# Code Summarization with Abstract Syntax Tree

Qiuyuan Chen[1⋆], Han Hu[2⋆], and Zhaoyi Liu[3⋆]

[1] College of Computer Science and Technology, Zhejiang University, Hangzhou, China
chenqiuyuan@zju.edu.cn
[2] School of Software, Tsinghua University, Beijing, China
hh17@mails.tsinghua.edu.cn
[3] School of Shenzhen Graduate, Peking University, Shenzhen, China, 518055
1701213615@sz.pku.edu.cn

**Abstract.** Code summarization, which provides a high-level description of the function implemented by code, plays a vital role in software maintenance and code retrieval. Traditional approaches focus on retrieving similar code snippets to generate summaries, and recently researchers pay increasing attention to leverage deep learning approaches, especially the encoder-decoder framework. Approaches based on encoder-decoder suffer from two drawbacks: a) Lack of summarization in functionality level; b) Code snippets are always too long (more than ten words), regular encoders perform poorly. In this paper, we propose a novel code representation with the help of Abstract Syntax Trees, which could describe the functionality of code snippets and shortens the length of inputs. Based on our proposed code representation, we develop Generative Task, which aims to generate summary sentences of code snippets. Experiments on large-scale real-world industrial Java projects indicate that our approaches are effective and outperform the state-of-the-art approaches in code summarization.

**Keywords:** Code Summarization · Code Clone · Code Representation

## 1 Introduction

There is much tacit knowledge in the source code which is not consistent with human intuition [1]. To better understand the source code, code summarization is used to transform this knowledge with low cost by automatically generating functional natural language description for a code snippet.

Typical code summarization includes summarizing commit message, log text, and code comment, which is vital to comprehend the source code. Researchers leverage Information Retrieval (IR) and learning-based techniques to generate comments automatically [11, 4].

IR techniques heavily rely on whether similar the code snippets can be retrieved, how similar the code snippets are, and fail to generate comments when encountering unmatched issues. However, if no similar code snippet exists, these techniques cannot output accurate summaries [2].

⋆ equal contribution

Current learning-based approaches often exploit deep neural network framework to build probabilistic models of source code [7, 2, 4] and prevalent learning-based approaches exploit mutant encoder-decoder [7, 15, 2] to learn the tacit knowledge in source code and transform them into descriptive natural language.

However, current learning-based approaches suffer from two drawbacks: a) Lack of summarization in functionality level. Vanilla deep neural networks always regard the code snippet as a sequence of plain words, only taking word-level features into account. When we meet a code snippet, we want to know more about the functionality of the code snippet, not word-level features. b) Code snippets are always too long (more than ten words). When faced with long sequences, as we know, regular encoders perform poorly.

Faced with drawback a), we need a new presentation of code snippets which is able to describe the functionality of code snippets. Abstract Syntax Tree (AST) is widely used in the field of Program Analysis. Figure 1(a) is a code snippet of counting characters in a string, Figure 1 is the visualized AST of Figure 1(a).

```
public int countCharacter(String str, char c){
    int num=0;
    for (int i=0;i<str.length();i++){
            if (str.charAt(i)==c)
                num++;
        }
        return num;
    }
```

(a) (b)

**Fig. 1.** Java Method and Its AST.

As shown in Figure 1, AST is a tree-structural representation of source code which describes code snippets in a specific programming language. The leaves of the tree usually refer to user-defined values which represent identifiers (such as *num*, *str*) or variable types (such as *String*, *int*) in the source code. Syntactic structures such as judgment statement (*ifStmt*) and loop (*ForStmt*) are represented as non-leaf nodes. In AST, every function module is represented as a path between two leaves or between a leaf and the root, which is called AST path. AST paths are marked red, yellow, blue or green respectively in Figure 1.

In this paper, to overcome drawback a), we utilize AST paths to represent code snippets. Faced with drawback b), as mentioned early, every code snippet is represented as AST paths in our paper, so we split every long code snippet to short AST paths which could improve the performance of our encoder.

Based on our new representation of code snippets, we build an encoder-decoder model to summarize source codes in Generative Task. Generative Task aims to generate

a sequence of words to describe the code snippet, the target of each code snippet is a sentence.

We collect 204,688 training pairs from 12 most popular open-source Java libraries, which are all starred more than 10,000 times on GitHub, and carry several experiments. The experiments results show that our model achieves better performance on three reliable metrics: Rouge-2, Rouge-L, and BLEU and can generate more accurate natural languages to describe the functionality of code snippets.

In summary, the contributions of our work are as follows:

- We propose a new representation of code snippets which is based on AST paths.
- We build an encoder-decoder model to summarize source codes based on our proposed code representation.
- We carry several experiments and compare the results with four baselines.

The rest of this paper is organized as follows: our approach is presented in Section 2. Section 3 introduces the experiments details. Section 4 introduces researches related to this work. The conclusion is shown in 5.

## 2 Approach

This section consists of two parts: Code Representation, and Generative Task.

### 2.1 Code Representation

As mentioned in Section 1, every input code snippet $X$ is represented as a series of AST paths. Every AST path consists of two leaves and non-leaf nodes, so every path is seen as a sequence of its non-leaf nodes' embedding vectors and a sequence of two leaves' tokens embedding vectors[4]. Let

$$V^{nodes} = (V_1^{node}, V_2^{node}, ..., V_i^{node}) \tag{1}$$

$$V^{leaves} = (V_1^{token}, V_2^{token}, ..., V_j^{token}) \tag{2}$$

where $V_{node}^{i}$ is denoted as the vector of $i_{th}$ node. We use bi-direction LSTM to encode the $V^{nodes}$ and $V^{leaves}$

$$h_1^{node}, h_2^{node}, ...h_i^{node} = BiLSTM(V_1^{node}, V_2^{node}, ..., V_i^{node}) \tag{3}$$

$$h_1^{leaves}, h_2^{leaves}, ...h_j^{leaves} = BiLSTM(V_1^{token}, V_2^{token}, ..., V_j^{token}) \tag{4}$$

and concatenate the bi-direction final hidden states of *LSTM* as the final representation of non-leaf nodes and leaf nodes.

$$E^{nodes}(V_1^{node}, V_2^{node}, ..., V_i^{node}) = [h_i^{node}; h_1^{node}] \tag{5}$$

$$E^{leaves}(V_1^{token}, V_2^{token}, ..., V_j^{token}) = [h_j^{leaves}; h_1^{leaves}] \tag{6}$$

[4] The camel-cased and underline identifiers are split into several words, for example, split *checkJavaFile* to three words: check, Java, File, so a leaf may consist of several tokens.

So every AST path $E^{path}$ is computed as

$$E^{path} = [E^{nodes}, E^{leaves}] \tag{7}$$

Suppose a snippet of code has $k$ AST paths, so the final representation of a code snippet is

$$E^{code} = (E_1^{path}, E_2^{path}, ..., E_k^{path}) \tag{8}$$

### 2.2 Generative Task

Generative Task aims to generate a sequence of words to describe the code snippet. The structure is shown in Figure 2. The task use $LSTM$ with $Attention$ as decoder to

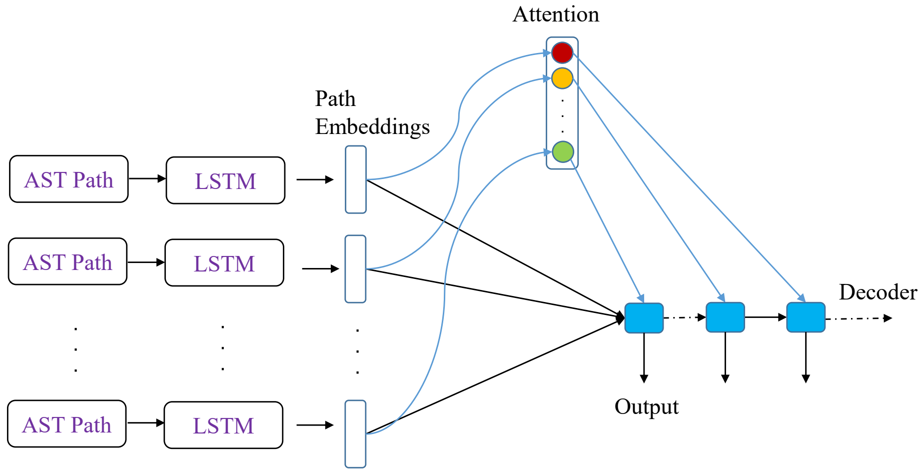


**Fig. 2.** The Structure of Generative Task

generate words one by one.

**Decoder** The initial state of decoder is the average of path embeddings $E^{average}$:

$$E^{average} = \frac{1}{k} \sum_{i=1}^{k} E_i^{path} \tag{9}$$

Other part is the same with the regular $LSTM$.

**Attention** Similar to the regular global attention, the decoder generates outputs while attending the encoder's output vectors $E^{path}$. At time $t$, let the hidden state of $LSTM$

is $h_t$, the output of $LSTM$ is $O_t$, apply $Softmax$ to convert every dot product between $h_t$ and $E_j^{path}$ to a probability distribution $\alpha_t^j$.

$$\alpha_t^j = softmax(E_j^{average} \cdot h_t) \tag{10}$$

where $\alpha_t^j$ refer to the $j_{th}$ result in $E^{average}$ at time $t$. The final context vector $C_t$ is computed as weight sum of $\alpha_t^1$, $\alpha_t^2$, ...$\alpha_t^j$ and $h_t$, let there are $k$ AST paths

$$C_t = \sum_j^k \alpha_t^j \cdot h_t \tag{11}$$

Then concatenate $C_t$ and $h_t$, feed it to a fully connected layer with $Softmax$ activation to the predict word at time $t$.

$$y_t = softmax(W_t \cdot [C_t; h_t]) \tag{12}$$

where $W_t$ is a $length_{dictionary} \times (d_{C_t} + d_{h_t})$ weight matrix.

The loss function of this task is Multi-Class Cross Entropy Loss Function.

## 3 Experiments

### 3.1 Datasets

We collect JDK source code and 12 most popular open source Java libraries, which are all starred more than 10,000 times on GitHub. The details of the dataset are shown in Table 1. We mainly extract Java methods and comments.

**Table 1.** Statistics of Collected Projects

| Project | Code lines | Comment lines | Java Files | Methods |
|---|---|---|---|---|
| JDK source code | 1,009,560 | 1,122,392 | 7,700 | 56,704 |
| druid | 290,239 | 59,086 | 4,023 | 25,836 |
| fastjson | 154,353 | 12,747 | 2,645 | 14,973 |
| fresco | 76,139 | 22,361 | 875 | 5,247 |
| glide | 76,726 | 14,546 | 649 | 3,712 |
| gson | 25,047 | 8,791 | 206 | 1,648 |
| guava | 501,702 | 171,651 | 3,170 | 19,940 |
| leakcanary | 5,703 | 1,512 | 81 | 261 |
| RxJava-2.x | 273,894 | 71,320 | 1,637 | 14,635 |
| spring-boot | 253,250 | 117,580 | 4,359 | 16,202 |
| spring-framework | 642,193 | 359,293 | 7,321 | 42,741 |
| tinker | 31,752 | 9,024 | 233 | 1,486 |
| zxing | 42,931 | 16,509 | 500 | 1,303 |
| TOTAL | 3,383,489 | 1,986,812 | 33,399 | 204,688 |

### 3.2 Data Preprocessing

For every Java method, the method body is treated as our inputs, method names, and method comments are extracted as targets. All words are lower-cased. The camel-cased and underline identifiers are split into several words, for example, split *checkJavaFile* to three words: check, Java, File, split *get_user_name* to three words: get, user, name. All punctuation marks are removed. We add *[CLS]* at the beginning of every sentence and add *[SEP]* at the end. *[UNK]* is used to represent words outside the vocabulary. After these steps, every word is tokenized to token.

Javaparser lib[5] is used to parse Java source codes. ASTParser lib[6] is used to build AST of code. We only use the first sentence of comments since they already describe the function of methods according to Javadoc guidance[7]. Several redundant comments, such as empty comments, one-word comments, and non-English comments, are filtered. We restrict both tasks to examples where the length of code snippet is at most 20 tokens, and the length of a comment is at most ten tokens.

Finally, we collect 1,04,963 pairs of $(Code, Comment)$ for Generative Task and split them into the training set, and test set in proportion with 8:2 after shuffling the pairs.

### 3.3 Experiment Setting

All $LSTM$ cells use 2-layers, and we set the dimensionality of the cells hidden states to 1024. In Generative Task, we set the token embeddings and hidden states to 1024. We use dropout [14] with p = 0.2. Adam[6] is used as our optimizer with an initial learning rate of 0.001 for optimization. Teacher forcing is used in both tasks' training phases, and the ratio of teacher forcing is 0.5. We use pytorch[8] and trained on cpus. In order to evaluate the quality of the output, following recent works in code summarization[8, 5, 3], we choose **Rouge-2** and **BLEU** as our metric of Generative Task, which is widely used in code summarization and machine translation.

To validate our approaches, we compare our model with four baselines in several state-of-the-art approaches:

- **CODE-NN:** CODE-NN [4] is the most famous model in code summarization.
- **Basic-Code-RNN & Code-GRU:** Basic-Code-RNN and Code-GRU are effective generation models recently proposed by Liang et al. [7].
- **Word-level Seq2Seq**: Sequence to Sequence model [15] is effective and widely used in the field of machine translation.

### 3.4 Experiment Results

Table 2 illustrates ROUGE-2, ROUGE-L, and BLEU results of 4 baselines and our model. The result shows that compared with other baselines, our approach outperforms

[5] https://github.com/javaparser/javaparser
[6] http://help.eclipse.org/mars/index.jsp
[7] https://www.oracle.com/technetwork/articles/java/index-137868.html
[8] https://pytorch.org/tutorials/

all baselines in three metrics, and achieves an improvement of 0.1546 Rouge-2 points, 0.0933 Rouge-L points, and 0.0611 BLEU point compared with the best results of other approaches.

The results of Word-level Seq2Seq and Generative Task in Table 2 indicate that our AST path features are more effective than word-level features, and our proposed new code presentation could demonstrate the features of code snippets better than regarding the code snippets as a plain sequence of words.

**Table 2.** ROUGE-2, ROUGE-L, BLEU of Baselines and our approaches

| Approaches | ROUGE-2 | ROUGE-L | BLEU |
| --- | --- | --- | --- |
| CODE-NN | 0.0685 | 0.1740 | 0.2333 |
| Basic CODE-RNN | 0.0682 | 0.1749 | 0.0322 |
| CODE-GRU | 0.1456 | 0.2004 | 0.0312 |
| Word-level Seq2Seq | 0.3225 | 0.5575 | 0.2640 |
| **Generative Task** | **0.4771** | **0.6508** | **0.3251** |

## 4 Related Work

In previous work, Information Retrieval (IR) approaches and training-based approaches are exploited to generate descriptive natural language for source code.

IR approaches are widely used in code summarization. They usually synthesize summaries by retrieving keywords from source code or searching comments from similar code snippets. Templates are utilized for several techniques [10, 13, 9] which select a subset of the statements and keywords from source code. Then the information is included from those statements and keywords in summary. Sridhara et al. [12] propose an automatic comment generator that identifies the content for the summary. Moreno et al. [10] proposed a template-based method to summarize Java classes rather their methods.

Recently, several studies generate natural language summaries leveraging deep learning approaches. Previous studies [11] predict comments using topic models and n-grams. Study of [1] applies a neural convolutional model with attention to summarize the source code snippets. Iyer et al. [4] propose CODE-NN that uses Long Short Term Memory (LSTM) networks with attention to produce summaries that describe C# code snippets and SQL queries. It takes source code as plain text and models the conditional distribution of the summary on two tasks, code summarization, and code retrieval. Hu et al. [2] combine the neural machine translation model and the structural information within the Java methods to generate the summaries automatically. Liang et al. [7] designed a customized recursive neural network called Code-RNN to extract features from the source code.

## 5 Conclusion

In this paper, we propose a novel code representation with the help of Abstract Syntax Trees, which could describe the functionality of code snippets and shortens the length

of inputs. Based on our proposed code representation, we develop the Generative Task, which aims to generate summary sentences of code snippets, and models for code summarization.

The experimental results show that our approaches, which outperform state-of-the-art baselines, are significantly effective and is able to generate high-quality summary sentences. We also introduce and open our dataset, which can be revisited by other researchers in the future.

## References


1. Allamanis, M., Peng, H., Sutton, C.: A convolutional attention network for extreme summarization of source code. In: International Conference on Machine Learning. pp. 2091–2100 (2016)
2. Hu, X., Li, G., Xia, X., Lo, D., Jin, Z.: Deep code comment generation. In: Proceedings of the 26th Conference on Program Comprehension. pp. 200–210. ACM (2018)
3. Hu, X., Li, G., Xia, X., Lo, D., Lu, S., Jin, Z.: Summarizing Source Code with Transferred API Knowledge. In: IJCAI. pp. 2269–2275 (2018)
4. Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: 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). vol. 1, pp. 2073–2083 (2016)
5. Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Mapping language to code in programmatic context (2018)
6. Kingma, D., Ba, J.: Adam: A method for stochastic optimization. Computer Science (2014)
7. Liang, Y., Zhu, K.Q.: Automatic generation of text descriptive comments for code blocks. In: Thirty-Second AAAI Conference on Artificial Intelligence (2018)
8. Ling, W., Grefenstette, E., Hermann, K.M., Kocisky, T., Senior, A., Wang, F., Blunsom, P.: Latent predictor networks for code generation (2016)
9. McBurney, P.W., McMillan, C.: Automatic documentation generation via source code summarization of method context. In: Proceedings of the 22nd International Conference on Program Comprehension. pp. 279–290. ACM (2014)
10. Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L., Vijay-Shanker, K.: Automatic generation of natural language summaries for java classes. In: 2013 21st International Conference on Program Comprehension (ICPC). pp. 23–32. IEEE (2013)
11. Movshovitz-Attias, D., Cohen, W.W.: Natural language models for predicting programming comments. In: Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers). vol. 2, pp. 35–40 (2013)
12. Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., Vijay-Shanker, K.: Towards automatically generating summary comments for java methods. In: Proceedings of the IEEE/ACM international conference on Automated software engineering. pp. 43–52. ACM (2010)
13. Sridhara, G., Pollock, L., Vijay-Shanker, K.: Generating parameter comments and integrating with method summaries. In: 2011 IEEE 19th International Conference on Program Comprehension. pp. 71–80. IEEE (2011)
14. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: a simple way to prevent neural networks from overfitting. The Journal of Machine Learning Research **15**(1), 1929–1958 (2014)
15. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Advances in neural information processing systems. pp. 3104–3112 (2014)