.size()];
3   Paint paint=new Paint();
4   Matrix matrix=new Matrix();
5   Canvas canvas=new Canvas();
6   for (int i=0; i < pieces.length; i++) {
         ...
35  }
36  Arrays.sort(pieces);
37 }
```

Fluency

Baseline Model : build the final bitmap - pieces to draw in animation
Proposed Model : build the final path to draw in animation
References : build the final bitmap-pieces to draw in animation

```
1  public static String encode(byte[] data){
2  int start=0;
3  int len=data.length;
4  StringBuffer buf=new StringBuffer(data.length * 3 / 2);
5  int end=len - 3;
6  int i=start;
7  int n=0;
8  while (i <= end) {
              ...
26 else   if (i == start + len - 1) {
27    int d=(((int)data[i]) & 0x0ff) << 16;
28    buf.append(legalChars[(d >> 18) & 63]);
29    buf.append(legalChars[(d >> 12) & 63]);
30    buf.append("==");
31  }
32  return buf.toString();
33 }
```

Relevance

Baseline Model : translates the specified byte array to a string .
Proposed Model : translates a byte array to a base 64 encoded string .
References : base64 encode the given data .

```
1 private static Stream<Method> extractMethods(Class clazz){
2  try {
3    Method[] methods=clazz.getMethods();
4    if (methods.length > 0) {
5      return Stream.of(methods);
6    }
7  }
8  catch (  Exception|Error e) {
9    LOG.warn("Problems loading class at startup: {}",clazz,e);
10  }
11  return Stream.empty();
12 }
```

Relevance

Baseline Model : extracts the method invocation from a methods if it exists .
Proposed Model : extracts the migration methods of the given method .
References : extract a set of methods from a given class .

```
1 public void deleteLeaves(String name){
2  for (int i=0; i < _leaves.size(); i++) {
3    CatalogTreeLeaf leaf=_leaves.get(i);
4    if (name.equals(leaf.getName())) {
5      _leaves.remove(i);
6    }
7  }
8 }
```

Coverage

Baseline Model : removes the specified name from the table
Proposed Model : removes a leaf with the specified name .
References : leaves can be used for many-to-many relations

```
1 public void releaseAnyConnection(Connection connection)throws SQLException {
2  try {
3    connection.close();
4  }
5  catch (  Exception ex) {
6    throw new RuntimeException(ex);
7  }
8 }
```

Coverage

Baseline Model : releases a connection back to the database
Proposed Model : releases a connection back .
References : release a non-shard-specific connection .

Figure 6: Examples for the qualitative comparison. The left examples are chosen from the ones that have 4 wins (all the annotators agree that our model is better), and the right examples are chosen from the opposite cases.

reinforcement learning to summarize source code. These approaches mainly focused on the sequential and context information of code, but little considered the structural information about the relationship between code tokens.

There are also studies that convert source code to AST to represent the structure of source code. Liang and Zhu (2018) proposed a tree-based recursive neural network to represent the syntax tree of code. Shido et al. (2019) represented source code using the tree structure encoder of tree-LSTM. Harer et al. (2019) adopted tree-transformer to encode the structure of ASTs. Fernandes et al. (2019) proposed the structured neural model for source code summarization. Alon et al. (2019) represented source code based on AST paths between pairs of tokens. LeClair et al. (2020) proposed a model that encoded the AST of source code using graph neural networks. These approaches utilized ASTs to capture structural features, but less considered the sequence characteristics of code in a program language.

6 Conclusion

We proposed a model that learned both the sequential and the structural features of code for source code summarization. We adopted the abstract syntax tree (AST) and graph convolution to model the structural information and the Transformer to model the sequential information. We also modified the AST to deliver more structural information.

We verified that modified ASTs and graph convolutions were very effective to capture the structural features of code through quantitative and qualitative analysis. We also showed the superiority of our model over the state-of-the-art for source code summarization by experiments and human evaluations.

Acknowledgments

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2019-0-00421, Artificial Intelligence Graduate School Program (Sungkyunkwan University)). This research was supported by the MSIT (Ministry of Science, ICT), Korea, under the High-Potential Individuals Global Training Program) (2019-0-01579) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation).

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007, Online. Association for Computational Linguistics.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2091–2100. JMLR.org.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72, Ann Arbor, Michigan. Association for Computational Linguistics.

Akiko Eriguchi, Kazuma Hashimoto, and Yoshimasa Tsuruoka. 2016. Tree-to-sequence attentional neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 823–833, Berlin, Germany. Association for Computational Linguistics.

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

Jacob Harer, Chris Reale, and Peter Chin. 2019. Tree-transformer: A transformer-based method for correction of tree-structured data. *arXiv preprint arXiv:1908.00449*.

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

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018b. Summarizing source code with transferred API knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 2269–2275. ijcai.org.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Thomas N. Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Wojciech Kryscinski, Nitish Shirish Keskar, Bryan McCann, Caiming Xiong, and Richard Socher. 2019. Neural text summarization: A critical evaluation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 540–551, Hong Kong, China. Association for Computational Linguistics.

Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 184–195.

Yuding Liang and Kenny Qili Zhu. 2018. Automatic generation of text descriptive comments for code blocks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 5229–5236. AAAI Press.

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

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80.

Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. Automatic source code summarization with extended tree-lstm. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 397–407.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 6559–6569.